Group-Based Distributed Computing

Programming & Distributed Platform Model

By

Kazi Farooqui

Thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of
the requirements of the degree of

Doctor of Philosophy
in
Computer Science

under the auspices of
Ottawa Carleton Institute of Computer Science

University of Ottawa
Ottawa, Ontario,
Canada

February 2000

Copyright © 2000 Kazi Farooqui, Ottawa, Canada.

Acknowledgments

A number of people have, directly or indirectly, contributed to this thesis. It is my pleasure to have the opportunity to thank them. First of all, I would like to express my deepest gratitude to my thesis supervisor, Dr. Luigi Logrippo. His encouragement, support and guidance have been very instrumental during my Ph.D studies. I'm thankful to him for the numerous discussions that I have had with him and for the suggestions that I received from him. His careful reading of the thesis improved the content and its form. Luigi has not only been my supervisor, but also my friend. I could knock his door at any time and still be received with a friendly smile. Luigi, thanks very much!

The weekly meetings of the Telecommunications Software Engineering Research Group is an excellent place to test the ideas. I would like to thank all the members of this group for their useful suggestions and patient listening. Jacques Sincennes and Daniel Amyot were always available for discussion. Many thanks to them for sharing uncountable hours, ideas, and fruitful comments.

I would like to thank the members of the ISO /ODP group - Andrew Herbert, Jean-Bernard Stefani, Kerry Raymond, Peter Linington, Peter Schoo, Gerd Schrumann, and others, for the numerous discussions that I had with them through the internet. I would like to thank them for sharing their experience on distributed systems architectural principles.

A project of this scale can seldom be accomplished without the moral support and affection from the family. The first picture that comes to my mind is that of a tall and elegant girl who is gifted with a beautiful smile, my wife Shehla. She has been a constant source of affection for me. I wish to thank her for her companionship during my tough times and for tolerating my evenings and weekends at the office. I would like to apologize her for my occasional slipping into my 'thesis world' while talking to her. Our little daughter Juhi kept me cheerful with her innocent talk and with her playful activities. My parents have always been supportive of my intellectual pursuits. They have been a constant source of inspiration. I'm grateful to them for their encouragement and remain indebted to them for their care. Thank you mummy and pappa. Finally, I wish to remember my Dadima for the love she gave me. To her, I dedicate this thesis.

This work was made possible by the financial support of National Science and Engineering Research Council (NSERC), Telecommunications Research Institute of Ontario (TRIO), Motorola and Nortel.

Abstract

The synergy of the client-server computing model, the object group model, and the group communication model leads to a new distributed computing paradigm - the *group-based distributed computing paradigm.* This paradigm is characterised by the extension of the existing point-to-point client-server distributed computing model to a model that explicitly addresses *one-to-many* and *many-to-one* client-server interactions, as well as other aspects of group-orientation such as *synchronised message invocation*, *filtered message delivery*, etc. The group-based distributed computing applications are structured as a *client group* interacting with a *server group*. The paradigm of next generation of information systems will involve a large number of distributed objects which are structured as object groups and which interact in a client-server manner.

Much research has been done in the past in the area of group communication. However most of this research exists in the low-level support for group communication, such as different types of ordered and reliable multicast protocols, membership management protocols, virtual synchrony, replication techniques, etc. Most of this research provides only low-level pieces of the complete puzzle. The big picture involves a vision of *group-based distributed computing.* This vision calls for a shift of focus from low-level issues of group communication to the high-level issues of an overall distributed environment capable of supporting *group-based distributed computing applications*. This thesis is focussed on the dual models of the *distributed environment* - the *distributed programming model* and the *distributed platform model,* required for the support of group-based distributed computing applications.

At the programming-level, we describe a *communication primitive* (analogous to the *interrogation* or *remote procedure call* of the basic client-server model), that explicitly addresses one-to-many and many-to-one interactions between a client group and server group. It is named *group interrogation*. It allows a singleton client to access a server group in one call, through the mediation of a group proxy object, and to receive multiple and variable number of replies in a controlled manner in response to that call. Similarly it allows a singleton server to receive multiple service requests from the client group as a single group service request and to issue multiple replies, one for each component request, in response to a group service request.

The semantics of the proposed group interrogation primitive supports some of the sophisticated group communication requirements, such as *multiple reply delivery*, *variable reply delivery*, *group reply delivery*, *controlled reply delivery*, *terminable reply delivery*, and *ordered reply delivery.* Transparency is an important issue in a programming primitive. The proposed model allows the programmer to configure the level of group transparency by specifying different message distribution and collation policies.

At the distributed platform level, the focus is essentially on the *group support middleware platform*, which resides on top of the low-level group communication protocols. The thesis presents the software architecture of an *agent-based and policy-driven group support platform* in an implementation independent manner. This is an extensible, configurable and programmable software architecture which permits the separation of group coordination aspects from the application issues. The goal is to enhance the level of middleware support provided by the current generation of distributed platforms such as CORBA.

The thesis identifies a set of “middleware-level” *group support services* (GSSs), such as message *distribution service*, *collation service*, *synchronisation service*, *filtering service*, etc. commonly required by group-based applications and the corresponding *group support agents* (GSAs). The thesis presents a framework for the organisation of these group support agents. This framework is called the *group support machine* (GSM). The GSM serves as a framework for the identification of new group support services and for the identification of interactions that take place between the corresponding GSAs within the GSM, in

order to support different application requirements. Each component of the group-based application is supported by GSM. The *group support platform* (GSP) is a set of inter-connected GSMs which communicate with each other through an inter-GSM protocol (IGP). This protocol defines the nature and format of interaction between the peer GSAs within the distributed GSMs.

In this model, the GSAs manage the group communication and coordination patterns on behalf of the user applications, who influence the behavior of these agents by means of policy specifications. The idea is to describe the functionality required of the group support platform (GSP) in a declarative language. The thesis presents such a declarative group support requirements specification language, the *group policy specification language* (GPSL), for the specification of a rich set of application requirements with respect to different group support services such as message distribution, collation, synchronisation, filtering, etc. Therefore the GSP offers selective group transparency by allowing applications to specify group support policies.

Finally, the thesis describes the different types of *group coordination models* that are supported by the GSP and how the corresponding *group coordination patterns* can be specified using GPSL by a combination of basic message *distribution policy*, *collation policy*, *synchronisation policy*, and *filtering policy.*

This thesis is based upon the architectural principles underlying *object-based distributed systems architectures* such as RM-ODP, ANSA, ROSA, OMA, etc., and is scoped within the ODP computational and engineering models. The thesis presents an integration of diverse *distributed object computing technologies* such as the *client-server model*, *object group model*, *distributed agent model*, *policy-driven agent models*, *group coordination models* into an advanced *group-based distributed computing model.*

List Of Contents

CHAPTER 1 Introduction to the Problem Domain 1

1.1 Introduction 1
1.2 Group-Based Distributed Computing: Emergence of a New Paradigm 2
1.3 Relationship with Distributed Systems Architectures 3
  1.3.1 RM-ODP Viewpoint Model 3
    1.3.1.1 Enterprise Model 3
    1.3.1.2 Information Model 3
    1.3.1.3 Computational Model 3
    1.3.1.4 Engineering Model 4
    1.3.1.5 Technology Model 4
  1.3.2 Relationship to RM-ODP Viewpoint Models 4
1.4 Review of Existing Object Group Models 4
  1.4.1 Object Group Terminology 4
    1.4.1.1 Object Group 5
    1.4.1.2 Interface Group 5
    1.4.1.3 Group Member 5
    1.4.1.4 Member Name 5
    1.4.1.5 Member Role 5
    1.4.1.6 Group Identifier 5
    1.4.1.7 Group Administrator 5
  1.4.2 Object Group Classification Schemes 5
    1.4.2.1 Client and Server Groups 5
    1.4.2.2 Open and Closed Groups 6
    1.4.2.3 Active and Passive Groups 6
    1.4.2.4 Transparent and Non-Transparent Groups 6
    1.4.2.5 Replica and Heterogeneous Groups 6
    1.4.2.6 Static and Dynamic Groups 6
    1.4.2.7 Anonymous and Explicit Groups 6
    1.4.2.8 Source and Sink Groups 6
  1.4.3 General Applications 6
1.5 Review of ODP Client-Server Interaction Model 7
  1.5.1 ODP Computational Model Communication Primitives 7
    1.5.1.1 Interrogation 7
    1.5.1.2 Announcement 7
  1.5.2 Operation, Notification, and Termination Message Signatures 7
1.6 Scope of Group-Based Distributed Computing: Application Domains 8
1.7 Group-Based Distributed Computing: Dual Levels of Support 9
1.8 Scope and Aim of Thesis 11
  1.8.1 Programming-Level Support for Group-Based Distributed Computing 11
  1.8.2 Distributed Platform Support for Group-Based Distributed Computing 12
1.9 Related Work and Differences 13
  1.9.1 Programming Level 13
  1.9.2 Distributed Platform Level 15
1.10 Structure of Thesis 16

Part-1: Distributed Programming Model: A Group Communication Primitive

CHAPTER 2 Requirements of Programming-Level Group Communication Primitive 20
2.1 Introduction 20
2.2 Client Group and Server Group: Definition & Properties 21
  2.2.1 Client and Server Interfaces 21
    2.2.1.1 Client Interface 21
    2.2.1.2 Server Interface 21
  2.2.2 Client and Server Groups 21
    2.2.2.1 Server Group 21
    2.2.2.2 Client Group 21
    2.2.2.3 How are Client Groups Formed 21
    2.2.2.4 Client Group Invocation Properties 22
  2.2.3 Categories of Client and Server Groups 23
    2.2.3.1 Replica Client Group 23
    2.2.3.2 Homogeneous Client Group 23
    2.2.3.3 Heterogeneous Client Group 24
    2.2.3.4 Replica Server Group 24
    2.2.3.5 Homogeneous Server Group 24
    2.2.3.6 Heterogeneous Server Group 25
2.3 Programming-Level Communication Requirements of Group-Based Applications 25
  2.3.1 'Singleton-client' and 'Server-group' interaction requirements 25
  2.3.2 'Singleton-Server' and 'Client-Group' interaction requirements 27
2.4 Limitations of ODP Interrogation Primitive 29
2.5 Conclusion 30

CHAPTER 3 Group Interrogation: A Group Programming Primitive 32
3.1 Introduction 32
3.2 ODP-Based Group Programming Primitives 33
  3.2.1 Group Interrogation 33
  3.2.2 Group Announcement 33
  3.2.3 Group (Operation | Termination) Message 33
3.3 Semantics of Group Interrogation 34
  3.3.1 Multiple Invoker and Multiple Invokee semantics 34
  3.3.2 Group Invocation Semantics 34
  3.3.3 Message collation semantics 34
  3.3.4 Controlled Reply Delivery Semantics 35
  3.3.5 Terminable Reply Delivery Semantics 35
  3.3.6 Invocation Completion Reporting Semantics or Variable Reply Delivery Semantics 36
3.4 Signature of Group Interrogation 36
3.5 Group Message Construction: Collation Schemes 37
3.6 Basic Group Message Construction Schemes 37
  3.6.1 Matrix-mode message collation 38
    3.6.1.1 Group-Application-1: Managed Group - Manager Object Application 38
    3.6.1.2 Group Application-2: Modified Group Application-1 39

3.6.1.3 Principles of Matrix-Mode Message Collation 39
3.6.1.4 Implementation of matrix-mode message collation 41
3.6.2 Linear-mode message collation 41
3.6.2.1 Group Application-3: Group Computing 41
3.6.2.2 Group Application -4: Parallel Computing Group 42
3.6.2.3 Principles of Linear-Mode Message Collation 42
3.6.2.4 Observations of Linear-mode invocation collation 42
3.7 Group Interrogation vs. Group Transparency 43
3.8 Comparison between Interrogation and Group Interrogation 43
3.9 Need for Group-Oriented Objects 44
3.10 What is a Group-Based Distributed Application 44
3.11 What is a Group-Oriented (Client | Server) 45
3.12 Identification of Group Invocations in Group-Oriented (Client | Server) 46
3.12.1 Invocation Instance Identifier 46
3.12.2 Unique Identifiers 47
3.13 Communication between Group-Oriented (Clients | Servers) and Local Proxy 47
3.13.1 Client Side 47
3.13.2 Server Side 47
3.13.2.1 Single reply to all the clients based upon the group input 47
3.13.2.2 Individual reply to each client based upon the group input 48
3.13.3 Reply Handling Protocol between the Server object and Proxy object 48
3.14 Conclusion 49

Part-2: Distributed Platform Model: Middleware Support for Group-Based Applications

CHAPTER 4 Group Support Services: Requirements of the Group Support Platform 51
4.1 Introduction 51
4.2 Why Middleware Support for Group-Based Distributed Applications 51
4.3 What Middleware Services in the Group Support Platform and Why 52
4.3.1 Basic Group Support Services 53
4.3.2 Secondary Group Support Services 53
4.3.3 Group Management Services: 54
4.4 Basic Issues of Group Support Services: Elements of Group Support Policy 54
4.4.1 Issues of Message Distribution: Elements of Distribution Policy 55
4.4.2 Issues of Message Collation: Elements of Collation Policy 55
4.4.3 Issues of Message Synchronisation: Elements of Synchronisation Policy 56
4.4.4 Issues of Message Filtering: Elements of Filtering Policy 56
4.5 Conclusion 57

CHAPTER 5 Group Support Machine: An Organisation of Group Support Services 59
5.1 Introduction 59
5.2 Group Support Agents: Realisation of Group Support Services 59
5.3 Group Support Machine: Configuration of Group Support Agents 60
5.3.1 Parallel Configuration of Group Support Agents 60
5.3.2 Functioning of Group Support Machine 61

5.4 Group Support Platform: A Parallel Configuration of Inter-Communicating GSMs 62
5.5 Agent-Based Approach and Separation of Communication Functions 63
5.6 Group Support Machine: An External, Configurable, and Programmable Architecture 64
  5.6.1 Separation of group-coordination aspects from application aspects 64
  5.6.2 Extensible and configurable architecture 64
  5.6.3 Programmable and policy-driven architecture 64
  5.6.4 Support for group transparency and group awareness 64
5.7 Conclusion 64

CHAPTER 6 An Abstract Model of Group Support Machine 65
6.1 Introduction 65
6.2 Middleware Box Between Group Member and Network: External Interfaces of GSM 65
  6.2.1 GSM - Group Member Interface 65
    6.2.1.1 GSM Invocation Interface (GII): 65
    6.2.1.2 GSM Management Interface (GMI) 66
  6.2.2 GSM - Network Interface 67
6.3 GSM Components 68
  6.3.1 G-Agent 68
  6.3.2 D-Agent 68
  6.3.3 C-Agent 69
  6.3.4 S-Agent 71
  6.3.5 F-Agent 72
  6.3.6 MM-Agent 72
  6.3.7 P-Agent 72
6.4 Interaction between GSAs in the GSM: Internal Interfaces of GSM 73
  6.4.1 Interaction between D-Agent and C-Agent: Coordination between basic group support functions73
  6.4.2 Interaction between D-Agent and S-Agent: Synchronise before message distribution 74
  6.4.3 Interaction between D-Agent and F-Agent: Insert the filtering constraints before message distribution at client side75
  6.4.4 Interaction between C-Agent and S-Agent: Synchronise before message delivery75
  6.4.5 Interaction between C-Agent and F-Agent: Filter the received messages before delivery 76
  6.4.6 Interaction between MM-Agent and other GSAs: Communicate group membership information 76
6.5 Conclusion 76

CHAPTER 7 Group Coordination Models: Platform Support and Policy Specification 77
7.1 Introduction 77
7.2 Basic Group Coordination Models 78
7.3 Basic Issues in Group Coordination Models 79
7.4 The Basic Message Distribution Model 80
  7.4.1 Group Application-1: Stock Exchange Application 80
  7.4.2 Message Distribution Requirements & Policy Specification 81

7.5 Advanced Message Distribution Models: Smart D-Agents 81
  7.5.1 Splitting Transformation 81
  7.5.2 Message Splitting Requirements & Policy Specification 81
  7.5.3 Renaming Transformation 82
  7.5.4 Group Application-2: Parallel Computational Group 83
  7.5.5 Renaming Requirements & Policy Specification 84
7.6 Reply Collation and Delivery Models 84
  7.6.1 Group Application-3: Stock Inventory System 86
  7.6.2 Delivery of Group Termination of a Single Reply Type: Matrix-Mode Collation 87
    7.6.2.1 Reply Collation Requirements & Policy Specification 87
    7.6.2.2 Transparency and Policy Interpretation 88
  7.6.3 Delivery of Group Termination of a Single Reply Type: Linear-Mode Collation 88
    7.6.3.1 Reply Collation Requirements & Policy Specification 88
    7.6.3.2 Transparency & Policy Interpretation 88
  7.6.4 Unordered Delivery of Singleton Terminations of a Reply Type 88
    7.6.4.1 Unordered Reply Delivery Requirement and Policy Specification 89
    7.6.4.2 Transparency & Policy Interpretation 89
  7.6.5 Ordered Delivery of Singleton Terminations of a Reply Type 89
    7.6.5.1 Ordered Reply Delivery Requirement & Policy Specification 90
    7.6.5.2 Transparency & Policy Interpretation 90
  7.6.6 Unordered Delivery of Multiple Reply Types as Singleton Terminations 90
    7.6.6.1 Reply Collation & Delivery Requirements and Policy Specification 90
    7.6.6.2 Transparency & Policy Interpretation 92
  7.6.7 Unordered Delivery of Multiple Reply Types as Group Terminations 92
    7.6.7.1 Reply Collation & Delivery Requirement and Policy Specification 93
    7.6.7.2 Transparency & Policy Interpretation 93
  7.6.8 Ordered Delivery Multiple Reply Types as Singleton Terminations 93
    7.6.8.1 Reply Collation & Delivery Requirement and Policy Specification 93
    7.6.8.2 Transparency & Policy Interpretation 94
  7.6.9 Disabling the Delivery of Other Reply Types by a Preferred Reply Type 94
    7.6.9.1 Group Application-4: Mobile Telecommunications 94
    7.6.9.2 Reply Collation & Deliver Requirement and Policy Specification 95
    7.6.9.3 Transparency & Policy Interpretation 95
  7.6.10 Choice between Multiple Reply Types 96
    7.6.10.1 Group Application-5: Group Survey 96
    7.6.10.2 Reply Collation & Deliver Requirement and Policy Specification 97
    7.6.10.3 Transparency & Policy Interpretation 97
    7.6.10.4 Group Application-6: Scheduling Group Meeting 97
    7.6.10.5 Reply Collation & Deliver Requirement and Policy Specification 97
    7.6.10.6 Transparency & Policy Interpretation 97
7.7 'Group-Service' Request Models: Service Request Collation Models 98
  7.7.1 Group Application-7: Network Management Application 98
  7.7.2 Constructinga'Group-Service'Request:Matrix-ModeCollation&PolicySpecification 99
  7.7.3 Transparency & Policy Interpretation 100
  7.7.4 Group Application-8: Target Location Acquisition Sonar System 100
  7.7.5 Constructing a Service Request from Partial Service Requests: Linear-Mode Collation & Policy Specification101

7.8 Replies to Group-Service Request: Reply Distribution Models 102
  7.8.1 Multiple Replies to Group-Service Request 102
  7.8.2 Transparency & Policy Interpretation 102
  7.8.3 Single Reply to Group-Service Request 102
7.9 Synchronised Invocation Model 103
  7.9.1 Why Synchronised Invocation in the Client Group 103
  7.9.2 What are Synchronisation Events in Client Groups 104
  7.9.3 What are Synchronisation Messages 105
  7.9.4 Communication between the Client Object and the S-Agent 106
  7.9.5 Group Application-9: Coordinated Testing Application 106
  7.9.6 Synchronisation Requirements & Policy Specification 110
  7.9.7 Interaction between GSM Agents to Support Synchronised Message Distribution from Client114
  7.9.8 Transparent & External Support for Synchronised Invocation in the GSM 117
7.10 Filtered Message Delivery Model 117
  7.10.1 Why Filtered Message Delivery in the Server Group 117
  7.10.2 Communication between the Server Object and F-Agent 118
  7.10.3 Group Application-10: A Printer-Pool 119
  7.10.4 Filtering Requirements & Policy Specification 119
  7.10.5 Interaction between GSM Agents to Support Filtered Message delivery to Server Object 120
  7.10.6 Transparent & External Support for Filtered Invocation 122
7.11 Conclusion 122

CHAPTER 8 Group Policy Specification Language: An Introduction 123
8.1 Introduction 123
8.2 Why Group Policy Specification Language 123
8.3 Basic Elements of GPSL 124
8.4 Syntax and Semantics of Group Policy Primitives 125
  8.4.1 Distribution Policy Primitive 125
    8.4.1.1 DPP Syntax 125
    8.4.1.2 DPP Semantics 126
  8.4.2 Collation Policy Primitive 126
    8.4.2.1 CPP Syntax 126
    8.4.2.2 CPP Semantics 126
  8.4.3 Synchronisation Policy Primitive 126
    8.4.3.1 SPP Syntax 126
    8.4.3.2 SPP Semantics 127
  8.4.4 Filtering Policy Primitive 127
    8.4.4.1 FPP Syntax 127
    8.4.4.2 FPP Semantics 128
8.5 Syntax and Semantics Of GPSL Elements 128
  8.5.1 Message Specifier Elements 128
  8.5.2 Membership Specifier Elements 128
  8.5.3 Cardinality Specifier Elements 128

8.5.4 Time Specifier Elements 129
8.5.5 Combination Mode Specification Elements 130
8.5.6 Attribute Combination Specification Elements 130
8.5.7 Message Ordering Specification Elements 130
8.6 Conclusion 132

CHAPTER 9 Inter-GSM Protocol 133
9.1 Introduction 133
9.2 Why Protocol between GSMs 133
9.3 Peer GSAs in Inter-GSM Protocol 134
9.4 A General Format of the Inter-GSM Protocol Data Unit 134
9.5 Encoding of GPDUs 135
9.6 Inter-GSM Protocol between D-Agent and C-Agent 136
9.6.1 Application Message Communication between D-Agent & C-Agent 136
9.6.2 Marshalling of Application Messages in GPDUs 137
9.6.3 Group Exception Handling Protocol Between C-Agents 137
9.7 Inter-GSM Protocol between Peer S-Agents 138
9.7.1 Solicited Synchronisation Protocol 139
9.7.2 Unsolicited Synchronisation Protocol 140
9.8 Inter-GSM Protocol between Peer F-Agents 141
9.9 Inter-GSM Protocol between Peer MM-Agents 143
9.9.1 Distributed Membership Monitoring 144
9.9.2 Membership Change Notification 144
9.10 Inter-GSM Protocol over Multicasting Protocol 146
9.10.1 Group Communication Layer 146
9.10.2 GSM - GCL Interface 147
9.11 Conclusion 147

CHAPTER 10 Group Support Platform: Implementation and Performance 149
10.1 Introduction 149
10.2 Implementation Details 149
10.2.1 Implementation of GSM Agents 149
10.2.1.1 GSM Class 150
10.2.1.2 G_Agent Class 150
10.2.1.3 D_Agent Class 151
10.2.1.4 C_Agent Class 153
10.2.1.5 P_Agent Class 154
10.2.2 Implementation of Inter-Agent Invocations 154
10.2.3 Implementation of Inter-GSM Communication 155
10.2.4 Implementation of Inter-GSM Protocol 155
10.2.5 Implementation Distribution and Collation Policies 156
10.2.6 Implementation of an API for Group Interrogation Primitive 156
10.2.6.1 Implementation of Unsolicited Group Reply Delivery - API 157
10.2.6.2 Implementation of Solicited Multiple Reply Delivery - API 157
10.2.6.3 Implementation of Unsolicited, Multiple and Terminable Reply Delivery - API 158

10.3 Performance Aspects 158
10.3.1 Message count 159
10.3.2 Message Complexity 159
10.3.3 Communication Network Speed 160
10.3.4 Message Marshalling and Un-marshalling Overhead 160
10.3.5 Intra-GSM Invocations Overhead 160
10.3.6 Internal Buffer Sizes and Queue Lengths Considerations 161
10.3.7 Concurrency and Multi-threaded architecture aspects 161
10.3.8 Timers 161
10.3.9 Collation Processing Overhead 162
10.3.10 Other Group Processing Overhead 162
10.3.11 Reliability and Robustness 162
10.3.12 Scalability 162
10.3.13 Ease of Use 163
10.3.13.1 Ease of use of GSM Invocation (GII) Interface and Group Interrogation primitive163
10.3.13.2 Ease of use of Group Policy Programming Interface 164
10.4 Comparison of Group Support Platform with CORBA Middleware 164
10.4.1 Comparison at Programming-Level 164
10.4.1.1 Group Interrogation vs. Remote Procedure Call 164
10.4.1.2 Ease of group request invocation 168
10.4.1.3 Support for Advanced Programming-level facilities in GSP vs. Corba 168
10.4.2 Comparison at Platform-Level 169
10.4.2.1 Middleware functions of GSP vs. Corba 169
10.4.2.2 Platform programmability Capability in GSP vs. Corba 169
10.5 Case Studies 170
10.5.1 Case Study-1: Group Reply Delivery, Matrix-Mode Collation 170
10.5.2 Case Study-2: Group Reply Delivery, Linear-Mode Collation 170
10.5.3 Case Study-3: Solicited Reply Delivery 171
10.5.4 Case Study-4: Group Request Deliver and Reply Distribution 171
CHAPTER 11 Conclusions and Directions for Future Work 173
11.1 Conclusion and Contribution of Thesis 173
11.1.1 Contribution at the Programming-Level 173
11.1.2 Contribution at the Platform-Level 174
11.2 Directions for Future Work 176
11.2.1 Research on Group-Oriented Programming Languages & Systems 176
11.2.2 Integration of GSM Model in CORBA 177
11.2.3 Extension of GSM Model 177
11.2.4 Extension to Group Policy Specification Language 178
List of References 179
APPENDIX BNF of Group Policy Specification Language (GPSL) 191

List Of Figures

Fig. 1.1 Group-Based Distributed Computing Model: Synergy of Client-Server Model & Object-Group Model 2
Fig. 1.2 Modeling of Conventional Distributed Applications as Group-Based Distributed Applications using Object Group Model and Client-Server Model 9
Fig. 1.3 Group-Based Distributed Computing: Application Domains 9
Fig. 1.4 Area of Research: Lightly Shaded Areas 10
Fig. 2.1 Homogeneous Client Group: Each member invokes instances of the same operation signature 22
Fig. 2.2 Heterogeneous Client Group: Each member invokes an instance of different operation signature 23
Fig. 2.3 Client object interrogates a server group 26
Fig. 2.4 Client Group interrogates a server object 28
Fig. 3.1 Interrogation Signature 36
Fig. 3.2 Matrix-mode message collation: An example of Manager Object and Managed Group Interaction 39
Fig. 3.3 Group Message Stub Using Matrix-Mode Collation: Array Structure Implementation 40
Fig. 3.4 Linear-mode message collation: An Example of Group Computing 41
Fig. 3.5 Group-Based Distributed Application and the Group Support Platform. 45
Fig. 3.6 Protocol between group-oriented (client | server) and proxy 48
Fig. 4.1 Group Support Platform: Middleware & Group Communication Services 53
Fig. 5.1 Group Support Machine (GSM): Configuration of Group Support Agents 61
Fig. 5.2 Group Support Platform (GSP): A Distributed Agent Model 63
Fig. 6.1 A Model of Group Support Machine (GSM) 67
Fig. 6.2 A Model of Policy-Driven Group Support Machine 70
Fig. 7.1 Group Coordination Model: Combination of coordination behavior and group organisation 78
Fig. 7.2 Stock Exchange Application: A Group-Based Distributed Application 80
Fig. 7.3 Message Distribution Policy Specification 81
Fig. 7.4 Splitting Policy Specification 82
Fig. 7.5 A Parallel Computational Group 83
Fig. 7.6 Renaming Policy Specification 84
Fig. 7.7 Stock Inventory System 86
Fig. 7.8 Reply collation and delivery policy of a single group termination (matrix-mode) 87
Fig. 7.9 Reply collation and delivery policy of a single group termination (linear-mode) 88
Fig. 7.10 Unordered delivery of singleton terminations of a reply type 89
Fig. 7.11 Ordered delivery of singleton terminations of a reply type 90
Fig. 7.12 Policy Specification for interleaved delivery of instances of multiple reply types 91
Fig. 7.13 Policy specification for Unordered Delivery of Multiple Reply Types as Group Terminations 92
Fig. 7.14 Policy Specification for Ordered Delivery of Multiple Reply Types as Singleton Terminations 94
Fig. 7.15 Group Interrogation in Mobile Telecommunications 95
Fig. 7.16 Policy Specification for Disabling the Delivery of Other Reply Types by a Preferred Reply Type 96
Fig. 7.17 Policy Specification for Choosing between reply types based upon cardinality requirements 96
Fig. 7.18 Policy Specification for Choosing between reply types based upon sender identity 97
Fig. 7.19 Group Interrogation in Telecommunications Network Management 99
Fig. 7.20 Operation Collation Policy Specification 100
Fig. 7.21 Group Interrogation in Sonar System 101

Fig. 7.22 Linear-Mode Collation of partial service requests 101
Fig. 7.23 Multiple Replies Distribution Policy 102
Fig. 7.24 Single Reply Distribution Policy 102
Fig. 7.25 Coordinated Testing Application 109
Fig. 7.26 Synchronisation Policy Specification for the S-Agent of TAdmin 110
Fig. 7.27 Synchronisation Policy Specification for the S-Agent of TA-1 110
Fig. 7.28 Synchronisation Policy Specification for the S-Agent of TB-1, TB-2 112
Fig. 7.29 Synchronisation Policy Specification for the S-Agent of TC-1 112
Fig. 7.30 Synchronisation Policy Specification for the S-Agent of TE-1 113
Fig. 7.31 Synchronisation Policy Specification for the S-Agent of TAdmin - (for Grade-B() message) 113
Fig. 7.32 Synchronisation Policy Specification for the S-Agent of TAdmin (for Object_Partially_Tested() message) 114
Fig. 7.33 Synchronised Message Distribution Policy 115
Fig. 7.34 Coordination between GSM Agents to Support Synchronised Message Distribution from Client 115
Fig. 7.35 Client’s Filtering Policy Specification 119
Fig. 7.36 Server’s Filtering Policy Specification 120
Fig. 7.37 Coordination between GSM Agents to Support Filtered Message Delivery (Server Side) 121
Fig. 9.1 A General Format of GSM Protocol Data Unit (GPDU) 135
Fig. 9.2 Inter-GSM Protocol between D-Agent & C-Agent 137
Fig. 9.3 Inter-GSM Protocol between S-Agents 139
Fig. 9.4 S-NTF-GPDU Format 141
Fig. 9.5 F-PAR-GPDU Format 142
Fig. 9.6 F-RES-GPDU Format 143
Fig. 9.7 Inter-GSM Protocol between F-Agents 143
Fig. 9.8 Inter-GSM Protocol over Multicast Protocol 147
Fig. 10.1 Which Agents are implemented 150
Fig. 10.2 GSM Implementation: GSM Agents and their Interaction 152

List Of Tables

Table 2.1:	Categories of Client Group	24
Table 2.2:	Categories of Server Group	24
Table 2.3:	Limitation of ODP Interrogation primitive	29
Table 3.1:	Comparison of Matrix and Linear mode Collation Schemes	43
Table 3.2:	Interrogation vs. Group Interrogation	44
Table 3.3:	Group-Oriented (Clients | servers)	46
Table 5.1:	Group Support Services Requirement on the Client and Server side	60
Table 6.1:	Interaction of D-Agent with other Agents before & after message distribution	75
Table 6.2:	Interaction of C-Agent with other Agents before message delivery to (Client | Server)	76
Table 7.1:	Reply Collation and Delivery Schemes	85
Table 8.1:	Relationship between Basic issues of Group Support Services and Elements of GPSL	125
Table 8.2:	Semantics of Collation Operators	131
Table 8.3:	Combined Semantics of Collation Time, Collation Cardinality, and Collation Mode	132
Table 9.1:	A Catalogue of GPDUs	145
Table 10.1:	Corba vs. GSP: How do they compare w.r.t. Crucial Performance Metrics	166
Table 10.2:	GSP vs. Corba: What are the Other Trade-Offs	169

CHAPTER 1 Introduction to the Problem Domain

Abstract

This chapter describes the subject of the thesis, beginning with the discussion of the background and motivation that led to the research and the material presented in the thesis. The focus of the thesis is on an emerging distributed computing paradigm - the "group-based distributed computing". We describe this paradigm. We describe the relationship of our work with the existing distributed systems architectures and introduce some basic models which are the basis of the thesis. We delineate and define the scope and the aim of the thesis.

1.1 Introduction

Group communication is increasingly becoming an important communication paradigm of modern distributed systems. Much research has been done in the past in the area of group communication, such as reliable group communication protocols, ordered multicast protocols, membership management protocols, virtual synchrony, object-group models, group communication paradigms, etc. However most of this research provides only low-level pieces of the complete puzzle. The big picture involves a vision of *group-based distributed computing* and of a distributed environment capable of supporting *group-based distributed computing applications*. This vision calls for a shift of focus from low-level issues of *group communication* to the issues of an overall *distributed environment* required for the support of *group-based distributed applications*, in which the users and application programmers have easy and flexible access to the services provided by the group support infrastructure.

We believe that the synergy of client-server computing model, the distributed object model, object group model and the group communication model leads to a compelling and powerful distributed computing paradigm - the *group-based distributed computing paradigm* (see figure 1.1). This paradigm is characterised by the extension of the existing point-to-point client-server distributed computing model to a model that explicitly addresses *one-to-many* and *many-to-one* client server interactions, as well as other aspects of group-orientation such as *synchronised message invocation*, *filtered message delivery*, etc. This paradigm involves a large number of distributed objects which are structured as *object groups* and which interact in a *client-server* manner.

Experience with the existing client-server distributed computing has demonstrated the importance of "*programming-level*" and "*distributed platform-level*" support for distributed computing applications. Such dual support greatly simplifies the task of building "client-server" distributed applications. The distributed platform model provides the "*middleware services*" necessary to support remote "client-server" interactions. The distributed programming model, amongst other things, provides the inter-object communication support, such as a "*remote procedure call*" which gives the programmer a powerful handle to invoke a remote object, as if it were a local one, and to receive the reply. Such a high-level support is missing in the case of group-based distributed computing model. The successful and large-scale deployment of group-based distributed computing applications is heavily dependent upon the availability of these two-levels of support. Our thesis addresses these two aspects: the *distributed programming-level* support and the *distributed platform* support required for the successful and large-scale deployment of group-based distributed computing applications.

1.2 Group-Based Distributed Computing: Emergence of a New Paradigm

We are witnessing the emergence of two new distributed computing paradigms: *mobile computing* and *group-based distributed computing*. Although mobile and group-based distributed applications have been around for quite some time now, there has been an enormous increase in the demand and deployment of these applications in the recent past. The major factors responsible for the growth of these distributed system paradigms are inherent in the very nature of the *business enterprises* we want to develop. The basic components of our distributed enterprise are the people or the application objects (software entities) that represent them. These objects are *mobile* and they work as a *group* in *client-server* manner to achieve their enterprise-specific goals. The growing importance and the large-scale deployment of these applications calls for a sound architectural framework for the support of these applications. This thesis is dedicated to the development of such an architectural framework for the support of group-based distributed computing applications.

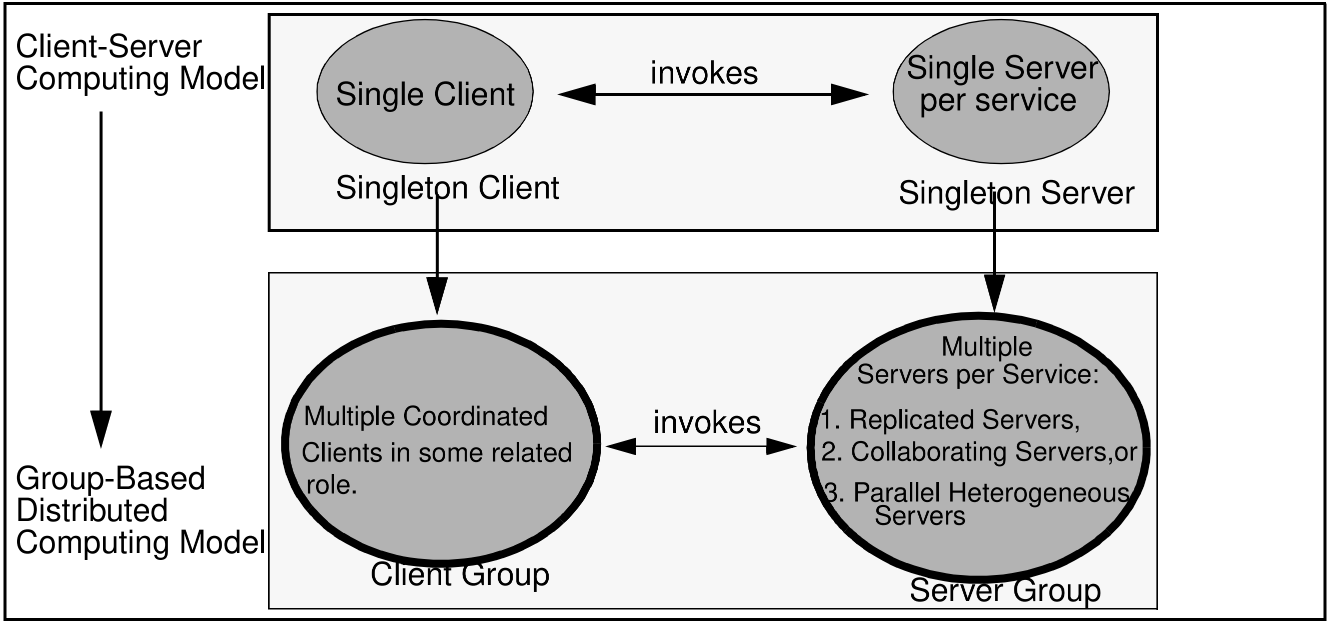

Client-Server Computing Model
Single Client
invokes
Single Server per service
Singleton Client
Singleton Server
Multiple Coordinated Clients in some related role.
invokes
Multiple Servers per Service:
1. Replicated Servers,
2. Collaborating Servers,or
3. Parallel Heterogeneous Servers
Group-Based Distributed Computing Model
Client Group
Server Group

Fig. 1.1 Group-Based Distributed Computing Model: Synergy of Client-Server Model & Object-Group Model

A major force responsible for the rapid growth of group-based distributed computing is inherent in the way in which distributed applications are organised (see figure 1.1). The concept of a '*single server per service*' is being replaced in many domains by the concept of '*multiple servers per service*'. One-to-one correspondence between services and server is neither realistic nor desirable in distributed environments. There is a clear separation of concepts of *services* and *servers*. These services could be realised by a set of replicated servers or a set of parallel heterogeneous servers or a set of collaborating servers. Clients need an interface to services rather than to individual servers. Similarly, the concept of a '*singleton client*' is being replaced in some domains by the concept of '*multiple coordinated clients in some* *related roles*'. These distributed applications could either represent a set of replicated clients or they could be a set of non-replicated, but coordinated clients in some related roles. We address these issues within the model of group-based distributed computing.

1.3 Relationship with Distributed Systems Architectures

Our model of group-based distributed computing is based upon the architectural principles underlying distributed system architectures such as the Reference Model of Open Distributed Processing (RM-ODP) [1 - 8], Advanced Networked Systems Architecture (ANSA) [9 - 15], RACE Open Services Architecture (ROSA) [16 - 17], OMG's Common Object Request Broker Architecture (CORBA) [18] and other related models [21 - 24]. In particular, our model is structured within the architectural framework of RM-ODP and its predecessor, the ANSA. Here we present a brief review of RM-ODP and explain how our work relates to a subset of its component models.

RM-ODP is a generic architecture for the design of object-based distributed systems. It can be considered as a meta-standard to coordinate and guide the development of domain-specific ODP standards, such as the Telecommunications Information Networking Architecture (TINA) [25 - 32] in the telecom domain and the Open Distributed Management Architecture (ODMA) [33] in the management domain.

1.3.1 RM-ODP Viewpoint Model

RM-ODP prescribes a set of abstractions or projections on a distributed system called *viewpoint models*, in order to deal with the full complexity of distributed systems. Each viewpoint model reveals different aspects of the same system. A viewpoint is a representation of the system with emphasis on a specific set of concerns, and the resulting representation is an abstraction of the system, i.e., a description of the system which highlights certain aspects of the system relevant to the viewpoint and abstracts others. Different viewpoints on the systems address different aspects or concerns of the system. The ODP prescribes the five viewpoint models. They are the *Enterprise Model*, the *Information Model*, the *Computational Model*, the *Engineering Model*, and the *Technology Model*.

1.3.1.1 Enterprise Model

The *enterprise model* is directed to the *needs* of the users of an information system. It describes the (distributed) system in terms of answering what it is required to do for the enterprise. It is the most abstract of the ODP framework of viewpoints stating high level enterprise *requirements* and *policies*.

1.3.1.2 Information Model

The *information model* focuses on the information content of the enterprise. The information modeling activity involves identifying *information elements* of the system, *manipulations* that may be performed on information elements, and the *information flows* in the system.

1.3.1.3 Computational Model

The *computational model* is a framework for describing the structure of a distributed application in terms of application components (called *computational objects*) and the interactions that occur between them, independent of any underlying "distributed platform". The computational model is a (distributed) object world populated with concurrent computational objects interacting with each other, in a *distribution transparent* manner, by invoking messages at their interfaces (called *computational interfaces*). An object can have multiple interfaces and these interfaces define the interactions that are possible with the object. The computational model *hides* from the application designer/programmer the details of the underlying distributed platform that supports the application. The computational model is analogous to the (distributed) *programming model*. Hence we use the terms computational model and the programming model interchangeably throughout the thesis. Similarly, the terms “application component” and “computational object” is used interchangeably.

1.3.1.4 Engineering Model

The *engineering model* addresses the issues of the “distributed platform” required for the support of distributed applications. The distributed platform includes the end-systems (such as processors, storage, operating systems) that support the application components and the intervening communication support mechanisms (such as networks, communication protocols, distribution transparency support mechanisms) required to support interactions between distributed application components. The engineering model defines a set of useful distribution transparency support mechanisms such as access transparency support mechanisms, location transparency support mechanisms, migration transparency support mechanisms, group transparency support mechanisms, etc. It can be viewed as an architectural framework of an object-based distributed platform. The set of “distribution support” services and mechanisms such as *messaging service*, *binding service*, *trading service*, *location management service*, *transaction support service*, *security related services*, etc. are modeled as *engineering objects*. A collection of interacting engineering objects together provide the necessary *engineering support* for the realisation of interactions between distributed application components (computational objects). Hence the engineering model animates the computational model. The engineering model is analogous to the distributed platform. Hence we used the term engineering model and the distributed platform interchangeably.

1.3.1.5 Technology Model

The *technology model* identifies the technical artifacts (i.e., the actual hardware and software) which implement the engineering objects, computational objects, information objects, and the enterprise objects. It focuses on the choice of technology required for the implementation of the systems, such as Unix operating system, ATMswitch, Java applets, CORBA distributed platform, etc.

1.3.2 Relationship to RM-ODP Viewpoint Models

Our work on the group-based distributed computing relates to the ODP computational and engineering models. This is explained in detail in section 1.7 and section 1.8.

1.4 Review of Existing Object Group Models

The basis of group-based distributed computing model is the *object group model* and the *client-server model*. In this section we present a brief review of the existing object group model as a predecessor to the higher level group concepts which are introduced later in the thesis. The next section presents a review of the client server interaction model.

1.4.1 Object Group Terminology

The concept of “*object groups*” proposed earlier by many researchers [34 - 42] is a nice architectural solution for the design of group-based distributed computing applications. It greatly simplifies the structuring and the interaction issues associated with a group of distributed objects. The following is the basic terminology used in the context of object groups.

1.4.1.1 Object Group

An *object group* is a collection of co-located or distributed objects which can be treated as a single logical entity. It forms a natural unit for performing computational tasks. The basic group abstraction is to treat the collection of objects in an object group as if it were a singleton object.

However an object may offer multiple interfaces. An object offers its service at an interface and receives its service through an interface. Hence it is the interfaces which are the members of a group. Hence we have the following definition.

1.4.1.2 Interface Group

An *interface group* is a collection of co-located or distributed object interfaces which can be treated as a single logical entity for the purpose of invocation. It forms a natural unit of addressing and invocation. An object may be a member of multiple interface groups.

1.4.1.3 Group Member

An (object | interface) group is composed of one or more (objects | interfaces) called the *group members*. If each member of an object group offers a single interface, then the object group and interface groups are synonymous.

1.4.1.4 Member Name

Each member of the group has a unique *name* or *identifier*. Individual members of the group are addressed by their names.

1.4.1.5 Member Role

The group members may play different *roles* in the group depending upon the services they offer. These roles are application-specific, such as a ‘manager role’, an ‘administrator role’, a ‘subscriber role’, etc.

1.4.1.6 Group Identifier

The members of a group are collectively identified by a name, called the *group name* or the *group identifier*. Any invocation on the group name results in an invocation on individual members of the group.

1.4.1.7 Group Administrator

In many groups, there is usually a special group member in the role of a *group administrator* which is responsible for allowing the objects or their interfaces to join or leave the group based upon certain group membership polices.

1.4.2 Object Group Classification Schemes

The literature on group communications [34 - 55] has proposed different kinds of classification schemes for object groups. The schemes are based upon different aspects of the object group which are of interest from different perspectives.

1.4.2.1 Client and Server Groups

A primary classification used in the context of object groups is with respect to their *client* and *server* roles. Object groups can be classified as *client groups* or *server groups* depending upon whether all the members of the group are clients or servers. The members of the client group offer *client interfaces* (i.e., invoke service requests and expect replies) and the members of the server group offer *server interfaces* (i.e., accept service requests and give replies). A precise and complete definition of the groups with respect to this aspect is presented in section 2.2.

1.4.2.2 Open and Closed Groups

The object groups can also be classified as either *open groups* or *closed groups*. In open groups, a non-member of the group can make an invocation on the group whereas in closed groups, communication with the group requires the membership of the group. Our client and server groups are open because clients which are not the members of the server group can communicate with the server group and vice versa.

1.4.2.3 Active and Passive Groups

Similarly, the groups are also classified as *active groups* or *passive groups.* All members of an active group receive and process the invocation whereas in passive groups only one designated member receives and processes the invocations while all other members act as standby who checkpoint their state periodically. Our server groups are active groups. Similarly the clients groups are active groups because all members of the group can send the invocation together.

1.4.2.4 Transparent and Non-Transparent Groups

Groups may be *transparent* or *non-transparent* depending upon whether the interaction with the group is indistinguishable from the interaction with a singleton object providing the same service. Our client and server group could either be transparent or non-transparent or semi-transparent depending upon the application requirements.

1.4.2.5 Replica and Heterogeneous Groups

The groups are categorised as *replica* or *heterogeneous groups.* All members of the replica group provide the same service and their state is identical all the time. The members of the heterogeneous group are functionally different and hence offer different types of service. Our client and server groups could either be replica groups or heterogeneous groups.

1.4.2.6 Static and Dynamic Groups

Groups may be *dynamic* or *static* depending upon whether the membership of the group can change during its life time. Our client and server groups could either be static or dynamic depending upon the application requirements.

1.4.2.7 Anonymous and Explicit Groups

Yet another classification is based upon whether the group membership is regulated or not. In *anonymous groups*, any object can join or leave the group at will and can receive data sent to the group. In *explicit groups*, there is usually a special group member in the role of a *group administrator* which is responsible for allowing the objects to join or leave the group based upon certain group membership policies. Our client and server groups are explicit groups.

1.4.2.8 Source and Sink Groups

We identify another classification based upon the direction of message invocation. A group becomes a *source group* when messages are invoked from it and it becomes a *sink group* when messages are invoked on it. A client group is a source group of operation and notification messages and a sink group for termination messages (see section 1.5). Similarly, the server group is a sink group for operation and notification messages and a source group of termination messages.

1.4.3 General Applications

The object groups are the basis of multi-endpoint communication, namely the ability to invoke operations on a collection of objects without the need to know the exact membership of the collection or the location

of the members. This capability provides the basis for distributing the implementation of a service over a set of objects. Generally objects are grouped for:
1. abstracting the common characteristics of the group members and the service they provide,
2. encapsulating the internal state and hiding interactions among group members, so as to provide a uniform interface and a single addressing mechanisms to the external world,
3. using groups as building blocks to construct larger system objects.

Traditionally, object groups have been used for load sharing, fault-tolerance, performance improvement, and as a single logical addressing mechanism. In this thesis we exploit the concept of object groups together with the client-server model in a variety of non-traditional application domains.

1.5 Review of ODP Client-Server Interaction Model

The client-server model is the most basic, widely understood, and much used interaction model which needs no further introduction. However our aim is to introduce the two client-server style interaction primitives that have been described in the ODP computational model and the associated concept of message signature, which are probably not widely known. These primitives and concepts are used later in the thesis for the definition of higher-level group communication primitives.

1.5.1 ODP Computational Model Communication Primitives

The ODP computational model defines two styles of interactions between computational objects in the *client* and *server* roles. These are the *interrogation* and *announcement*. Interrogation is a *request-response* style communication between a client and a server object, and is similar to the familiar *remote procedure call* [56 - 63]. Announcement is a *request-only* communication style between a client and a server object.

1.5.1.1 Interrogation

An *interrogation* is defined in the ODP computational model as an interaction between a pair of client and server object consisting of

- the invocation of an *operation message* by the client object, resulting in the conveyance of information from that client object to a server object, requesting a function be performed by the server object, followed by
- the invocation of a *termination message* by the server object, resulting in the conveyance of information from the server object to the client object in response to the *operation message.*

1.5.1.2 Announcement

An *announcement* is defined in the ODP computational model as one way interaction between a pair of client and server object consisting of

- the invocation of a *notification message* by the client object, resulting in the conveyance of information from that client object to a server object, requesting a function be performed by that server object.

1.5.2 Operation, Notification, and Termination Message Signatures

In the ODP computational model, the client and server objects communicate by exchanging operation message, termination message, and notification message. In the remote procedure call model, an operation message corresponds to a ‘service request’ and a termination message corresponds to a ‘reply’. Hence the corresponding terms are used interchangeably throughout the thesis. Moreover, in the rest of the thesis, an operation message is also identified as "**OPR-message**", a termination message as "**REP-message**", and notification message as "**NTF-message**".

The computational model uniquely defines a message by its *signature.* A *message signature* consists of the *name* of the message followed by *parameter specification*. The parameter specification consist of the number, names and types of the parameters present in the message. Hence we have an *operation message signature*, a *termination message signature*, and a *notification message signature* in the client-server model. A *message signature* also defines the *message type*.

The computational model recognizes that the invocation of an operation message on a server object may result in distinct outcomes (replies), each of which can convey different types of results. Each outcome (reply) is identified by a unique name and carries its own set of parameters, hence each outcome has its own termination signature. A client object receives an instance of one of these termination signatures from the server object in response to the invocation of an operation message. Hence an *interrogation signature* consists of an *operation message signature* and a finite, non-empty set of *termination message signatures*, one for each possible outcome (reply) from the server object (see figure 3.1). An *announcement signature* consists of a *notification message signature*.

1.6 Scope of Group-Based Distributed Computing: Application Domains

The previous research on group communication has focussed its efforts on traditional group-based applications such as those used for replication, fault-tolerance, availability or load sharing. However an object group model is a powerful application *structuring* mechanism and client-server model is a simple, yet powerful *interaction* model. The scope and applicability of *group-based distributed computing paradigm* can be enhanced by applying the object group abstraction and the client-server model to the conventional distributed applications.

Many conventional distributed applications have multiple client and server components. Moreover these applications have a certain degree of parallelism and independence between these components. The client components of the conventional distributed applications have to deal with multiple server components on a separate and individual basis, thus sacrificing the independence and parallelism inherent in the application.

As shown in figure 1.2, we use object group abstraction as a mechanism for *structuring* distributed applications, and client-server model as a basis of *interaction* model. When used in this way, conventional distributed applications, in many domains (see examples in chapter 7) can be transformed into "*group-structured and client-server based distributed applications*". Hence, we now have a broad range of distributed applications which fall in the category of *group-structured* and *client-server based distributed applications*, and hence under the scope of our work. For simplicity, we call these applications the "*group-based distributed applications*" or in short "*group-based applications*". The essential characteristics of these applications is that they are organised as a *client group* interacting with a *server group.* Such groups could either be *replica groups* or *homogeneous groups* or *heterogeneous groups* (see section 2.2.3).

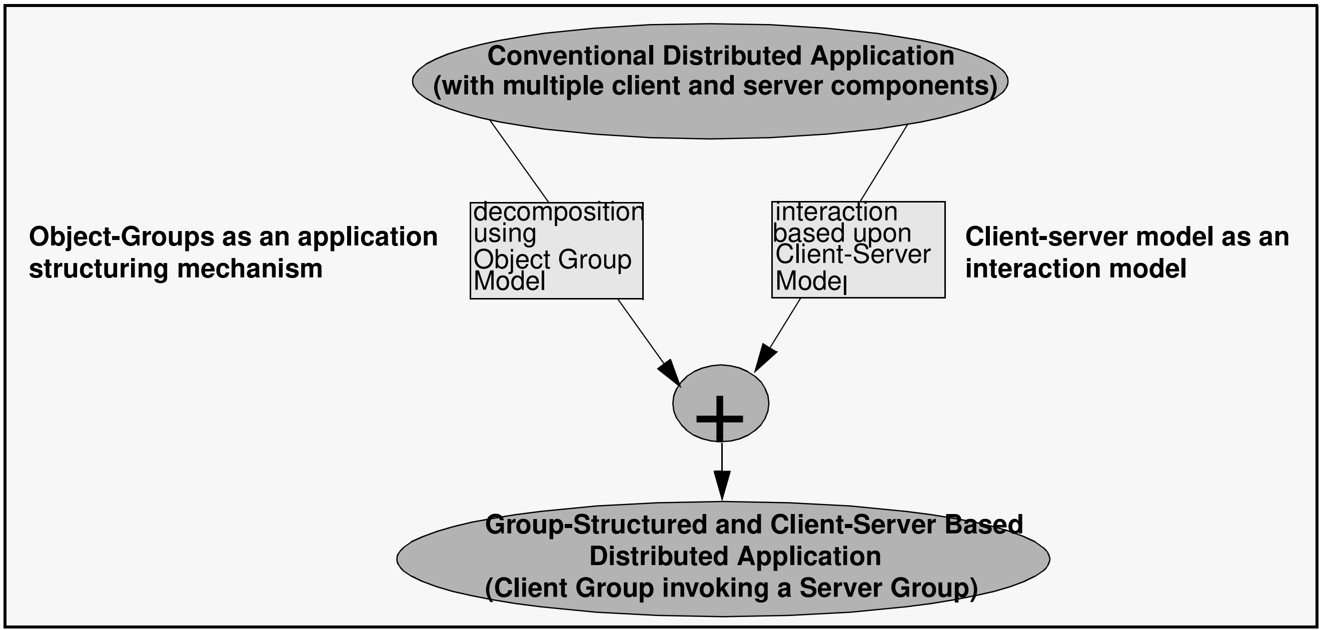

Conventional Distributed Application
(with multiple client and server components)
decomposition using Object Group Model
interaction based upon Client-Server Model
Object-Groups as an application structuring mechanism
Client-server model as an interaction model
+
Group-Structured and Client-Server Based Distributed Application
(Client Group invoking a Server Group)

Fig. 1.2 Modeling of Conventional Distributed Applications as Group-Based Distributed Applications using Object Group Model and Client-Server Model

As shown in chapter 7, there are numerous real-world applications in many domains that can be modeled as group-based distributed applications (see figure 1.3). These include applications in a wide spectrum of domains spanning telecommunications, network management, parallel computing, collaborative work groups, office automation, factory floor automation, process control, aviation, manufacturing, and in commercial domains such as stock exchange, banking, insurance, brokerage, etc.

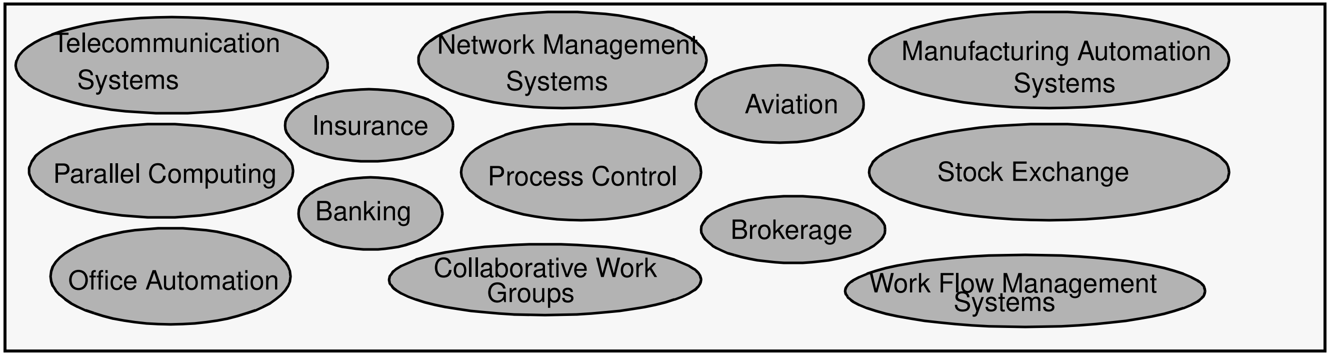

Telecommunication Systems
Network Management Systems
Manufacturing Automation Systems
Insurance
Aviation
Parallel Computing
Process Control
Stock Exchange
Banking
Brokerage
Office Automation
Collaborative Work Groups
Work Flow Management Systems

Fig. 1.3 Group-Based Distributed Computing: Application Domains

1.7 Group-Based Distributed Computing: Dual Levels of Support

The focus of the thesis is on the *distributed environment* required for the support of group-based distributed applications, as identified by the lightly shaded areas in figure 1.4. In general, the *distributed environment* required for the support of distributed applications is composed of *distributed programming* and the *distributed platform* model. These correspond to the ODP *computational* and *engineering* models respectively. The distributed platform is composed of low-level *communication support* services and the *middleware support* services. The communication support services include communication protocols which ensure the end-to-end reliable and ordered message delivery between distributed application components. The *middleware support* services provide high-level and commonly required application-specific services. The type of the middleware services varies with the nature of applications that it is required to support. The distributed programming model, amongst many other things, supports an inter-object communication facility to facilitate communication between distributed application components.

In the case of client-server based distributed applications, the middleware layer provides general useful services to support distribution-transparent interactions between client and server components. Example of such services are object-discovery services (*trading*), object-binding services, object-location services (*location-transparency support mechanisms*), mobility management services (*migration-transparency support mechanisms*), transaction support services, security services, programming-language heterogeneity support services (*access transparency support mechanisms*), etc. These services are an asset to the application developer. The question then is what are the corresponding middleware-level services required for the support of group-based distributed applications and how to configure these services in the group support platform. The thesis is devoted to these aspects.

Similarly, the existing distributed programming models for client-server applications provide a inter-object *communication primitive*, the *remote procedure call* or *interrogation*, to support application-level communication between remote client and server components. In this thesis we focus on corresponding communication primitives required to support *one-to-many* and *many-to-one* communication styles found in group-based applications and the semantics of such primitives.

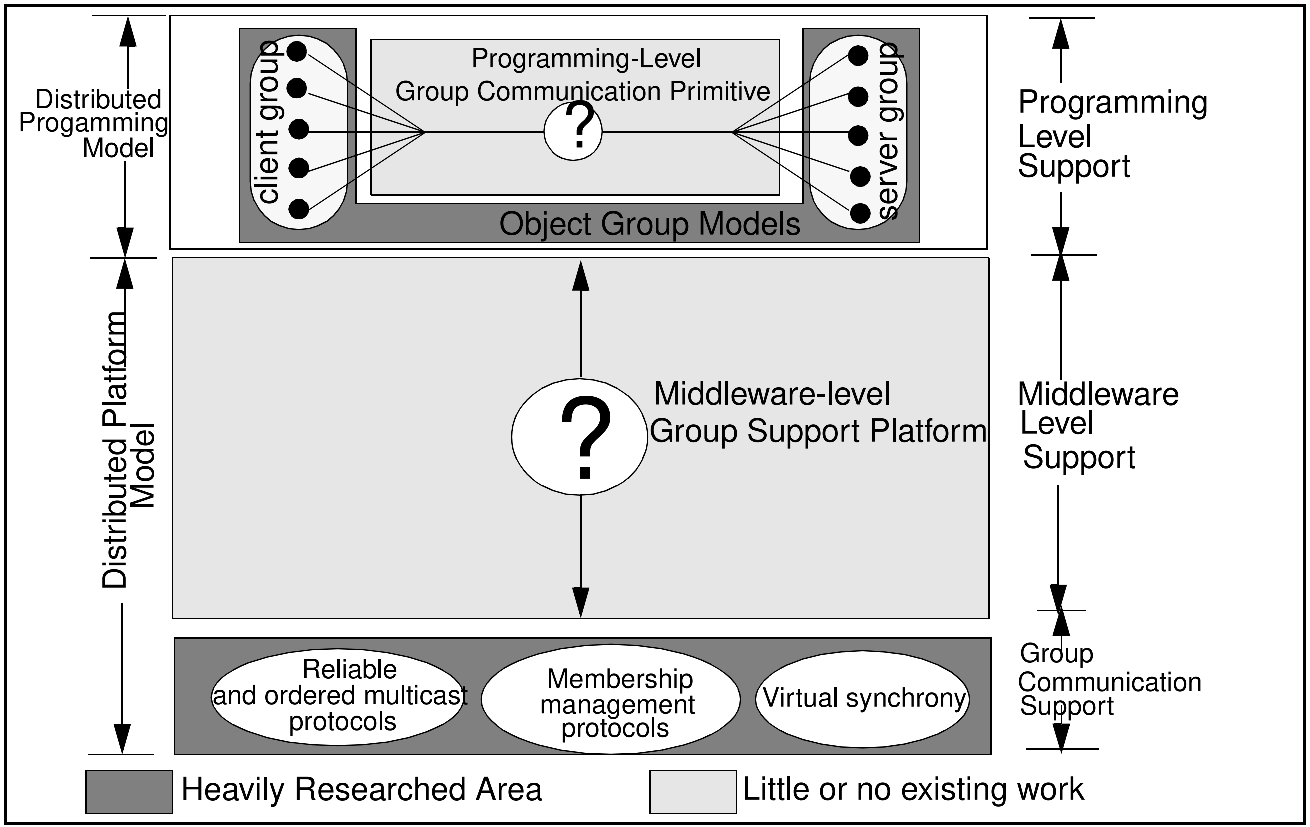

Distributed Progamming Model
client group
Programming-Level Group Communication Primitive
?
server group
Object Group Models
Programming Level Support
Distributed Platform Model
?
Middleware-level Group Support Platform
Middleware Level Support
Reliable and ordered multicast protocols
Membership management protocols
Virtual synchrony
Group Communication Support
Heavily Researched Area
Little or no existing work

Fig. 1.4 Area of Research: Lightly Shaded Areas

1.8 Scope and Aim of Thesis

The thesis is targeted at the programming-level (ODP computational model) and distributed platform level (ODP engineering model) support for *group-structured* and *client-server based* distributed applications (or *group-based distributed applications*, in short), which are discussed in section 1.6.

One of the major source of problems in group-based distributed applications is related to the new styles of interactions found in these applications. The most common form of interaction in these applications involves a client object invoking a server group and the client group invoking a server object. The former represents *one-to-many* invocation model and the latter represents *many-to-one* invocation model. This requires the support of flexible message distribution and collation schemes, based upon application's requirements. More sophisticated interaction styles involve message invocation synchronisation and message filtering schemes. The combination of these schemes results in complex coordination patterns between client group and server group. The question then arises how to support these interactions between client group and server group at the *programming-level* and at the *platform-level*. This is the subject of the thesis. The thesis investigates the issues arising at both these levels.

1.8.1 Programming-Level Support for Group-Based Distributed Computing

The first part of the thesis (chapter 2 to chapter 4) describes the programming-level (computational) support required for group-based distributed applications. This is the upper lightly shaded area in figure 1.4. At the programming-level, we describe a *communication primitive* analogous to the *interrogation* (or remote procedure call) of the basic client-server model, that explicitly addresses one-to-many and many-to-one interactions between client and server groups. We also discuss some of the sophisticated requirements of group communication support at the programming-level in order to incorporate appropriate semantics in the group communication primitive. Some of the issues that are focussed are:

1. how multiple services are requested and how they are organised,
2. how much knowledge do clients need to have about the server group in order to invoke service requests on them and be able to handle multiple replies,
3. how multiple replies from the service group are combined into a *group reply* and the order in which they are delivered to the client in case of separate reply delivery requirement,
4. how to give the client the control to receive multiple replies at the pace it wants,
5. how to give the client the control to terminate the replies when it has received sufficient number of them or when it has received whatever it was interested in receiving,
6. how should the client be informed of the end of replies in case of transparent server groups,
7. how multiple service requests from the client group are coordinated into a *group service request* and how the multiple clients are organised,
8. how much knowledge do servers need to have about the client group in order to receive and process group service requests from them,
9. how should the server respond to group service requests from the client group and the number of replies generated by it,
10. how a single reply or multiple replies generated in response to a group service request is split and distributed to appropriate clients,
11. how invocations from multiple clients in the client group are coordinated to bring about desired state change in the server applications.
12. how service requests are selectively filtered in the service group in order to satisfy specific client and server requirements for message delivery.

Possible answers to these questions have a strong impact on the degree of group transparency that is available to client and server applications. The solutions to these issues must take into account individual application requirements for message distribution, collation, synchronisation, filtering, etc. As shown in the thesis, these requirements tend to be varied and complex. A precise and unambiguous specification of these requirements can be given by a suitable policy specification language. This thesis develops such a language in order to express application requirements for group communication to the underlying group support engineering mechanisms.

1.8.2 Distributed Platform Support for Group-Based Distributed Computing

The second part of the thesis (chapter 4 through chapter 9) addresses the issues of distributed platform (engineering) support required by group-based applications. This is the lower lightly shaded area in figure 1.4. Most of the existing distributed platforms, such as Corba and DCE do not provide adequate support for this new class of applications. This has forced application developers and programmers to deal with low-level issues related to group communication that could be better provided as uniform and standard mechanisms by the underlying distributed platform. The aim of the thesis is to address issues of group communication, such as message distribution, collation, synchronisation, etc., that arise at the application-level, but are common to a wide range of applications and to put these issues in the distributed platform. The provision of such a support at the platform level will not only enrich the existing distributed platforms such as CORBA, DCE, etc., but will also substantially simplify the design and construction of group-based applications. The application designer can now focus on the application aspects leaving the *group communication and coordination* aspects to the underlying distributed platform. The group coordination aspects can be separately specified thus separating the application logic from group coordination logic. This also enables the group coordination requirements to be changed without recompiling the application. With this objective, the focus of the thesis is on the following aspects of the distributed platform. We call such a platform the *Group Support Platform* (GSP).

1. what group support services are most commonly required by the applications which could be provided as separate services in the distributed platform, and what functionalities are required in the corresponding group support engineering objects,
2. what is the relationship between these group support services and what are the possible interactions between the corresponding group support engineering objects,
3. how should these objects be organised or configured in the distributed platform,
4. how do the group support objects in the remote machines communicate with each other and what is the protocol between them,
5. what group coordination patterns exist in group-based applications and how can they supported by the combination of group support engineering objects in the distributed platform,
6. what requirements do applications place on individual group support objects, and how to specify these requirements,
7. what information do group support objects need to know from the applications in order to perform their tasks, and how to specify this information,
8. what is the notation for the precise representation of group communication requirements and what it is the syntax and semantics of the corresponding group policy specification language,
9. what interface does the group support platform offer to the applications, and what interactions take place at this interface,
10. how can group transparency be realised through an autonomous, albeit requirements-driven middleware layer between application components and the low-level group communication protocols.

Our solution to these questions defines an agent-based, configurable, extensible, and policy-driven distributed platform for the support of group-based applications. In this model, the group support objects manage the group communication and coordination patterns on behalf of the user applications, which influence the behavior of the objects by means of policy specifications. We present a software architecture or a framework for the organisation of group support services in the distributed platform. This framework serves as a basic unit of the group support platform within which new group support services may be identified and their interaction with the existing ones defined. This framework gives architectural elegance and simplicity in the design of the group support platform. The group support platform supports diverse application requirements and offers selective group transparency by allowing applications to specify group communication requirements through group support policies.

1.9 Related Work and Differences

As mentioned in section 1.1, the area of *group communication* has been a subject of extensive research in the past, whereas *group-based distributed computing* has received much less attention. Our research is on higher-level support of group-based distributed computing which use group communication at the lowest level. This is the major difference.

Some attempts have been made in the past at the *programming-level* and the *platform-level* support for group-based applications, as cited below. However such attempts were focussed on a limited set of applications. In particular, as discussed below, such attempts have either been incomplete or partial with respect to considering the overall requirements of group-based applications at the programming and the distributed platform level. In fact, the very concept of group-structured and client-server based distributed computing that we present in this thesis and which is basis of our work is lacking in the previous work. We identify the requirements of group-based applications within a broader framework which recognizes the group-structured nature of these applications as well client-server nature of interactions in them. We discuss some notable efforts that have been made earlier and their shortcomings and explain the differences of our work from the previous ones.

1.9.1 Programming Level

At the programming-level, previous work such as [64 - 79] has identified the need for one-to-many communication between client object and server group. Since much of the early work aimed at providing server groups for fault-tolerance and for replication management, server groups were assumed as replicated groups in most of the cases and hence the proposed solutions are based upon simplified assumptions about the nature of the server groups and kind of replies expected from them in response to a service request. Most of the authors have chosen to pick the first or the first ‘n’ matching replies and discard the rest. Our work takes into consideration not only replica server groups, but also the homogeneous and heterogeneous server groups [80]. This allows us to take into account general requirements of a one-to-many group communication primitive.

An important consideration in case of homogeneous server groups is how to handle (or collate) multiple instances of replies corresponding to the same termination signature. These replies may not necessarily be identical (in their parameter values) - an assumption made in the previous work. The previous work has dealt with multiple replies using content-based collation schemes such as giving the average or the maximum or the minimum of the replies. This scheme is not general enough. We propose a *signature based collation scheme* which is general enough and which allows the client to gain access to all the replies, through a single reply message, and to process them in application-specific manner. Content-based collation can also be supported in our model because our reply collation mechanisms are generic and are driven by application-specific collation policies.

A major difference between ours and previous work is the lack of a general policy-driven collation framework in all previous attempts. We propose a general collation framework which takes into account not only the collation mechanism but also the collation time, collation cardinality, and the identities of the source group members whose message the sink object is willing to accept, as well as the preferred order of reply delivery.

An important distinction between ours and previous work is the *manner* and the *format* in which the replies are returned to the client and the *control* the client has on the receipt of the replies. These are important aspects of a group communication primitive.

Our work considers a general reply format such as the one proposed for existing client-server applications in the ODP computational model - the *interrogation* primitive, in which each reply that is understood by the client is identified by a name and carries its own set of reply parameters. This, coupled with the general collation framework and the reply delivery control, results in a powerful group communication primitive.

In many group-based applications, clients need to handle not only multiple replies, but also different types of replies individually and separately and in a controlled manner. Server group transparency is impossible and in many cases undesirable for clients in such applications.

The previous work completely neglects the interaction requirements of a heterogeneous server group. For example what to do when different types of replies are returned from the server group. How to combine these different types of termination signatures. The *signature-based* collation schemes proposed in the thesis present an elegant solution to these requirements. Moreover to deal with transparent server groups, the client needs to know the end of replies. This aspect is missing in previous work. Another aspect of a group communication primitive is the order of delivery of replies to the client. As shown in later chapters, many clients not only need to process multiple replies, but have a certain preference with respect to the order of delivery of replies. This aspect is also missing in the previous work. Our collation framework gives the client application the ability to specify the reply delivery ordering requirements.

Our work goes beyond in identifying more sophisticated requirements of group-based applications, such as *solicited* versus *unsolicited* reply delivery, *terminable* reply delivery, etc. which gives the client desired-level of control on communication with different types of server groups.

The existing work has been mostly one-sided. It has only considered one-to-many aspects of group communication. The other important group invocation paradigm which deals with many-to-one communication is missing. This involves a client group invoking a server object. This paradigm is required by many applications as shown in chapter 7. The notion of client group exists but the notion of a combined invocation from a client group does not exist. Similarly, the notion of combined reply from a server object in response to group service request from a client group does not exist in the existing work.

Some client groups lead to a *group service request* invocation semantics, wherein each member of the client group periodically makes a service request of the same type, not necessarily identical, on the server object. Moreover the server object needs to receive all the service requests from the client group together before it gives the replies. This reply is based upon the client group input. Such class of applications are totally excluded from the existing work. Our work gives a precise treatment to this type of group invocation paradigm by taking into account different aspects of many-to-one communication such as periodic nature of service requests invocations from the client group, group service request construction, multiple reply generation by the server, reply splitting and distribution, etc.

The level of transparency is an important issue in a programming-level communication primitive.

The issue is to what extent should the semantics of group communication primitive be configurable by the programmer. We address this issue by giving the programmer the ability to specify different message distribution and collation policies within the underlying group support platform.

Our work goes beyond the previous work in group programming model in that it provides a very general and logical enhancement of the existing remote procedure call or interrogation paradigm which meets the needs of one-to-many, and many-to-one group communication. Our group communication primitives represent a synergy of client-server interaction model and the group communication model.

1.9.2 Distributed Platform Level

At the distributed platform level there exists a big void in existing research, which has mostly been done at the lowest level. There exists low level support for group communication, in terms of different types of ordered multicast protocols [81 - 94], membership management protocols[95 - 99], virtual synchrony [100 - 103], etc. These low-level group communication protocols have been used in distributed systems for group support, such as ISIS [104 - 106], Horus [107 - 108], Electra [109 - 110], Amoeba, [111 - 112], Transis [113 - 114], Rampart [115 - 116], Totem [117], Relacs [118], V Kernel [119], Consul [120], Delta-4 [121], and others [122 - 123]. In all these systems, the applications are directly tied to the low-level group communication layer, without the support of a *middleware layer* to separate applications from the low-level group communication issues. In particular, there is no flexibility in specifying different types of group communications support required by the applications. This results in very low-level reasoning about the group communication issues still being part of the application.

As shown in chapter 7, group-based applications exhibit a wide variety of group coordination patterns. These group coordination and cooperation aspects should be specified external to the applications in order to separate application logic from group coordination logic as well as to be able to dynamically modify the latter without affecting the former. These pertain to the issues of the middleware layer which can be supported at the lowest level by above mentioned group communication protocols.

As discussed in section 1.7, the middleware-level frees the application designer from worrying about the issues of group communication at the application level. The middleware-level needs to provide high-level group support services which can be tailored to application requirements. Hence the programmability of middleware components is an important requirement. The middleware layer should also offer a uniform interface to the applications in accessing its services.

In case of group-based distributed applications, the middleware layer is almost non-existent. Reference to some of the middleware level services in [34 - 35] is promising, but there exist no architecture or framework for the organisation of those services. Our work [124 - 126] fills a big gap at the middleware-level.

We identify a set of group support services which are required by a number of group-based applications in different domains. Then we present a software architecture or a framework for the organisation of these services in the group support platform. We identify the relationship between these services and the interactions that take place between the corresponding group support objects in order to support applications. Finally, we present the protocol for communication between peer group support objects in distributed nodes. This completes the design of the group support platform.

Programmability of the middleware services is an important requirement. Our work goes further in identifying the various issues of group support services and putting these issues in a language framework, resulting in a group policy specification language. This language can be used by the applications to specify their different group support requirements, such as message distribution, collation, etc.

1.10 Structure of Thesis

This thesis is structured in two parts. Part-1 deals with *distributed programming support* and part-2 deals with *distributed platform support* for group-based distributed computing applications. Part-1 of the thesis contains three chapters, chapter 2 through chapter 4, and part-2 contains seven chapters, chapter 4 through chapter 9. The last chapter, chapter 11, contains some concluding remarks and directions for future work. The following is a brief description of the contents of each chapter.

Chapter-1 is an introduction to the problem domain. It contains the scope and the aims of the thesis and a brief description of the issues of group-based distributed computing which are the focus of the rest of the thesis. This chapter also compares our work with the previous work.

Chapter-2 is the first chapter of the part-1 of the thesis. It identifies the requirements of the programming-level group communication between application components. It also describes the limitations of the existing programming-level communication primitives such as remote procedure call or interrogation.

Chapter-3 introduces the programming-level communication primitives which provide semantic support for multi-endpoint interaction between a client-group and a server-group. These primitives are called *group interrogation* and *group announcement*. Message collation is the basis for the construction of group communication primitives. Some generic signature-based message collation schemes are proposed in this chapter. The impact of these primitives on group transparency is described. This chapter also describes the impact of group interrogation primitive on the message invocation, reception, and processing requirements of the client and server objects. Such clients and servers are called group-oriented clients and servers. The communication between these objects and the local group support proxy object to which they are bound is also explained.

Chapter-4 is the first chapter of the part-2 of the thesis. It identifies some of the basic middleware-level *group support services*, such as message distribution, collation, synchronisation, filtering, etc., that are required in the distributed platform for the support of group-based distributed computing applications. The different aspects and issues involved in the provision of these service are also identified.

Chapter-5 describes how the set of group support services, introduced in the previous chapter, can be configured together inside an architectural framework called the *group support machine* and how the components of this machine work together in the provision of middleware-level service to the applications. Each member of the group-based distributed application is supported by a group support machine. The set of group support machines communicating with each other through an inter-machine protocol constitutes a *group support platform*.

Chapter-6 describes in detail the internal components of the group support machine, the functionality of these components, the interfaces between these components, and the interactions that occur at these interfaces. The internal structure and the behaviour of the group support machine is described in an abstract and implementation independent manner. The group support machine offers standardised interfaces both to the application components and to the underlying group communication layer. These interfaces are described in detail.

Chapter-7 describes *group coordination models* implicit in group-based distributed applications. The group coordination model is characterised by the *structure* of the application which is a configuration of a client-group and server-group and the *interactions* that occur between the members of these groups. The coordination behaviors inherent in these models can be specified at a high-level using a group policy specification language. This language is introduced informally in this chapter through examples. This chapter contains the examples of various group-based applications in different enterprise domains spanning telecommunications, network management, parallel computing, etc.

Chapter-8 is an introduction to the syntax and the semantics of the *group policy specification language* which has been presented informally through examples in the previous chapter. The language permits the specification of message distribution, collation, synchronisation, and filtering requirements of an application, at a high-level independent of the mechanisms or protocols needed to implement them. These policy specifications are associated with individual message types and are stored as *policy scripts* in the group support machine.

Chapter-9 is the last chapter of the part-2. The definition of group support platform is incomplete without describing the communication between the basic component of the platform - the group support machine. This chapter describes the remote communication protocol between the peer group support agents located in different group support machines - the information that is exchanged between the peer group support agents, the format in which this information is exchanged, and the handshake involved between the group support agents.

Chapter-10 describes the implementation and performance aspects of the proposed Group Support Platform (GSP) and the group interrogation primitive. A partial model of GSP is implemented in Java. This chapter compares the performance of GSP with that of a conventional middleware platform, such as Corba.

Chapter-11 highlights the contributions of the thesis and gives pointers to future work directions.

We suggest that the reader start with chapter-1 in order to find out the scope and the aim of the work and to gain an insight into the general research area. This allows the reader to put our work in perspective. Chapter 7 contains numerous examples of group-based applications. The reader new to the subject may glance through the examples in this chapter before starting the rest of the chapters. The remaining chapters of the thesis are generally organised in the order in which it is suggested that they be read.

PART-1

Distributed Programming Model: A Programming-level Group Communication Primitive

CHAPTER 2 Requirements of Programming-Level Group Communication Primitive

Abstract

Group-based distributed computing is becoming an increasingly important computing paradigm of modern distributed systems, but programming-level support for group communication is hitherto missing. Remote procedure call is a familiar programming-level abstraction to support "request-response style" communication in the point-to-point client server computing model. This chapter investigates the requirements of the corresponding abstraction for the support of multi-point communication in group-based distributed applications.

2.1 Introduction

Experience with *remote procedure call* has demonstrated the importance of *programming-level* support for point-to-point communication in client-server based distributed systems. It greatly simplifies the task of point-to-point communication between singleton client and singleton server at the programming level. The programmer is given a powerful handle to invoke a remote object, as if it were a local one, and to receive the reply.

In the case of group-based applications, this high-level support is missing. Interestingly enough, group communication is supported by many kinds of local area networks, such as ethernet, token ring, etc. and radio broadcast systems. The lowest level communication medium often supports the group communication that the applications need, it is the operating systems and the programming languages that do not provide support for group communication at the application-level.

There exists low level support for group communication, in terms of different types of ordered multicast protocols [81 - 94], membership management protocols [95 - 99], virtual synchrony [100 - 103] in many distributed system platforms such as ISIS [104 - 106], Horus [107 - 108], Electra [109 - 110], Amoeba [111 - 112], Transis [113 - 114], Rampart [115 - 116], Totem [117], etc. However, no facility is available to the application programmer to access and exploit group communication at the application level. The middleware support and the programming-level support for group-based computing is missing in these platforms. The middleware support is discussed in chapter 4 to chapter 9 and the programming-level support is discussed in chapter 2 to chapter 4 of the thesis.

This chapter investigates the "programming-level" communication requirements of group-based distributed applications. We also evaluate the capabilities of the currently available point-to-point "programming-level communication primitives" - the *remote procedure call* and *interrogation*, against the requirements of group-based distributed applications.

2.2 Client Group and Server Group: Definition & Properties

Before we start examining the requirements of group-based distributed applications, we investigate the basic properties of the client group and server group which are the basis of these applications.

2.2.1 Client and Server Interfaces

As a precursor to the definition of the client and server groups, we start with the definition of client and server interfaces. The *client* and *server* are the primary roles in distributed computing.

2.2.1.1 Client Interface

A *client interface* is an object interface which is characterised by the following properties:

1. the object invokes *operation messages* through this interface and expects to receive a *termination message* at this interface in response to the operation message.
2. the object invokes *notification messages* through this interface.

2.2.1.2 Server Interface

A *server interface* is an object interface which is characterised by the following properties:

1. the object expects to receive an *operation message* at this interface and invokes a *termination message* through this interface in response to an operation message.
2. the object expects to receive a *notification message* at this interface.

An object offers multiple interfaces. These interfaces could be client interfaces or server interfaces. The interfaces offered by an object could be members of different interface groups.

2.2.2 Client and Server Groups

The basic definition of an object group and an interface group are given in section 1.4. We make use of these concepts and of the ones introduced above in the definition of server groups and client group.

2.2.2.1 Server Group

A *server group* is an interface group in which all member interfaces are *server interfaces*. These interfaces could be of the same or different types.

2.2.2.2 Client Group

A *client group* is an interface group in which all member interfaces are *client interfaces*. These interfaces could be of the same or different types.

2.2.2.3 How are Client Groups Formed

The nature and the creation of client groups is not obvious. It deserves special mention. It is important to note that a client group is formed with respect to a given server interface. There are two main cases of the creation of client groups. The existing literature on object group models [34 - 42] recognizes only one case of the formation of a client group, the first one listed below. We identify the second important case of the creation of client groups. The proposed group communication primitive supports both these cases.

1. *From replicated server groups*: Client groups are formed when the members of an actively replicated server group need to invoke another server object in order to perform their service. This happens in cases when a server object containing reference to another server object is replicated thereby forming a replicated server group. Any invocation on the replicated server group which causes an invocation on the referenced server object by one member will generate invocations by all other members. In this

kind of application we need only give one operation invocation to the referenced server object and discard the other identical invocations. The replies must however be sent to all the members of the replica group.

2. *Object groups under the service provision of a singleton server object*: Client groups are also formed when the client components of a distributed application, which need not be identical or replicated, but are *related* to each other in some application-specific manner and which require the same *type of service*, are organised as an object group and are placed under the service provision of the same server interface. A client group is always formed with respect to a given server interface. An important characteristic of these applications is that each member of the client group invokes
   a. instances of the same service request, each of which is an instance of the same operation signature, not necessarily identical (in their parameter values), such as shown in figure 2.1,

   or

   b. partial service requests, each of which is an instance of the different part of the same operation signature at the server side, such as shown in figure 2.2,

   periodically (or at fixed time intervals) on a given server interface and the server's reply to each group member is based upon the total service request which is obtained by combining the individual service requests from the client group. The server may give the same reply or different replies to each member of the client group. Examples of these type of client groups are given in section 7.7. In this kind of applications we need to give all operation invocations from the client group to the server object because the reply of the server is based upon the total group input.

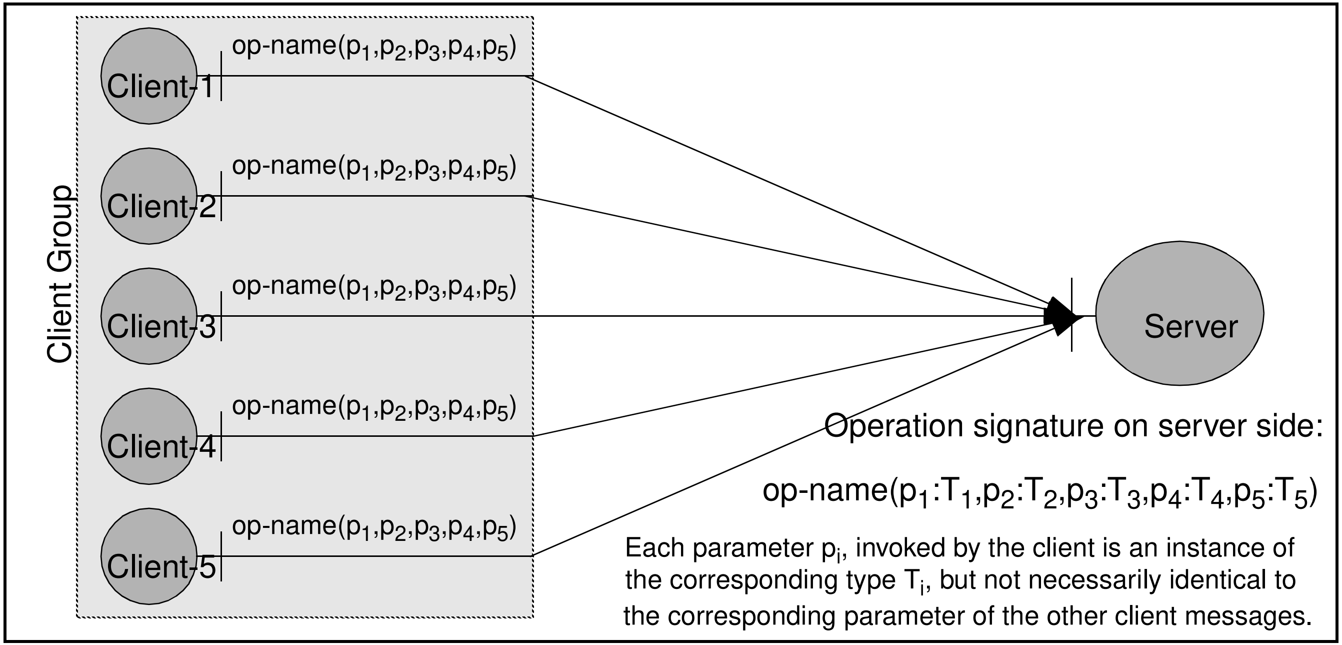

Client Group
Client-1
Client-2
Client-3
Client-4
Client-5
op-name(p1,p2,p3,p4,p5)
op-name(p1,p2,p3,p4,p5)
op-name(p1,p2,p3,p4,p5)
op-name(p1,p2,p3,p4,p5)
op-name(p1,p2,p3,p4,p5)
Server
Operation signature on server side:
op-name(p1:T1,p2:T2,p3:T3,p4:T4,p5:T5)
Each parameter pi, invoked by the client is an instance of the corresponding type Ti, but not necessarily identical to the corresponding parameter of the other client messages.

Fig. 2.1 Homogeneous Client Group: Each member invokes instances of the same operation signature

2.2.2.4 Client Group Invocation Properties

Based upon the nature of the client groups, we identify the following properties of the client groups. The message invocations from the client group exhibit the following unique properties.:

1. *Nature of service invocations*: The members of the client group invoke identical operation messages or instances of the same operation message signature or the instances of the different parts of the same operation message signature, on a given server interface.

2. *Timing of service invocations*: The message invocations from the client group occur during a well-defined time interval. The operation (or notification) messages are invoked by the group members during a fixed time interval or periodically.
3. *Reply to service invocations*: The server's reply (or replies) is based upon the group operation message which is obtained by combining the individual instances of the operation message from the group members. Therefore, a client's future state is dependent upon the current state of *all* the group members.

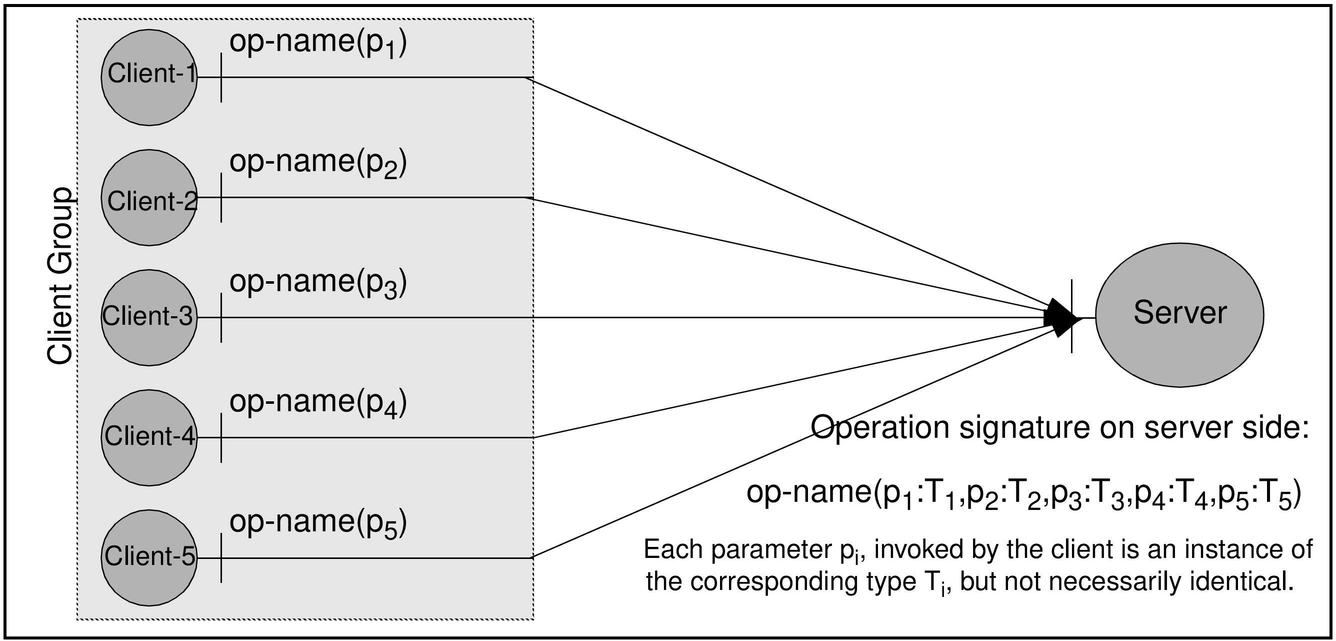

Client Group
Client-1
op-name(p_1)
Client-2
op-name(p_2)
Client-3
op-name(p_3)
Client-4
op-name(p_4)
Client-5
op-name(p_5)
Server
Operation signature on server side:
op-name(p_1:T_1,p_2:T_2,p_3:T_3,p_4:T_4,p_5:T_5)
Each parameter p_i, invoked by the client is an instance of the corresponding type T_i, but not necessarily identical.

Fig. 2.2 Heterogeneous Client Group: Each member invokes an instance of different operation signature

2.2.3 Categories of Client and Server Groups

We categorise the client and server groups as *replica groups* or *homogeneous groups* or *heterogeneous groups* based upon the type of messages (i.e., operation, notification, and termination message signatures) offered or invoked by the members of the groups and the state of the group members. The members of the replica group are identical or replicas of each other at all times. The members of the homogeneous group offer or invoke the same *service type* (i.e., operation or notification message type) but their state need not be identical. The members of the heterogeneous group offer or invoke different service types.

2.2.3.1 Replica Client Group

The members of the replica client group invoke identical operation or notification messages, usually simultaneously or within a fixed time interval. Examples of these type of groups abound in existing literature.

2.2.3.2 Homogeneous Client Group

The members of the homogeneous client group invoke instances of the same operation or notification message signature, usually at regular periodic time intervals. These instances need not be identical. An example of this type of group is given in section 7.7.1.

2.2.3.3 Heterogeneous Client Group

The members of a heterogeneous client group invoke instances of different parts of the operation or notification message signature supported at the server interface, usually at regular periodic time intervals. Heterogeneous client groups are formed when each member of the group gives partial inputs (i.e., different parts of the same operation message), but is interested in receiving a total reply from the server object. These type of groups are termed heterogeneous, because each member of the group invokes an instance of different operation or notification signature. An example of this type of group is given in section 7.7.4.

Table 2.1: Categories of Client Group

	Nature of Service Invocation	Timing of Service Invocation	Replies to Service Invocation
Replica Client Group	identical operation messages	invoked at the same time or within a fixed time interval	server's reply based upon a single client input, other inputs can be discarded
Homogeneous Client Group	non-identical, but instances of the same operation message signature	invoked periodically	server's reply based upon total client group input
Heterogeneous Client Group	instances of the different parts of the same operation message signature	invoked periodically	server's reply based upon total group input

2.2.3.4 Replica Server Group

The members of the replica server group respond with identical termination messages to an operation message from the client object. Examples of these type of groups abound in existing literature.

Table 2.2: Categories of Server Group

	Nature of Reply Invocation	Reply Delivery to Client Object	Server Group Transparency to Client Object
Replica Server Group	identical replies	single reply	possible
Homogeneous Server Group	same type of replies, not necessarily identical	single group reply (replies may also be delivered separately)	impossible
Heterogeneous Server Group	multiple types of replies	multiple group replies, one for each reply type (replies may also be delivered separately)	impossible

2.2.3.5 Homogeneous Server Group

The members of the homogeneous server group respond with instances of the same types of termination message to an operation message from the client object. These instances need not be identical. An example of this type of group is given in section 7.6.2.

2.2.3.6 Heterogeneous Server Group

The members of the heterogeneous server group respond with same or different types of termination messages to an operation message from the client object. An example of this type of group is given in section 7.6.6. The properties of these groups are summarized in table 2.1 and table 2.2 .

2.3 Programming-Level Communication Requirements of Group-Based Applications

Group-based applications are composed of a *client group* interacting with a *server group*. They have a unique set of communication requirements which arise due to this multiple-client and multiple-server characteristics of the application. In these applications, each member of a client group invokes a server group and/or each member of a server group receives an invocation from the client group.

We investigate the 'programming-level' (computational-level) inter-object communication requirements of group-based applications. Our focus is not restricted to the traditional group-based applications, which come in the category of *replicated groups*, such as those used for fault-tolerance or load sharing, where the composition of the group is, essentially, a set of *replicated objects*. We also analyze the programming-level communication requirements of a more general category of group-based distributed applications which fall under the category of *homogeneous* and *heterogeneous groups*. We present these requirements by analyzing the following basic interaction paradigms:

1. 'Singleton-Client' and 'Server-Group' interaction paradigm
2. 'Client-Group' and 'Singleton-Server' interaction paradigm

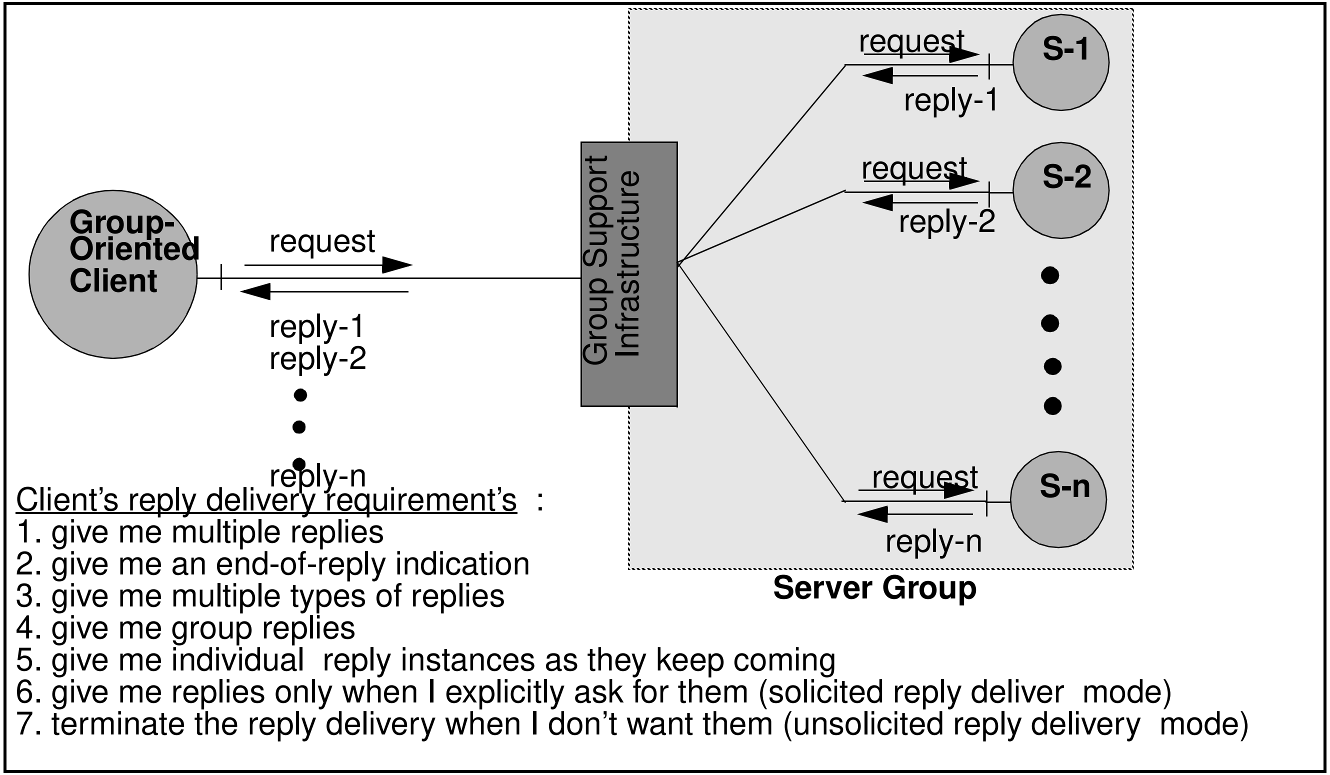
2.3.1 'Singleton-client' and 'Server-group' interaction requirements

In this paradigm, a singleton client *interrogates* (or invokes) a server group (see figure 2.3). The following properties are required from the programming-level group communication primitive. These properties also have some implications on the capabilities of *group-oriented clients* (see chapter 4).

1. *Multiple replies*: When a client interrogates a server group, it receives multiple replies, one from each member of the server group, in response to its operation invocation. *Group-oriented clients* (see definition in section 3.11) need an *invocation primitive* which can handle multiple replies from the server group.
2. *Variable number of replies*: For some clients, the membership of the server group is transparent. So the client does not know how many replies to expect. Moreover, the number of received replies is variable also because the membership of the server group may change dynamically due to member failures and new members joining the group. Similarly, the number of received replies is variable due to communication failures. This raises the question of how many replies shall a client application expect in response to its operation invocation and consequently how long shall the client wait for the replies. The group invocation primitive needs a special termination to convey "end-of-replies" to the client application. This termination would be locally generated by the underlying group support platform which is aware of the group membership.
3. *Multiple reply types*: Clients often need to invoke a *heterogeneous server group*, and be able to collect not only multiple replies, but also different types of replies. The replies received from a heterogeneous server group, in response to an operation invocation, have different *termination signatures*. A termination signature corresponds to a reply type and also denotes the context in which instances of the corresponding reply type are to be processed together. The group-oriented clients should be capa-

ble of receiving and processing multiple reply types.

4. *Requirement to combining multiple instances of a reply type*: In many group-based applications a client receives multiple instances of replies corresponding to a *termination signature* (reply type) from the server group, in response to its operation invocation. A termination type corresponds to an application context in which the corresponding reply instances are to be processed. Often a client has a requirement to process all instances of a given reply type together as a single unit, but it is inefficient for the client application to be interrupted to collect every individual reply instance and process it separately. Yet in some other cases it is impossible for the client to take any application-specific decision until all instances corresponding to a given reply type are received. And if the server group is transparent to the client, as in most cases, the client does not know how many replies to expect, and consequently when to start analyzing the results. It is desirable that the group invocation primitive support a "*group termination*" facility, so that all instances of replies corresponding to a given termination signature (reply type) can be *combined* together and handed over to the client as a single unit by the underlying engineering mechanisms. This in turn requires that the group-oriented clients be capable of processing multiple instances of replies contained in a '*group termination*'.



Client's reply delivery requirement's :
1. give me multiple replies
2. give me an end-of-reply indication
3. give me multiple types of replies
4. give me group replies
5. give me individual reply instances as they keep coming
6. give me replies only when I explicitly ask for them (solicited reply deliver mode)
7. terminate the reply delivery when I don't want them (unsolicited reply delivery mode)

Fig. 2.3 Client object interrogates a server group

5. *Unsolicited and separate delivery of individual reply instances*: In some group-based applications, the clients are on-line and they process the replies as soon as they are delivered. In such applications, the clients want to receive the individual instances of replies, as they keep coming in. They do not want to wait until all of them have been received and combined into a single group reply. The unsolicited reply delivery means "*give me the reply as soon as it has arrived*".
6. *Solicited reply delivery*: In some cases, the client applications wish to have the control to receive the replies as and when they are required by the client. This prevents the clients from being overwhelmed

with huge number of replies, and also gives the client the ability to receive the next reply only when required by it. In such applications, the clients also specify a certain order of the reply delivery, based either upon the type of the reply or the sender of the reply. The controlled reply delivery coupled with ordered reply delivery gives the client the capability to process the desired replies first and to ignore the rest. This corresponds to the *solicited reply delivery*, i.e., "*give me the reply only when I want it*".

7. *Terminable reply delivery*: In certain client applications, each reply is processed at the moment of its reception, without waiting for the receipt of all the replies (see bullet 5). In such cases, the clients wish to abandon or terminate the group interrogation as soon as the replies already collected by it are sufficient for it to proceed. Hence the clients need the control or handle to stop the subsequent flow of incoming replies. This allows the client to dynamically control the number of replies, subsequent to the operation invocation. Terminable reply delivery is most commonly required in combination with unsolicited reply delivery.
8. *Non-blocking invocation*: When a client makes an operation invocation on a server group, multiple replies are expected. There is, also, a varying amount of delay involved in the reception of replies. In many applications, the client does not want to be blocked until the receipt of all the replies. Moreover a client thread may need to make multiple operation invocations on a server group, without waiting for the replies of the previous invocations. Group-based client applications require the ability to perform other processing while the replies are in transit.

2.3.2 'Singleton-Server' and 'Client-Group' interaction requirements

In this paradigm, a client group *interrogates* (or invokes) a server object (see figure 2.4). Typically a client group is formed when a set of client objects, related to each other in some application-specific manner, organise themselves as a group in order to be (managed or supervised or otherwise) serviced by a single (manager or supervisor or) server object. The reader is referred to the examples of this type of interaction paradigm in section 7.7. This type of interaction paradigm has the following characteristics, which have implications on the capabilities of *group-oriented servers* (see chapter 4).

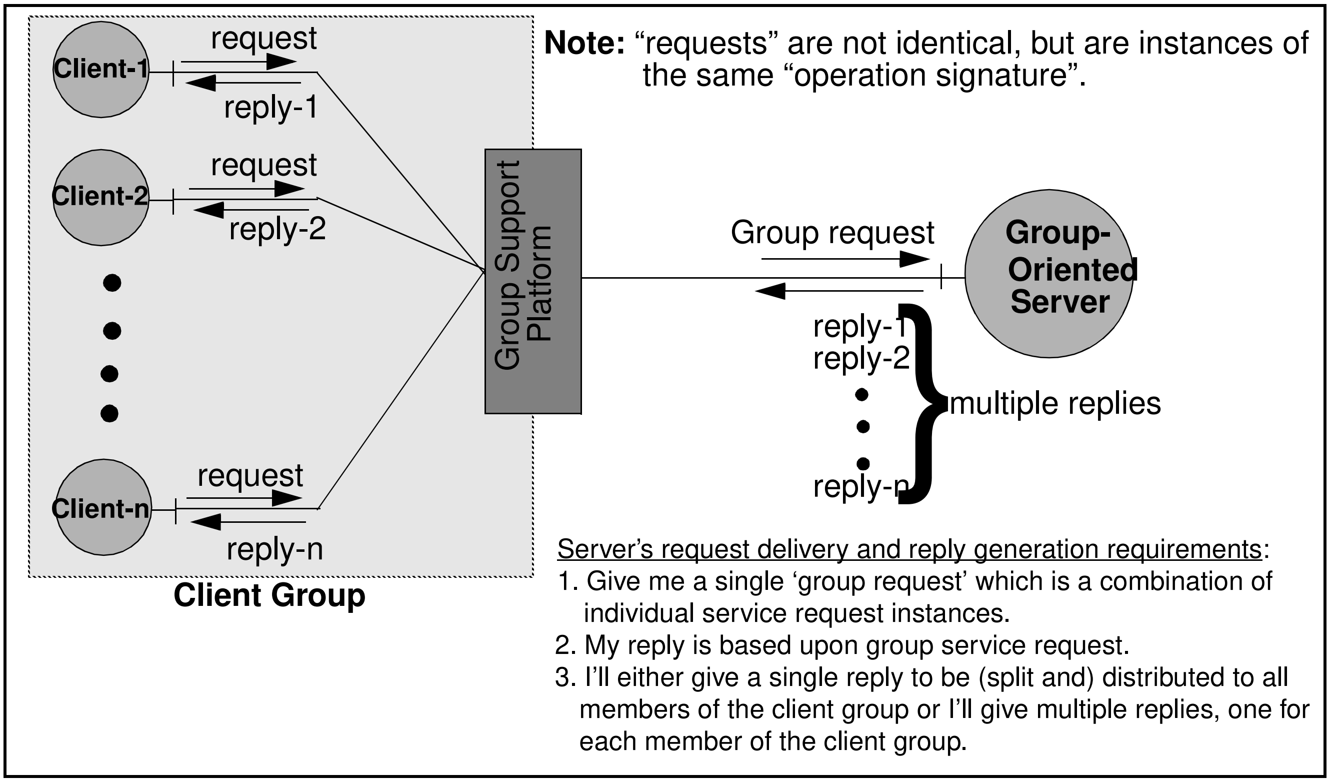
1. *Multiple instances of same service request*: The members of the client group have identical service requirements and hence they invoke either identical service requests or instances of the same service request (i.e., instances of the same *operation signature,* but with non-identical parameter values) on the server object.
2. *Periodic service requests* (or *notifications*): Operation or notification message invocations from the client group often occur periodically or within a specific time interval. For example, a group of managed objects (client group) send their status reports along with the associated status parameters, in the form of an operation message, to the manager object (server) periodically, and expect to receive the management command, in the form of a termination message, from the manager object.
3. *Reply based upon client group-input- need to combine multiple instances of a request type*: In this type of applications, the members of the client group are related in an application-specific manner. Instances of the same service request are invoked periodically by the members of the client group on the server object. These individual instances need to be analysed and processed together as a single unit in order to generate the reply. Essentially, the final output or the decision of the server object depends upon the combined input from the client group. The server cannot start processing the service requests (operation messages) until it has received the service requests from all the client group members. After processing the group input, the server may either give the same reply or different reply to each member of the client group. In either case the reply is dependent upon the cumulative group input. Hence the server wants to receive a combined set of instances of a service request (cor-

responding to an *operation signature*) as a *single group operation message* so that it can efficiently perform the processing of the combined group input. Additionally, the server cannot handle (or receive) individual inputs from the client group because of:

(a). *client group transparency*: In many cases, the client group is transparent to the server object. A question that arises in these cases is how long shall the server keep waiting expecting inputs from the client group before it can start processing. Moreover, the server does not know about the dynamic situations occurring in the client group such as member failures or communication failures.

(b). *increased load on servers*: Even if the client group is not transparent, it is inefficient for the server object to be continually interrupted to receive every individual input (OPR or NTF message) from the client group, keep accumulating them and keeping record of the number of inputs that are received. This would waste the server's time and resources in collecting and combining the inputs and it would require the server object to deal with the issues of a distributed client group. *Group-oriented servers* (section 3.11) benefit by receiving a single group input, a *group operation* or a *group notification* invocation, from the underlying engineering mechanisms. However, in order to achieve this the group programming primitive must have a 'group input' semantics.

Hence there is a requirement to collate the instances of operation messages or notification messages corresponding to an *operation signature* or *notification signature*, so that a single collated group input can be offered to the *group-oriented server object*. This collation must be performed by the underlying group support platform before message invocation on the server object. This facilitates the server object in processing the entire client group's service requests in a single processing without undue waste of time and resources of the server.



Note: "requests" are not identical, but are instances of the same "operation signature".

Server's request delivery and reply generation requirements:

1. Give me a single 'group request' which is a combination of individual service request instances.
2. My reply is based upon group service request.
3. I'll either give a single reply to be (split and) distributed to all members of the client group or I'll give multiple replies, one for each member of the client group.

Fig. 2.4 Client Group interrogates a server object

2.4 Limitations of ODP Interrogation Primitive

The ODP *interrogation* primitive (see definition in section 1.5) or the *remote procedure call* primitive support the basic point-to-point ‘single request - single reply’ communication semantics. They do not scale up to the requirements of the communication between client group and server group.

These primitives accept a single request from the client and give a single reply to the client. They terminate with the return of a single reply to the client. It is possible to construct a single group reply through the use of appropriate collation mechanisms as described in the next chapter and to return the multiple replies as a single group reply. But the requirement of receiving individual reply instances as they keep coming in from the server group, in a controlled manner, is not supported by the interrogation or the remote procedure call primitive.

On the server object side, these primitives give a single service request to the server and accept a single reply from the server. They terminate with the receipt of a single reply from the server object, whereas the group-oriented servers need to give multiple replies, in response to a group operation invocation.

Table 2.3: Limitation of ODP Interrogation primitive

Requirements of a group communication primitive	Support in ODP Interrogation Primitive
Multiple reply delivery requirement (client side)	not supported
Group reply delivery requirement (client side)	supported, through the use of appropriate message collation mechanisms
Variable reply delivery requirement (client side)	not supported
Separate delivery of individual reply instances (client side)	not supported
Solicited reply delivery requirement (client side)	not supported
Terminable reply delivery requirement (client side)	not supported
Multiple reply acceptance requirement (server side)	not supported
non-blocking invocation semantics (client side)	supported in some implementations

Therefore while the ‘group request’ and ‘group reply’ semantics can easily be integrated in the ODP interrogation primitive through the use of appropriate collation mechanisms, the multiple and variable reply delivery support is not available in it. Also the capability to receive and process multiple reply types separately is a key requirement of group-based applications. This enables the client to receive the desired reply types before the others and to terminate the reply delivery when it has received the required number or types of replies. As shown in table 2.3 , the ODP interrogation primitive also lacks other requirements of group communication such as solicited reply delivery semantics, terminable reply delivery semantics, and non-blocking invocation semantics.

These semantics have an impact on the group message processing, invocation generation and invocation reception capabilities of the client and server. This is described in chapter 4.

2.5 Conclusion

The programming-level communication requirements of group-based applications are fundamentally different from those of singleton-client and singleton-server' communication. It requires major extensions to the semantics of the ODP interrogation primitive. This support is crucial for the large-scale development and deployment of group-based distributed applications.

CHAPTER 3 Group Interrogation: A Group Programming Primitive

Abstract

Group-based distributed applications, structured as a client-group and server-group, have distinct "programming-level" communication requirements. The "interrogation" or the "remote procedure call" is a familiar programming-level primitive to support "request-response style" communication in point-to-point client server computing. In this chapter we present the corresponding communication primitive for the support of multi-endpoint and client-server style communication between a client group and server group. This primitive is called group interrogation. We present the semantics of group interrogation. Message collation is a key requirement for the construction of group interrogation. We present some generic signature-based collation schemes which preserve the contents of the messages received from the source group. The semantics of group interrogation has has an impact on the message invocation, reception, and processing requirements of application objects. We describe properties or capabilities required of such group-oriented client and server objects.

3.1 Introduction

Many applications can profit from the "programming-level" group communication support, but such a support is lacking in currently available programming languages and operating systems. In this chapter we propose group programming primitives that are general enough to cover the requirements of many types of group-based applications and can be integrated in real programming languages and systems. We define the semantics of these primitives.

This chapter introduces the ODP-based "programming-level" communication primitives which provide semantic support for multi end-point interaction between 'client-group' and 'server-group'. These are the *group interrogation* and *group announcement*.

The proposed primitives are a logical extension of the ODP *interrogation* and *announcements* primitives. They extend the basic point-to-point client-server interaction model in order to address *one-to-many*, *many-to-one*, and *many-to-many* client server interactions required in an group-based application. These primitives provide partial group transparency to the client and server applications.

The proposed group programming primitives imply the use of some *message collation mechanisms* in order to construct 'group request' and 'group reply'. We propose *signature-based message collation schemes* which construct a single group message from the component messages as well as preserve the contents of these messages. They preserve the client-server style computing in a group-based distributed application.

3.2 ODP-Based Group Programming Primitives

Object groups give rise to new invocation semantics which apply to the collection as a whole and is known as group invocation semantics. The synergy of the object group model and the client-server model gives rise to a new and a very powerful invocation semantics.

Our aim is to provide a general programming primitive for the support of *group communication*. Especially, we wish to integrate *client-server style interaction* in a *multi-endpoint object group environment*. We present a generic definition of the proposed group-programming primitives followed by the signature and semantics of these primitives.

We adopt the programming-level interaction primitives of the ODP model, the *interrogation* and *announcement*, as the basis for the definition of the group programming primitives, for the support of the group communication requirements listed in the previous chapter. Here we present a simple and logical extension to the basic ODP-interaction styles.

3.2.1 Group Interrogation

A *group interrogation* is a *multi-endpoint* interaction between the *client group* and the *server group* consisting of

a. one *group operation invocation*: one (or more) *operation invocation*(s), which are instances of the same *operation signature* or are instances of different parts of the same *operation signature*, initiated by a *single* (or *multiple*) *client*(*s*) in a *client group*, resulting in the conveyance of information from the invoking client object(s) to the invoked *server group* members, followed by

b. one or more *group termination invocations,* received by each invoking member of the client group in either solicited or unsolicited manner, in response to the *group operation invocation*, resulting in the conveyance of information from the invoked members of the *server group* to the invoking members of the *client group*; where each *group termination invocation* is composed of one or more termination invocations, which are instances of the same *termination signature* or are instances of different parts of the same *termination signature*, initiated in response to the group operation invocation by the members of the server group.

The construction of *group operation* and *group termination invocations* is described in section 3.6.

3.2.2 Group Announcement

A *group announcement* is a *multi-endpoint* interaction between a *client group* and a *server group* consisting of:

a. one *group notification invocation*: one (or more) *notification invocation*(*s*), which are instances of the same *notification signature* or are instances of different parts of the same *notification signature*, initiated by a *single* (or *multiple*) client(s) in the client group, resulting in the conveyance of information from the invoking client object(s) to the invoked server group members.

Group announcement is a one-way communication, i.e. from client group to server group, while group interrogation is two-way. The latter subsumes the former. Hence the former is not discussed henceforth.

3.2.3 Group (Operation | Termination) Message

A *group operation* is a single message obtained through a combination of multiple instances of an operation signature (or instances of different parts of an operation signature) issued by the members of the client group. A *group operation* is invoked as a single operation on each member of the server group. A *group operation* corresponds to a ‘group service request’ from the client group.

A *group termination* is a single message obtained through a combination of multiple instances of a termination signature issued by the members of the server group in response to a client's (or a client group's) operation invocation (or group operation invocation). A *group termination* is invoked as a single termination on each member of the client group. A client may receive multiple group terminations, one corresponding to each termination signature, in response to its operation invocation on the server group. A *group termination* corresponds to a 'group reply' from the server group. The message combination scheme, also known as the collation scheme, is described in section 3.6.

3.3 Semantics of Group Interrogation

The *fundamental* basis of group interrogation primitive is the *distribution* of an operation message from the client object to the server group, the *collation* of replies received from the server group into a group reply before delivery to the client object, and the ability to deliver multiple replies individually in a *controlled* manner to the client object. It involves multiple (message) invokers and receivers, group invocations, message collation, reply soliciting and termination. These constitute the *inherent* characteristics of the proposed group interrogation primitive. They are described below.

3.3.1 Multiple Invoker and Multiple Invokee semantics

There are multiple clients and multiple servers involved in a group interrogation. The group interrogation provides the basis for application-level multi-endpoint interaction between a singleton client and a server group, client group and a singleton server, and client group and a server group.

3.3.2 Group Invocation Semantics

The group interrogation primitive allows a server to access multiple service requests from the client group through the receipt of a single *group operation* invocation. Similarly, it allows a client to access multiple replies from the server group through the receipt of a single *group termination* invocation.

3.3.3 Message collation semantics

The 'group request' and 'group reply' semantics imply the existence of some (engineering) mechanisms which combine individual (*operation* | *termination*) *messages* into corresponding *group* (*operation* | *termination*) *messages* at the (server | client) side. The following semantics are inherent in the message collation process:

a. *Parameter collation semantics*: (Operation | termination) messages carry information, from the (client | server) object to the (server | client) group, in the form of a set of *parameters*. Each parameter in the (*operation* | *termination*) carries a certain *type* of information. The group (request | reply) semantics imply that the information contained in the parameters of individual messages need to be *combined* in order to construct group (operation | termination). How the information, i.e., parameter types, in the (operation | termination) messages is combined to construct a *group* (*operation* | *termination*) *message* is discussed in detail in section 3.6.

b. *Collation cardinality semantics*: A (server | client) object may receive multiple (operation | termination) invocations from the (client | server) group. A question that arises in such cases is how long shall the (server's | client's) infrastructure continue to accumulate messages before starting the collation process. The group interrogation semantics implies the existence of a finite collation cardinality, for example the size of the (client | server) group, the knowledge of which is available to the underlying engineering mechanisms. When the required number of (operation |termination) messages are received, the underlying engineering mechanisms will collate these messages into a single group (operation | termination) message and invoke it on the (server | client) object.

c. *Collation duration semantics*: In some group-based applications the knowledge of the cardinality of the (client | server) group is either unavailable or is of no significance. Instead it is required to collate messages received during a certain period of time. In many cases, the collation duration represents the maximum time interval to accumulate messages, in order to avoid indefinitely waiting for the reception of messages. Messages received during this period are input to the collation mechanisms, and the resulting group (operation | termination) message is invoked on the (server | client) object.

d. *Collation membership semantics*: Yet, some other group-based (server | client) applications are interested in receiving (operation | termination) messages from specific members of the (client | server) group, for example to ignore some members temporarily or because of some other application criterion.

3.3.4 Controlled Reply Delivery Semantics

There are two modes of reply delivery in a group communication primitive. These are *unsolicited reply delivery* and *solicited reply delivery.* In the unsolicited reply delivery mode, the reply is delivered to the client object as soon as it is received by the underlying local group support infrastructure. The solicited or the controlled reply delivery mode gives the client the control to receive the replies as and when it is required. This requires a special primitive, such as "*poll_reply*()", to be associated with the group interrogation. This allows the client to dynamically control reply reception, subsequent to an operation invocation on the server group.

The "poll_reply()" has local semantics. It is intercepted and interpreted by the local group support proxy mechanisms (see next chapter). Therefore the format or the signature of this primitive is programming-language specific or can be mutually agreed between a client application and the underlying group support mechanisms in order to avoid any conflict with the client's message signatures.

The "poll_reply()" could be implemented as either a blocking or non-blocking call. In the former case, the client is blocked until a reply is received by the underlying group support mechanisms. In the latter case, either the reply, if available, is returned to the client or an appropriate no reply availability indication is returned to the client immediately.

In case of non-blocking invocation, multiple operation messages may be invoked on the server group by the client object. Each group invocation is uniquely identified by the *invocation instance identifier (*iiid). Therefore, the request for replies is also identified, such as "poll_reply(iid)", in order to request the reply corresponding to a specific group invocation.

3.3.5 Terminable Reply Delivery Semantics

Terminable reply delivery capability gives the client the control to terminate the receipt of subsequent replies when it does not want them any more. This is mostly required in conjunction with unsolicited reply delivery. This semantics can be realized through the use of local primitive, such as a *terminate_replies*(*iid*) from the client to the underlying group support engineering mechanisms which terminates the subsequent flow of replies corresponding to the specified group invocation instance.

The "*terminate_replies*()" also has local semantics and is intercepted and interpreted by the local group support mechanisms. Its format or signature is programming language specific or can be mutually agreed between a client application and the underlying group support mechanisms in order to avoid any conflict with the client's message signatures. It is non-blocking.

3.3.6 Invocation Completion Reporting Semantics or Variable Reply Delivery Semantics

In certain applications, the server group is transparent to the client and the client cannot process the replies until all of them have been received. Even if the replies can be processed as soon as they are received, in some applications a client cannot take an (application-specific) decision unless it is known that all the expected replies have been received.

Replies may be given to the client either individually as soon as they are received or they may be collated and offered as group terminations to the client. In any case the underlying group support mechanisms must inform the client when delivering the last reply. One possibility is to include a special termination signature, say *end_of_replies*(), in the group interrogation which is invoked by the local group support mechanisms after the delivery of the last reply to the client. Since "*end_of_replies*()" has local semantics - it is generated by the local group support mechanisms (proxy) and is interpreted by the client, the format (or signature) of this message can be mutually agreed between a client application and its group support mechanism in order to avoid any conflict with the client's termination signatures.

These semantics call for the design of a new programming language communication primitive. However, as shown in section 10.2.6, some of the semantics of the group interrogation can be implemented or simulated in existing programming languages by using a series of multiple remote procedure calls.

3.4 Signature of Group Interrogation

The semantics of *group interrogation* permits the use of the *interrogation signature*. The *group interrogation signature* is the same as an *interrogation signature*, with the "end_of_replies()" message (see section 3.3.6) included in the list of termination signatures. It may be noted that the other *group interrogation control messages* such as "poll_reply()" and "terminate_replies()" (see section 3.3.4 and section 3.3.5) are implicitly a part of the group interrogation. The interrogation signature consists of an *operation signature*, and a finite, non-empty set of *termination signatures,* as shown in figure 3.1. An (*operation* | *termination*) *signature* comprises of the following elements:

1. *name* of the (operation | termination), and
2. zero or more *parameter specifications*; each parameter specification consists of a parameter name and a parameter type.

In case of a matrix-mode group (operation | termination) (section 3.6.1) each parameter name in the corresponding (operation | termination) signature identifies a multi-element data structure which contains elements of the associated type.

operation-name(p_1: T_1, p_2: T_2, ..., p_n: T_n)::
termination-name-1(p_{11}: T_{11}, p_{12}: T_{12}, ..., p_{1r}: T_{1r})
termination-name-2(p_{21}: T_{21}, p_{22}: T_{22}, ..., p_{2q}: T_{2q})
:.:.:.:.:.:.:.:.:

....
termination-name-m(p_{m1}: T_{m1}, p_{m2}: T_{m2}, ... , p_{mk}: T_{mk}),
where, p's stand for parameter names and T's stand for the corresponding type names

Fig. 3.1 Interrogation Signature

In case of *interrogation*, a client expects to receive a *single instance* of one of the termination signatures, specified in its interrogation signature. Similarly, the server responds with a *single instance* of one of the termination signatures, specified in its interrogation signature.

In case of *group interrogation*, a client expects to receive *zero or more* instances of each termination signature, specified in its interrogation signature, in response to its operation invocation on the server group. Similarly, the server responds with *zero or more instances* of each termination signature, specified in its interrogation signature, in response to a group operation message from the client group.

3.5 Group Message Construction: Collation Schemes

The 'group request' and 'group reply' semantics associated with the *group interrogation* implies the existence of some (engineering) mechanisms in order to *combine* the individual (operation | termination) messages into corresponding group (operation | termination) messages which can be invoked on the (server | client) object as a single invocation. The combination of individual messages into a single group message is called message *collation*. The main question that arises is how to combine the messages, i.e., what *collation scheme* to use.

There are many collation schemes which can be used for the construction of group messages. Some of the collation schemes modify the contents of the message and tend to be very application-specific, while others do not alter the contents of the message but rather arrange the component messages in a certain order such that the contents of the components can be scanned and processed by the recipient object. We broadly classify the message collation schemes in the following two categories:

1. *Content-based collation scheme*: These collation schemes perform the 'mixing' of the messages by modifying and processing the contents, i.e., parameters of the messages. For example, a client object may wish to obtain the 'average' value or the 'maximum' or 'minimum' value of all the replies received from the server group. These schemes may employ a variety of content-modification procedures such as the use of mathematical functions (addition, multiplication, etc.) to combine messages, synchronization functions to display the audio and video contents of the messages in certain synchronized manner, etc. The collation process may also modify the original message signature, i.e., the signature of the collated group message is different from the original message signature the instances of which were combined. These schemes are application-specific, and not the scope of this thesis.
2. *Signature-based collation scheme*: Each operation and termination message carries different types of information in the form of different 'parameter types'. The message name together with the information types that it carries constitute the *message signature.* These collation schemes do not modify or process the contents of the message, instead they combine the instances of a given message signature by *linking* the instances of parameter types of a message signature in a certain order. The collation process does not modify the original message signature, i.e., the signature of the collated group message is the same as the original message signature the instances of which were combined.

3.6 Basic Group Message Construction Schemes

A group (OPR|REP) message is composed of multiple (OPR|REP) messages. *Collation* is the basis for the construction of group message. We propose some basic message collation (or combination) mechanisms such that the resulting group message is compatible with the ODP interrogation type system.

We adopt the *signature-based collation scheme* for the construction of group (operation | termination) message because it preserves the content of the component messages as well as their signature. This enables the group-oriented (client | server) object to scan and process the original and unmodified component messages which are presented to it as a single group invocation. Moreover content-based collation schemes are application-specific, whereas signature-based collation schemes are general and permit the recipient (client | server) object to gain access to the parameters of the component messages of a group message, and hence be able to later modify them in application-specific manner.

Before presenting the message collation modes, let us look at the elements of a message signature, the instances of which are to be *combined* into a group message. The elements of an (operation | termination) signature are the (*operation* | *termination*) *name* and its *parameters*. Each parameter is typed and has a name. Essentially, an (operation | termination) signature is a collection of typed information, the parameters, which are collectively identified by the (operation | termination) name. The collation process therefore deals with the combination of message parameters. The operation name identifies a service (or a function) provided by the server object. The termination name corresponds to an application context in the client object, in which identically named replies can be analyzed (or processed).

We have two basic *parameter collation* requirements corresponding to the following scenarios found in group-based distributed applications:

- Each member of the (client | server) group sends a complete instance of an (operation | termination) signature to the (server | client) object.
- Each member of the (client | server) group sends instances of different parts of the same (operation | termination) signature to the (server | client) object.

Corresponding to these requirements, we propose the following message collation mechanisms. They permit the combination of multiple instances of a message signature or instances of different parts of a message signature into a single group message.

- *matrix-mode message collation*
- *linear-mode message collation*

However it must be recognized that these are only two of the many other possible collation schemes. They are proposed because they are simple and straightforward to implement (see section 3.6.1.4) using simple programming language data structures.

3.6.1 Matrix-mode message collation

This collation scheme combines multiple instances of a message signature into a single group message which is then invoked on the sink object. We explain this collation scheme with an example of the Open Distributed Management Application [33], from the network management domain.

3.6.1.1 Group-Application-1: Managed Group - Manager Object Application

The application consists of a group of managed objects (MO) managed by a manager object, as shown in figure 3.3. Each managed object represents some physical network resource such as a switch, a multiplexor, a communication link, etc. A collection of managed objects representing identical resource types (such as all switches in a certain geographic area) are organized as a group and a manager object is assigned to manage the group of identical resource types. The collection of managed objects can be viewed as a *client group,* because each MO periodically sends the status information of the physical network resource that it represents, to the manager object (the *server object*), in the form of an *operation message*: `my_status(`sp_1`:`T_1`, .., `sp_n`:`T_n`)`, and expects an advice, for example, a management signal to appropriately modify the physical resource attribute values, from the manager object in the form of a *termination message*: `modify_status(`sp_1`:`T_1`, ..., `sp_n`:`T_n`).`The status parameters in the operation messages convey the current values of different attributes of the managed resource, and in the terminations they represent a modified value suggested by the manager object.

Each MO sends its status information in `my_status(`sp_1`:`T_1`, .., `sp_n`:`T_n`)`. The manager object is not invoked, by its group support proxy (section 3.11), until the status notifications from all the MOs are received. It is required to present a single group operation message, which contains the set of status parameters received in the individual operation messages, to the manager object. This enables the manager object to obtain a network-wide status information through a single operation invocation. Moreover, since the total performance of network is dependent upon the performance of the component network resources, the manager can decide to modify the individual attribute values for each managed object based upon this complete status information in the group operation message. After analyzing the status parameters received in the group operation message, the manager object sends appropriate *termination messages* to the MOs. This application represents the need to combine operation messages before they are invoked on the server object.

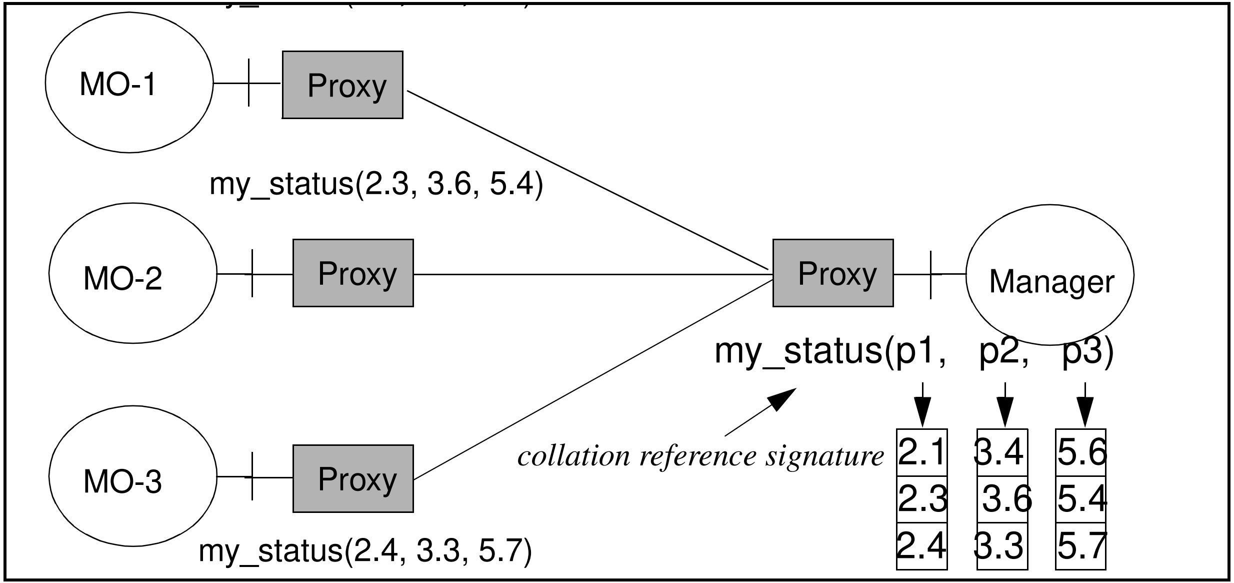


Fig. 3.2 Matrix-mode message collation: An example of Manager Object and Managed Group Interaction

3.6.1.2 Group Application-2: Modified Group Application-1

Consider group application-1 with the following modifications. The manager object, now becomes the client and the managed objects (MO) become the members of the server group. The manager object issues an operation invocation: `report_status()`, on the MO-group and expects to receive multiple terminations, $\texttt{my\_status(sp}_1\texttt{:T}_1\texttt{, .., sp}_n\texttt{:T}_n\texttt{)}$, one from each member of the MO-group. The terminations carry the management status information in the form of a set of parameters. The manager wants the set of status parameters, received in each termination message to be presented together as a single unit to it, so that it can analyze the status reports collectively. This application represents the case of termination message collation, before they are invoked on the client object.

3.6.1.3 Principles of Matrix-Mode Message Collation

1. *Single composite invocation*: The collation process results in the construction of a single group message, which is invoked on the sink object. Each component of the matrix-mode group message is called a *row message* (figure 3.3). Each parameter in a row message is *typed* and has a unique *name*.
2. *Collation reference*: The reference for the construction of matrix-mode group (operation | termination) message is the corresponding[1] (operation | termination) *signature* at the (server | client) object (see figure 3.3). This is called the *collation reference signature*. Therefore, the *names* and *types* of parameters in the final matrix-mode group (operation | termination) are the same as those in the col-

lation reference signature.

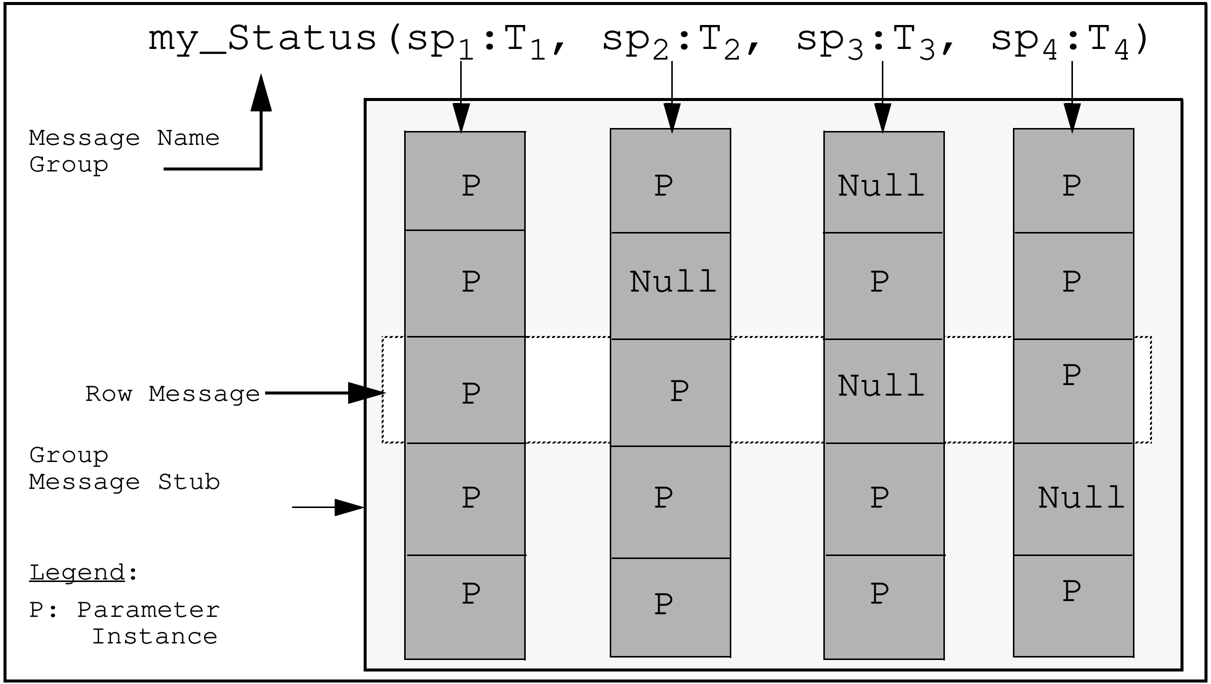


Fig. 3.3 Group Message Stub Using Matrix-Mode Collation: Array Structure Implementation

3. *Parameter collation principle*: Since there are multiple row messages in a group message, a group message consists of multiple instances of parameters corresponding to each parameter name. The parameter instances corresponding to a given parameter name in the individual *row messages* are collected and *collated* in a *multi-element data structure*. Hence there are as many multi-element data structures in a group (operation | termination) message as the number of parameters in the collation reference signature. As shown in figure 3.3, the multi-element data structures are combined into a single *group* (*operation* | *termination*) message *stub*, which is identified by the name of the collation reference signature. This stub is then invoked on the (server | client) object.

   Since the individual *row messages* may carry a variable number of parameters, which could be less than or equal to or greater than the number of parameters in the *collation reference signature*, the following semantics are implied in the parameter collation process:

3.1 *Null binding semantics*: If a row message contains no instance of a parameter name in the reference signature, the corresponding parameter in the group message is bound to a (programming-language-specific) "null" value (figure 3.3), and this value is included in the corresponding multi-element data structure. The sink object interprets that the corresponding parameter is not provided by the source.

3.2 *Chopping parameter semantics*: If a row message contains more parameters than those specified in the reference signature, then the parameter instances for which there is no corresponding parameter name in the collation reference signature are deleted. This implies that only those parameters which are desired by the invoked (server | client) object are retained.

These principles ensure the integrity of the computational type system of the group message.

---

1. Operation signature with the same name. In case of termination, it is the termination signature with the same name in the corresponding operation signature.

3.6.1.4 Implementation of matrix-mode message collation

In the previous section we have outlined the general principles of message collation. However the actual implementation of the collation scheme is specific to the data structures of the programming language used by the client and server applications. The construction of 'multi-element data structures' and the organisation of these data structures in a *group* (*operation* | *termination*) *stub*, can be realised in programming languages using a variety of data aggregation schemes. Here we outline some of them:

1. *Array structure*: A multi-element data structure is a linear organization of components. An array data structure in many programming languages is used for holding multiple data elements of a given type. An implementation of matrix-mode collation using this scheme is shown in figure 3.3.
2. *Linked-list structure*: A linked-list of elements is a more dynamic data structure capable of storing arbitrary number of elements of a given type.

3.6.2 Linear-mode message collation

This collation scheme combines instances of different parts of a message signature into a single group message which is then invoked on the sink object. We explain this scheme with an example of 'group computing'. A very simple sub-set of this application has been chosen in order to demonstrate the *linear-mode collation principle*.

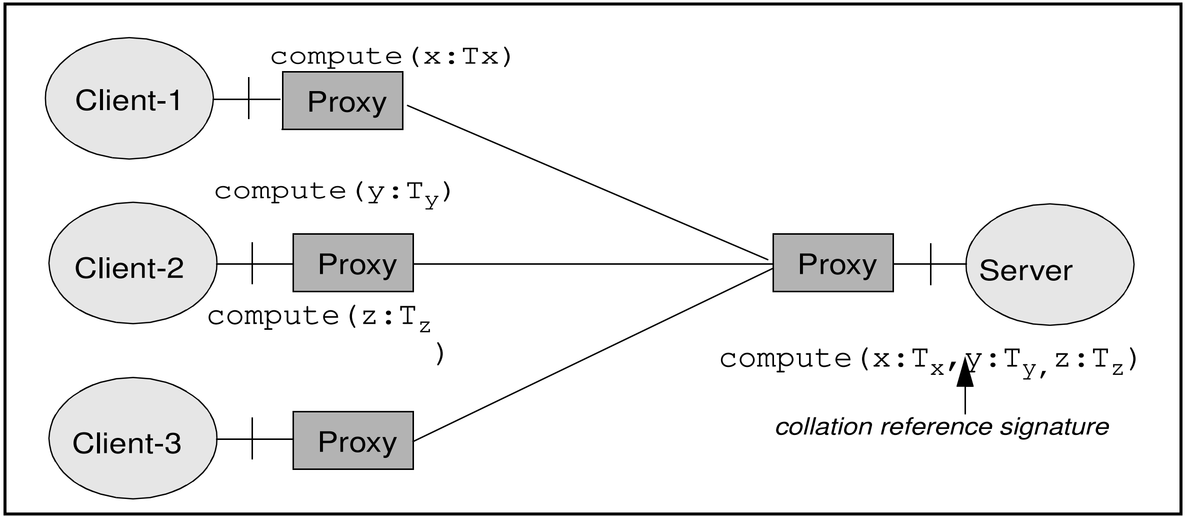


Fig. 3.4 Linear-mode message collation: An Example of Group Computing

3.6.2.1 Group Application-3: Group Computing

The application consists of a three-member client group bound to a server object. The server provides a 'computational service', say, the computation of a mathematical function: $(x^2+y^2+z^2)$. It supports an operation: `compute (x:`T_x`, y:`T_y`, z:`T_z`)`. Each parameter of the server's operation signature comes from a different source, one from each member of the client group. The individual clients invocations: `compute(x:`T_x`)`, `compute(y:`T_y`)`, and `compute(z:`T_z`)` are collated to produce a single operation: `compute(x:`T_x`,y:`T_y`,z:`T_z`)`, which is then invoked on the server object. This represents a very tightly coupled group-application, in which any change in the client group membership has an impact on the operation collation at the server object. Many group-based applications fall in this category (see section 7.7.4), where each client provides the partial input, say a fixed number of parameters, but is interested in the complete reply given by the server object. The total operation message is constructed from component messages.

3.6.2.2 Group Application -4: Parallel Computing Group

Consider another tightly-coupled group-based application. It consists of a client group and a server group. The server group is a ‘parallel computing group’, i.e., each server performs different processing on the same client input, and hence produces different result types. As a simple example, consider the server group composed of five members: the *adder*, *substractor*, *multiplier*, *divisor*, and the *averager.* Each client provides two parameters in its operation invocation: compute(x:T_x,y:T_y),and expects to receive the sum, difference, product, quotient, and average of the numbers from the server group. Each server provides part of the reply: result(a:T_{add}), result(b:T_{sub}), result(c:T_{mul}), result(d:T_{div}), and result(e:T_{avr}), and the total reply: result(a:T_{add}, b:T_{sub}, c:T_{mul}, d:T_{div}, e:T_{avr}), is constructed by the linear combination of the server replies. This kind of application represents the case of termination message collation. It also implies that a client can bind to the server group if each member is capable of giving the client a subset of the reply and the total set of replies received from the server group meets the client’s requirements.

3.6.2.3 Principles of Linear-Mode Message Collation

1. *Single composite invocation*: The linear-mode collation process results in the construction of a group (operation | termination) message at a (server | client) object, from instances of different parts of the same (operation | termination) signature issued by the members of a (client | server) group (see figure 3.4). Each component of the linear-mode group message is called a *tuple message*, e.g., compute(x:T_x) is a tuple message. Each parameter in a tuple message is *typed* and has a unique *name*. The set of parameters in a tuple message is called the *tuple parameter set*, e.g., {x} is a tuple parameter set. A linear-mode group (operation | termination) message is offered to the (server | client) object through a single invocation.
2. *Collation reference*: The reference for the linear-mode group (operation | termination) message construction is the corresponding (operation | termination) signature at the (server | client) object. This is called the *collation reference signature*. The *number*, *names* and *types* of parameters in the linear-mode group (operation | termination) is the same as those in the collation reference signature.
3. *Parameter collation principle*: Each tuple message carries a variable number of parameters, which may be less than or equal to the number of parameters in the collation reference signature, such that:
    1. the total number of parameters in the component tuple messages is equal to the number of parameters in the collation reference signature, and
    2. no parameter in the collation reference signature is received more than once, and
    3. for every parameter in the tuple message, there is a corresponding parameter with the same name and type in the collation reference signature at the (server | client) object.

    Since no more than one instance is received for each parameter in the collation reference signature from the individual tuple messages, the parameter collation process is a simple linear combination of parameters received in the tuple messages. Each received parameter instance (in the tuple message) is assigned to the corresponding parameter in the reference signature. Parameters for which no instances are received within a *collation duration* (if specified), are assigned (programming-language specific) “null” values.

3.6.2.4 Observations of Linear-mode invocation collation

A linear-mode group (operation | termination) message is essentially a singleton message obtained through the combination of parameters received from (clients | servers).

1. *Limitation on component invocations*: The number of components (tuple messages) of a linear mode group message is equal to the number of ‘tuple parameter sets’ in the collation reference signature. In the extreme case the minimum number of components is one (when the number of parameters in the ‘tuple parameter set’ is equal to the number of parameters in the reference signature) and maximum number of components is equal to the number of parameters in the reference signature (when each tuple message contains one parameter).
2. *Limitations on source group membership*: The number of (clients | servers) that can contribute to linear-mode group (operation | termination) is equal to the number of ‘tuple parameter sets’ in the corresponding (operation | termination) signature at the (server | client) object. Therefore the number of (clients | servers) in the (client group | server group) is equal to the number of tuple parameters sets in the corresponding (operation | termination) signature at the (server | client) object.

3.7 Group Interrogation vs. Group Transparency

Transparency is an important issue in a programming primitive. By giving the group (operation | termination) reception capability to the (server | client) applications, the members become *group-aware* and hence lose some *group-transparency.* We present the effect on group transparency to the (clients | servers) which are capable of accepting and processing group (terminations | operations). The difference between the properties of matrix-mode and linear-mode group messages is summarized in table 3.1 .

Table 3.1: Comparison of Matrix and Linear mode Collation Schemes

	Group message components	Membership Cardinality Transparency	Member Identity Transparency	Limitation on the size of the source group
Matrix-mode	Variable	No	Yes	No. (Variable-size source group)
Linear-mode	Fixed	Yes	Yes	Yes. (Fixed-size source group)

1. *Membership cardinality transparency*: When a (server | client) receives the matrix-mode group (operation | termination) invocation it becomes aware of the number of (client | server) group members that have sent the (operation | termination) invocations from the length of the *group operation stub.* However, the number of contributors to the linear-mode (operation | termination) invocation is transparent to the (server | client) object, unless the knowledge of ‘tuple parameter sets’ is known to the (server | client) application.
2. *Member identity transparency*: The issue here is the source of the component (row or tuple) invocations; how does the invoked (server | client) object know which member in the (client | server) group has sent which component invocation. In both the matrix-mode and linear-mode group messages, the identity of the source objects is hidden to the sink objects, unless there is an explicit parameter in the group (operation | termination) signature to convey this information to the sink application. The underlying engineering mechanisms have a knowledge of the source of the (operation | termination) messages and hence this information can be locally provided to the applications via this parameter.

3.8 Comparison between Interrogation and Group Interrogation

While the ‘group request’ and ‘group reply’ semantics can easily be integrated in the ODP interrogation primitive through the use of the proposed collation mechanism, the multiple and variable reply invocation and delivery semantics are not available in it. Similarly, group interrogation possesses some unique properties (see section 3.3) which are not required in a client-server case. The requirements for interaction between an client group and server group are fundamentally different from those of a singleton client-server interaction. The differences are summarized in table 3.2 .

Table 3.2: Interrogation vs. Group Interrogation

Interrogation	Group Interrogation
1. **single request - single reply** communication semantics (client and server side)	1. **single request - multiple reply** communication semantics, each reply may be of different type (signature) (client side) 2. **group request - single reply** communication semantics (server side) 3. **group request - multiple reply** communication semantics, each reply may be of different type (signature) (server side)
sender blocking call (client side)	**non-blocking call** (client side)
single reply delivery mechanism	**multiple reply delivery** mechanism
location transparent call (client and server side)	**location transparent** + fully or partially **group transparent call** (client and server side)
(no such requirement)	**terminable reply delivery semantics** (client side)
(no such requirement)	**invocation completion reporting semantics** (client side)
(no such requirement)	**controlled (or solicited) reply delivery semantics** (client side)

3.9 Need for Group-Oriented Objects

The semantics of the group interrogation primitive proposed earlier has an impact on the message invocation, reception, and processing requirements of the client and server objects. These semantics imply that the clients should be capable of handling not only multiple and variable number of replies from the server group, but also capable of receiving multiple reply types, of processing group replies, of interpreting special '*end-of-reply*' semantics, and of invoking '*terminate_replies*()' and '*poll_reply*()'. Similarly, the servers should be capable of receiving 'group operation messages', of processing these messages, and of generating multiple (and different types of) replies in response to a 'group operation message'. These semantics imply that certain special capabilities are required of clients and server objects which are involved in group interrogation. These are described in the following sections.

3.10 What is a Group-Based Distributed Application

We define a "*group-based distributed application*" as consisting of:
1. *Group-based client application*, and 2. *Group-based server application.*

The *group-based* (*client* | *server*) *application* is a distributed application organized as a (*client* | *server*) *group*. Such an application is composed of *group-oriented* (*clients* | *servers*). A model of the group-based distributed application is shown in figure 3.5.

3.11 What is a Group-Oriented (Client | Server)

The (client | server) object which is involved in communication with a (server | client) group and is capable of supporting the group interrogation semantics is called a *group-oriented* (*client* | *server*). A *group-oriented* (*client* | *server*) possesses the following properties (see figure 3.5).

1. *Partial group-awareness*: A group-oriented (client | server) object is partially aware of the (server | client) group. However it may not be aware of the cardinality or the identities of the group members.
2. *Group invocation interface*: It offers a *group invocation interface* (gii). On the client side, *operation messages* are invoked and *group termination messages* are received at the gii. On the server side, *group operation messages* are received and *termination messages* are invoked at the gii.
3. *Group management interface*: A group-oriented object optionally offers a *group management interface* (gmi). The gmi is used for communicating management or other policy related information from the object to the underlying *group support platform*. Depending upon the available group transparency, each object may obtain certain information about the group such as group membership, etc. through the gmi. This is discussed in detail in section 6.2. It may be noted that gii and gmi are logical interfaces and may be combined in some implementations.

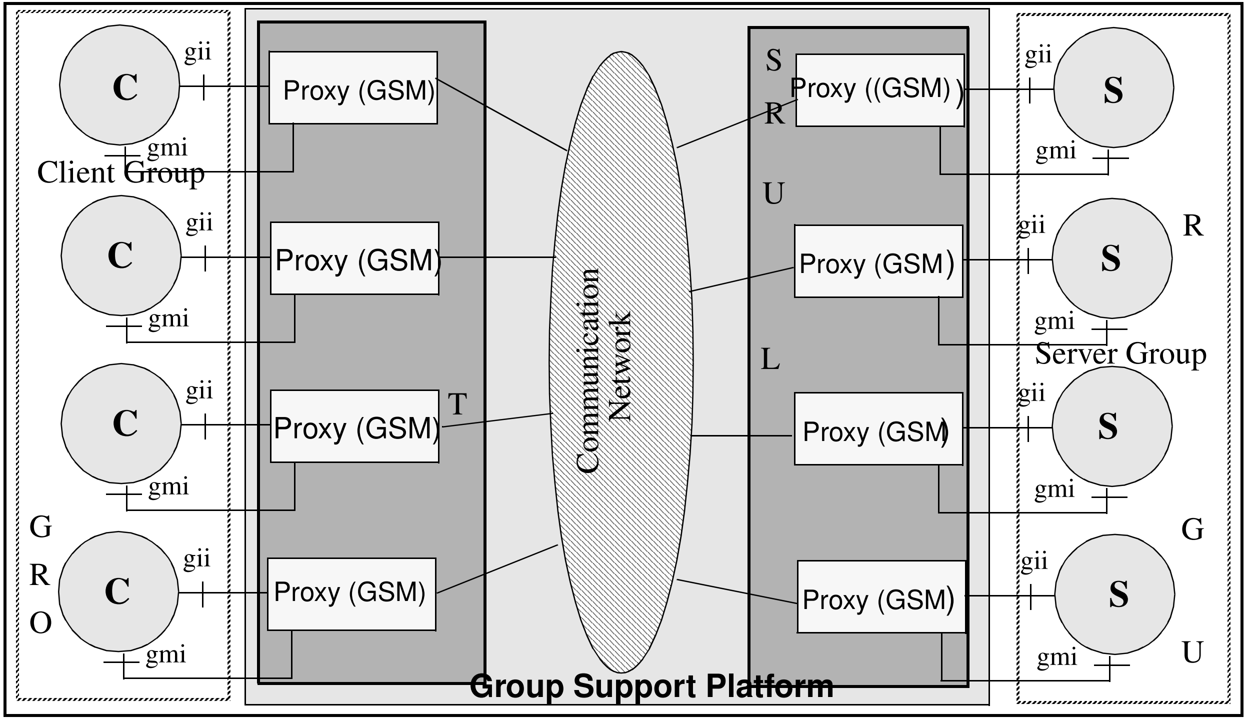


Fig. 3.5 Group-Based Distributed Application and the Group Support Platform.

4. *Message (invocation | reception) (on | from) local proxy*: Each group-oriented (client | server) object is supported (i.e., bound to) by a local proxy object called the Group Support Machine (GSM), which acts as a proxy of the (server | client group) with which it is interacting. The client invokes the operation messages on this proxy which is then transparently multicast to the all the objects in the server group as specified in the distribution policy. Similarly, the replies received from the server group are transparently collated by the proxy into group termination message before being invoked on the client object. The proxy also supports the other group interrogation semantics described in section 3.3.
5. *Group message reception and processing capability*: The *group-oriented client* is capable of invoking an *operation message* and of accepting multiple *group termination messages* at its “gii”. The *termination*

handlers in the group-oriented clients are capable of analyzing and processing multiple messages in a *group termination message*. Similarly, the *group-oriented server* is capable of accepting *group operation message* and of generating and invoking multiple *termination messages* (in response to the group operation) at its "gii". The *operation handlers* in the group-oriented server are capable of analyzing and processing multiple messages in a *group operation message*.

6. *Group interrogation semantics support capability*: The group-oriented clients are capable of interpreting "*end-of-reply*" semantics and of invoking "*poll_reply()*" and "*terminate_replies()*" at its gii.

The properties of group-oriented (client | server) objects are summarized in table 3.3 .

Table 3.3: Group-Oriented (Clients | servers)

Group-Oriented Clients	Group-Oriented Servers
Multiple reply handling capability (in response to an operation invocation).	Multiple reply generation and invocation capability (in response to a group operation).
Variable reply handling capability through "end-of-reply" notification from the proxy (GSM) object.	Number of returned replies is either one or equal to the number of components in the group operation.
Multiple reply types handling capability (in response to an operation invocation).	Multiple reply types generation and invocation capability (in response to a group operation).
Group termination processing capability.	Group operation processing capability.
Reply delivery soliciting capability (solicited reply delivery)	
Reply delivery termination capability (unsolicited reply delivery)	
Non-blocking invocation capability.	

3.12 Identification of Group Invocations in Group-Oriented (Client | Server)

Our group interrogation primitive supports both *blocking* and *non-blocking invocation semantics*. In case of *blocking invocations*, the client is blocked until the receipt of a single *group reply* (consisting of collated replies from server group) from the local proxy, the GSM.

In case of a non-blocking invocation, a client does not have to wait for the receipt of the replies after invoking an operation on the server group. A client can issue multiple operation invocations on the server group, one after the other, without waiting for the replies of the previous operation invocations. Due to communication delays, different link speeds, etc., replies are received by the client in any order. The question then is how shall the client know which reply is for which operation invocation.

3.12.1 Invocation Instance Identifier

The solution to the issue raised above lies in being able to uniquely identify an operation invocation. The client may invoke different operation messages or multiple instances of the same operation message one after the other. So it should be possible to identify every instance of the operation message issued by the client.

In our model the proxy object provides the solution. Whenever, an operation is invoked by the client object, it is intercepted by the local proxy, the GSM, which returns a handle, an *invocation instance identifier*(iiid), to the client object (only in case of non-blocking invocation). This handle is used both by the proxy and the client. When the proxy receives a termination message (reply), identified with an iiid, from the server group, the proxy either gives the message along with the iiid to the client or it invokes the message on a special interface of the client, so the client can relate the received reply with the corresponding operation invocation. The client may also use this handle to solicit the next reply or to terminate the replies corresponding to an operation invocation.

On the server side, the proxy invokes the *group operation message* on the server and is blocked until the receipt of the replies. The replies received from the server are tagged, by the proxy, with the iiid that was associated with the operation message. This process is described in section 6.4.1.

3.12.2 Unique Identifiers

An *invocation instance identifier* is unique in a given group-based application in order to uniquely identify an instance of *operation message* and its corresponding *termination messages*. It is represented as a combination of client-group identifier, the client identifier, the operation message identifier and the invocation instance count. For example, it can be represented as, *inv_instance_id:= group_id. member_id. op_inv_id. instance_count*. Hence it uniquely identifies the sender of the invocation, the name of the invocation, and the number of times this operation has been invoked by the sender.

3.13 Communication between Group-Oriented (Clients | Servers) and Local Proxy

There is a local protocol between the group-oriented (client | server) object and the group proxy object to which it is bound. This protocol pertains to the exchange of *application messages* (operation, notification, and termination) and *group interrogation control messages* (poll_reply(), terminate_replies(), end_of_reply()) between the (client | server) object and the proxy through the GII.

3.13.1 Client Side

As shown in figure 3.6, singleton (operation | termination) messages are invoked by the group-oriented (client | server) on their local proxy while group (operation | termination) messages are invoked by the proxy on the group-oriented (server | client) object.

The client expects to receive an invocation instance identifier from the proxy after it invokes an operation message on it (for non-blocking invocation). How the invocation instance identifier is returned to the client or is associated with a message is implementation dependent. It may be returned via the programming language specific invocation mechanism or the reply may be delivered on a special interface (a call back interface) of a client object (see section 10.2.6).

Additionally, as shown in figure 3.6, there is a local exchange of group interrogation control messages between the client object and the proxy. The *poll_reply*() and *terminate_replies*() are generated by the client and intercepted and interpreted locally by the proxy. The *end_of_reply* message is generated by the proxy and interpreted by the local client object. Group-oriented clients have the capability to invoke and receive these *group interrogation control messages*.

3.13.2 Server Side

A *group operation* received by a server object contains the ‘service requests’ (of a given type) of individual members of the client group. The server object’s reply is based upon this ‘group input’. After scanning, analyzing, and processing component service requests in the group input, the server object may either give a single reply or multiple replies, one for each source of the component invocation.

3.13.2.1 Single reply to all the clients based upon the group input

In some applications, a server may generate a single reply based upon the group input. This reply is to be sent to all the members of the client group whose service requests were present in the group operation message. This is programmed in the proxy object (see section 7.8.3).

3.13.2.2 Individual reply to each client based upon the group input

In some other applications, a server may give a different reply to each client. Although the replies are still based upon the group input, but each reply is different, and possibly of a different type (an instance of different termination signatures). In this case the number of replies generated by the server is equal to the number of components (*row messages*) of the *group operation message* (see section 7.8.1).

A server object may have multiple terminations listed in its *interrogation signature*. It may respond to a *group operation message* with any of these terminations. It may generate zero or more instances of terminations corresponding to a given termination signature in response to a group operation message.

3.13.3 Reply Handling Protocol between the Server object and Proxy object

A question that arises on the server side is how many replies should the underlying proxy object expect to receive from the server in response to a group operation message, and in case of multiple replies, how does the proxy know which reply is meant for which client. (The proxy object is responsible for sending the replies to the clients.) These issues pertain to a 'local understanding' between the server and the proxy object or they can be explicitly programmed in the proxy object.

The proxy object is explicitly programmed (see section 7.8) to expect either a single reply or multiple replies from the associated server object. In case of multiple replies, the proxy object is programmed regarding the order in which replies are expected from the server object. For example, one policy could be that a separate reply is received from the server object, one for each client, in the order in which the corresponding component message (*row* message) was arranged in the *group operation message stub*. Since the proxy had earlier collated the *group operation message stub*, it knows the identities of the clients and the order on which their *operation messages* were collated. So it can send the replies received from the server objects to the appropriate clients.

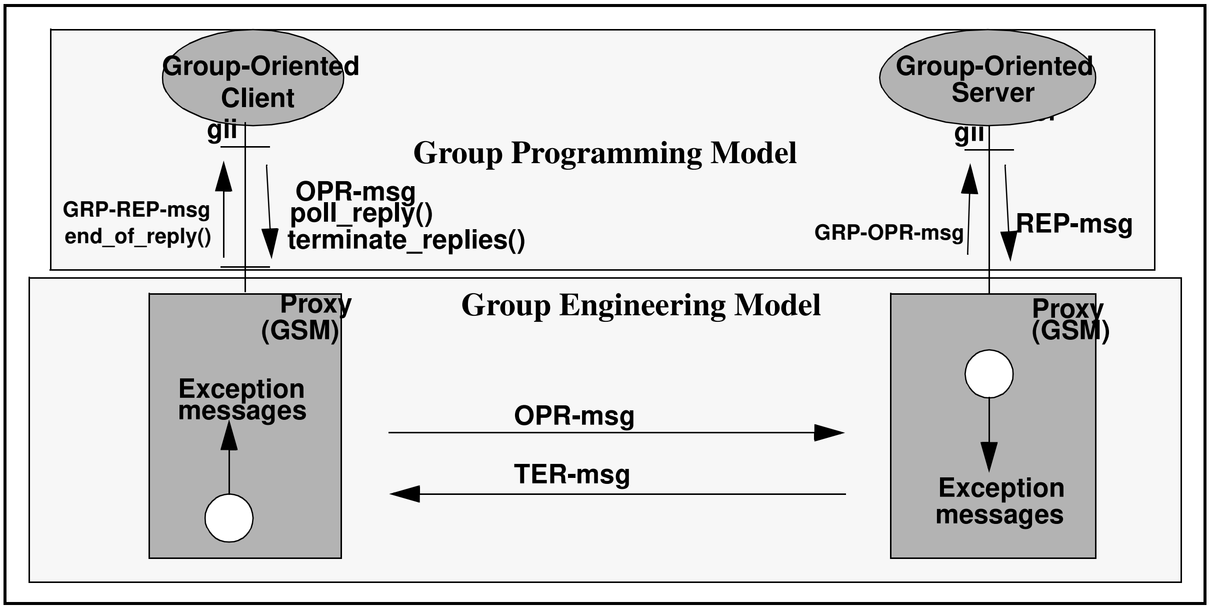

Group-Oriented Client
gii
Group Programming Model
Group-Oriented Server
gii
GRP-REP-msg
end_of_reply()
OPR-msg
poll_reply()
terminate_replies()
GRP-OPR-msg
REP-msg
Proxy (GSM)
Group Engineering Model
Proxy (GSM)
Exception messages
OPR-msg
TER-msg
Exception messages

Fig. 3.6 Protocol between group-oriented (client | server) and proxy

3.14 Conclusion

Group Interrogation is a programming-level communication primitive that gives the programmer the access to the low-level group communication at the application level. It provides the semantic support for combining multiple service requests from client group into a single *group operation* which can be invoked on a (group-oriented) server object. Similarly, it provides the semantic support for combining multiple replies from the server group into a single *group termination* which can be invoked on a (group-oriented) client object. This allows the server object to access multiple service requests from the client group in a single invocation. Similarly, it allows the client to access multiple replies from the server group, in response to a group interrogation, in a single invocation.

The semantics of the group interrogation is simple and easy to understand, yet powerful enough to express different message collation and reply delivery requirements. The client is provided with a handle to receive multiple and variable number of replies in a controlled manner. Group Interrogation provides partial group transparency to client and server applications.

PART-2

Distributed Platform Model: “Middleware Support” for Group-Based Distributed Computing Applications

CHAPTER 4 Group Support Services: Requirements of the Group Support Platform

Abstract

Currently available distributed platforms such as CORBA, DCE, DCOM, etc. offer middleware services for the support of client-server based distributed applications. This chapter investigates what "middleware services" are required in a "group support platform" for the support of group-based distributed applications. We identify the different aspects involved in the provision of each group support service in order to identify the functionality required by the corresponding group support agents.

4.1 Introduction

Much research has been done in the past in the area of group communication. However most of this research is devoted to the low level support for group communication, such as different types of ordered multicast protocols [81 - 94], membership management protocols [95 - 99], virtual synchrony [100 - 103], etc. These protocols provide low-level support to group-based distributed applications.

Experience with currently available distributed platforms such as CORBA [18], DCE [19], DCOM [20] demonstrates the importance of *middleware-level* support for point-to-point and distributed client-server communication. These platforms provide the "middleware services" necessary to support client-server computing, such as *messaging*, *binding*, *trading*, *location management*, *transaction*, *security*, etc. In this chapter we address the following question. What "middleware services" are required in a "group support (distributed) platform" in order to provide the engineering support for the group-based applications (see section 3.10). We identify some basic group support services that are required by a wide variety of group-based applications and describe the different aspects of those services.

4.2 Why Middleware Support for Group-Based Distributed Applications

The "middleware layer" is, in general, a layer of services sandwiched between the application and the low-level communication facilities. It is characterised by the following properties:

1. *Value-added service components*: The middleware layer contains the most-commonly needed services which are required by a wide variety of applications, such as access transparency service, location transparency service, mobility support services, transaction support services, security services, etc. These services make use of the low-level communication facilities.
2. *Programmable service components*: This layer provides generic services and hence they should be tailored to application requirements. The middleware service components can be programmed according to application requirements and policies.

3. *Interacting service components*: The distributed application consist of distributed 'application components' which are spread on different network nodes. Each application component is supported by a middleware layer. The middleware service components in these distributed network nodes interact with their peer components or with other service components in order to provide some value-added service to the application.

Some platforms, such as ISIS [104 - 106], Horus [107 - 108], Amoeba [111 - 112], Electra [109 - 110], Transis [113 - 114], Rampart [115 - 116], etc. provide low-level group communication support, such as message multicasting protocols, membership management protocols, virtual synchrony, etc. to distributed applications. However in these platforms the applications are very closely tied to these low-level group communication facilities, thereby sacrificing the flexibility in obtaining different types of services required in group-based applications. There is no middleware-level support in these platforms.

Group-based applications do not merely require message multicasting service or membership management service. As shown in chapter 7, group-based applications exhibit a rich variety of inter-object interaction patterns, message collation scenarios, inter-object synchronisation scenarios, etc. These rich interaction patterns need a well organised, configurable and programmable "middleware support", which should be programmed according to application requirements and policies. Such a sophisticated and organised "middleware services" is lacking in the case of group support platforms. It is an aim of this thesis to identify these services and to configure them as a logical and programmable entity in a group support platform.

4.3 What Middleware Services in the Group Support Platform and Why

The next step is the identification of the "middleware-level" services required for the support of group-based distributed applications. We call these services the *Group Support Services* (GSS), which are part of the *Group Support Platform* (GSP). As shown in figure 4.1, the GSP consists of the "group communication layer" and the "middleware services layer".

The group communication layer consists of the low-level message multicasting protocols, such as unordered broadcast protocol (UBCAST), source-ordered broadcast protocol (FBCAST), causally-ordered broadcast protocol (CBCAST), atomic-ordered broadcast protocol (ABCAST), etc. and membership management protocols, such as virtual synchrony (VSYNC) etc. These protocols are extensively researched in existing literature [81 - 99]. The middleware layer consists of application-specific services which provide extra functionality other than the low-level group communication function.

Our aim is to identify the issues of 'group communication' that arise at the application level, but are common to a wide range of applications and to put these issues in the "middleware" support. The provision of such a support at the platform level will substantially simplify the design and construction of group-based distributed applications. The application designer can now focus on the application aspects leaving the *group communication and coordination* aspects to the underlying distributed platform.

It may be noted that the set of group support services identified below do not represent the service requirements of all the group-based applications. These requirements vary considerably from one application to another. Our aim is to identify some common and widely applicable services which are required by many group-based applications, such as the ones discussed in chapter 7. Some of these services have earlier been identified in [34 - 35]. We categorize these services as *basic services*, *secondary services*, and *management services* depending upon their role.

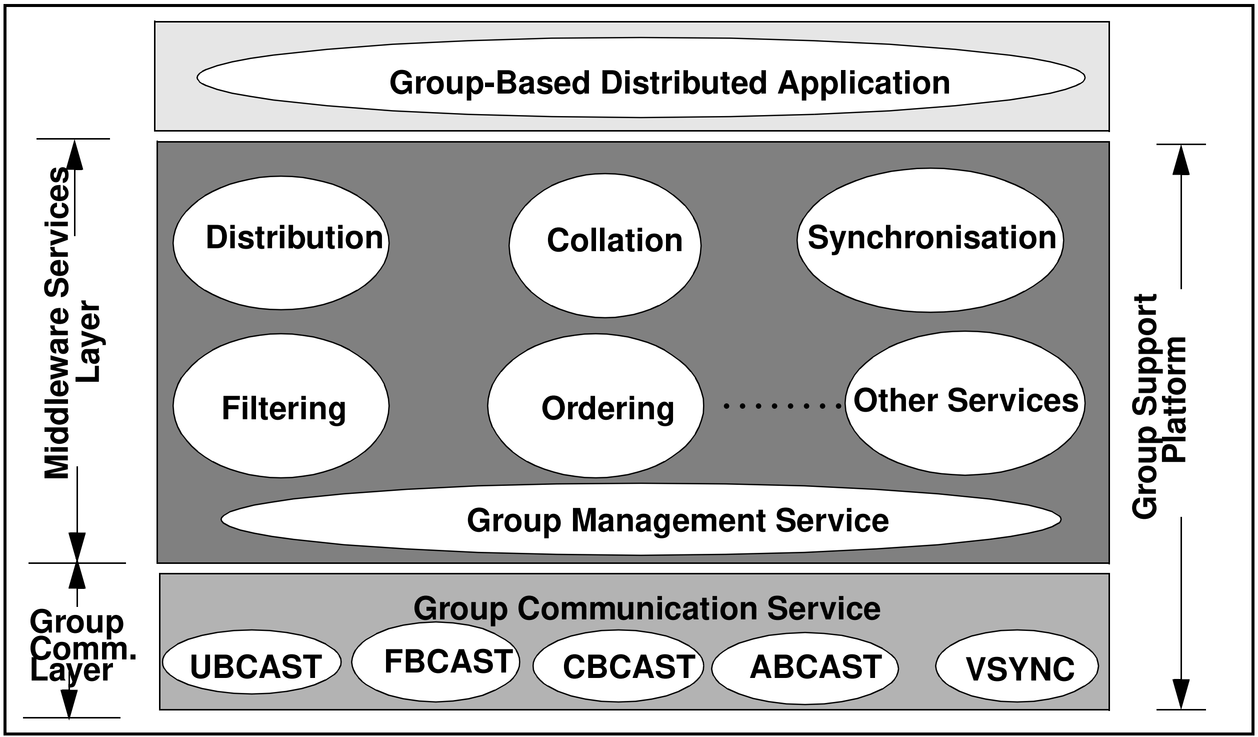

Group-Based Distributed Application
Middleware Services Layer
Distribution
Collation
Synchronisation
Filtering
Ordering
Other Services
Group Management Service
Group Support Platform
Group Comm. Layer
Group Communication Service
UBCAST
FBCAST
CBCAST
ABCAST
VSYNC

Fig. 4.1 Group Support Platform: Middleware & Group Communication Services

4.3.1 Basic Group Support Services

The *distribution* of a message from a source object to the sink group and the *collation* of (related) messages received from the source group at a sink object is a fundamental requirement of almost all group-based applications. These two are the core services required for the support of group-based distributed computing applications, which are organised as a client group interacting with a server group.

1. *Message Distribution Service*: The *distribution* of a message from the source object to a sink group makes use of the underlying multicasting protocols. However, the distribution service needs to specify different aspects of distribution such as when to distribute the message, the recipients of the message, the message delivery ordering requirements, (see section 4.4) etc. These items are application specific and as such are programmed by the application.
2. *Message Collation Service*: In group-based applications, (related) messages from the source group need to be collected and combined into a single group message before being offered to the sink object. However there are many aspects of collation (see section 4.4) that need to be programmed in the collation service in order to meet the different message collation requirements of the applications.

4.3.2 Secondary Group Support Services

These services are the utility services which are required by many group-based applications. However we intend to capture a wide variety of service requirements through these middleware-level services. The provision of these services is dependent upon application requirements.

3. *Message Synchronisation Service*: Synchronised activity is a characteristic feature of many group-based applications. For example in some applications, a message cannot be *distributed* to the sink group until a permission (or quorum) is received from other members of the source group. Similarly, in some other applications, a message cannot be *distributed* to the sink group until some specific event occurs in the

source group. Yet in some other applications, members of the client group need to invoke messages on the server group in some synchronised way in order to gain mutually exclusive access to the server group or to bring certain application-specific state changes in the server group (see example in section 7.9). Synchronisation includes a wide range of application requirements. It may imply some kind of turn taking protocol within the source group or it may imply concurrency control in which messages are delivered to the sink object (a shared resource) in some serialised order. There are many aspects of synchronisation (see section 4.4) that need to be programmed in the synchronisation service in order to meet the different message synchronisation requirements of the applications.

4. *Message Filtering Service*: Filtering of received messages before their delivery to the group members, based either upon the client's criterion or the server's criterion or both is a very common requirement in server groups (see example in section 7.10). However there are many aspects to the filtering service (see section 4.4) that must be specified by the application. Filtering also requires "m-out of-n selection" in order to select the best 'm' out of 'n' qualified contenders in the server group.
5. *Message Ordering Service*: Ordering of the message delivery to the sink objects is a common requirement of message distribution service and message collation service, in order to ensure state consistency in replicated groups or to satisfy some other application requirement. In case of message distribution service, the ordering requirement is usually satisfied by the choice of appropriate multicasting protocols. In case of collation, the message delivery is ordered based upon factors such as the type of the message (for example some reply types must be delivered before others) or the source of the message (for example replies from certain server group members be delivered before those from other members), etc.

4.3.3 Group Management Services:

The management services are required for the support of both the basic services and secondary services. A commonly required group management service is the following.

7. *Membership Management Service*: These include services which control the membership of the group such as new members joining the group, current members leaving the group, monitoring the failure of group members, and notifying the change in the group membership. The membership of the group has an impact on distribution, collation, synchronisation, and filtering procedures.

4.4 Basic Issues of Group Support Services: Elements of Group Support Policy

In this section we identify the basic issues involved in the *distribution*, *collation*, *synchronisation* and *filtering* of a message. These pertain to the different aspects of a group support service that must be specified by the application in order to obtain that service. They can also be viewed as *group transparency parameters.* As shown in chapter 8, these issues also represent the basic elements of a group policy specification language. These policy elements are interpreted by the "policy-neutral" *group support agents* which contain the required mechanisms for the execution of these policies.

The notion of *policy* has been discussed earlier by many authors in different domains - management domain [127 - 133], trading domain [134], cooperative work domain [135 - 143], etc. A *policy* is a high-level statement of the objectives that an object or a mechanism is required to accomplish. The policy does not describe the behavior of an object in detail. It only prescribes the requirements which can later be transformed into behavior using appropriate translation mechanisms. In the following sections we identify the basic elements of *policy* as applicable to the group support middleware platform.

4.4.1 Issues of Message Distribution: Elements of Distribution Policy

Distribution of a message from a member to a group is a basic requirement of group-based applications. The *distribution service* requires the specification of the following items:

a. *What to distribute*: This involves the specification of the *message signature*, the instances of which are to be distributed.

b. *To whom to distribute*: This involves the specification of member identifiers to whom the message is to be distributed.

c. *When to distribute*: This involves the specification of a synchronisation condition which must be fulfilled before a message is distributed. This enables the distribution of a message to be synchronised with other events in the group.

d. *How to distribute*: Distribution usually has some *delivery ordering* requirements in the sink group. The following delivery guarantees are required in message distribution:

  - *Unordered delivery*: Messages from the source object are delivered to the sink group in any order using the Unordered broacast (UBCAST) protocol.
  - S*ource ordered delivery*: Messages from the same source are delivered to the sink group in the order in which they were sent using the FIFO broadcast (FBCAST) protocol.
  - *Causal ordered delivery*: Messages from different source objects are delivered to the sink group based upon the causal order in which those messages were invoked from their sources using the CBCAST protocol.
  - *Destination ordered delivery*: Messages from different sources are delivered to the members of the sink group in the same order; which may not necessarily be the order in which they were sent.

e. *Resilience of distribution*: Some applications require the guarantee that a message be delivered to atleast some minimum members of the sink group or to none of them, in the same order. This is achieved using an ABCAST protocol.

4.4.2 Issues of Message Collation: Elements of Collation Policy

Collation is the process of collecting and combining (related) messages received from the source group into a single group message before delivery to the sink object. The *collation service* requires the specification of the following items:

a. *What message to collate*: This involves the specification of the message signature, the instances of which are to be collected and collated into a single group message. According to our collation scheme (see section 3.6), only messages of the same type can be combined into a group message. Messages of different types are collated and delivered separately to the sink object.

b. *Whose messages to include in the collated group input*: A message of a given type is usually received from multiple members of the source group. A sink object may wish to include either all messages in the final collated group input or a subset of them based upon their source.

c. *How many messages to include in the collated group input*: A sink object may wait until all of the expected inputs are received or it may put a limit on the *collation period* by restricting the number of inputs to a certain minimum.

d. *How long to wait accumulating the inputs*: Messages may be lost due to communication failures or a message may not be generated at all due to member failure. A time limit must be explicitly imposed on the collation period in order to avoid indefinite delay in constructing and delivering the final collated group input to the sink object. In some cases messages of a given type are generated periodically by members of the source group. In such cases collated group inputs are delivered to the sink object *periodically*. However, if multiple inputs are generated by a source object during a collation period, then the

sink object may decide to include:
- *All inputs* received from a given source during the collation period in the group message.
- *First Input* from a given source in the collated group message, and the subsequent inputs from that source are included in the subsequent collation periods for the construction of subsequent group messages.
- *Recent Input* received from a given source during the collation period in the group message, while the earlier inputs are discarded.

e. *How to collate*: The final issue in the collation process is how to *combine* the received inputs into a group message. This is discussed in chapter 3.

f. *When to deliver collated message*: Once a group message is constructed, it may either be delivered immediately to the sink object or it may be deferred until some *synchronisation condition* is satisfied. This allows the message delivery to be synchronised with other events in the group.

g. *In what order to deliver collated messages*: Group messages may be delivered immediately once they are constructed or the sink object may impose certain *ordering requirements* on message delivery in order to receive the most important message types first and the rest later.

4.4.3 Issues of Message Synchronisation: Elements of Synchronisation Policy

Synchronisation is the process of coordinating the (*distribution* | *delivery*) of a message (from | to) a member with respect to:

(a). the distribution and delivery of other messages, i.e., occurrence of other events in the group or

(b). the receipt of quorum messages (approval) from other members of the group.

The following issues are involved in synchronising the *distribution* or *delivery* of a message:

a. *What message to synchronise*: This involves the specification of the message signature, the instances of which are to be synchronised with other messages.

b. *With whom to synchronise*: A member needs to perform synchronised activity with respect to the activities of other members in the group. So we need to identify the synchronisation enabling messages as well as the sources of these messages.

c. *How many messages are required for synchronisation*: A message may require single or multiple synchronisation enabling messages before it can be scheduled for distribution or delivery.

d. *How long to wait to receive synchronisation*: Synchronisation enabling messages may be lost due to communication failures. In order to avoid indefinite delay, an explicit time limit may be imposed on the receipt of synchronisation enabling messages, after which an exception condition is reported to the message sender (and the message is not distributed or delivered).

e. *Disabling of synchronisation*: The *distribution* or *delivery* of a message may be abandoned if an exception condition is reported by the members of the synchronisation group.

4.4.4 Issues of Message Filtering: Elements of Filtering Policy

Filtering is the process of selecting the received messages for delivery to the sink object based upon the *filtering criterion* (see section 7.10) specified either by the source object or the sink object or both. The following issues are involved in the filtering of a received message:

a. *What message to filter*: This involves the specification of message signature, the instances of which are to be filtered in.

b. *On what basis to filter*: Filtering is usually performed on the basis of a *filtering criterion* which is a boolean expression of *filter attributes*. Client's attributes are evaluated against server's filtering criterion and server's attributes are evaluated against client's filtering criterion. The received message is filtered in at the server side only when both the clients and the server's filtering criterion are satisfied.

c. *How many of the filtered objects to select*: If a message is groupcast to a server group, multiple objects in the server group may satisfy the filtering criterion. However, the client object may want only one or a fixed number of filtered objects to process the message. Therefore the client must specify the number of filtered objects that must be finally selected to execute the service request.

4.5 Conclusion

This chapter has identified the "middleware-level" group support services that are required in a group support platform in order to support the group-based applications. There are different aspects to each service. These aspects must be specified by the application in order to obtain the required group support service. These aspects are part of the group policy specification language.

CHAPTER 5 Group Support Machine: An Organisation of Group Support Services

Abstract

This chapter describes how the set of group support services, introduced in the previous chapter, can be configured together inside an architectural framework called the "group support machine" and how the components of this machine work together in the provision of middleware-level service to the applications. Each member of the group-based distributed application is supported by a group support machine. The set of group support machines communicating with each other through an inter-machine protocol constitutes a "group support platform". This chapter introduces an abstract model of an agent-based group support machine.

5.1 Introduction

In the previous chapter we have introduced a set of *group support services* which cater to the group communication requirements of a wide range of group-based distributed applications. These services can be provided as standardised "middleware" mechanisms by the underlying distributed platform. In this chapter we turn our attention to the organisation of these services in a common distributed platform so that the combination of them can be used in a logical and flexible way by the applications.

It is our intention to design a generic software architecture for the provision of group services. In this chapter we present such an architecture for the middleware support of group-based distributed applications. Each *group support service* is realised by a corresponding *group support agent*. A set of these agents, locally bound together and interacting with each other through inter-agent interfaces, constitutes a framework called the *group support machine* (GSM). Each member of the (client | server) group is supported by a GSM. The combination of GSMs communicating with each other through an inter-GSM protocol constitutes a *group support platform* (GSP).

5.2 Group Support Agents: Realisation of Group Support Services

Each group support service, introduced in the previous chapter, provides a distinct function. In our model each group support service is realised by the corresponding *group support agent* (GSA). Hence we have the *Distributor Agent* (D-Agent), the *Collator Agent* (C-Agent), the *Synchroniser Agent* (S-Agent), the *Filter Agent* (F-Agent), and the *Membership Manager Agent* (MM-Agent) corresponding to the distribution, collation, synchronisation, filtering, and membership management services respectively. The functionality of these GSAs is described in detail in chapter 6.

5.3 Group Support Machine: Configuration of Group Support Agents

Group-based applications do not need single isolated group support services; instead a combination of these services is required for the provision of group communication requirements of an application. Moreover these services need to *interact* with each other in order to provide group communication support to the applications.

The message distribution and collation are the basic services required by every application while synchronisation and filtering are the additional services required by some of the applications (see chapter 7). These services are required by every group member and hence they should be accessible to every group member. Moreover since the group members are distributed on a set of network nodes, it is preferable that these services are *locally* available at each member node so that the failure of one node (or the links leading to that node) does not prevent other members in obtaining the group support services. We propose a configuration of the group support agents, available to each group member, within a single logical architectural framework called the *Group Support Machine* (GSM). A model of GSM is shown in figure 5.1. The GSM is a configuration of Group Support Agents (GSAs), the components of which interact with each other locally via inter-GSA interfaces (see chapter 6) and remotely through inter-GSM protocol (see chapter 9) for the provision of group support service. GSM is a software machine, the components of which are software entities.

The group support services are required on both the client and server sides. However, these services are required by different message types on the client and server sides, as shown in table 5.1 .

Table 5.1: Group Support Services Requirement on the Client and Server side

	Distribution of	Collation of	Synchronisationof	Filtering of
At Client	**OPR-message** **NTF-messages**	**REP-messages**	the distribution of **OPR-messages NTF-messages**	**Not required**
At Server	**REP-messages**	**OPR-messages** **NTF-messages**	the delivery of **OPR-messages** **NTF-messages**	**OPR-messages** **NTF-messages**

5.3.1 Parallel Configuration of Group Support Agents

The next question is how to organise the group support agents within the framework of GSM. The answer is that this is a design issue and there is no unique configuration. The proposed solution is one of the many possible configurations. It reflects the *independence* of the GSAs as well as the *interactions* that occur between them (see chapter 6).

The GSAs are independent entities which perform orthogonal functions. Hence they are configured as a system of parallel agents which can be accessed by the (client | server) group members. The MM-Agent provides the supporting function to the rest of the GSAs. It monitors the group membership and feeds information about the group membership to the rest of the GSAs which need this information in order to provide their service. Hence it is placed as an agent in the supporting role.

The GSM offers its services to the individual components of the group-based application. These components are the members of the (client | server) group. These group members should not have to interface with the individual GSAs in the GSM in order to obtain their service. The composition and the configuration of the GSM should be transparent to the group member. Moreover the members should be able to receive services from the GSM as single logical entity. Additionally, the GSM should be able to (accept |

deliver) messages (from | to) the member object in its local programming language. Hence a *Group Agent* (G-Agent) which represents the GSM and interfaces with the local member object is bound to the GSAs as shown in figure 5.1. The G-Agent acts as a single point of contact with the GSM.

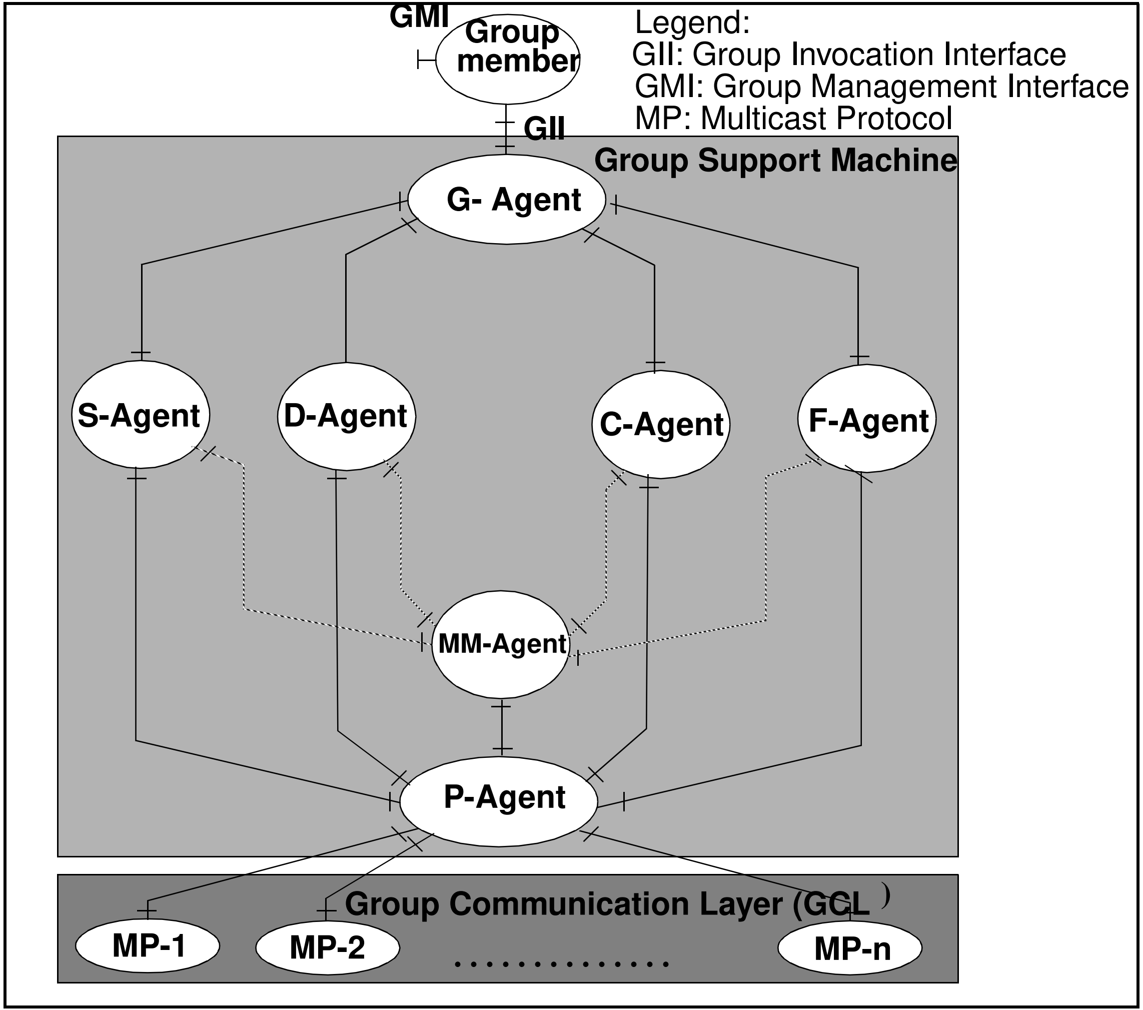

GMI
Group member
GII
Legend:
GII: Group Invocation Interface
GMI: Group Management Interface
MP: Multicast Protocol
Group Support Machine
G- Agent
S-Agent
D-Agent
C-Agent
F-Agent
MM-Agent
P-Agent
Group Communication Layer (GCL)
MP-1
MP-2
....
MP-n

Fig. 5.1 Group Support Machine (GSM): Configuration of Group Support Agents

The GSAs inside a GSM need to communicate with their peer agents in other GSMs through an inter-GSM protocol (see chapter 9). Each GSA generates certain types of messages in order to communicate with its peer agents in other nodes. Hence a GSM receives many different types of messages from other GSMs. There is a need for an agent to intercept the different types of messages received from the network (i.e., the underlying group communication protocols), parse these messages and give them to the appropriate GSAs for respective service processing. This function is performed by the *Parser Agent* (P-Agent). The P-Agent presents a single logical interface to the underlying network (or group communication protocols).

So, the parallel configuration of GSAs is bound to the G-Agent on one side and to the P-Agent on the other. The G-Agent interfaces with the member object and the P-Agent interfaces with the network.

5.3.2 Functioning of Group Support Machine

The GSM is composed of multiple GSAs which interact with each other before the (distribution | delivery) of the message (from | to) a group member in order to support the group communication requirements of the application. The next question is how do the components of the GSM work together in the provision of group communication service to the applications. In this section we will present a brief description of the functioning of the GSM. A detailed description is presented through example applications in chapter 6 and in chapter 7.

The GSM is placed between the member object and the communication network. It intercepts the messages received from both sides for processing by the appropriate GSAs before the *distribution* or *delivery* of message.

When a message is received from the member object for *distribution* to the sink group, it is intercepted by the G-Agent. The G-Agent gives this message to the D-Agent as well as to the other agents such as the S-Agent and the F-Agent. This allows the S-Agent to obtain the appropriate synchronisation of the message w.r.t other events in the source group, before its distribution. Similarly, the filtering constraints to be associated with the message are obtained from the F-Agent. On the receipt of the synchronisation signal (if any) from the S-Agent and the filtering constraints (if any) from the F-Agent, the D-Agent distributes the message to the sink group using the appropriate multicasting protocol.

Similarly, when a group protocol data unit GPDU (see chapter 9) is received from the network, it is intercepted by the P-Agent. There are two main types of GPDUs, the *control GPDUs* and *data GPDUs*. The control GPDUs carry the inter-GSM protocol control information only and data GPDUs carry both control and user messages. The P-Agent decodes the GPDUs and depending upon its type, it gives it to the appropriate GSAs for processing. If a data GPDU is received, the P-Agent gives it to the C-Agent as well as to the S-Agent and the F-Agent. This allows the S-Agent to obtain the appropriate synchronisation of the message w.r.t. the other events in the sink group, before its delivery to the member. Similarly, it allows the F-Agent to evaluate the filtering attributes of the source object against the filtering criterion of the sink object. Once the appropriate synchronisation signal and the filtering signal is received from the S-Agent and the F-Agent respectively, the C-Agent performs the collation of the received message with other appropriate messages and delivers the group message to the sink object.

Thus, while the components of the GSM perform orthogonal functions, the final distribution and delivery of the message involves the interaction between the components of the GSM. In chapter 6, we present, in detail, the relationship and the interaction between GSAs.

The GSM framework serves as a middleware component placed between the application and the underlying group communication protocols. In particular it can use a range of underlying message multicasting protocols with different reliability and ordering guarantees.

5.4 Group Support Platform: A Parallel Configuration of Inter-Communicating GSMs

Our aim is to design a middleware platform for the support of group-based applications, which are structured as a client group and a server group. The members of the client group and the server group are distributed on network nodes. Each member needs local access to the group support services organised within the GSM framework in order to avoid loss of group service support due to communication failures are node failures. Hence we propose a *distributed agent model* of the group support platform in which each member of the (client | server) group is supported by an instance of the GSM.

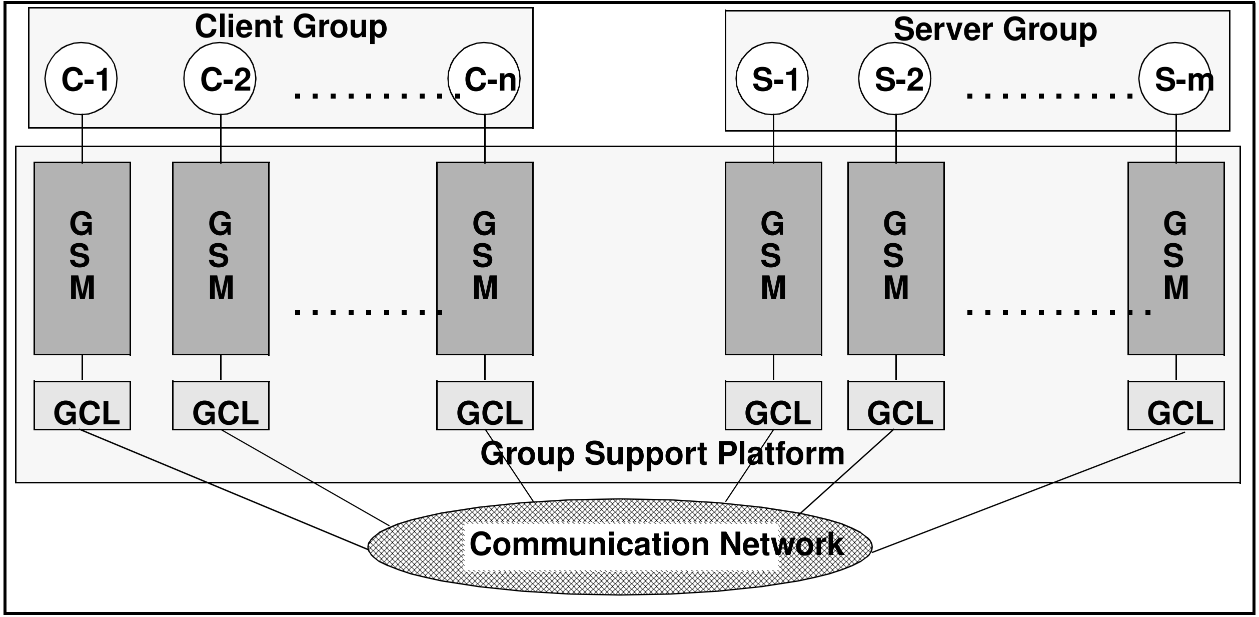


Fig. 5.2 Group Support Platform (GSP): A Distributed Agent Model

The model of the proposed group support platform is shown in figure 5.2. Each member is supported by a GSM. Each GSM is supported by the underlying *Group Communication Layer* (GCL) which provides group communication protocol support such as message multicasting protocols, membership management protocols, etc. The multicast protocol objects in the GCLs are connected to and communicate with each other through a communication network. This network could be a local or wide area network depending upon the distribution of the member objects.

The *Group Support Platform* (GSP) is composed of inter-communicating GSMs. The GSP is a distributed agent model, because the GSMs are distributed and the group support agents in each GSM interact with their peers in other GSMs via the inter-GSM protocol (chapter 9) for the provision of the group support service required by the applications.

5.5 Agent-Based Approach and Separation of Communication Functions

An agent is an entity or a software object (in an object oriented paradigm) which provides a “policy-neutral” functionality within a given application domain and whose behavior can be modified by the application of external user requirements or domain-specific *policies*. The GSM framework is an agent-based approach. The entities within the GSM framework are policy-neutral. For example, the C-Agent can support any collation function or accept any number of messages for collation, etc. The actual message collation process is guided by the policies which are external to the entity (see chapter 7).

The communication functions within the GSM are separated into different agents for the sake of “modularity” and “separation of concerns”. Each group support service identified in the previous chapter is a distinct and independent function which warrants a separate “module” of its own. Moreover, these separate modules need to communicate with each other in order to accomplish certain specific application requirements as shown in section 6.4. It may be noted that GSM is a model and a particular implementation of GSM may choose to combine all functions in a single class definition.

5.6 Group Support Machine: An External, Configurable, and Programmable Architecture

The GSM is an agent-based software architecture for the organisation of group support services. This framework of organising group support services offers the following advantages to the applications:

5.6.1 Separation of group-coordination aspects from application aspects

The group-support services, such as message distribution, collation, synchronisation, etc., are organised within the GSM framework, and are layered below the application components. This layered architecture lets the complex coordination behaviors (see chapter 7), found in group-based applications, to be modeled and executed external to application elements. The resulting partitioning of application and group coordination behaviors yields improved modularity, maintainability, and extensibility of group-based applications. The group communication and coordination aspects are separated from the application aspects. The designer can focus on the application aspects while leaving the group coordination aspects of the application to the GSM. The latter can be separately programmed as discussed below.

5.6.2 Extensible and configurable architecture

The GSM promotes explicit identification of group coordination services and a place for the insertion of these services. The GSM is a framework within which new group support services can be added and interaction with the existing service identified. Therefore the model allows the group support platform to evolve as new group support functionality is required.

5.6.3 Programmable and policy-driven architecture

The GSM is intended to be a generic and a programmable service architecture. As shown in chapter 7, it enables individual member preferences w.r.t. group services, such as to whom to distribute the messages, how to distribute the messages, how to collate the messages, with whom to synchronise the messages, etc. to be programmed through policy specifications. The GSM consists of ‘policy-free’ and neutral group support agents which can be programmed according to different application requirements. Therefore the architecture permits changes to the group coordination behaviors to be modeled external to the applications by modifying appropriate group support policies, such as distribution policy, collation policy, etc., without re-compiling the whole application.

5.6.4 Support for group transparency and group awareness

Depending upon the user requirements, the GSM can provide varying levels of group transparency to applications. Through the group management interface (chapter 6), the applications can be notified about the current group membership and other group management information.

5.7 Conclusion

The GSM represents an agent-based software architecture for the organisation of group support services. It is composed of group support agents which provide specialised support for group communication requirements of applications. We have presented the architecture in an abstract form allowing its implementation in different programming languages and computing environments. The GSM is a primary unit of the middleware support for group-based applications. It is a modular, configurable, extensible, and programmable software framework. The group support platform, is essentially a distributed agent model composed of inter-communicating GSMs.

CHAPTER 6

An Abstract Model of Group Support Machine

Abstract

This chapter describes in detail the internal components of the GSM, the functionality of these components, the interfaces between these components, and the interactions that occur at these interfaces. The aim is to describe the internal structure of GSM in an abstract and implementation independent manner. This chapter contains the summary of the functions of the GSAs illustrated later in the thesis through example applications in subsequent chapters.

6.1 Introduction

The GSM is the basic entity of the group support platform. It is a multi-agent software framework, the components of which offer diverse group support services and interact with each other in complex ways for the support of numerous types of group-based applications. In this chapter we describe in detail the services offered by the component agents of the GSM, the interactions that occur between the components of the GSM, and the interfacing of the GSM with the member object.

6.2 Middleware Box Between Group Member and Network: External Interfaces of GSM

The GSM acts as a single logical middleware entity placed between the member object and the underlying group communication layer. It offers interfaces both to the member object and to the group communication layer, as shown in figure 6.1. In this section we will describe the characteristics of these interfaces and the nature of information that is exchanged at these interfaces.

6.2.1 GSM - Group Member Interface

The internal components of the GSM and the way these components work together are hidden from the member object. The GSM appears as a single logical proxy object to the group member. Apart from application messages (i.e., OPR, NTF, REP messages) which are exchanged between the group members through the GSMs, the member object also need to input certain management information to the GSM which is required for the proper functioning of the GSAs. Hence the GSM offers two interfaces to the member object. These are the *GSM Invocation Interface* (GII) and the *GSM Management Interface* (GMI).

6.2.1.1 GSM Invocation Interface (GII):

The GII is used by the (client | server) member object to invoke (operation, notification | termination) messages on the GSM for *distribution* to the (server | client) group. It is used by the GSM to invoke group (operation, notification | termination) messages on the local (server | client) object. The group messages are constructed locally in the GSM. These application messages are treated as data by the GSMs and are exchanged transparently between the GSMs after appropriate group service processing. The member object interfaces with the GII via its *group invocation interface* (gii). The following are the characteristic features of the GII:

a. *Compatibility with member's invocation interface*: The message invocations are accepted from and delivered to the member object at the GII in the member's native invocation style. The invocation mechanism of the GII is compatible with the member's programming-language specific invocation mechanisms.

b. *Support for group interrogation capability*: Apart from using GII for invoking operation and notification messages, this interface is also used by the client object for invoking *group interrogation control messages* (see section 3.3), such as "*poll_reply*()" and "*terminate_replies*()" on the GSM. These messages give the client the capability to control the delivery of replies in response to an operation invocation on server group.

6.2.1.2 GSM Management Interface (GMI)

The GMI is used by the member object to perform certain management operations on the GSM or to communicate some local information to the GSM which is required for the functioning of the GSAs in the GSM (see section 7.9.4 and section 7.10.2). Similarly, the GMI is used by the GSM to communicate the object group related information, such as the current membership of the client and server group, to the member object, depending upon the group transparency available to the member.

GMI is a proprietary interface between the member object and the GSM. Every group-based application has different management requirements, and hence different types of information is exchanged at this interface. The following are the examples of types of information that are exchanged between the member object and the GSM through the GMI:

a. *Application-specific information*: The GSAs in the GSM are to a large extent application-neutral and perform their functions independent of the applications they support. However in some cases, they need certain application-specific information in order to perform their function. For example (see section 7.10.2), the F-Agent needs the values of some dynamically changing server attributes, such as the current load on the server (e.g., the queue length of the printer), current values of performance parameters, etc., based upon which it performs the filtering of the received messages. This information is input by the server object to the GSM through the GMI. Similarly, as shown in section 7.9.4, the S-Agent needs certain information from the client application such as which members of the server group have successfully executed its operation messages and which ones have not, so that the S-Agent can send appropriate 'synchronisation notification messages' to the other members of the client group, so that next operation messages are invoked only on those server group members who have successfully executed the previous operation messages. This information is given to GSM via GMI.

b. *Group membership information*: An object group is a dynamic entity. Members may join and leave the group, and there are member failures due to node and link failures. In some applications, the group members are interested in being informed about the current group membership. Depending upon the group transparency subscribed by the member object, the GSM (i.e., the MM-Agent) notifies group membership updates to the member object through the GMI.

c. *Group support policies*: The GMI may also be used by the member object to specify group support policies such as message distribution policy, collation policy, filtering policy, and synchronisation policy to the respective GSAs, which offer their services based upon the user's requirements specified in these policies.

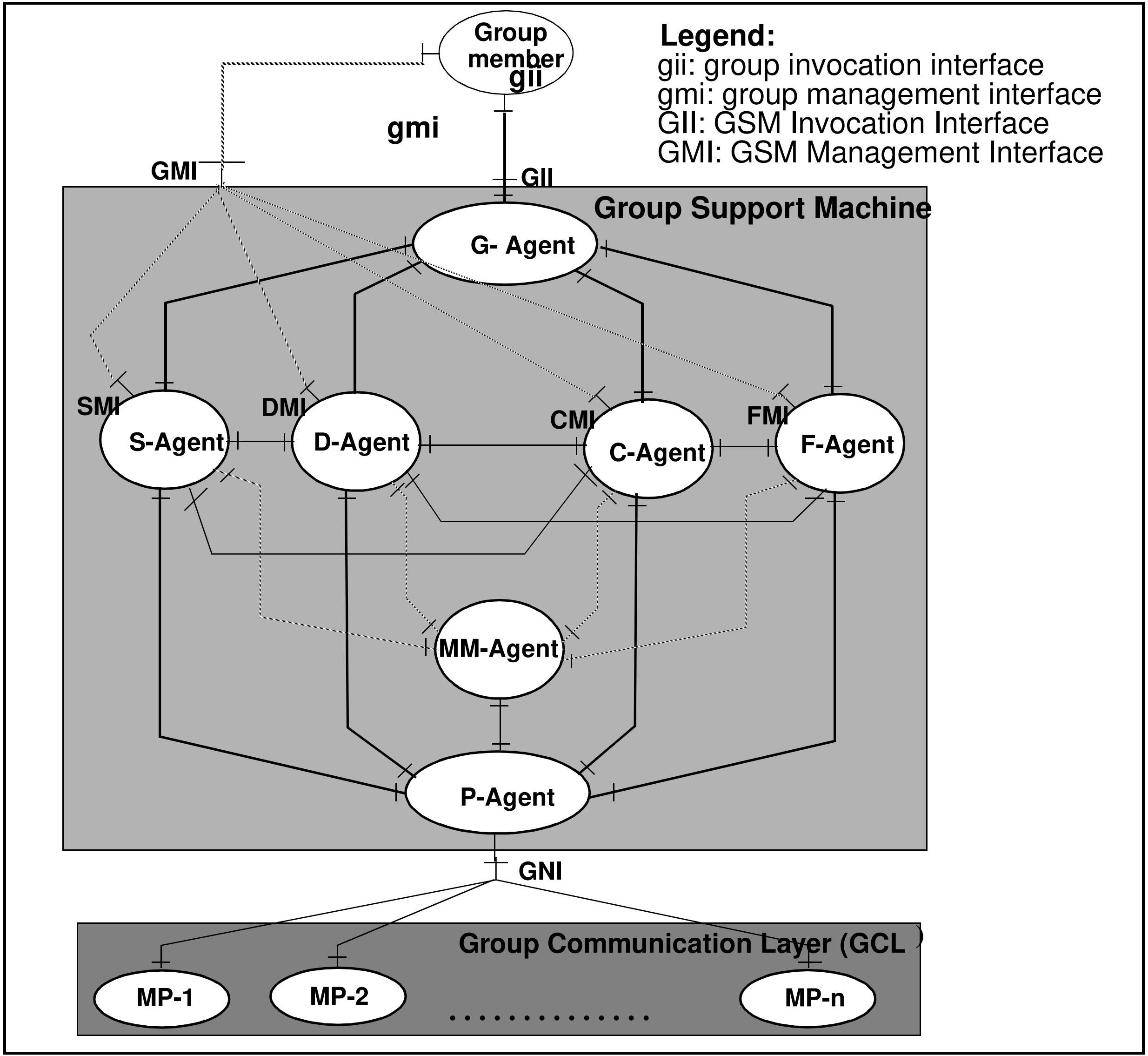

Group member
gii
gmi
Legend:
gii: group invocation interface
gmi: group management interface
GII: GSM Invocation Interface
GMI: GSM Management Interface
GMI
GII
Group Support Machine
G- Agent
SMI
DMI
CMI
FMI
S-Agent
D-Agent
C-Agent
F-Agent
MM-Agent
P-Agent
GNI
Group Communication Layer (GCL)
MP-1
MP-2
....
MP-n

Fig. 6.1 A Model of Group Support Machine (GSM)

The GMI is a logical interface. As shown in figure 6.1, each GSA supports its own management interface such as the *Distributor Management Interface* (DMI), the *Collator Management Interface* (CMI), the *Synchroniser Management Interface* (SMI), and the *Filter Management Interface* (FMI). These interfaces are used to input or output the information types explained above.

6.2.2 GSM - Network Interface

The GSM makes use of the low-level message multicasting and ordered delivery service provided by the underlying group communication layer, to transport the group protocol data units between the GSMs. The interface between the GSM and the underlying group communication layer is called the *GSM Network Interface* (GNI). A minimal information needs to be exchanged through this interface.

The group communication layer consists of different types of multicast protocol objects (MP-Objects) which provide different message ordering and delivery guarantees. The GSM (i.e., the P-Agent) only needs to select the appropriate MP-Object that meets its ordering and delivery requirement and give it the GPDU and the list of the group members to whom the GPDU is to be multicast. The GSM (the P-Agent) invokes the following message on the MP-Object: multicast(this_GPDU, to_these_members). Similarly, the MP-Object transfers the received GPDU to the GSM through by invoking the following message on GNI: input(this_GPDU).

6.3 GSM Components

The GSM is a multi-component software machine. Each component of GSM performs a specialized group support function. In this section we describe in detail the function offered by each GSA and the exception conditions that are encountered in group service processing. The functionality of the GSAs is described in an abstract manner. The reader is referred to appropriate examples in chapter 7.

The GSAs are *policy-driven* agents. They perform their respective functions based upon the corresponding *group support policy* described in the previous chapter. These policies can be input to the GSAs through their management interfaces (such as DMI, CMI, SMI, and FMI) or the policies can be specified as *policy scripts*. The policy scripts are stored in *policy repository objects* (PROs). The GSAs communicate with the PROs to find out what actions to perform when they receive a message.

6.3.1 G-Agent

The G-Agent acts as a gateway to the GSM. It supports the *GSM Invocation Interface* (GII). It interfaces with the member object on one side and with the rest of the GSAs on the other side, as shown in figure 6.1. The G-Agent provides the engineering support for the group interrogation semantics, i.e., it supports the non-blocking invocation semantics, solicited reply delivery semantics, terminable reply delivery semantics, as discussed in the previous chapters.

On the client side, the G-Agent gives the (operation, notification) message and the invocation instance identifier that it has generated for that message instance to the D-Agent for distribution to the server group. If the client application requires that the message distribution be synchronised with respect to other events in the client group and if filtering constraints are to be associated with the message before its distribution, then the G-Agent gives the message to the D-Agent, as well as to the S-Agent and to the F-Agent. This allows the S-Agent to start synchronisation processing and the F-Agent to search for filtering constraints associated with the message type, in parallel with the functions of the D-Agent such as message marshalling and GPDU construction. On the server side, the G-Agent gives the termination message and the invocation instance identifier that is associated with the message to the D-Agent for distribution to the client group.

6.3.2 D-Agent

The D-Agent performs the *distribution* of messages based upon the *distribution policy* specified in the *D-Policy Script*. At the client side, there is a D-Policy Script, one for each (operation, notification) message, thus allowing instances of each message type to be distributed to different members of the server group or to be sent using different ordering protocols depending upon application requirements. At the server side, there is a D-Policy Script, one for each operation message, which specifies reply distribution policy for the distribution of replies corresponding to the group operation message. The D-Agent performs the following functions.

1. *Message encoding, splitting, and renaming*: The D-Agent encodes the (OPR, NTF | REP) message received from the G-Agent as parameter name and value tuples and constructs the (OPR, NTF | REP)-GPDU by encapsulating the (OPR, NTF | REP) message, the invocation instance identifier associated

with the message, and the filtering constraints, if any, obtained from the F-Agent (see section 6.4.3). The filtering constraints are associated only with operation and notification messages. If *name transformations* (see section 7.5.3) are specified in the D-Policy Script, then appropriate name transformations are performed on the message name and/or its parameter names. If *splitting transformation* (see section 7.5.1) is specified in the D-Policy Script, then multiple GPDUs are constructed one for each message component.

2. *Synchronised message distribution*: In applications which require synchronised message distribution (see section 7.9), the D-Agent delays the distribution of the message until the reciept of a positive synchronisation signal from the S-Agent. The D-Agent then delivers the GPDU, the information about the members to whom it is to be distributed and the type of multicasting protocol to be used to the P-Agent, which in turn selects the appropriate MP-Object for message distribution. Finally, if an operation message is distributed, the D-Agent informs the C-Agent of the identities of the server group members from whom to expect the replies.
3. *Associating appropriate identifiers with termination messages*: On the server side, the D-Agent is responsible for the distribution of termination messages received from the server object (in response to group operation message) to the respective clients. The replies received from the server object are identified by the 'group invocation instance identifier' (see section 6.3.3) which is different from the 'invocation instance identifiers' which were associated with the individual components of the group operation message. However, the D-Agent must associate the appropriate invocation instance identifiers with the replies before sending the replies to the corresponding client objects. This is accomplished as follows.

   The D-Agent receives from the C-Agent the list of clients whose operation messages were collated into a group operation message, the invocation instance identifiers which were associated with each operation message, and the group invocation instance identifier which was associated with the group operation message.

   If a single reply is received from the server object, then this reply is distributed to all the clients whose components were included in the group operation message, with appropriate invocation instance identifiers tagged to them. In case of multiple replies, the order in which the replies are received from the server object corresponds to the order in which the component messages were arranged in the group message and it also corresponds to the order in which the clients are listed in the client list given to the D-Agent by C-Agent. Based upon the list of clients and the list of the invocation instance identifiers received from the C-Agent, the D-Agent, associates appropriate identifiers with the received replies and sends them to the corresponding clients.

6.3.3 C-Agent

The C-Agent performs the *collation* of messages based upon the *collation policy* specified in the *C-Policy Script*. At the client side, there is a C-Policy Script, one for each operation message type, which specifies the reply collation and delivery policy for the replies received in response to the operation message. At the server side, there is a C-Policy Script, one for each operation and notification message type, thus allowing instances of each message type to be collated using a different collation scheme as required by the application. The C-Agent performs the following functions.

1. *Message decoding, collation, and ordered delivery*: On the (client | server) side, the C-Agent receives the (REP | OPR, NTF) GPDUs from the P-Agent. It decodes these GPDUs and extracts the message and the invocation instance identifier associated with the message from the GPDU. Then it constructs the message

in a format that is understood by the local object. When the required number of the instances of the message are received from the specified group members within the specified collation time period, the C-Agent constructs the group message using the specified collation scheme and gives the group message along with the invocation instance identifier to the G-Agent, which in turn invokes the message on the member object using the local invocation mechanism.

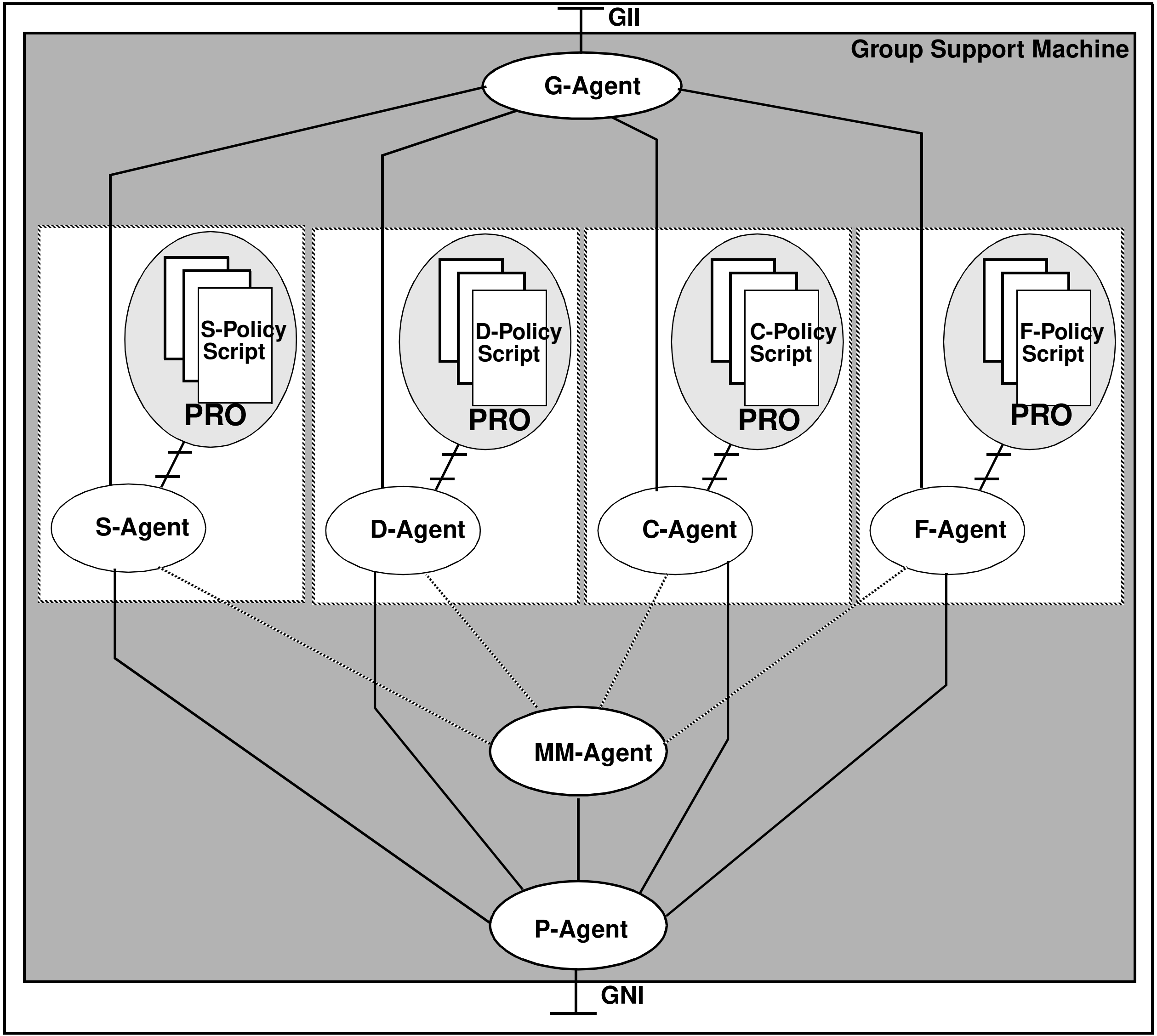

GII
Group Support Machine
G-Agent
S-Policy Script
PRO
D-Policy Script
PRO
C-Policy Script
PRO
F-Policy Script
PRO
S-Agent
D-Agent
C-Agent
F-Agent
MM-Agent
P-Agent
GNI

Fig. 6.2 A Model of Policy-Driven Group Support Machine

In some applications, the client may wish to receive certain reply types before others or replies from certain sources before those from others. The C-Agent performs the ordered reply delivery function according to the ordering scheme specified in C-Policy Script.

2. *Message delivery after filtering constraint processing*: Some group-based server applications require that (operation, notification) messages be delivered to the server objects only if the server object satisfies the client's filtering criterion or the client satisfies server's filtering criterion or both (see section 7.10). Filtering of messages based upon client's and server's attributes and filtering criterion is performed by the F-Agent.

   In such applications, the C-Agent includes the received message in the collation process and delivers it to the server object only after the receipt of a proper filtering signal from the F-Agent.

3. *Associating a unique identifier with group operation message*: On the server side, the C-Agent is responsible for the collation of service requests of the same type (i.e., instances of the same operation message type) received from the client group within a (periodic) collation window (see example in section 7.7), into a group service request (group operation message), before delivery to the server object. The group operation messages must be locally identified between the GSM and the server object, with a "*group invocation instance identifier*". This allows the G-Agent to identify its termination messages with a single group invocation instance identifier.

   In such cases, the C-Agent deletes the "invocation instance identifiers" associated with the component operation messages and generates a locally unique "group invocation instance identifier" which is associated with the group operation message. All replies received in response to this group operation message from the server object are identified with the group invocation instance identifier. Finally, the C-Agent gives to the D-Agent, a list of invocation instance identifiers of component messages (which were replaced with a group invocation instance identifier) and the list of the clients, so that the D-Agent can send the termination messages to the clients with the appropriate invocation instance identifiers tagged to them.
4. *Generation of the special "end_of_reply()" termination message*: On the client side, the C-Agent is responsible for the collation and ordered delivery to termination messages. In some applications, the membership of the server group is transparent to the client and the client cannot start processing the replies until it has received all of them. Since the knowledge of the server group is available to the C-Agent, it sends a special "*end_of_reply*(*invocation_instance_id*)" termination message to the client, after it has received all the expected replies. This is treated as a normal termination by the G-Agent and forwarded transparently to the client object.
5. *Generation of group exception messages*: The exception conditions arising due to collation, synchronisation and filter processing on the server side may result in the non-delivery of the operation message to the server object. In such cases, the clients must be informed and the reason for non-delivery reported. These exception conditions are reported to the C-Agent (see section 9.6), which in turn informs them to the local client object by constructing appropriate exception termination messages.

6.3.4 S-Agent

The S-Agent is responsible for the *synchronisation* of the (distribution | delivery) of the operation and notification messaged (from | to) the (client | server) object, based upon the *synchronisation policy* specified in the *S-Policy Script*. There is an S-Policy Script, one for each operation and notification message on the client side as well as on the server side. This allows the (distribution | delivery) of instances of each message type to be synchronised with different events in the (client | server) group, depending upon the application requirements.

In applications which require synchronised message (distribution | delivery), whenever an operation or a notification message is received from the (client | network) for (distribution | delivery), the (G-Agent | P-Agent) gives the message not only to the (D-Agent | C-Agent), but also to the S-Agent in order to synchronise the (distribution | delivery) of the message with other events in the (client | server) group. The (D-Agent | C-Agent) defer the (distribution | delivery) of the message until the receipt of the synchronisation signal from the local S-Agent.

In order to obtain the required synchronisation, the S-Agent sends the *synchronisation soliciting requests* (S-SOL-GPDU) to the *synchroniser group members* from whom the synchronisation or permission to (distribute | deliver) the message is to be obtained, waits for their responses (S-RES-GPDU), and when the required number of responses are received within the specified time period, it evaluates the responses to check if the synchronisation condition specified in the S-Policy Script is satisfied (see also, section 7.9 and section 9.7). Based upon its evaluation, the S-Agent either sends a positive or negative synchronisation signal to the local (D-Agent | C-Agent) responsible for message (distribution | delivery).

6.3.5 F-Agent

The F-Agent is responsible for the filtering of the received operation or notification messages at the server side, based upon the client's message *filtering policy* or the server's *filtering policy* or both, as specified in the *F-Policy Script*. There is an F-Policy Script, one for each operation and notification message, on both the client and server side. This allows the clients and servers to specify their own *filtering criterion* and *filter attributes* (object attributes) independently, so that a message is delivered to the server object only if both the client's and server's message delivery criterion are satisfied.

1. *Filter criterion evaluation*: On the server side, the F-Agent is responsible evaluating the server's filtering criterion using the client's filter attributes (received in the OPR or NTF-GPDU) and the client's filtering criterion (received in the OPR or NTF-GPDU) using the server's filter attributes. If both the client's and the server's filtering criterion is satisfied, the F-Agent gives a positive filter signal to the C-Agent, otherwise a negative filter signal is given.
2. *m-out of n selection*: If the client specifies that a specific number of servers which satisfy the client filtering criterion be selected for message delivery and execution, then the F-Agents in the server group enter into an "m -out of- n" selection process in order to select the best 'm' out of the 'n' contenders (see section 9.8).

6.3.6 MM-Agent

The MM-Agent is the local agent in charge of maintaining the current group membership information, such as a record of current group members, their identities, their application-specific roles, addresses, etc. It is responsible for monitoring the current group membership, such as the communication link failures and node failures (see section 9.9). Similarly it receives member addition and removal notifications from the group administrator. The MM-Agent notifies any change in the group membership information to the other GSAs which need this information in order to perform their functions.

6.3.7 P-Agent

The P-Agent acts as a gateway to the GSM from the network side (i.e., from the underlying MP-Objects). It supports the single logical GSM-Network Interface (GNI). On the other side it interfaces with the rest of the GSAs.

The P-Agent receives from the GSAs the GPDUs, the identities of the members to whom the GPDU is to be sent, and the information about the type of MP-Object to be used for multicasting the GPDU. The P-Agent selects the appropriate MP-Object and gives it the GPDU and the addresses of the group members to whom it is to be distributed.

The P-Agent receives different types of GPDUs from the network side (i.e., from MP-Objects). It decodes the type field of the received GPDU, and based upon the type of the GPDU, the P-Agent gives the GPDU to the appropriate GSA for processing.

6.4 Interaction between GSAs in the GSM: Internal Interfaces of GSM

The GSAs are to a large extent independent in the provision of their respective services. However they need to interact with each other locally, within the GSM, in order to provide the total group service required by the applications. In the following sections we describe the *relationship* and the *interaction* between the GSAs and the kind of information that need to be exchanged between them. Our intention is to identify the most commonly required interactions between the GSAs. Application-specific requirements may add more interactions or more information to the interactions identified below. Some examples of complete scenarios of interaction between GSAs are given in section 7.9.7 and in section 7.10.5.

6.4.1 Interaction between D-Agent and C-Agent: *Coordination between basic group support functions*

The basic support for *group-based distributed computing* is achieved through coordination between *distribution* and *collation* functions, as shown below for the client and server side considerations.

1. *Client side*: The D-Agent distributes the operation messages to the members of the server group. Depending upon the application requirements, an operation message may be sent to all the members of the server group or it may be sent only to partial server group membership. The C-Agent is responsible for the collation of replies received from the server group in response to an operation message. The C-Agent must know how many replies to expect and from whom to expect. The D-Agent gives this information to the C-Agent. The D-Agent invokes the following notification on the C-Agent, via the DC-interface, after the successful distribution of the operation message.

```
dc_collate_replies_from(OPR_inv_instance_id: inv_instance_id_type,
                        membership_list: member_id_list_type)
```

   This message notifies the C-Agent to expect replies, associated with the identifier specified in the `OPR_inv_instance_id,` from the server group members specified in the `membership_list`.

2. *Server side*: On the server-side, the C-Agent is responsible for the collation of the instances of an operation signature (i.e., instances of the same service request type) received from the client group, within a (periodic) collation window, into a *group operation message* (see examples in section 7.7). These messages may not necessarily be received from all the members of the client group within the specified collation window. Hence the replies received from the server in response to the group operation message must be sent only to those clients whose operation messages were received within the collation window.

   The C-Agent deletes the "invocation instance identifiers" associated with individual operation messages and assigns a local "group invocation instance identifier" with the group operation message before invoking it on the server object. The C-Agent has the knowledge of the components of the *group operation message* as well as of the identities of the clients who have sent these messages. The replies generated by the server object in response to the group operation message are identified with the "group invocation instance identifier". However the replies must be sent to the appropriate members of the client group with their original "invocation instance identifiers" associated with them. The D-Agent is responsible for the distribution of replies. It must know to whom to distribute the replies and what identifier to associate with each reply. The C-Agent gives this information to the D-Agent, because the C-Agent knows the identities of the clients whose operation messages were included in the group operation message as well as the original invocation instance identifiers that were associated with individual operation messages.

   Therefore the C-Agent invokes the following notification on the D-Agent, via the CD-interface after the successful collation of the group operation message and its delivery to the server object.

```
cd_send_replies_to(GRP_OPR_inv_instance_id: inv_instance_id_type,
                    membership_list: member_id_list_type,
                    OPR_inv_instance_id_list: inv_instance_id_list_type)
```

This message notifies the D-Agent to send the replies received from the server object to the clients listed in the "`membership_list`".

Either a single reply or multiple replies are received from the server object. In case of multiple replies, the number of replies is equal to the number of components in the group operation message and the order in which the replies are received corresponds to the order in which the operation messages were packed in the group operation message, i.e., the n_{th} reply is in response to the n_{th} operation message and must be sent to the corresponding client.

The replies received from the server object are identified with "`GRP_OPR_inv_instance_id`". However the D-Agent must send the replies to the respective clients tagged with their original "operation invocation instance identifier" which was associated with the corresponding operation message. The order in which the clients are listed in the "`membership_list`" corresponds to the order of identification of their respective operation messages in the "`OPR_inv_instance_id_list`". So the D-Agent sends the n_{th} reply received from the server object to the n_{th} client listed in the "`membership_list`" and tags that reply with the n_{th} "operation invocation instance identifier" listed in the "`OPR_inv_instance_id_list`".

6.4.2 Interaction between D-Agent and S-Agent: *Synchronise before message distribution*

Synchronised activity is a characteristic feature of many group-based applications. Either the *distribution* of a message from a source object to the sink group may need to be synchronised with other *events* in the source group or the *delivery* of a message to a sink object may need to be synchronised with other events in the sink group or both. In this section we will look into the former case.

In some applications, the distribution of (operation | notification) messages issued by the client need to be synchronised with other events in the client group(section 7.9). As shown in the example in section 7.9.5, the messages from the members of the "coordinated client group' are *distributed* in *synchronisation* with the message distribution from other members. Similarly, in other applications, a client may need to take the approval or quorum of other members in the group before a message can be distributed to the server group. In our model, the S-Agent is responsible for obtaining the required synchronisation (or quorum) according to the application's *synchronisation policy* by soliciting, receiving and processing "synchronisation-enabling messages". In such applications, the D-Agent does not distribute the message until the receipt of a local *synchronisation signal* from the S-Agent. So the S-Agent needs to inform the D-Agent the outcome of the synchronisation process in order for it to perform synchronised distribution. The S-Agent also has to advise the D-Agent to whom the message should be distributed, because the message distribution to all the group members may not be approved. In such cases the S-Agent invokes the following notification message on the D-Agent, via the SD-interface, after obtaining and processing the required synchronisation (or quorum) from the synchroniser objects (see details in section 7.9 and in section 9.7).

```
sd_distribute_message_to(inv_instance_id: inv_instance_id_type,
                         membership_list: member_id_list_type)
```

This messages notifies the D-Agent to send the (operation | notification) messages identified by the "`inv_instance_id`" to the server group members specified in the "`membership_list`".

Table 6.1: Interaction of D-Agent with other Agents before & after message distribution

	S-Agent	F-Agent	C-Agent
Client side	D-Agent receives the synchronisation signal from S-Agent before the distribution of (OPR | NTF).	D-Agent receives the 'filtering constraints' from F-Agent before the distribution of (OPR| NTF) message.	D-Agent informs the C-Agent about the number and sources of expected replies, after the distribution of OPR-message.
Server side	no interaction required.	no interaction required.	D-Agent receives the identities of the clients to whom a REP-message is to be distributed from C-Agent.

6.4.3 Interaction between D-Agent and F-Agent: *Insert the filtering constraints before message distribution at client side*

In some applications, such as in example in section 7.10.3, the client specifies some '*filtering constraints*' that must be satisfied by the members of the server group in order for the client's service request (operation message) to be delivered to them for processing. These filtering constraints are specified as set of client attributes (which are evaluated against server's filtering criterion) and client's filtering criterion (which is evaluated against server's attributes). These attributes and constraints, called the *filter attributes* and *filter constraints* respectively, are pre-specified in the F-Agent as part of the client's *filtering policy* for each message type. If the client wishes that its service request be executed by "m - out of- n" servers in the group, based upon its filter criterion, then *filter cardinality* is also included in the filtering policy. When a service request (operation message) is received from the client for distribution, the F-Agent interacts with the D-Agent in order to give the 'filtering attributes' and 'filtering criterion' to the D-Agent so that they are also distributed along with the message. The F-Agent invokes the following notification on the D-Agent, via the FD-interface, to communicate the filter constraints.

```
fd_include_filter_constraints(inv_instance_id: inv_instance_id_type,
                              filter_attribute_list: attribute_list_type
                              filter_criterion: constraint_expression_type,
                              filter_cardinality: cardinality_type)
```

6.4.4 Interaction between C-Agent and S-Agent: *Synchronise before message delivery*

In some cases, the delivery of a message to a group member need to be *synchronised* with other events in the group. This is a particular requirement in some server groups, in which the delivery of an (operation | notification) message to a server object needs the approval or quorum of some specific members of the server group in some supervisory roles. In such applications the message delivery to the server object by the C-Agent is withheld until the receipt of a *synchronisation signal* from the S-Agent. The S-Agent must obtain the required quorum (or synchronisation) from the specified members according to application-specific *synchronisation policy* and then inform the C-Agent about the outcome of the quorum. Hence the S-Agent invokes the following notification on the C-Agent, via the SC-interface, to inform the result of the synchronisation process ("`sync_result`"), which is usually a binary information.

```
sc_synchronisation_result(inv_instance_id: inv_instance_id_type,
                          sync_result: sync_result_type)
```

Table 6.2: Interaction of C-Agent with other Agents before message delivery to (Client | Server)

	S-Agent	F-Agent	D-Agent
Client side	no interaction required	no interaction required	receive from the D-Agent, the number and sources of expected replies.
Server side	receive the synchronisation signal from S-Agent before the delivery of the (OPR | NTF) message.	receive the filtering signal from the F-Agent before the delivery of the (OPR | NTF) message.	inform the D-Agent about the identities of the clients to whom the REP-message(s) is(are) to be distributed, after the delivery of the corresponding GRP-OPR-message.

6.4.5 Interaction between C-Agent and F-Agent: *Filter the received messages before delivery*

In applications which require filtered message delivery to server object (see section 7.10), although each GSM (C-Agent) in the server group receives the client's service request (operation message), only some of them may actually deliver the message to the server object due to the filtering constraints specified by the client. In such cases the C-Agent cannot include the received message for collation process (and/or for subsequent delivery to server object) until the filtering constraints and attributes sent along with the message are evaluated by the local F-Agent according to a pre-specified server's *filtering policy* and a permission received from it. Hence, the F-Agent needs to interact with the C-Agent in order to communicate the outcome ("`filter_result`") of the *filtering process* and the *m -out of- n selection process* to the C-Agent. This interaction occurs over the FC-interface.

```
fc_filtering_result(inv_instance_id: inv_instance_id_type,
                    filter_result: filter_result_type)
```

6.4.6 Interaction between MM-Agent and other GSAs: *Communicate group membership information*

The group membership, the member identities, and the location of members may be transparent to the (client | server) application components bound to the GSM. However this information must be available to GSAs in order for them to perform their functions. The MM-Agent is responsible for maintaining the current group membership information. It is responsible for monitoring the current group membership, including communication link failures and node failures. Similarly it receives member addition and removal notifications from the group administrator. Hence the MM-Agent interacts with the other GSAs, via the MM-GSA interfaces, to communicate the group membership information whenever there is a change in it.

```
add_member_notification(group_id: group_id_type,
                        member_id: name_type,
                        member_role: role_type,
                        member_location: address_type)
delete_member_notification(group_id: group_id_type,
                           member_id: name_type)
```

6.5 Conclusion

The GSM is a software architecture of group support middleware. The components of the GSM offer specialised group support services based upon the application requirements specified in the group policy scripts. Although the components of the GSM perform orthogonal group support functions, they need to interact with each other in order to provide the total group support service required by the applications.

CHAPTER 7 Group Coordination Models: Platform Support and Policy Specification

Abstract

The basic "client-server" model describes how a singleton client obtains service from a singleton server. In this chapter we describe the extended coordination models found in group- based distributed applications - the "group coordination models". These extended models coordinate multiple, independent servers to provide complex services for the clients. Similarly they permit the coordination of operation invocations from multiple, independent clients in order to request a 'group service' from a server object or to bring a desired state change in the server object. In this chapter we describe these extended coordination models which enhance the basic client-server interaction model in a multi-object environment. The coordination behaviors inherent in these models can be specified at a high-level using a group policy specification language. This language is introduced informally through examples.

7.1 Introduction

In general, a coordination model is characterized by a multi-component configuration and the interactions that occur between the components of the configuration [143 - 148]. The focus of this chapter is on coordination models found in *group-based* applications. We call such coordination models the "*group-based client-server coordination models*" or in short the "*group coordination models*".

A *group coordination model* is characterized by the *structure* of the application consisting of the configuration of a client group and a server group and the *interactions* that occur between the members of these groups. The former aspect of the coordination model is static while the latter aspect is dynamic and thus programmable. The intra-group and the inter-group interactions between the members of the client and server groups can be viewed at a high-level as *group coordination patterns* or *group coordination behaviors*. Therefore, as shown in figure 7.1, a *group coordination model* is a combination of a *group coordination behavior* within a given *group organisation*.

In this chapter we show how different *group coordination patterns* (*or behaviors*) can be obtained by composing the basic group support services such as message *distribution*, *collation*, *synchronisation*, *filtering*, etc. in different combinations. The combination of these basic services in different group organisations yields different group coordination models.

In this chapter we introduce some basic group coordination models and discuss associated group coordination patterns (or behaviors). These high-level group coordinations patterns are expressible as a combination of basic message distribution schemes (policy), message collation schemes (policy), message synchronisation schemes (policy), message filtering schemes (policy), etc.

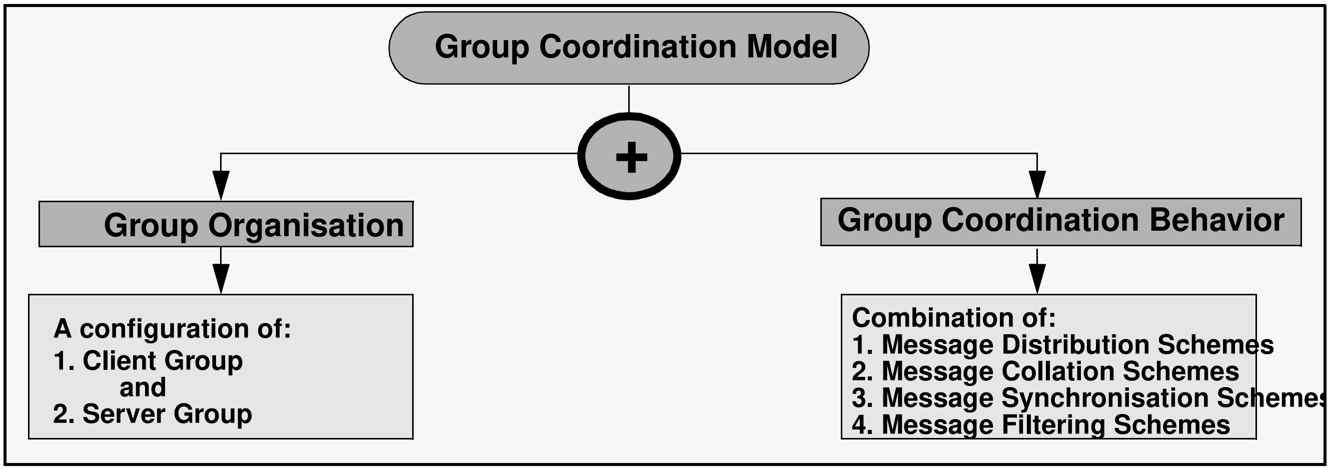

Group Coordination Model
+
Group Organisation
Group Coordination Behavior
A configuration of:
1. Client Group
and
2. Server Group
Combination of:
1. Message Distribution Schemes
2. Message Collation Schemes
3. Message Synchronisation Scheme
4. Message Filtering Schemes

Fig. 7.1 Group Coordination Model: Combination of coordination behavior and group organisation

Our aim is to represent group coordination patterns as programmable coordination behaviours which can be specified as message distribution, collation, synchronisation, and filtering policies. These policies are interpreted by the generic and policy neutral group support agents within the GSM. Therefore the group support agents support group coordination behaviours which can be specified and programmed within the GSM, external to the application logic.

In this chapter we informally introduce GPSL, the group policy specification language, as a language framework which is capable of specifying different group coordination patterns. A formal introduction to the syntax and semantics of the language is given in chapter 8.

7.2 Basic Group Coordination Models

Group-based applications exhibit a wide spectrum of coordination models. These coordination models involve numerous and complex coordination behaviors and multi-object organisation schemes. In this section we introduce some basic group coordination models found in group-based applications and discuss what group support services, such as distribution, collation, synchronisation, etc., are involved in the provision of the associated group coordination behaviors.

1. *Singleton Client - Server Group Coordination Model*: This model is characterized by a client object interrogating a server group and receiving multiple replies, one from each member of the server group. This coordination model involves the *distribution* of an operation message from the client object to the server group and the *collation* of multiple termination messages received from the server group before delivery to the client object. Message distribution scenarios are presented in section 7.4 and section 7.5. The reply collation and delivery scenarios are described in section 7.6.
2. *Client Group - Singleton Server Coordination Model*: This model is characterized by a client group interrogating a singleton server and receiving multiple replies, one for each member of the client group, from the server object. In this model, a set of client objects, related to each other in an application-specific manner, invoke instances of the same operation message signature (not necessarily identical) on the server. This group coordination model involves the *collation* of instances of the same operation signature into a *group operation message* (*group service request)* which is invoked on the server object and the *distribution* of the server's reply (or replies), which is *based upon the group service request*, to the client group. In this model the subsequent behavior of each client is dependent upon or biased by the previ-

ous behavior of the rest of the members of the client group. Service request collation models are described in section 7.7and reply distribution models are described in section 7.8

3. *Client Group - Server Group Coordination Model*: This model is a generalisation of the previous models. It involves multiple servers (or managers) in different *roles* serving (or managing) a client group. Each server (or manager) takes care of the different service aspect of the clients.
4. *Synchronised Invocation Model*: This is a special case of the "client group - server group" model in which distribution of an operation message from each member of the client group is synchronised with respect to the other events in the client group. This enables a group of clients to perform synchronised invocations on the server group in order to gain an exclusive access to the server group and to bring about a desired state change in the members of the server group. This coordination model involves *synchronising* the *distribution* of operation messages to the server group and *collating* server replies before delivery to the client object. This model is described in section 7.9.
5. *Filtered Invocation Model*: While the synchronisation of the distribution of (operation | notification) messages occurs in the client group, the filtering of these messages, based upon some filtering criterion, before delivery to the server objects, occurs in the server groups. This model is characterized by the *distribution* of (operation | notification) messages to all the members of the server group but the *filtering* of these messages only at the subset of server group members based upon either the client's filtering criterion or the server's filtering criterion or both. This model is described in section 7.10.

7.3 Basic Issues in Group Coordination Models

In most of the group-based applications, the clients often need to invoke multiple servers, coordinated to reflect how those servers interrelate and contribute to the overall application. Similarly the servers often need to receive multiple service requests from a group of (related) clients, coordinated as a single 'group service request'. Similarly, the members of the client group need to invoke service requests on the server group coordinated to bring about certain specific state change in the server application. These multi-client and multi-server applications give rise to many interaction issues. The basic issues that arise in these *one-to-many* and *many-to-one* coordination models are:

1. how multiple services are requested and how those services are organised,
2. how multiple replies from the service group are combined and the order in which they are delivered to the client,
3. how multiple service requests from client group are coordinated into a group service request and how multiple clients are organised,
4. how multiple replies generated in response to group service requests are distributed to the clients,
5. how invocations from members of the client group are coordinated to bring about the desired state change in the server application,
6. how service requests are selectively filtered in the service group in order to satisfy specific client and server requirements, etc.

Coordination models represent one way to handle these diverse group organisation and group interaction issues. We illustrate through examples the different group coordination models which involve a combination of message distribution, collation, synchronisation and filtering services. We start with the basic coordination models which involve single group support services and proceed to more complex ones which involve multiple services.

7.4 The Basic Message Distribution Model

The most basic coordination pattern in a group-based application is the distribution of a service request or a notification from a client object to a server group. The basic message distribution scheme involves the specification of server group membership to whom the message is to be distributed and the type of ordered multicast protocol to be used for message distribution, to the D-Agent.

7.4.1 Group Application-1: Stock Exchange Application

An automated stock exchange serves as a good example of group-based application in commercial domain. As shown in figure 7.2, the stock exchange is composed of three main entities: the stocks, the brokers, and the customers (each supported by a PC or a workstation). These entities are distributed on a combination of local and wide area networks. Each of them can be logically organised as a *stock group*, a *broker group*, and a *customer group*. A stock object is responsible for the management of its broker group and the broker object is responsible for the management of its customer group.

Each member of the stock group represents a company whose shares are traded in the stock market and periodically notifies the stock information of that company to the set of brokers who have subscribed to that information. The following notification is broadcast periodically by the stock object to its broker group: `stock_info(stock_id,time_of_day,stock_value,stock_volume)`.

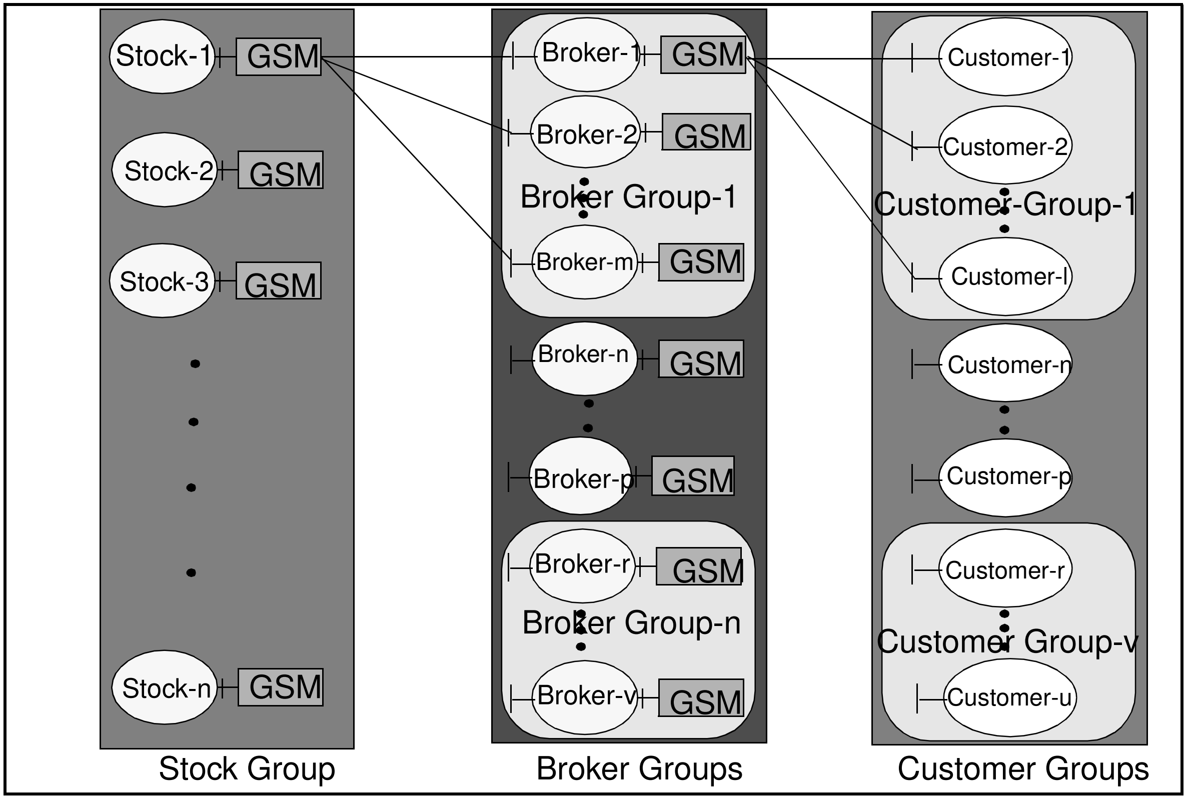

Stock-1
GSM
Stock-2
GSM
Stock-3
GSM
Stock-n
GSM
Broker-1
GSM
Broker-2
GSM
Broker Group-1
Broker-m
GSM
Broker-n
GSM
Broker-p
GSM
Broker-r
GSM
Broker Group-n
Broker-v
GSM
Customer-1
Customer-2
Customer-Group-1
Customer-l
Customer-n
Customer-p
Customer-r
Customer Group-v
Customer-u
Stock Group
Broker Groups
Customer Groups

Fig. 7.2 Stock Exchange Application: A Group-Based Distributed Application

Each broker deals with the stocks of many companies, and hence a broker may be a member of “broker groups” of many stock objects. Brokers may also be specialised in certain types of stocks (or companies) and hence the customers must take services of specific brokers in order to buy the stocks of specific companies. A customer may purchase different stocks through a broker. A broker is required to (periodically) notify the customers about the value of the stocks which they have purchased.

Each broker has a customer group which buys, sells, and receives other information about the stock through the broker. A customer may be a member of customer groups of many brokers.

7.4.2 Message Distribution Requirements & Policy Specification

The above mentioned application represents a case in which information distribution is a major requirement. Each stock object notifies the current stock information to its broker group. Since this information is distributed periodically, there is no stringent requirement for the atomicity of message delivery; it is sufficient that stock notifications are delivered to the brokers in the order in which they are sent, i.e., source ordered multicast is sufficient. Hence the D-Agent in the GSM bound to each stock object is programmed with the following policy:

```
notification_distribution_policy
for stock_info
    distribute stock_info(stock_id,time_of_day,stock_value,stock_volume)
    to my_broker_group
    using SOURCE_ORDERED_MULTICAST
end_policy
```

Fig. 7.3 Message Distribution Policy Specification

7.5 Advanced Message Distribution Models: Smart D-Agents

Message distribution is the most basic group communication service. However coordination behaviors within group-based applications tend to be complex and require some syntactical *message transformations* before distribution. In the following subsections we present the two commonly required message transformations through examples and policy specification.

7.5.1 Splitting Transformation

The message distribution in its most basic form involves the distribution of complete (operation, notification, or termination) message to the sink group. The source object invokes the complete message, but in some cases, the members of the sink group are not interested in the entire message, or they may not be capable of accepting or interpreting the complete message. Each member of the sink group may be interested in different parts of the message and hence capable of interpreting only limited parts of the message signature. This implies that the message contents be selectively distributed to the sink group. Splitting transformation is a syntactical message transformation which splits a message into multiple component messages. Each component message is identified by the name of the message and contains one or more parameters of the original message.

This type of message transformation facilitates many group coordination models as shown in this chapter. In particular it allows a computation to be divided amongst server group members by splitting the service request (operation message) into multiple component service requests (component operation messages) and distributing them to the members of the server group. The partial results can be combined upon receipt, and a single answer can be presented to client. This type of distribution and collation scheme fully exploits the multiprocessing capabilities available in distributed systems.

7.5.2 Message Splitting Requirements & Policy Specification

In group application-1, a broker object receives stock information from different stock objects at periodic intervals. It is required to distribute the average stock price information daily (at the end of the business day) to its customer group. The broker object sends the following notification to its customer group:

`stock_info(price_Nortel,vol_Nortel,price_ATT,vol_ATT,price_IBM,vol_IBM, price_Cognos,vol_Cognos).` However the customers are interested only in the information about the stocks that they have purchased. Hence relevant information needs to be distributed to individual members of the customer group. The D-Agent, on the broker side, can be programmed to perform *splitting transformation* before the distribution of the message, as shown in figure 7.4.

```
notification_distribution_policy
for stock_info

distribute stock_info(price_Nortel,vol_Nortel,price_ATT,vol_ATT,
                price_IBM,vol_IBM,price_Cognos,vol_Cognos)
transformed_as
          [
           component_message stock_info(price_Nortel, vol_Nortel)
           to customer-1,customer-5,cutomer-9
           using SOURCE_ORDERED_MULTICAST


           component_message stock_info(price_ATT, vol_ATT)
           to customer-1, customer-2, customer-3, customer-4
           using SOURCE_ORDERED_MULTICAST


           component_message srock_info(price_IBM, vol_IBM)
           to customer-2,customer-5,customer-6,customer-7
           using SOURCE_ORDERED_MULTICAST


           component_message stock_info(price_Cognos, vol_Cognos)
           to customer-4,customer-8, customer-9
           using SOURCE_ORDERED_MULTICAST
           ]
end_policy
```

Fig. 7.4 Splitting Policy Specification

The source (broker) object need not be concerned about sending multiple component messages to individual members of the sink (customer) group; it can make a single message invocation. The source may also not necessarily need to know who the recipients are and what part of the information they are interested in. Moreover the broker application need not be modified as the customers are dynamically added or removed from the customer group. These object group related aspects can be modified external to the (broker) application in the GSM.

7.5.3 Renaming Transformation

Another common group coordination model is the binding of a client object to a heterogeneous server group. Clients often require access to multiple heterogeneous servers to obtain independent services in distributed applications such as parallel computational groups, process control applications, office automation, etc. In such applications each member of the server group provides a different service. As shown in the example below, each member of the server group accepts the same client input, but performs different processing on it and hence produces different types of results. In particular the client's service request is identified by different names by each member of the server group. In some cases it is also possible that the service request (operation message) parameters are identified by different names on the client and server side.

The above mentioned situation is also possible in homogeneous service groups, in which each member of the server group can perform the requested operation and produce the same result, but the same service request is identified by different names by each member. In all such cases, it is desirable that the clients be able to invoke the server group through a single generic service request in order to maintain the server group transparency.

This implies that the (operation, notification, termination) messages be appropriately renamed at the source before distribution to the sink group so that these messages can be identified by their recipients. Naming transformation is a syntactical message transformation in which a message and/or its parameters are renamed appropriately before distribution. Since the GSM on the source side has the knowledge of the sink group, this transformation is performed by the D-Agent before message distribution.

7.5.4 Group Application-2: Parallel Computational Group

In many cases a distributed application is organised as group-based application in order to exploit the parallelism and heterogeneity of application components. In such applications, the service is offered not by a single server, but by a set of independent and heterogeneous server objects. One such application is a *parallel computational group*, which is an example of heterogeneous server group. In this example we consider a simple computational group which is composed of set of server objects which offer different services, such as an *Adder*, *Multiplier*, *Arithmetic-Mean Generator*, *Geometric-Mean Generator*, and *Harmonic-Mean Generator.*

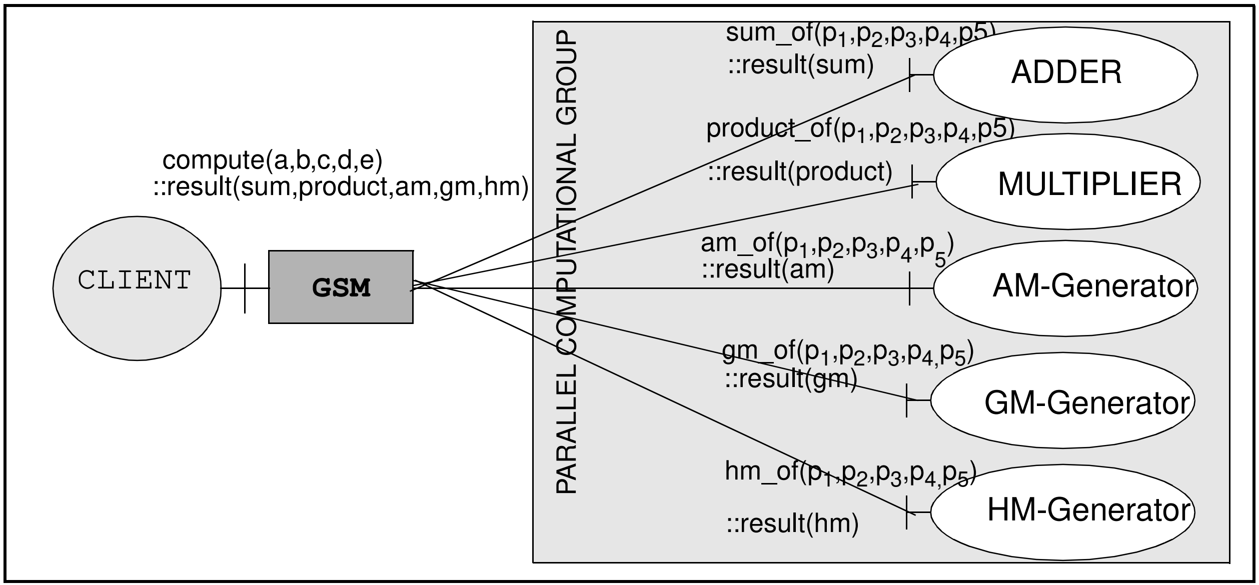


Fig. 7.5 A Parallel Computational Group

The composition of the computational group is unknown to the client. The client invokes a single generic service request: `compute(a,b,c,d,e)` on the group in order to obtain the sum, product, arithmetic-mean, geometric-mean, and the harmonic-mean of the five numbers. However this service request is recognized by different names by each member of the server group, as shown in figure 7.5. Hence the client's generic service request must be appropriately renamed before distribution. The client expects to receive a single reply in the following termination signature: `result(sum, product, am, gm, hm)`. However each member of the server group sends part of the reply, namely the `result(sum)`, `result(product)`, `result(am)`, `result(gm)`, and `result(hm)`. Therefore the partial replies sent by the members of the computational group must be assembled together into a single reply in the client's termination signature format.

Therefore the client can obtain the total service by invoking a single request on a group of heterogeneous servers and collating the received replies. The parallel computational group exemplifies a group coordination model which involves operation message renaming and distribution on the client side and reply collation on the client side. All these coordination patterns are rendered transparent to the client and server application by the GSM. This type of coordination model also implies that a client can bind to a server group if each group member provides partial service and the total service provided by individual members meets the clients requirements.

7.5.5 Renaming Requirements & Policy Specification

In Group Application-2, the client's generic service request must be renamed appropriately before distribution to the computational group, so that it can be recognised by the members of the computational group. The following renaming and distribution policy specification programs the D-Agent on the client side GSM to rename the message before distribution.

```
operation_distribution_policy
for compute
distribute compute(a,b,c,d,e)
transformed_as
              [
               renamed_message sum_of(p1,p2,p3,p4,p5)
               to ADDER
               using SOURCE_ORDERED_MULTICAST

               renamed_message product_of(p1,p2,p3,p4,p5)
               to MULTIPLIER
               using SOURCE_ORDERED_MULTICAST

               renamed_message am_of(p1,p2,p3,p4,p5)
               to AM-GENERATOR
               using SOURCE_ORDERED_MULTICAST

               renamed_message gm_of(p1,p2,p3,p4,p5)
               to GM-GENERATOR
               using SOURCE_ORDERED_MULTICAST

               renamed_message hm_of(p1,p2,p3,p4,p5)
               to HM-GENERATOR
               using SOURCE_ORDERED_MULTICAST
               ]
end_policy
```

Fig. 7.6 Renaming Policy Specification

The client is unaware of the existence of the multiple servers involved in the provision of the computational service and of the need to invoke individual servers by a different operation name.

7.6 Reply Collation and Delivery Models

The most common *group coordination model* is the binding of a *client object* to a *server group*. The server group could either be a homogeneous group or a heterogeneous group. In this type of coordination model, the client object *interrogates* a server group and it receives multiple replies, one from each member of the group. The question that arises is how does the client want these replies to be collated and the order in which the replies to be delivered to it. As shown in the examples in this section, the clients have their own requirements (or preferences) with respect to the collation of replies and the order in which those replies are delivered to them. The following issues arise with respect to the reply collation and its delivery to the client object.

1. *Reply collation based upon cardinality*: The reply collation policy implements the failure semantics of the group interrogation. It dictates how many servers must successfully reply in order for the group

interrogation to be considered successful. In some other cases, the client may want to receive fixed-size *group terminations*, separately, in order to ease the processing of the terminations.

2. *Reply collation based upon sender identity*: In some other applications, the client wants to receive the replies sent by some specified senders together in a single group termination, in order to separately analyse the replies received from different sender groups.
3. *Reply collation based upon reply type*: In case of heterogeneous server groups, different types of replies are received from the group in response to the client's operation invocation. In such applications the clients wish to receive all instances of replies of a given type together in a single group termination, in order to separately analyse different types of replies.

Table 7.1: Reply Collation and Delivery Schemes

	Collation and delivery of instances of single reply type to client
1.	multiple reply instances delivered as a single group termination: *matrix-mode collation*
2.	multiple reply instances delivered as a single group termination: *linear-mode collation*
3.	multiple reply instances delivered separately in any order
4.	multiple reply instances delivered separately in specific order
	Collation and delivery of instances of multiple reply types to client
5.	delivery of multiple instances of each reply type as separate group terminations in any order
6.	delivery of multiple instances of each reply type as separate group terminations in specified order
7.	delivery of multiple instances of each reply type as singleton terminations in any order
8.	delivery of multiple instances of each reply type as singleton terminations in specific order
	Special Cases
9.	discarding the rest of the reply types on the arrival of the specific reply type
10.	choosing between reply types

4. *Separate delivery of singleton replies*: Many clients wish to receive singleton replies as soon as they are received by their local GSM because they do not want to wait for a long time to receive all the replies as a single group reply. In these applications, the replies are processed as soon as they are received.
5. *Ordered reply delivery*: Some clients wish to receive and process the replies in a certain order. Sequential delivery of singleton is requested in the following cases:
   - *Pick the chosen few and discard the rest*: In some cases, the clients want to receive and process certain desired reply types or replies from certain important (crucial) members before others and if the replies already collected are sufficient, the client requests to terminate the delivery of the rest of the replies.
   - *Process all of them in a certain order*: When the client application requires processing of certain reply types or replies from certain senders in some sequential order, for example to bring a certain desired state change, etc.

   In such cases the clients pre-specify the order of reply delivery to their proxy object, the GSM (C-Agent), and the replies are delivered to the clients only when the reply delivery is explicitly solicited. This is typical of clients which have 'solicited reply reception capability'.
6. *Disabling of reply delivery*: In some cases, the client is interested in receiving a certain reply and if that reply type is received from any of the senders, it may require other reply types to be abandoned and only the desired one delivered to it. Similarly if an exception termination is reported by any of the senders, then the client may wish to abandon the rest of the replies.

These requirements are typical of group-oriented client applications. They dictate a sophisticated reply collation and delivery policy specification in order to program the C-Agent in the client's GSM about how and when to deliver the replies. The GPSL is capable of specifying these complex reply collation and delivery policies as shown in the examples below.

Some of the important reply collation and delivery schemes commonly required by group-oriented client applications are summarized in table 7.1 . In the following subsections we illustrate these scenarios through examples, together with the corresponding policy specification.

7.6.1 Group Application-3: Stock Inventory System

A stock inventory system is a good example of group-based application which exhibits different reply collation and delivery scenarios. The consumers, suppliers, and the inventory manager are the main components of an inventory system.

Every retail business (such as department store, grocery store, stationery store, etc.) has its own inventory system. Each retail business sells multiple types of merchandise. The inventory manager is responsible for keeping sufficient merchandise in the inventory storage. It must order the merchandise from the suppliers whenever the merchandise goes below a certain threshold level. Typically there are multiple suppliers for each type of merchandise. The suppliers of a particular merchandise are organised as a *supplier group*. Hence there are multiple supplier groups, one for each type of merchandise.

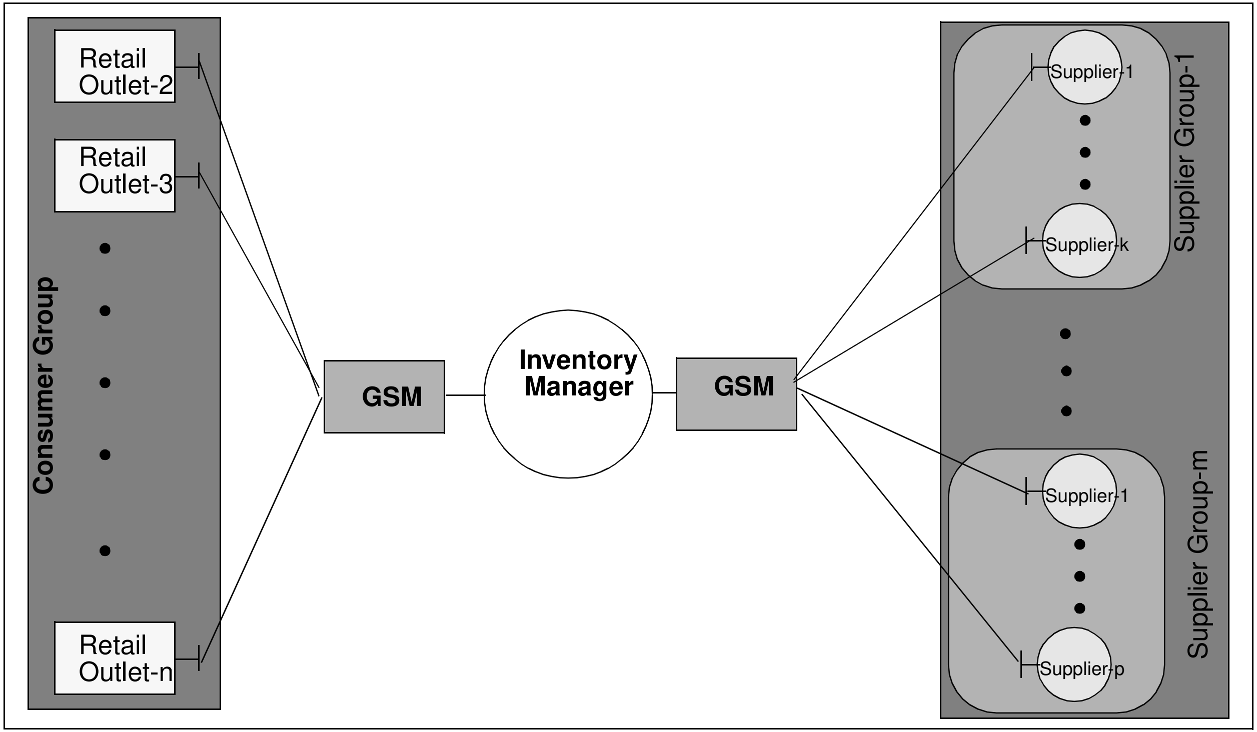


Fig. 7.7 Stock Inventory System

Consider the configuration of a distributed inventory system shown in figure 7.7. Every retail business has a set of retail outlets (for example check-out machines) through which products are sold to customers. These retail outlets act as consumers and they are organised as a *consumer group*. The inventory manager periodically checks the current levels of merchandise availability by sending the following service request (operation message) to the consumer group: `query_sale_status()`. On receipt of this request, each member of the consumer group responds with the following reply (termination message): `sale_status(merchandise-1, merchandise-2,....,merchandise-n)`, indicating the number of products of each type that have been sold since the previous query through a given retail outlet (check-out machine).

After receiving the current merchandise sale information from the consumer group, the inventory manager calculates the current merchandise availability for each merchandise type. If any merchandise falls below a certain threshold level in the inventory, the inventory manager must start the process of ordering the merchandise from the corresponding merchandise's supplier group.

The inventory manager starts with finding out how much quantity of a given merchandise each supplier can provide and consequently how many suppliers it should contact (or place an order) in order to meet its inventory requirements. So it invokes the following service request (operation message) "`query_merchandise_availability(merchandise_id)`" on the supplier group (acting as a server group) to find out their supply capability. Each member of the supplier group responds with the following reply to indicate the quantity of merchandise it can supply: `merchandise_availability(merchandise_id,quantity)`.

7.6.2 Delivery of Group Termination of a Single Reply Type: Matrix-Mode Collation

When a client interrogates a homogeneous server group, it receives multiple replies which are instances of the same reply type (i.e., termination signature), but they are not identical replies. The client wants to receive all the replies together in a single group termination so that it can analyse (or process) them together and make some application-specific decision based upon group reply. Moreover the client cannot start processing the individual replies until it has received all of them. In such cases, a single group termination is constructed (in either matrix or linear mode depending upon whether complete or partial reply instances are received) after all the replies are received by the C-Agent in the client's GSM. The previous example illustrates this as well as some other reply collation requirement.

7.6.2.1 Reply Collation Requirements & Policy Specification

In group application-3, the inventory manager cannot calculate the current merchandise level (merchandise availability) until it has received the 'sale status' information from all the members of the consumer group (check-out machines). So it wants to receive this information from all the members of the consumer group together through a single group termination. Since every member of the consumer group sends the sale status of each merchandise, the replies must be combined in matrix-mode (see section 3.6.1). Moreover the inventory manager does not want to wait an indefinite period of time for the delivery of group reply. Usually the 'sale status' replies are received immediately from the consumer group, so the inventory manager wants *any* number of replies received within 5 minutes of issuing the "`query_sale_status`" be combined into a group termination and delivered to it.

```
for query_sale_status
     [
      deliver sale_status(merchandise-1,................,merchandise-n)
      from consumer_Group
      within 5 minutes
      collation_cardinality UNSPECIFIED
      collation_mode MATRIX
     ]

end_policy
```

Fig. 7.8 Reply collation and delivery policy of a single group termination (matrix-mode)

The policy specification shown in figure 7.8 captures these requirements for the collation of replies. The C-Agent in the GSM of the inventory manager is programmed with this policy.

7.6.2.2 Transparency and Policy Interpretation

The policy specifies the collation of any number of the instances of the reply “`sale_status(merchandise-1,.,merchandise-n)`” received from the consumer group within 5 minutes of the issuing of the corresponding operation message “`query_sale_status`” in the matrix mode. The client (inventory manager) need not be concerned about collecting multiple and possibly variable number of replies, if the consumer group is transparent.

7.6.3 Delivery of Group Termination of a Single Reply Type: Linear-Mode Collation

In some cases a client object is transparently bound to a server group, as if it was bound to a singleton server. The client expects to receive a single reply as a singleton termination message. If each member of the server group gives partial reply (i.e., part of the termination signature), then the total reply must be constructed by assembling together partial replies using linear-mode collation (see section 3.6.2).

7.6.3.1 Reply Collation Requirements & Policy Specification

In group application-2, the client expects to receive a single reply in the following termination signature: `result(sum, product, am, gm, hm)`. However each member of the computational group sends part of the reply, namely the `result(sum)`, `result(product)`, `result(am)`, `result(gm)`, and `result(hm)`. Therefore these partial replies must be assembled together into a single reply in the client’s termination signature format, using linear mode collation. Moreover replies from all the members of the computational group must be received, otherwise the reply cannot be constructed (and an exception termination has to be sent to the client by the GSM). The client wants the complete reply to be delivered to it within, say 5 minutes, of the corresponding operation invocation, (otherwise an exception termination is sent to the client by the GSM).The following policy specification captures the above mentioned requirements for the collation of replies received from the computational group. The C-Agent in the GSM of the client is programmed with this policy.

```
termination_collation_policy
for compute
     [
     deliver result(sum, product, am, gm, hm)
     from computational_Group
     within 5 minutes
     collation_cardinality ATLEAST(ALL)
     collation_mode LINEAR
    ]
end_policy
```

Fig. 7.9 Reply collation and delivery policy of a single group termination (linear-mode)

7.6.3.2 Transparency & Policy Interpretation

The policy specifies to collate the partial instances of the termination signature “`result(sum, product, am, gm, hm)`” which are received from the computational group, within 5 minutes of the issuing of the corresponding operation message “`compute`”, in the linear mode, only if all inputs are received from all members of the computational group, otherwise an exception termination must be constructed by the C-Agent and delivered to the client object.

The client is unaware of the existence of the multiple servers involved in the provision of the computational service and of the need to combine partial replies sent by each member of the group.

7.6.4 Unordered Delivery of Singleton Terminations of a Reply Type

In some cases the client wants the replies from the server group to be delivered to it as soon as they are received by the GSM in order to avoid the delay associated with the reception of the group reply, particularly when the members of the server group are prone to sending late replies or when the client has immediate reply requirements. So singleton replies need to be delivered to the client separately in the order in which they arrive. In such cases the client is not necessarily interested in receiving all the replies. In particular, when the replies already delivered are sufficient for it to proceed, the client may request the reply delivery to be terminated (i.e., terminable reply delivery semantics). In order to control the delivery of replies, the client may use the 'polled reply delivery scheme', so that the GSM will deliver the reply only when client explicitly requests it, provided that the reply is available.

7.6.4.1 Unordered Reply Delivery Requirement and Policy Specification

In group application-3, the suppliers give their replies (i.e., merchandise availability information) to the inventory manager's query only when they have the product currently available, otherwise they delay their reply. The suppliers may also delay their replies due to some other considerations. However the inventory manager is interested in finding out the product availability as soon as possible (so that it can place an order). So the inventory manager is interested in receiving the replies from the supplier group as soon as they are received by its GSM. Once the inventory manager can obtain the required quantity of the merchandise from the suppliers whose replies it has analysed, it may request the delivery of other replies to be terminated. The inventory manager queries the supplier group periodically. The inventory manager puts a certain time limit on the receipt of replies, after which it will place an order with the suppliers who have responded to its query within the time limit. The following policy specification captures the above mentioned requirements for reply collation and delivery. The C-Agent of the inventory manager's GSM is programmed with this policy.

```
  termination_collation_policy
  for query_merchandise_availability
      [
      deliver merchandise_availability(merchandise_id,quantity)
      from supplier_Group
      within 60 minutes
      collation_cardinality UNSPECIFIED
      collation_mode SINGLETON(ANY-ORDER)
     ]
end_policy
```

Fig. 7.10 Unordered delivery of singleton terminations of a reply type

7.6.4.2 Transparency & Policy Interpretation

The policy specifies the delivery of the instances of reply "`merchandise_availability(merchandise_id,quantity)`", as singleton termination messages, to the inventory manager object in the order in which they are received from the supplier group, within 60 minutes of the corresponding operation invocation.

7.6.5 Ordered Delivery of Singleton Terminations of a Reply Type

The order of delivery of singleton replies to the client object is an important criterion in some applications. The client not only wants to receive individual replies, but it also wants some replies to be delivered before others, based upon the sender of the reply. This allows the client to pick and process the replies from some of its favored servers before others. When the replies already delivered are sufficient for it to proceed, the client may request the delivery of the replies from the rest of the servers to be abandoned.

7.6.5.1 Ordered Reply Delivery Requirement & Policy Specification

Consider the following variation in the requirement of group application-3. The suppliers of a given merchandise are distributed and located at different places. If the inventory manager wants to order the merchandise from the suppliers nearer to it than those farther from it, in order to reduce transportation cost or delivery times, then it would like to receive the replies (merchandise availability information) from the nearest vendors first. As soon as the required quantity of the product is available from the nearest suppliers, the inventory manager may not be interested in hearing from other suppliers. The following policy specification captures the above mentioned requirements for reply collation and delivery. The C-Agent of the inventory manager's GSM is programmed with this policy.

```
 termination_collation_policy
 for query_merchandise_availability
      [
      deliver merchandise_availability(merchandise_id,quantity)
      from supplier-3, supplier-2, supplier-5, supplier-1,supplier-4
      within 60 minutes
      collation_cardinality UNSPECIFIED
      collation_mode SINGLETON(ORDERED)
     ]
end_policy
```

Fig. 7.11 Ordered delivery of singleton terminations of a reply type

7.6.5.2 Transparency & Policy Interpretation

The policy specifies to the C-Agent to deliver the instances of reply "`merchandise_availability(merchandise_id,quantity)`", received from the suppliers listed in the "`from`" clause, as singleton termination messages, in the *order* in which the suppliers are listed in the "`from`" clause, within 60 minutes of the corresponding operation invocation.

7.6.6 Unordered Delivery of Multiple Reply Types as Singleton Terminations

When a client object invokes a heterogeneous server group, it not only receives multiple replies, but the received replies are of different types, i.e., instances of different termination signatures. A question that arises in such a case is how to collate and deliver instances of multiple reply types to the client object. As shown in the following examples, the client applications exhibit different requirements with respect to these options. One such option is the unordered delivery of multiple reply types as singleton terminations. In such a case the client application is characterised by the following requirements:

1. client is interested in receiving all reply types in any order
2. the client is interested in receiving instances of each reply type as singleton terminations so that they can be delivered to it as soon as they are received by GSM and hence processed immediately; when sufficient number of instances of a given reply type are received, the client may ignore the rest of the instances of that type.
3. the client may require the instances of a given reply type to be delivered to it in certain order in order to process the replies from its preferred servers earlier than the rest.

7.6.6.1 Reply Collation & Delivery Requirements and Policy Specification

Consider a specific example of stock inventory system (group application-3) in retail grocery business. In this case the inventory manager deals with different supplier groups, for example there is a *dairy group*, *bakery group*, *meat group*, *fruits group*, etc. If the policy of the grocery store is to order the required quantities of merchandise periodically, say every two days (in order to get the fresh supplies), then the inventory manager invokes the following service request (operation message): "`query_merchandise_availability(my_store_id)`"on all the suppliers groups to find out the quantity of merchandise in the respective domains each supplier can provide. The members of the supplier groups respond to this generic service request with the following different reply types, indicating the quantity of each product that they can supply:

Dairy group: `diary_availability(milk,cheese,butter,yougurt)`
Bakery group: `bakery_availability(white_bread,brown_bread,muffins)`
Meat group: `meat_availability(lamb,beef,chicken)`
Fruit group: `fruit_availability(apple, banana, orange, strawberry)`

The inventory manager has a requirement of specific quantities of each merchandise type and is interested in placing the purchase order with the nearest suppliers. Therefore the inventory manager wants the replies from each supplier group to be delivered to it sequentially starting with the nearest supplier to the farthest one, and as soon as the required quantity of product is achievable from the nearest suppliers whose replies it has processed first, the inventory manager is not interested in replies from the rest of the suppliers in that group and may ignore those replies.

```
termination_collation_policy
for query_merchandise_availability
     [
     deliver diary_availability(milk, cheese, butter, yogurt)
     from diary_Supplier-3,diary_Supplier-2,diary_Supplier-4,
               diary_Supplier-1,diary_Supplier-4
     within 50 minutes
     collation_cardinality UNSPECIFIED
     collation_mode SINGLETON(ORDERED)
    ]
    interleaved_with
    [
     deliver bakery_availability(white_bread, brown_bread, muffins)
     from bakery_Supplier-3,bakery_Supplier-2,bakery_Supplier-5,
               bakery_Supplier-1,bakery_Supplier-4,bakery_Supplier-6
     within 50 minutes
     collation_cardinality UNSPECIFIED
     collation_mode SINGLETON(ORDERED)
    ]
    interleaved_with
    [
     deliver meat_availability(lamb, beef, chicken)
     from meat_Supplier-3,meat_Supplier-2,meat_Supplier-1
     within 50 minutes
     collation_cardinality UNSPECIFIED
     collation_mode SINGLETON(ORDERED)
    ]
    interleaved_with
    [
     deliver fruit_availability(apple, banana, orange, strawberry)
     from fruit_Supplier-3,fruit_Supplier-2,fruit_Supplier-4,fruit_Supplier-1
     within 50 minutes
    collation_cardinality UNSPECIFIED
    collation_mode SINGLETON(ORDERED)
   ]
end_policy
```

Fig. 7.12 Policy Specification for interleaved delivery of instances of multiple reply types

In this example, the inventory manager is interested in receiving all the reply types. Moreover it want replies from a given supplier group delivered to it in a certain order (in the order of the proximity of the supplier). However replies from different supplier groups can be interleaved in any order. The inventory manager can accept the replies received within 50 minutes of corresponding query for merchandise availability. The policy specification shown in figure 7.12 captures the above mentioned requirements. The C-Agent of the inventory manager's GSM is programmed with this policy.

7.6.6.2 Transparency & Policy Interpretation

The policy specifies the delivery of the instances of each reply type as singleton termination messages, to the client (inventory manager) object in the order in which the suppliers are listed in the corresponding "`from`" clause, within 50 minutes of the corresponding operation invocation. The suppliers are listed in the "`from`" in the order of their proximity to the grocery store. The instances of different reply types may be interleaved and delivered in any order.

7.6.7 Unordered Delivery of Multiple Reply Types as Group Terminations

In some 'multiple reply type' client applications, the client cannot make an application-specific decision (i.e., it cannot proceed further) until it has received all instances of each reply type. In such cases it is better to collate instances of each reply type into group terminations and to deliver multiple group terminations, one for each reply type, to the client object. These group terminations can be delivered to the client in any order, as soon as the all the instances of a reply type are received by the GSM, because the client application wants to receive all the reply types and is unaffected by the order of delivery of those reply types.

```
termination_collation_policy
for query_merchandise_availability
     [
     deliver     dairy_availability(milk_quantity,milk_price,cheese_quantity,
                cheese_price,butter_quantity,butter_price)
     from diary_Supplier_Group
     within 50 minutes
     collation_cardinality UNSPECIFIED
     collation_mode MATRIX
    ]
    interleaved_with
    [
     deliver       bakery_availability(white_bread_quantity,white_bread_price,
                                   muffins_quantity,muffins_price)
     from bakery_Supplier_Group
     within 50 minutes
     collation_cardinality UNSPECIFIED
     collation_mode MATRIX
    ]
    interleaved_with
    [
    deliver meat_availability(lamb_quantity,lamb_price,beef_quantity,beef_price)
    from meat_Supplier_Group
    within 50 minutes
    collation_cardinality UNSPECIFIED
    collation_mode MATRIX
   ]
   interleaved_with
   [
    deliver      fruit_availability(apple_quantity,apple_price,orange_quantity,
                                  orangeprice,banana_quantity,banana_price)
    from fruit_Supplier_Group
    within 50 minutes
    collation_cardinality UNSPECIFIED
    collation_mode MATRIX
   ]
end_policy
```

Fig. 7.13 Policy specification for Unordered Delivery of Multiple Reply Types as Group Terminations

7.6.7.1 Reply Collation & Delivery Requirement and Policy Specification

Consider a slight variation in requirement of the previous example. If the inventory manager makes a purchase decision based upon the cost of the product rather than the proximity of the product supplier, then the inventory manager must receive all the replies from a supplier group to find out which supplier sells a given product at the lowest price. In this case, as soon as all the replies from a given supplier group are received by the GSM, a group termination must be constructed and delivered to the inventory manager. Here it is convenient for the inventory manager to process a single group reply. The group terminations from different supplier groups can be delivered in any order. In this case each supplier not only provides the information about the quantity of the product that it can supply, but also the price of the product in its reply to the inventory manager's query for product availability. The policy specification shown in figure 7.13 captures the above mentioned requirements for reply collation and delivery. The C-Agent of the inventory manager's GSM is programmed with this policy.

7.6.7.2 Transparency & Policy Interpretation

The policy in figure 7.13 specifies to the C-Agent to deliver all instances of each reply type as group termination messages as soon as all replies from a given supplier group are received. The instances of different reply types may be interleaved and delivered in any order.

7.6.8 Ordered Delivery Multiple Reply Types as Singleton Terminations

In some 'multiple reply type' client applications, the clients have preference for certain types of replies. They wish to receive certain reply types before others and when sufficient number of the desired reply types are received, they may terminate the flow of rest of the replies. In such applications the order of reply delivery is based upon the 'type of the reply'. In some applications, the client also knows how many instances of a given reply type to expect and who sends those reply types.

7.6.8.1 Reply Collation & Delivery Requirement and Policy Specification

Consider another instance of a stock inventory system (group application-3) in a department store. We illustrate the requirement of ordered delivery of multiple reply types as singleton terminations through a simple example. A department store, amongst many other merchandise, carries winter jackets. These jackets come in different colors. The customers show a certain preference for the color, say green, blue, and red in the order.

So the inventory manager wishes to keep as many green (and then blue) jackets in stock as available from the suppliers. There are multiple suppliers of winter jackets and each supplier provides a specific-colored jacket. During the winter season, the inventory manager wishes to keep his stock of jackets at a certain threshold level, so it periodically sends the following query to the supplier group to find out the type (color) of the jacket and the quantities available from each of the supplier:
"`query_merchandise_availability(my_store_id)`".
On receipt of this query, the jacket suppliers respond with one of the following replies:

```
green_jacket(supplier_id,quantity,date_available,cost_per_piece)
blue_jacket(supplier_id,quantity,date_available,cost_per_piece)
red_jacket(supplier_id,quantity,date_available,cost_per_piece)
```

If the inventory manager can get the required quantities of green jackets, then it is not interested in receiving other types of replies. If the required quantity of green jackets is not available, the it wishes to receive the replies of the blue jacket suppliers, etc. Moreover, the inventory manager has a certain time limit associated with receiving the replies, after which it will order what ever is available from the suppliers who have responded to its query. The following policy specification captures these requirements.

```
termination_collation_policy
for query_merchandise_availability

    [
     deliver  green_jacket(supplier_id,quantity,date_available,cost_per_piece)
     from green_jacket_supplier_Group
     within 60 minutes
     collation_cardinality UNSPECIFIED
     collation_mode SINGLETON
    ]
    followed_by
    [
     deliver  blue_jacket(supplier_id,quantity,date_available,cost_per_piece)
     from blue_jacket_Supplier_Group
     within 60 minutes
     collation_cardinality UNSPECIFIED
     collation_mode SINGLETON
    ]
    followed_by
    [
     deliver    red_jacket(supplier_id,quantity,date_available,cost_per_piece)
     from red_jacket_Supplier_Group
     within 60 minutes
     collation_cardinality UNSPECIFIED
     collation_mode SINGLETON
    ]
end_policy
```

Fig. 7.14 Policy Specification for Ordered Delivery of Multiple Reply Types as Singleton Terminations

7.6.8.2 Transparency & Policy Interpretation

The policy in figure 7.14 specifies to the C-Agent to deliver instances of each reply type as singleton terminations. The instances of the reply "`blue_jacket()`" are delivered to the client only when all the instances of the reply "`green_jacket()`" are received from its supplier group or after the expiry of its time out period, whichever occurs first. Similarly, the instances of the reply "`red_jacket()`" are delivered only when all the instances of the reply "`blue_jacket()`" are received from its supplier group or after the expiry of its time out period.

7.6.9 Disabling the Delivery of Other Reply Types by a Preferred Reply Type

Although a client object may receive multiple reply types from the server group, in some applications, the client is interested in only one specific reply type and if this reply type is generated by any one member of the server group, all other reply types are of no significance to the client. If this desired (or preferred) reply type is received, then all other reply types need to be discarded and only the desired reply type delivered to the client. In the absence of the receipt of the desired reply type the client may be interested in receiving the other replies. The following example demonstrates this requirement.

7.6.9.1 Group Application-4: Mobile Telecommunications

Mobile telecommunications gives the subscriber the freedom of mobility. It gives the subscriber the ability to receive and initiate phone calls on his or her terminal anywhere in the 'mobility domain' and be charged to his home phone account. The subscriber (or terminal) is registered in one administrative domain (a country or a telecommunications company). The subscriber (terminal) registration service within an administrative domain is usually implemented as a distributed name service in which the global name space (subscriber identification information) is partitioned, and a different name server maintains each partition. In case of mobile telecommunications, the terminal registration service is distributed between multiple Home Location Registers (HLRs) located in different cities and organised as a single logical service, called an ‘HLR Group’. A subscriber (terminal) is registered in one of the HLRs.

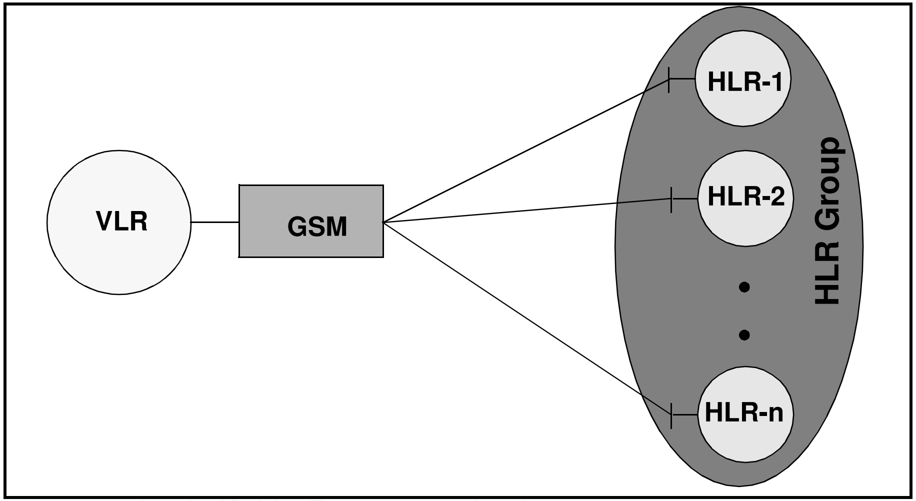


Fig. 7.15 Group Interrogation in Mobile Telecommunications

When the user visits a foreign administrative domain and wishes to initiate or receive calls, he must register with the Visitor Location Register (VLR) of the visited domain. The VLR groupcasts an authentication request, `authenticate_terminal(terminal_id)`, to the HLR Group of the users domain. If the user is registered, only one of the members of the HLR group responds positively with the reply `registered_terminal(terminal_id)`, while all others respond negatively with the reply `unregistered_terminal(terminal_id)`. If the user is unregistered, then all HLRs respond with `unregistered_terminal(terminal_id)`.

7.6.9.2 Reply Collation & Deliver Requirement and Policy Specification

In group application-4, the VLR is interested in receiving a `registered_terminal()` reply from any one of the members of the HLR group. If this reply is received from any one of the HLRs, then all other replies are of no significance to the VLR (client). If the terminal is not registered in any HLR, then one `unregistered_terminal()` reply is sufficient, say the most recent one. Since the VLR cannot wait indefinitely, a time limit has to be imposed for the acceptance of replies. If reply from any one of the HLR group members is not received within the specified time limit, then the VLR should receive an exception from its GSM indicating the receipt of insufficient replies within the specified time limit.

It is irrelevant to the client (VLR) which name server (HLR) responds. The policy specification in figure 7.16 captures the above mentioned reply collation and delivery requirements. The C-Agent of the VLR’s GSM is programmed with this policy.

7.6.9.3 Transparency & Policy Interpretation

It may be noted that if the “`registered_terminal(terminal_id)`”reply is received from any one member of the HLR group, then delivery of the rest of the replies is disabled, and this reply is delivered to the client. However in order to deliver the “`unregistered_terminal(terminal_id)`” reply, the GSM must wait until the receipt of this type of reply from all members of the HLR group. Since one reply of this type is sufficient, only one reply (the most recent one) is picked out of the many “`unregistered_terminal(terminal_id)`” replies for delivery to the client (VLR).

```
termination_collation_policy
for authenticate_terminal
     [
      deliver unregistered_terminal(terminal_id)
      from HLR_Group
      within 900 msec
      collation_cardinality ATLEAST(ALL)
      collation_mode MATRIX(RECENT)
     ]
    disabled_by
     [
     deliver registered_terminal(terminal_id)
     from HLR_Group
     within 900 msec
     collation_cardinality ATMOST(1)
     collation_mode SINGLETON
    ]
end_policy
```

Fig. 7.16 Policy Specification for Disabling the Delivery of Other Reply Types by a Preferred Reply Type

7.6.10 Choice between Multiple Reply Types

In some applications, the clients may show choice between reply types based upon some criterion such as the number of the received reply instances, for example to know what do the majority of the interrogants say. In other cases this choice is based upon the source of the reply type, for example to know what do specific members of the interrogated group have to say. The following example demonstrates this requirement.

7.6.10.1 Group Application-5: Group Survey

Group survey is a common practice in many application domains. A *surveyor object* wishes to collect the opinion of the members of the *surveyed group* on a particular subject or topic. The surveyor object broadcasts a “`query(topic)`” message to the surveyed group and waits for their reply. In such applications a binary reply is expected from the surveyed group. It could be a `Yes()` or `No()` reply or an `approved()` or `unapproved_reply()`, etc. In the simplest case the surveyor object is interested in the reply returned by the majority of the surveyed group members. In some other cases, the surveyor object is interested in all the received reply types and the number of instances of those reply types.

```
termination_collation_policy
for query
     [
      deliver Yes(member_id)
      from Surveyed_Group
      within 1 day
      collation_cardinality ATLEAST(MAJORITY),ATMOST(ALL)
      collation_mode MATRIX
     ]
    choice
    [
     deliver No(member_id)
     from Surveyed_Group
     within 1 day
     collation_cardinality ATLEAST(MAJORITY),ATMOST(ALL)
     collation_mode MATRIX
    ]
end_policy
```

Fig. 7.17 Policy Specification for Choosing between reply types based upon cardinality requirements

7.6.10.2 Reply Collation & Deliver Requirement and Policy Specification

In group application-5, the surveyor object is interested in receiving only the reply type which is returned by majority of the surveyed group members. Moreover, it wants to know the identities of these members. The other reply types should be discarded. The policy specification figure 7.17 captures the above mentioned reply collation and delivery requirements. The C-Agent of the surveyor's GSM is programmed with this policy.

7.6.10.3 Transparency & Policy Interpretation

The choice operator in figure 7.17 explores both the sides simultaneously and delivers only one of the message for which the 'cardinality' (or collation time out) clause is satisfied first and drops the other.

7.6.10.4 Group Application-6: Scheduling Group Meeting

Scheduling group meetings is another common requirement in many organisations. A *secretary* is responsible for scheduling these meetings. The secretary sends a query "`are_you_available(day,time)`" to the members of the group to find out their availability on a particular day and time. The members respond with "`Yes(member_id)`" or "`No(member_id)`" replies. Usually there are some minimum membership requirements for scheduling meetings, such as a meeting cannot be scheduled without the presence of at least three managers and at least five staff members.

7.6.10.5 Reply Collation & Deliver Requirement and Policy Specification

In group application-6, a meeting cannot be scheduled without the presence of at least three managers and at least 5 staff members. Moreover, the secretary wants to know the identities of all members who are willing to attend the meeting. If this condition cannot be satisfied, the secretary wants to get a single 'No' reply.

The following policy specification captures the above mentioned reply collation and delivery requirements. The C-Agent of the secretary's GSM is programmed with this policy.

```
termination_collation_policy
for are_you_available
     [
     deliver No()
     from MANAGER_Group, STAFF_Group
     within 1 day
     collation_cardinality UNSPECIFIED
     collation_mode MATRIX(RECENT)
    ]
    disabled_by
    [
    deliver Yes(member_id)
    from MANAGER_Group, STAFF_Group
    within 1 day
    collation_cardinality ATLEAST(ANY(3,MANAGER),ANY(5,STAFF)),
                          ATMOST(ALL)
    collation_mode MATRIX
   ]
end_policy
```

Fig. 7.18 Policy Specification for Choosing between reply types based upon sender identity

7.6.10.6 Transparency & Policy Interpretation

This policy specification delivers one of the two reply types to the client based upon the above mentioned rule. Moreover a single "No" reply (the most recent one) is sent if the above condition is not satisfied.

7.7 ‘Group-Service’ Request Models: Service Request Collation Models

The basic client-server model is a ‘*single request - single reply*’ model. Some group-based applications give rise to ‘*group-service request - multiple reply models*’, as shown in the examples below. These type of applications are examples of “*client group - singleton server*” coordination models in which a set of clients which are *related* to each other in an application-specific manner and which require the same type of service are organised as a *client group* and are placed under the supervision or management or service of a single (supervisor or manager or) *server object.* Such coordination models are typically characterized by the following properties (see section 2.2.2.4):

1. Multiple instances of the same service request (operation signature) or instances of different parts of the same service request are invoked from client group to server object.
2. Service requests are invoked *periodically* from client group to server object.
3. Server’s reply is based upon the group-input (*group service request*).

These type of applications require that the individual service requests of the same type from the members of the client group be *combined* together into a single “*group-service request*” which is invoked on the server object. The server object now has access to the total group service request through a single operation invocation. The server object can then analyse or process the total group input together and give its reply (or replies) to the client group members based upon the group input. The subsequent activities (or behavior) of the client group members is modified upon the receipt of the reply from the server object. Hence these types of coordination models are characterised by the following additional property: “the subsequent activities (or behavior) of each client is dependent upon and | or biased by the previous activities (or behavior) of the rest of the members of the client group”.

This type of group coordination model involves the *collation* of clients service requests into a group service request which is invoked on the server object and the *distribution* of multiple replies generated by the server object (in response to the group service request) to the members of the client group. There are many applications which exhibit this type of group coordination model, such as network management systems, land or mobile traffic control systems, process control applications, etc.

7.7.1 Group Application-7: Network Management Application

Client groups offer a convenient solution to the problem of *organising* a set of related objects under the management (or control) of a manager object and for *disseminating* monitoring and management information in distributed applications. Network management systems exhibit two prominent characteristics of client group-based applications:

1. *Client group models*: The two main entities of the network management systems are the *manager objects* and the *managed objects*. A group of related managed objects in a certain geographic area or in a particular administrative domain are organised together as a single logical entity which is managed by a manager object. Each managed object sends its partial (and local) status information to the manager object and expects to receive a management signal from it based upon the total group input. Here we have a case of “client group” interacting with a single server object.
2. *Global decision models*: Network management systems are based upon global decision models which rely on network-wide status information for management function. Examples of global decision mechanisms include routing algorithms that compute routes based upon network-wide traffic conditions such as link failures, congestion conditions, etc. Network managers require global network input before they can make any management decisions in order to optimise network-wide performance characteristics.

Consider a simple example of telecommunication network management. A telecommunication network consists of a set of switches connected together by communication lines. A set of *switches* in a particular geographic area (such as a city) are placed under the management of a *traffic manager* which is responsible for managing switches and for maintaining balanced traffic conditions on the network links.

The set of switches is organised as a *managed-object group* under the management of the *traffic manager object*. As shown in figure 7.19, each switch is represented by a managed object (MO), a software entity, which periodically reports its status information to the traffic manager (in the form of an operation message) and expects a reply, a management command, from the traffic manager. The managed object group represents a *client group* of the traffic manager object which acts as a *server object*.

The traffic manager object expects to receive the combined status reports of all the switches (MOs) in its domain in order to evaluate the overall network condition. It is expected to give a management command, in response to the group status report, to the individual MOs in the form of replies to the MOs. The reply of the traffic manager is based upon the group input and could be different for individual switches. The replies suggest a change in the traffic routing plan or no modification to the existing routing table. Each switch periodically sends the following operation message to the traffic manager:

`my_status(buffer_space,throughput,delay,link_condition)`, and expects to receive one of the following termination messages from it:

```
route_traffic (in_link, out_link)
status_OK()
```

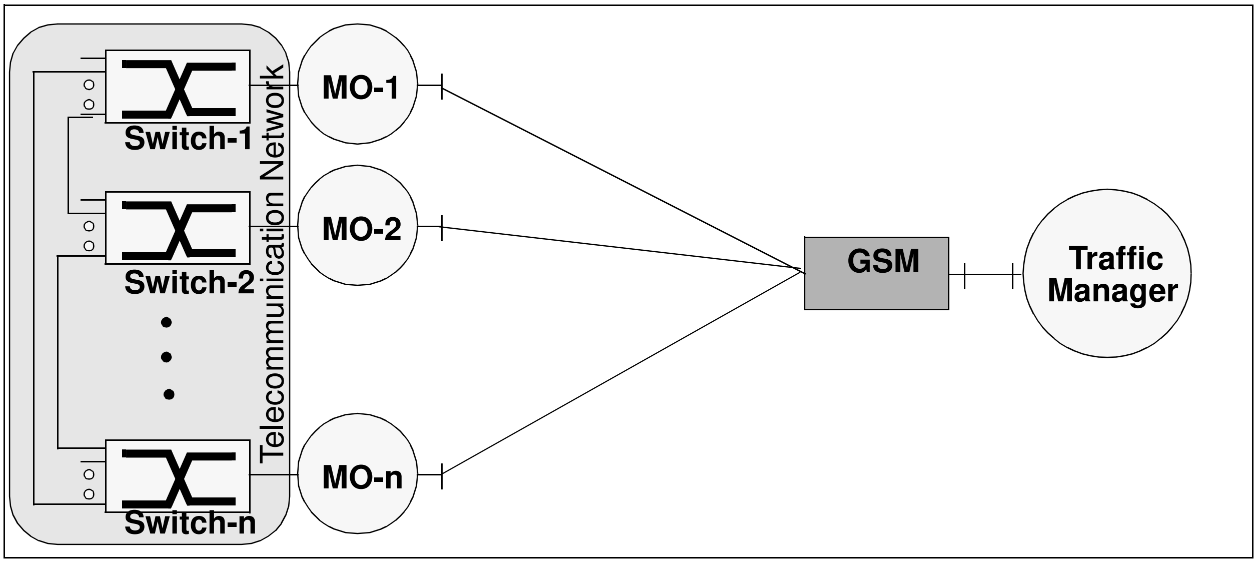


Fig. 7.19 Group Interrogation in Telecommunications Network Management

7.7.2 Constructing a ‘Group-Service’ Request: Matrix-Mode Collation & Policy Specification

The next question is how to combine individual service requests (operation message) from the client group into a group-service request (group operation message). In section 3.6, we have described two basic collation schemes: the *matrix-mode* and the *linear-mode* collation schemes, as simple and straightforward message grouping mechanisms. The matrix-mode collation scheme is applicable in cases in which each member of the client group sends a complete instance of the operation signature. In group application-7, each member of the managed-object group sends a complete instance of the “`my_status(buffer_space,throughput,delay,link_condition)`” operation. Now let us examine the other requirements of message collation.

In group application-3, the managed-object group sends status report to the traffic manager object every 60 minutes. The traffic manager wants to receive the combined status report from all the MOs as a single group operation invocation. In this example, the managed object group consists of 11 members. The traffic manager wants status inputs from at least 7 managed objects, otherwise the group input is not sufficient for the traffic manager to make any decision about the global network traffic condition. If multiple status reports are received from the managed object during a collation period, then the traffic manager wants the most *recent* input to be included in the group operation message, because it conveys the latest status information. The following policy specification, captures these requirements. The C-Agent in the GSM of the traffic manager is programmed with this policy.

```
 operation_collation_policy
 for my_status
      [
      deliver       my_status(buffer_space,throughput,delay,link_condition)
      from switch_Group
      every 60 minutes
      collation_cardinality ATLEAST(7), ATMOST(11)
      collation_mode MATRIX(RECENT)
     ]
end_policy
```

Fig. 7.20 Operation Collation Policy Specification

7.7.3 Transparency & Policy Interpretation

The above policy specification programs the C-Agent to collect the instances of the message specified in the "`deliver-clause`" which are received from the sources specified in the "`from-clause`" during the time duration specified in the "`every-clause`". If the required minimum number of messages are received within the specified collation period, then the C-Agent constructs the group message in the specified collation mode, otherwise an exception termination is constructed and sent to the client group members. The group operation message is invoked on the traffic manager (server) object at the end of the collation period.

7.7.4 Group Application-8: Target Location Acquisition Sonar System

The military sonar system, which is used to locate underwater targets such as submarines, etc., is another variation of above mentioned *group coordination model* in which related client objects, organised as a client group, send their inputs, as operation messages, to the server object, and each of them is *dependent* upon the cumulative server's reply (which is based upon group input) to position their 'firing stations' on the target. The difference in this case being that each member of the group sends *partial input* (partial operation signature) to the server object, but each of them is interested in the complete reply (the complete termination signature) from it. The following is a simplified version of the sonar system.

The sonar system consists of a set of sonar stations which are placed at strategic locations on the sea surface or on the sea shore, so as to scan a wide-volume of sea water. Each sonar station is equipped with a 'firing device', which when given the location of the target can point and fire at it. The sonar stations operate using the sound-waves technology. They are capable of giving the distance of the target object from the sonar and an approximate direction, but not the actual location (i.e., the 'x', 'y', and 'z' coordinates) of the target. Obviously a single sonar is not sufficient for target location acquisition. Hence a minimum of three sonars are employed to obtain sufficient information (i.e., target distance from each of them) to calculate the target location (target's coordinates) in the sea.

As shown in figure 7.21, the set of sonar stations is organised as a client group under the guidance of an off-shore 'target tracking system'. Each sonar station sends the distance of the target from itself to the tracking system, in the form of an operation message: "`target_distance(D)`". The existence of the sonar group is transparent to the tracking system. The tracking system expects to receive the following input, as an operation message from the sonar group: `target_distance(D1, D2, D3)`.

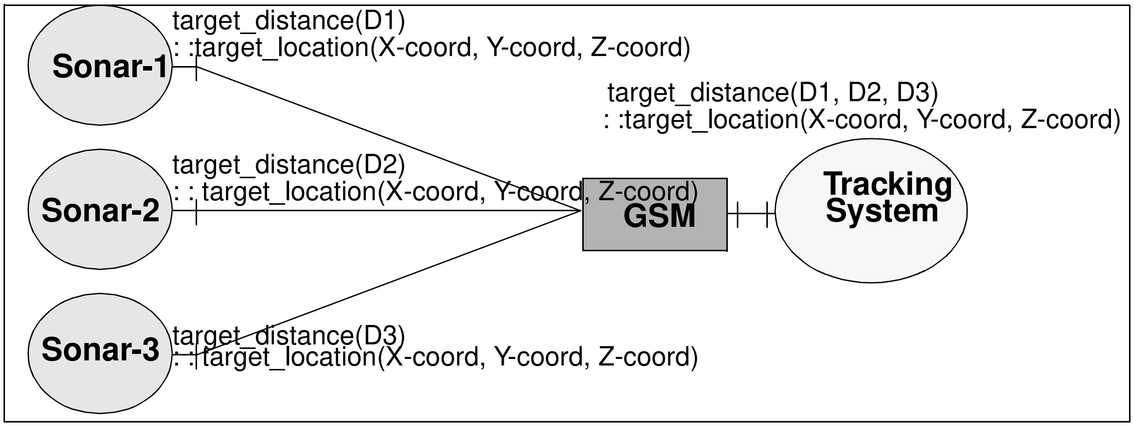

Sonar-1
target_distance(D1)
: :target_location(X-coord, Y-coord, Z-coord)
Sonar-2
target_distance(D2)
: : target_location(X-coord, Y-coord, Z-coord)
Sonar-3
target_distance(D3)
: :target_location(X-coord, Y-coord, Z-coord)
GSM
target_distance(D1, D2, D3)
: :target_location(X-coord, Y-coord, Z-coord)
Tracking System

Fig. 7.21 Group Interrogation in Sonar System

The tracking system functions as a server object, because it computes the location (i.e., coordinates) of the target based upon the client group input and its own knowledge about the location of the individual sonars and sends the target coordinates back to each sonar in the form of a termination message: "`target_location(X-coord,Y-coord,Z-coord)`". The sonars feed this information to their local firing devices, which use it to point at the target and fire at it.

This application represents a tightly-coupled and non-scalable application. The tracking system is tightly coupled to the three member sonar station group. The tracking system can only accept inputs from these three members. Any change in the client group membership cannot be accommodated by the server object. Moreover the client inputs, although of the same type, must be bound to the corresponding parameters of the server's operation signature.

7.7.5 Constructing a Service Request from Partial Service Requests: Linear-Mode Collation & Policy Specification

Group Application-8 is an example of client-group-based application in which each client gives partial input (partial service request) to the server object. The complete service request is constructed, on the server side, by combining the partial service requests using the linear-mode collation scheme. Moreover inputs from all the sonar stations are required, otherwise the target coordinates cannot be calculated. If each sonar station sends the target distance, say every 3 minutes, then the following policy specification captures these requirements. The C-Agent in the GSM of the tracking system is programmed with this policy.

```
 operation_collation_policy
 for target_distance
      [
       deliver target_distance(D1, D2, D3)
       from sonar_station_Group
       every 3 minutes
       collation_cardinality ATLEAST(ALL)
       collation_mode LINEAR
       ]
end_policy
```

Fig. 7.22 Linear-Mode Collation of partial service requests

7.8 Replies to Group-Service Request: Reply Distribution Models

As mentioned in section 4.2.3, the *group-oriented server* may generate either a single reply or multiple replies, one for each member of the client group, in response to the group-service request (i.e., group operation message). In case of a single reply, it is meant for all the members of the client group and hence, a copy of it is sent to each member of the group. In case of multiple replies, the order of reply generation is based upon a local protocol between the server and its GSM. Hence the GSM (or the D-Agent) knows which reply is to be sent to which client.

7.8.1 Multiple Replies to Group-Service Request

In group application-7, the traffic manager object, which is a *group-oriented server object*, gives multiple replies, one for each managed object, in response to the group operation message. These replies are generated by the traffic manager in the order in which the component operation messages, of each managed object, were arranged in the corresponding group operation message. The D-Agent must send the replies only to those managed objects whose component operation messages were included in the corresponding group operation message (i.e., only to those clients who have actually requested the service and not to all members of the client group). The following distribution policy specification programs the D-Agent to send the nth reply received from the traffic manager to the managed object whose operation message was placed in the nth row in the corresponding group operation message.

```
 termination_distribution_policy
 for my_status
     distribute _REPLIES_
     to SENDERS_IN_ROW_ORDER
     using UNORDERED_MULTICAST
end_policy
```

Fig. 7.23 Multiple Replies Distribution Policy

7.8.2 Transparency & Policy Interpretation

The above policy specifies that the replies received in response to the group operation specified in the "`for` clause" be distributed to the senders of the component messages of the group operation. The 'n_{th}' reply is sent to the sender of the 'n_{th}' component of the group operation message.

7.8.3 Single Reply to Group-Service Request

In group application-8, a single reply, "`target_location(X-coord,Y-coord,Z-coord)`", is received from the tracking system in response to the group operation message, "`target_distance(D1,D2,D3)`".This reply is to be sent to all the sonar stations, because their component operation messages were included in the corresponding group operation message. The following policy specification programs the D-Agent of the tracking system to send a copy of the reply to all the 'senders' of the component operation messages of the corresponding group operation message.

```
termination_distribution_policy
for target_distance
    distribute _REPLY_
    to SENDERS
    using UNORDERED_MULTICAST
end_policy
```

Fig. 7.24 Single Reply Distribution Policy

7.9 Synchronised Invocation Model

The most general group coordination model is the binding of a client group to a server group. Many coordination behaviors can be seen in this coordination model. In this section we discuss a specific class of coordination behavior which is the characteristic feature of applications in which a group of clients interrogates a server group in a *synchronised* manner.

Whenever a client group, instead of a singleton client, interrogates a server group, a basic question that arises is how to coordinate the invocation of multiple service requests (operation messages) from the members of the client group. In section 7.7, we described one method of synchronising the invocation of multiple service requests from a client group on a server object. In this case, each member of the client group periodically invokes (operation | notification) messages of the same type (i.e., instances of the same message signature), and the synchronisation is achieved by *collating* the message instances of the same type, invoked during a given periodic interval, into a group service request. In this section we describe another method of synchronising the invocation of multiple service requests from a client group. In this case, each member of the client group invokes (operation | notification) messages of different type, and the synchronisation is achieved by scheduling these messages in a certain sequential or parallel order in order to give a mutually exclusive access of the server group to the client group members, so as to bring a desired state change in the members of the server group. In this case the distribution of messages from the members of the client group are coordinated based upon some synchronisation policy.

The invocation of (operation | notification) messages from the client group in an arbitrary order on the server group can lead to inconsistent or otherwise undesirable state changes in the members of the server group. Synchronised invocation of messages from the client group is the solution. There are numerous invocation synchronisation behaviors corresponding to different application requirements resulting in a wide variety of synchronisation policies.

7.9.1 Why Synchronised Invocation in the Client Group

The distribution of (operation | notification) messages from the client group members is synchronised in “coordinated client group applications” due to many application requirements. In this section we list the major ones:

1. *To obtain quorum* (*permission*) *of superior roles before message distribution*: In some client-group based applications, the group members have different roles, and subordinate roles are required to seek the quorum or permission of superior roles before they send their service requests or information notifications to the server group. The quorum is provided based upon the contents of the message or the source and destination of the message or a combination of all of them.
2. *To grant fair access to the server group to the client group members*: In some applications the clients want a mutually exclusive access to the services of the server group. In such cases a round robin or a prioritised access policy may be employed in order to schedule the distribution of the service requests from the members of the client group.
3. *To bring a desired state change in the server group members through synchronised message invocations*: In some client-group based applications, members of the client group need to invoke messages on the server group in some synchronised way in order to
   a. gain mutually exclusive access to the server group, and to
   b. bring certain application-specific state change in the server group, through the coordinated invocation (or distribution) of messages from the client group members.

In such applications, a client cannot invoke an (operation | notification) message on the server group until some previous message(s) have been invoked by some specific member(s) of the client group, and a confirmation received by them that the desired state change has occurred in the server group. Hence a message is not scheduled for distribution by the GSM until a '*synchronisation message*' is received from other GSM(s), signalling successful delivery and | or execution of the previous messages.

There are many aspects of *synchronisation policy* that need to be programmed in the S-Agent of the GSM in order to specify the different message synchronisation requirements of the applications. An example of synchronisation policy is shown in section 7.9.6.

7.9.2 What are Synchronisation Events in Client Groups

In a synchronised invocation model, an (operation | notification) message invoked by a client on its GSM is not distributed to the server group by the GSM until the GSM either explicitly seeks the permission of other members of the client group or it (i.e., the GSM) is implicitly informed when an event of some significance to the application occurs in other members of the client group (i.e., given the permission to invoke the message). In the latter case, the event of significance is called a *synchronisation event*, and the message that conveys the occurrence of this event to other members of the group is called a *synchronisation message*.

The objective of the synchronised invocation model is to give a coordinated or mutually exclusive access of the server group to the client group members. This means that whenever a client relinquishes its use of the server group, it (or its GSM) must inform the other members (or their GSMs) so that the next message can be invoked (distributed) on the server group. The question that arises is what information is required by the client (or its GSM) to know that its message has been delivered to the server group and the state of the server group has been appropriately modified. The receipt of this information constitutes the synchronisation event which triggers a synchronisation message. This information is application specific and the nature of this information varies with application requirements. In the following section we list some events which are considered as synchronisation events by the "coordinated client group applications":

1. *Receipt of message delivery confirmation from the GSM*: In some applications it is sufficient to know that a (operation | notification) message has been delivered to all the server group members. As soon as this confirmation is received, a synchronisation message may be sent to the other members of the client group (so that messages can be scheduled for distribution from those sites). This confirmation can be obtained from the underlying multicasting protocols, such as an atomic ordered broadcast protocol. As soon as the D-Agent receives this confirmation from the underlying multicast protocol object, it informs the local S-Agent in the client's GSM. The S-Agent then constructs the appropriate synchronisation message and sends it to the other members of the client group (i.e., to the S-Agents in their GSMs), as specified in the synchronisation policy.
2. *Receipt of message delivery confirmation from the servers*: In some other applications, a simple message delivery notification from the underlying multicast protocol object is insufficient to establish that the desired state change has occurred in the server group members. The requirement is to issue the synchronisation message only when replies are received from the server group members. These replies are confirmation that an action has been performed by the servers. As soon as the required number of replies are received by the C-Agent, it informs the local S-Agent in the client's GSM. The S-Agent then constructs the appropriate synchronisation message and sends it to the other members of the client group (i.e., to the S-Agents in their GSMs), as specified in the synchronisation policy.

3. *Receipt of desired state change notification from the previous client*: In some other synchronised invocation models, such as the example in section 7.9.5, a (operation) message can be invoked only on those server group members which have successfully executed the previous messages invoked by other clients, according to some application-specific criterion. In such applications the mere receipt of replies from the server group members is insufficient to establish that the desired state change has occurred in the servers or a successful execution of the client's message was performed by the servers. This requirement arises in applications in which the server group members are of the same type (possess similar capabilities or functionalities), but are not replicas, such as a group of robots, or a group of students which perform the same requested operation with different levels of precision or correctness. In such applications it is required to analyse (or process) the replies to determine that the desired state change has occurred in the servers or the results of the message execution returned by the servers, in their replies, conform to the desired level of acceptance.

   Therefore when all the replies are received from the server group, the C-Agent collates them into a group termination message and delivers it to the client object. The client object analyses each reply to determine which members of the server group have successfully executed its (operation) message and / or which of them have undergone the desired state change, according to some application-specific criterion. The client, then informs the GSM (the S-Agent) which members of the server group have successfully executed its message and which of them have not. This communication between the client object and the GSM occurs through the Group Management Interface (GMI), as described in detail in section 6.2.1. When the S-Agent receives this notification from the client, it constructs the appropriate synchronisation message and sends it to the other members of the client group (i.e., to the S-Agents in their GSMs), as specified in the synchronisation policy.

This type of synchronisation is not based upon the number of acknowledgments received from the server group, but rather on the contents of the replies sent by the server group members.

7.9.3 What are Synchronisation Messages

The synchronisation process involves seeking the permission of other members of the client group before the distribution of the message and / or receiving notifications from other members of the client group which grant permission for the distribution of the message to the server group. The client (or its S-Agent) which seeks the permission of other members of the group, in order to distribute a message, is called the *synchronisation seeker* while the group members which give (or deny) such a permission are called *synchronisation providers*.

A synchronisation message is constructed and sent by the S-Agent to solicit the permission to distribute a member message or to grant a permission to distribute the message. There are three types of synchronisation messages which are exchanged between the S-Agents. They are:

1. *Synchronisation soliciting message*: This message is constructed by the *synchronisation seeker* S-Agent and sent to the *synchronisation provider* S-Agents, to solicit the permission to distribute the message. This message is constructed by the S-Agent when an (operation | notification) message is received from the client object for distribution to the server group.
2. *Synchronisation response message*: This message is returned in response to the *synchronisation soliciting message* by the *synchronisation provider* S-Agents to the *synchronisation seeker* S-Agent, to convey the result of the synchronisation soliciting request.
3. *Synchronisation notification message*: This is an unsolicited message which is constructed by the *synchronisation provider* S-Agent and sent to the *synchronisation seeker* S-Agents to inform them that a pending message at their site can be distributed and the identities of the server group members to whom

the message can be distributed. This message is issued by the *synchronisation provider* S-Agent as soon as the confirmation of the message delivery and / or successful state change is received from the local D-Agent or the C-Agent or the client object, as discussed in section 7.9.2. This synchronisation message says to its recipients: "it's now your turn to distribute the message" and "distribute the message to these members" (see example in section 7.9.5).

The information contained in these synchronisation messages is application-specific and is interpreted by the S-Agents which are programmed to interpret a limited set of information content. The format of these and the information contained in them is discussed in chapter 9.

7.9.4 Communication between the Client Object and the S-Agent

When a synchronisation soliciting message is received by an S-Agent, it is required to decide the type of response (for e.g., authorisation or denial to distribute a message) to give in a synchronisation response message and when to send that response. In some cases, the S-Agent can decide locally what response to return and when to return that response based upon some synchronisation policy (such as a round-robin or prioritised message scheduling algorithm) that has been pre-specified to the S-Agent. However in some other cases, the subordinate roles are required to take the permission of superior roles before distributing the message. In such cases the permission to distribute a message is granted based upon the contents of the message. The members in the superior role need to analyse the message contents before deciding the type of response (such as an authorisation or denial to distribute a message) to give. In such cases the S-Agent needs to contact its member object to give it the message contents and the information about the source and destinations of the message. This communication between the S-Agent and the member object occurs through the *synchroniser management interface* (SMI) of the S-Agent and the *group management interface* (gmi) of the member object (see also section 6.2.1).

Similarly in applications, such as described in section 7.9.5, in which a desired state change is to be brought in the server group through the coordinated message invocations from the client group, the permission to *distribute* the next message from *successor group members* is given after the analysis of the replies (received from the server group) by the *current token holder* (client). The current token holder (client) is required to inform the local S-Agent which members of the server group have successfully executed its message and which of them have not, so that next message from the successor group members is *distributed* only to those server group members which have successfully executed the previous message. Hence there is a need to communicate some application-specific information to the GSM in order to enable the synchronised message distribution. This information could be a simple "pass | fail" information together with the identities of the server group members who have "passed | failed" the operation message execution, as is the case in the example in section 7.9.5. The interpretation of this information by the S-Agent is defined by the application. Again this communication between the client and the S-Agent occurs through the *synchroniser management interface* (SMI) of the S-Agent and the *group management interface* (gmi) of the member object. On receiving this information from the client, the S-Agent constructs the appropriate synchronisation notification message and sends it to successor group members.

7.9.5 Group Application-9: Coordinated Testing Application

In this section we describe a generic example of a coordinated client group application in order to illustrate the principles and requirements of synchronised message distribution from client group members. Our intention is not to describe any particular application, but to give an example which covers a broad range of applications of this kind. Applications in the product testing domain and product manufacturing domain are some of the examples that fall in this category. These applications require the coordinated actions of many agents (testers or builders) in order to carry out the testing or manufacturing function. In the following we consider a generic example from a testing domain. This application is characterised by the following features:

1. *Components of a testing application*: The testing activity involves different types of tests which are conducted on *objects-under-test* (OUTs) by different *tester agents* (TAs) which are specialised in giving those tests and in evaluating the results of those tests. In an automated testing environment these tests are conducted on a group of OUTs in a certain sequential (or parallel) order in order to conform to the application-specific testing logic. For example, some tests must be performed before others in order to bring the OUTs to the desired state or to discard those OUTs which fail the preliminary tests from the rest of the testing activities.

   The TAs represent any software or hardware entities which contain domain-specific testing logic including human agents such as professors specialised in specific subjects. The OUTs represent software or hardware or mechanical or human entities which are of the same type (i.e., possess similar capabilities and functionalities), but are not replicas, such as a group of robots or a group of students which perform the same (testing) function with different levels of precision and correctness.
2. *Organisation of testing application as a client group and server group*: The coordinated testing application consists of a group of TAs organised as a client group and the group of OUTs organised as a server group, as shown in figure 7.25. The TAs are categorized as clients and OUTs as server objects because the TAs invoke testing operations (operation message) on the OUTs, and the OUTs execute those testing operations and return the result of their test execution in the form of replies (termination messages) to the TAs. In some applications, the replies of the OUTs depend upon their previous invocation history.

   The TAs analyse the replies received from the OUTs to determine which of them conform to their required level of acceptance and which of them do not. Once the TA determines which of the OUTs have passed its test and which of them have not, it must communicate this information to its *successor TA*(*s*) so that it (they) can invoke its (their) testing operations on those OUTs which have passed the previous tests.
3. *Separation of testing logic from test coordination logic*: As shown in the example below, the test coordination (and sequencing) logic could be very complex. Moreover the test coordination logic changes with different application testing requirements. Hence it is desirable to keep this logic external to the application elements (i.e., tester agents), so that tester agents do not have to be modified for different testing requirements. In our model the test invocation synchronisation logic and the identities of the client and server group members are made transparent to the tester agents and they reside in the GSM (S-Agent) as programmable *synchronisation policy.* The TAs only communicate their pass or fail verdict of their test operations to the GSM (S-Agent) via their gmi (see section 6.2.1) and leave the test coordination and test progress (i.e. sending the synchronisation notification message to successor members) functions to the GSM (S-Agent).
4. *Testing Cycles*: In an automated testing environment, there are many items which come from production environment to testing environment, but there is only a limited number of ‘testing facilities’. Hence the tests are conducted in cycles. A fixed number of objects are tested during each cycle. A testing cycle consists of different types of tests, administered by different TAs, which are invoked one after the other according to the specific test coordination logic.

   The test operation (operation messages) for the next cycle is invoked by the tester agents on their local GSM at the GII, as soon as they give the ‘pass’ or ‘fail’ verdict of their previous operation message to the GSM at its GMI (see also section 7.9.7). However, the new test operation is not scheduled for distribution until a synchronisation message is received from the predecessor clients (i.e., their GSMs).

5. *Protocol between Client and GSM*: The client is required to invoke the test on the GII and to communicate the results of the test to the GSM (S-Agent) on the GMI. The client is obliged to inform the GSM about the outcome of the current test operation. It implies that the synchronised activity in a client group requires that the group members are capable of (and obliged to) inputting certain application-specific information which is required for the progress of the synchronised activity, such as the one above, to the GSM. It is the understanding (protocol) between the member and the GSM that the member invokes the next operation on the GII after giving the verdict of the previous operation on the GMI. The following activities are performed by the client object in every testing cycle:
   a. Invoke the test message (operation message) at the GII.
   b. Evaluate the replies (delivered through the GII).
   c. Convey the result of the test evaluation to the GSM (S-Agent) at the GMI.

Example: Let us consider a specific testing example in order to illustrate the requirements of test coordination. Consider a group of TAs, each specialised to perform a specific testing task on a group of OUTs. The following types of tests are performed:
1. Initialisation procedure at the beginning of a testing cycle: `init_test()`,
2. 'Type-A' test: test-A(a_1, a_2, a_3),
3. 'Type-B' test: test-B(b_1, b_2)
4. 'Type-C' test: test-C(c_1, c_2, c_3, c_4)
5. 'Type-D' test: test-D(d_1, d_2)
6. 'Type-E' test: test-E(e_1, e_2, e_3, e_4, e_5), where a_1, a_2, b_1, c_1, c_2, etc. are test parameters.

Each of these tests are performed by different TAs. In the most general case, there are multiple TAs assigned to perform a test of a given type, and in the extreme case there is a single TA per test type. In the general case of multiple TAs per test type, each TA performs the same type of test, but not necessarily identical tests, i.e., tests with identical parameter values. Moreover their test assessment and evaluation criterion is different. In such cases, the application is interested in obtaining an opinion of different testers of a given type before declaring a 'pass' or 'fail' outcome.

In our example the TA group consists of the following members:
1. A test administrator: `TAdmin`, which initialises the testing activity
2. One agent which performs 'Type-A' test: `TA-1`
3. Two agents which perform 'Type-B' test: `TB-1, TB-2`
4. One agent which performs 'Type-C' test: `TC-1`
5. One agent which performs 'Type-D' test: `TD-1`
6. One agents which performs 'Type-E' test: `TE-1`

After evaluating the results of the testing operation, each tester agent communicates the 'pass' or the 'fail' verdict of the corresponding test operation to its local S-Agent through the following message invocation on the GMI:

`sync_enabling_info(successful_member_list, failed_member_list).`

The following are the application-specific test coordination requirements:
1. "Type-A" test can be offered only to those OUTs who have passed init_test().
2. "Type-B" test can be offered only to those OUTs who have passed "Type-A" test.

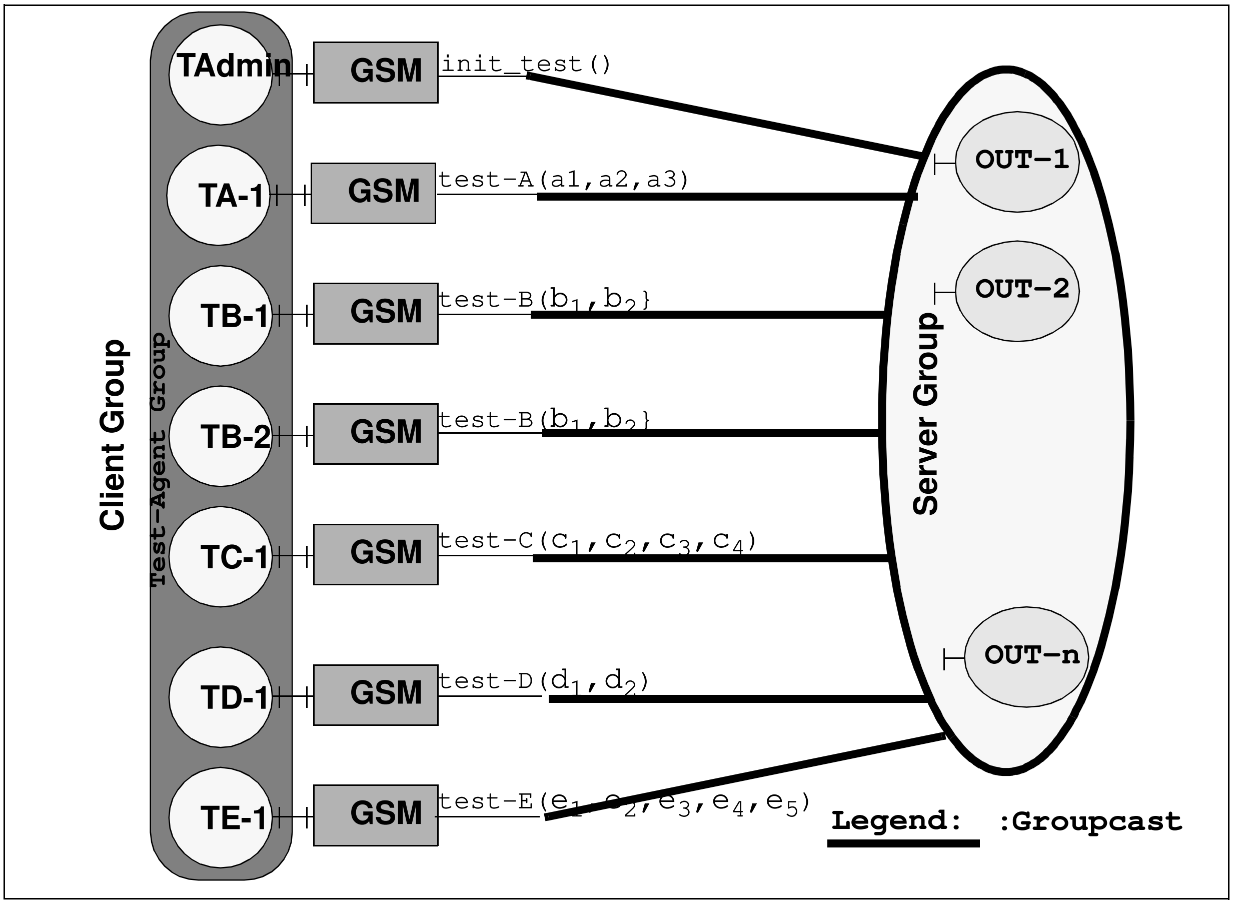

Client Group
Test-Agent Group
TAdmin
TA-1
TB-1
TB-2
TC-1
TD-1
TE-1
GSM
init_test()
test-A(a1,a2,a3)
test-B(b1,b2}
test-B(b1,b2}
test-C(c1,c2,c3,c4)
test-D(d1,d2)
test-E(e1,e2,e3,e4,e5)
Server Group
OUT-1
OUT-2
OUT-n
Legend: :Groupcast

Fig. 7.25 Coordinated Testing Application

3. "Type-C" and "Type-D" tests can be offered in parallel, i.e., the execution of these tests can be interleaved at OUTs, however they can be offered only to those OUTs who passed at least one (of the two) "Type-B" test.
4. "Type-E" test can be offered only to those IUTs who have passed either "Type-C" or "Type-D" test.
5. The test administrator, TAdmin, is notified by the other test agents about the outcome of the test. A member of the OUT group may either pass the test or fail the test or not respond to the test (i.e., its reply is not received within the specified time limit). After the completion of the "Type-E" test, the TAdmin must send the final pass or failure notifications, with appropriate grades if required by the application (for product certification, etc.), to the OUTs. So the TAdmin is required to send a:

5.1 "Grade-A()" notification to those OUTs who have passed "Type-E" test.
5.2 "Grade-B()" notification to those OUTS who have passed "Type-C" or "Type-D" test but not "Type-E" test.
5.3 "Grade-C()" notification to those OUTs who have passed "Type-B" test but not "Type-C" or "Type-D" test.
5.4 "Grade-D()" notification to those OUTs who have not passed "Type-A" test.
5.5 "Object_Partially_Tested()" notification to those OUTs who have not responded to any one of the tests.

These notifications are sent as operation messages to the OUTs. When the TAdmin receives the confirmation of the receipt of these notifications from the OUTs in the form of replies (or when the C-Agent sends an exception termination message to it when the replies are not received within a certain time limit), it starts the next testing cycle.

7.9.6 Synchronisation Requirements & Policy Specification

The test messages are distributed to the OUTs when a specified synchronisation condition is satisfied. The synchronisation condition is specified in the S-Agent in GPSL as a synchronisation policy specification. The synchronisation condition represents the test coordination logic. In this section we describe the test synchronisation requirements and the test outcome notification requirements of the tester agents in group application-9 and the corresponding synchronisation policy specification.

1. *Test Initialisation*: The test administrator, `TAdmin`, initialises the OUTs and prepares them for the subsequent testing activity, at the beginning of every testing cycle. There is no precondition to the *distribution* of the `init_test()` operation invoked by the test administrator. However, only those OUTs who have successfully initialised themselves should be included in the subsequent testing activity. Hence the result (pass or fail) of the `init_test()` should be notified to the next tester agent, i.e., TA-1. The following policy specification captures the "`init_test()`" synchronisation requirements. It is specified to the S-Agent associated with `TAdmin`.

```
synchronisation_policy
sync init_test()
with
[nil]
notify
[
 sync_events passed(),failed(),reply_not_received(),test_not_scheduled()
  to TA-1,TAdmin
]
end_policy
```

Fig. 7.26 Synchronisation Policy Specification for the S-Agent of TAdmin

2. *Sequencing "Type-A" test after test initialisation*: The "Type-A" test can only be scheduled for *distribution* after the completion of test initialisation process by the test administrator. Moreover, only those OUTs who have successfully initialised themselves are offered the "Type-A" test. The outcome of the "Type-A" test must be notified to tester agents of "Type-B". The outcome of all the tests is notified to the test administrator. The following policy captures these requirements. It is specified to the S-Agent associated with TA-1.

```
synchronisation_policy
sync test-A(a1, a2, a3)
with
   [
    unsolicited_reception_of passed(member_list)
    from TAdmin
    within Time_Limit_1
   ]sync_cardinality ATLEAST(ALL)
notify
   [sync_events passed(),failed(),reply_not_received(),test_not_scheduled()
   ]to TB-1, TB-2, TAdmin
end_policy
```

Fig. 7.27 Synchronisation Policy Specification for the S-Agent of TA-1

Policy Interpretation: In the above policy specification, we introduce a few new language primitives. These are explained in detail in chapter 8. Intuitively, the semantics of this policy specification is to synchronise the distribution of the message specified in the 'sync' clause with the (unsolicited) reception of a synchronisation notification message, "passed()", from the TAdmin within a certain time of the invocation of the message, `test-A(`a_1`,`a_2`,`a_3`)`, by TA-1. If the synchronisation notification message is not received within the specified time period, then the message is not distributed and an exception termination is returned to the member (TA-1) along with the appropriate exception information. The synchronisation notification message contains the list of the members who have successfully executed the previous message invoked by TAdmin. The message is required to be distributed only to these members. If the list is empty, then the message is not distributed and an exception termination is sent to the member (TA-1) along with the appropriate exception information. Moreover, the S-Agent sends a synchronisation notification message, "`test_not_scheduled()`", to the members specified in the 'notify' clause to inform them that none of the members could qualify for my test. This enables the progress of the other testing activities within the testing cycle and brings the testing cycle to its termination.

As mentioned in section 7.9.4, the S-Agent expects to receive a notification from the associated member (TA-1), via the SMI, which contains the identities of the reply messages which are declared 'pass' and 'fail' by the member. These identities are mapped onto the corresponding member identities by the S-Agent (see section 7.9.7). The S-Agent then constructs a synchronisation notification message which contains identifiers of the server group members who have passed, failed, and not responded to the test, and sends it to the members specified in the 'notify' clause.

At any given site, the S-Agent is required to notify the following events, through the S-NTF-GPDU (see section 9.7.2), to the other S-Agents associated with the members specified in the 'notify' clause These events may be generated by the group member (test agent), the local C-Agent, or due to a time-out condition in the S-Agent.

1. The list of the members who have passed the test: `passed(member_list)`. This event is generated by the group member (test agent).
2. The list of members who have failed the test: `failed(member_list)`. This event is generated by the group member (test agent).
3. The list of members who have not responded to the test, i.e., whose replies were not received by the C-Agent within the specified reply collation period: `reply_not_received(member_list)`. This is generated by the local C-Agent.
4. If the above mentioned synchronisation related information is not received by any S-Agent within the specified 'synchronisation interval', then the test (operation message) cannot be distributed to server group, an exception termination is returned to the client (test agent), and the following message is sent to all the members specified in the 'notify' clause, in order to avoid any deadlock: `test_not_scheduled()`. This event is generated due to synchronisation time-out.
5. If "`test_not_scheduled()`" is received from any one of the S-Agent in the S-NTF-GPDU, then the S-Agent which receives this message sends an exception termination to its local client (test agent), and sends the "`test_not_scheduled()`" message to the other S-Agents specified in the 'notify' clause.

3. *Sequencing "Type-B" test after the completion of "Type-A" test*: "Type-B" test can only be offered to those OUTs who have passed the "Type-A" test. "Type-C" and "Type-D" testers require a notification of the completion of "Type-B" test and the identities of the members who have passed this test.

.

```
synchronisation_policy
for test-B()
sync test-B(b1, b2}
with
   [
    unsolicited_reception_of passed(member_list)
    from TA-1
    within Time_Limit_2
    sync_cardinality ATLEAST(ALL)
   ]
notify
   [
   sync_events passed(),failed(),reply_not_received(),test_not_scheduled()
   to TC-1, TD-1, TAdmin
   ]
end_policy
```

Fig. 7.28 Synchronisation Policy Specification for the S-Agent of TB-1, TB-2

4. *Parallel Scheduling of "Type-C" and "Type-D" test after the completion of "Type-B" test*: "Type-C" and "Type-D" tests can be scheduled in parallel only on those IUTs who have passed at least on of the two "Type-B" tests conducted by TB-1 and TB-2. "Type-E" tester agent requires a notification of the completion of "Type-C" and "Type-D" tests and the identities of the members who have passed this test.

```
synchronisation_policy
for test-C()
sync test-C(c1, c2, c3, c4)
with
    [
     unsolicited_reception_of passed(member_list)
     from TB-1, TB-2
     within Time_Limit_3
     sync_cardinality ATLEAST(1)
    ]
notify
    [
     sync_events passed(),failed(),reply_not_received(),test_not_scheduled()
     to TE-1, TAdmin
    ]
end_policy
```

Fig. 7.29 Synchronisation Policy Specification for the S-Agent of TC-1

5. *Sequencing "Type-E" test after the completion of "Type-C" and "Type-D" test*: "Type-E" test can be offered only to those IUTs who have passed either "Type-C" or "Type-D" test.

```
synchronisation_policy
for test-E()
sync test-E(e1, e2, e3, e4, e5)
with
(
   [
    unsolicited_reception_of passed(member_list)
    from TC-1
    within Time_Limit_4
    sync_cardinality ATLEAST(ALL)
   ]
   or
   [
    unsolicited_reception_of passed(member_list)
    from TD-1
    within Time_Limit_4
    sync_cardinality ATLEAST(ALL)
   ]
)
notify
   [
    sync_events passed(),failed(),reply_not_received(),test_not_scheduled()
    to TAdmin
   ]
end_policy
```

Fig. 7.30 Synchronisation Policy Specification for the S-Agent of TE-1

6. *Sequencing final test outcome notifications after the completion of testing activity*: Let's take two requirements of this final testing activity.

6.1 Grade-B()" notifications are sent to those OUTs who have passed "Type-C" or "Type-D" test but not "Type-E" test.

```
synchronisation_policy
for Grade-B()
sync Grade-B()
with
(
   ([
    unsolicited_reception_of passed(member_list)
    from TC1
    within Time_Limit_5
    sync_cardinality ATLEAST(ALL)
    ]
   or
    [
    unsolicited_reception_of passed(member_list)
    from TD-1
    within Time_Limit_5
    sync_cardinality ATLEAST(ALL)
    ])
    and
    [
    unsolicited_reception_of failed(member_list)
    from TE-1
    within Time_Limit_5
    sync_cardinality ATLEAST(ALL)
    ]
) notify
    [
    sync_events _NONE_
    to _NONE_
end_policy
```

Fig. 7.31 Synchronisation Policy Specification for the S-Agent of TAdmin - (for Grade-B() message)

P*olicy Interpretation*: The "_NONE_" in the "sync_events" clause means that no synchronisation events are generated. The "_NONE_" in the "to" clause means that there is no need to send a S-NTF-GPDU" to any one in the group.

6.2 The TAdmin must send "Object_Partially_Tested()" message to those OUTs who have not responded to any one of the tests or if a test is not scheduled (performed) by any one of the test agents.

```
synchronisation_policy
for Object_Partially_Tested()
sync Object_Partially_Tested()
with
[
unsolicited_reception_of reply_not_received(member_list)
from TAdmin,TA-1,TB-1,TB-2,TC-1,TD-1,TE-1
within Time_Limit_5
sync_cardinality ATLEAST(1), ATMOST(ALL)
]
or
[
unsolicited_reception_of test_not_scheduled()
from TAdmin,TA-1,TB-1,TB-2,TC-1,TD-1,TE-1
within Time_Limit_5
sync_cardinality ATLEAST(1), ATMOST(ALL)
]
notify
[
sync_events _NONE_
to _NONE_
]
end_policy
```

Fig. 7.32 Synchronisation Policy Specification for the S-Agent of TAdmin (for Object_Partially_Tested() message)

7.9.7 Interaction between GSM Agents to Support Synchronised Message Distribution from Client

The synchronisation of message distribution from client group members is achieved through the cooperation of multiple GSM agents, such as, the S-Agent, the D-Agent, and the C-Agent, as shown in figure 7.34. These agents interact with each other locally via the inter-agent interfaces and remotely through the inter-GSM protocol, for the support of synchronised message distribution.

The G-Agent intercepts the (operation | notification) messages received from the client object. In applications involving synchronised message distribution, the G-Agent gives this message not only to the D-Agent (arrow 1 in figure 7.34), but also to the S-Agent (arrow 2 in figure 7.34), because the latter must inform the former when and whom to distribute the message to after receiving the appropriate synchronisation notification messages and processing the synchronisation policy associated with the message (specified as S-Policy Script, as shown in figure 6.2).

1. *Interaction between S-Agent and D-Agent*: (*Synchronise before distribution*). When an operation message is received from the client for distribution to the server group, the D-Agent does not distribute the message until a synchronisation message is received from the local S-Agent. This message is issued by the S-Agent when the synchronisation condition specified by the client in the S-Policy Script is satisfied. The message contains the identifier of the operation message for which the required synchronisation has been achieved and the identities of the server group members (for example OUTs in figure 7.25) to whom it can be distributed. This is shown in arrow 3 in figure 7.34.

```
sd_distribute_message_to(inv_instance_id: inv_instance_id_type,
                         membership_list: member_id_list_type)
```

As shown in figure 7.33, the *distribution policy* of the D-Agent contains an indication in the ‘to’ clause (i.e., “NOTIFIED_MEMBERS”) that the message distribution requires a *notification* from the S-Agent, and that it can be distributed only to those server group members identified (in the “membership_list”) by the S-Agent. Therefore the D-Agent delays the distribution until notified by the S-Agent.

```
operation_distribution_policy
for test-A
    distribute test-A(a1, a2, a3)
    to NOTIFIED_MEMBERS
    using SOURCE_ORDERED_MULTICAST
end_policy
```

Fig. 7.33 Synchronised Message Distribution Policy

2. *Interaction between D-Agent and C-Agent*: (*Expect replies from these members*). After the D-Agent distributes the message to the ‘notified members’, it informs the C-Agent to expect replies from those members, as shown in arrow 4. The D-Agent gives the identities of these members to the C-Agent.

```
dc_collate_replies_from(OPR_inv_instance_id: inv_instance_id_type,
                        membership_list: member_id_list_type)
```

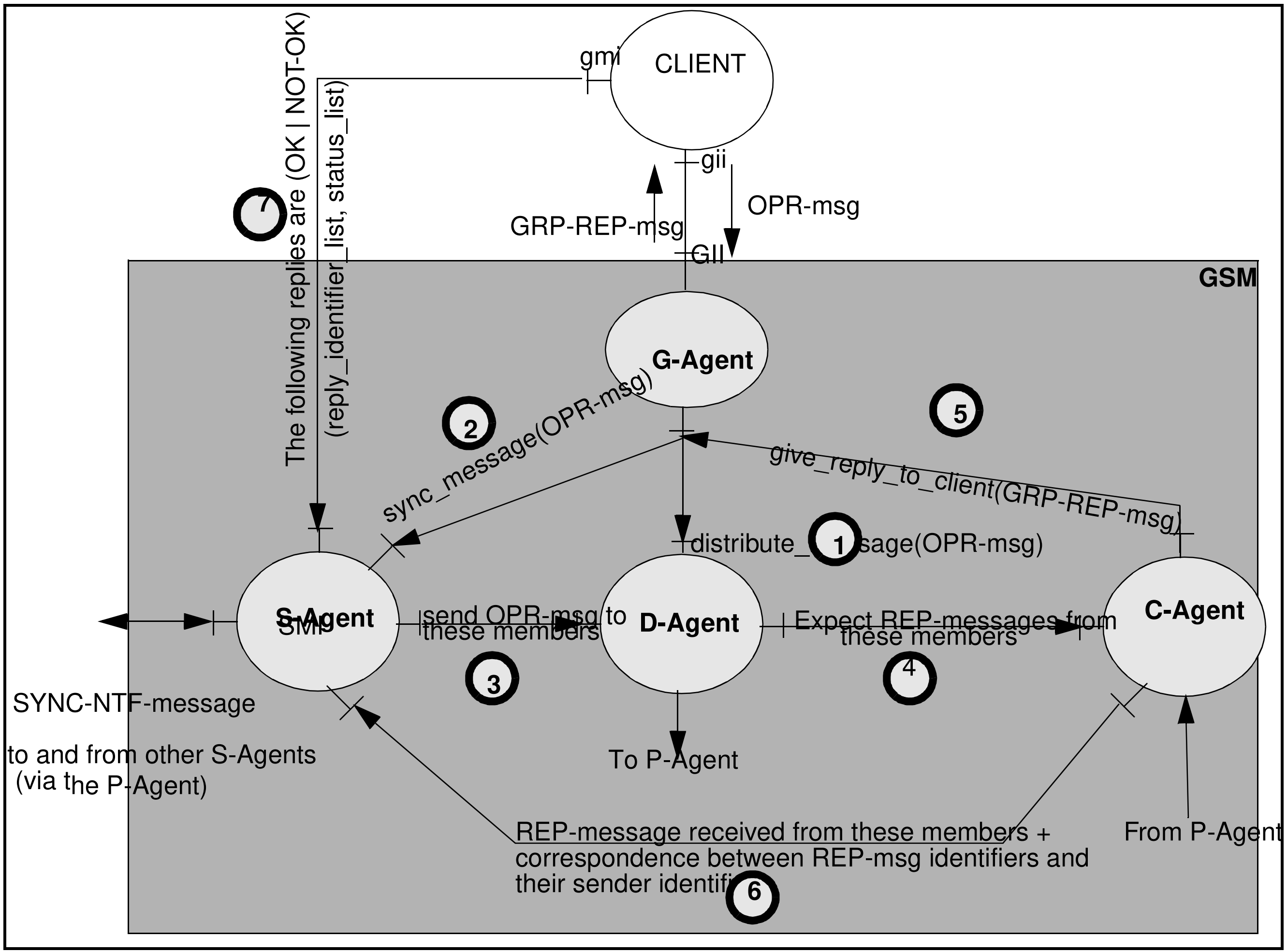

gmi
CLIENT
gii
OPR-msg
GRP-REP-msg
GII
GSM
The following replies are (OK | NOT-OK)
(reply_identifier_list, status_list)
7
G-Agent
2
sync_message(OPR-msg)
5
give_reply_to_client(GRP-REP-msg)
distribute_
1
sage(OPR-msg)
S-Agent
SMI
send OPR-msg to these members
D-Agent
Expect REP-messages from these members
C-Agent
3
4
SYNC-NTF-message
to and from other S-Agents
(via the P-Agent)
To P-Agent
From P-Agent
REP-message received from these members + correspondence between REP-msg identifiers and their sender identifi
6

Fig. 7.34 Coordination between GSM Agents to Support Synchronised Message Distribution from Client

3. *Interaction between C-Agent and G-Agent*: (*Replies received from these members*). The C-Agent expects to receive the replies from the 'notified members' (for example OUTs in figure 7.25). It collects the replies received within the collation period, as specified in the collation policy, and gives them to the client object, via the G-Agent as shown in arrow 5 in figure 7.34. These replies may be returned separately or they may be collated into a single group termination message which is returned to the client. In many applications, the replies are locally identified (for example by their offset in the group termination message) between the C-Agent and the client; the identifiers of the servers which have sent these replies are not explicitly given to the client (unless such a parameter exists in the termination message itself).
4. *Interaction between C-Agent and S-Agent*: On receiving the replies from the local GSM (G-Agent), the client (for example the tester agent in figure 7.25) analyses these replies and gives a "pass/fail" verdict to the local S-Agent (see arrow 7 in figure 7.34) for each of the received reply, by identifying the replies with their local identifiers (for example by their offset in the group termination message). However, the S-Agent must know the identities of the servers (for example OUTs in figure 7.25) which have passed/failed the test (i.e., the operation message sent by client), so that it can include this information in the synchronisation notification message (S-NTF-GPDU) to other S-Agents in the client group (test agent group in figure 7.25). Therefore, the C-Agent must inform the S-Agent the local reply identifiers and their corresponding sender identifiers, as shown in arrow 6 in figure 7.34.
   `cs__replies_received_from(reply_identifier_list, server_identifier_list)`
   This enables the S-Agent to know which servers have not sent their replies, so that appropriate information (such as "reply_not_received()") is included in the synchronisation notification message (S-NTF-GPDU) that are sent to other S-Agents in the client group. The correspondence between the reply identifiers and the server identifiers also enables the S-Agent to know which servers have successfully executed the client's operation message and which ones have not, as explained in the next bullet.
5. *Interaction between client object and S-Agent*: (*Receive synchronisation related information from the client*). In some applications, such as in group application-9, the S-Agent receives a message from the client object (test agent), through its *synchronisation management interface* (SMI), which contains the identities of the replies (for example their offset in the group termination message) which are satisfactory and unsatisfactory (or passed and failed) according to the client's application-specific criterion (such as the test result evaluation criterion in group application-9). This is shown in arrow 7 in figure 7.34. Based upon this information and the correspondence between the reply-ids and the server-ids received from the C-Agent, the S-Agent can decide which server group members have passed or failed the test (operation message) sent by the client. The client invokes the following message on the S-Agent via the SMI (arrow 7):
   `reply_evaluation_results(reply_identifier_list, status_list)`
6. *Inter-GSM communication between S-Agents*: (*Notify synchronisation events to other S-Agents*) When the S-Agent determines the identities of the server group members who have passed, failed, and not responded to the test, it must communicate this information to the *synchronisation seeker* S-Agents in the client group through the inter-GSM protocol, so that the synchronised message distribution activity can proceed in the group. For example, in group application-9, when the replies of the operation message, `test-A(`a_1`, `a_2`, `a_3`)`, are received, analysed, and communicated to the local S-Agent by TA-1 (client), the S-Agent of TA-1 (synchronisation provider) sends the identities of the OUTs who have passed, failed, and not responded to the test, to the synchronisation seeker S-Agents associated with TB-1 and TB-2, in the synchronisation notification message (S-NTF-GPDU).

7.9.8 Transparent & External Support for Synchronised Invocation in the GSM

As shown in the previous example, the logic for the synchronised distribution of messages from the client group members could be very complex. Moreover this logic changes with different message synchronisation requirements of an application, such as different test message coordination logic in group application-9. Hence it is desirable to keep this logic external to the application elements (client objects), so that the application elements do not have to be modified for different message synchronisation requirements.

In our model the message synchronisation logic and the identities of the client and server group members are made transparent to the application elements and this information resides in the GSM (S-Agent) as programmable *synchronisation policy*. The application elements only communicate certain application-specific information which is required for the synchronisation, such as the pass or fail verdict of the test operation in group application-9, to the GSM (S-Agent) through the "gmi" and SMI and leave the rest of the message synchronisation functions to the GSM. The GSM transparently performs the following synchronisation functions for the application elements:

1. *Synchronisation constraint (policy) evaluation*: The constraints on the distribution of message from the client objects, specified as *synchronisation policies*, are evaluated by the S-Agents, and when these constraints are satisfied, the D-Agents are notified to distribute the message.
2. *Progress of the synchronised activity in the client group*: The synchronisation events, such as the receipt of all replies from the server group or the successful execution of operation messages in the server group, are notified to other *synchronisation seeker* S-Agents via the synchronisation notification messages, in order to progress the synchronised message distribution activity in the client group.

7.10 Filtered Message Delivery Model

Filtering of (operation | notification) messages in a server group before delivery to the server objects is a common requirement in group-based applications, in which a message groupcast by a client object to the server group is delivered to a sub set of server group members based upon some message filtering criterion specified either by the client or the server or both.

Typical examples of "filtered message delivery in server groups" include a wide range of applications from traditional 'service group' applications to some non-traditional network management applications, such as

1. requesting services from a group of service providers based upon cost or some performance criterion, or
2. a manager object (acting as a client) wishes to selectively address the managed objects in a managed object group (acting as a server group) based upon their diverse attributes such as location, device characteristics, etc.

7.10.1 Why Filtered Message Delivery in the Server Group

Although each member of a (homogeneous) service group provides the same type of service, but the quality of the provision of the service may differ from member to member due to the difference in the non-functional attributes associated with the service. The *quality of service* is characterised by these non-functional attributes such as the queue length of the print server, load or other performance metrics of a server object, cost of service provision, etc.

In many applications, the selection of the sub set of servers from an organised service group, to handle a given service request, is based upon these "*quality of service*" related attributes. A sub set of servers are selected, for service request handling, from the server group, based upon their quality of service attributes. Hence these attributes are called *filter attributes*. A *filtering criterion* is a boolean expression composed of *filtering clauses* joined with boolean operators. A *filtering clause* is a filter attribute compared with a constant value or a function using *relationship operators*.

The filtering of messages, before delivery to the servers, is performed in the server group for the following reasons:

1. *To give the client the ability to choose the servers to execute its service request based upon client's filtering criterion*: In some applications the clients wish to select specific servers from the server group, for execution of its service request, based upon its filtering criterion. So the client specifies its filtering criterion as a boolean expression of server's filter attributes, which is groupcast along with the (operation | notification) message. In some cases the client also specifies the number of servers required. The client's filtering criterion is evaluated on the server side by each GSM (F-Agent) in order to find if the server object satisfies the client's filtering criterion. All those servers (i.e., their GSMs) who satisfy the client's filtering criterion enter into an *m-out of-n selection* process (see section 9.8) in order to select the fixed number of servers to whom the message will be finally delivered.
2. *To give the server the ability to choose the clients it wishes to service based upon server's filtering criterion*: The server objects in the server group may not necessarily wish to accept service requests from all possible clients. In such a case they specify their requirements for 'service offer' which the clients must satisfy in order to obtain the service. This is specified as *server's filtering criterion* which is a boolean expression of client's *filter attributes*, such as the geographic location of the client or the cost of service offer. The server's filtering criterion is specified to its local GSM (F-Agent). In these applications the client specifies the value of its filtering attributes which are group cast along with the message. The GSM (or F-Agents) on the server side uses the client's filtering attributes to evaluate the server's filtering criterion. The message is delivered to those servers at which the server's filtering criterion is satisfied by the client's filter attributes.

7.10.2 Communication between the Server Object and F-Agent

Filtering is done by the F-Agent based upon the values of the filter attributes. These attributes are application-specific. They can be *static* or *dynamic*. The values of static attributes do not change with time whereas the values of the dynamic attributes change with the execution history of the server objects, such as the 'queue length of the printer object' or the 'load' and the 'resource availability' of a server object, etc. Hence the values of the dynamic attributes must be communicated to the GSM (i.e., its F-Agent) whenever there is a change in its value, in order to filter the messages based upon the current values of the filter attributes.

Therefore the GSM needs some application-specific information for the correct functioning of the message filtering process. As described in section 6.2.1, this information is communicated from the member object to the GSM through the group management interface (gmi) of the member and GSM Management Interface (GMI) of the GSM. The F-Agent receives this information through its Filter Management Interface (FMI). The server object can also notify its ready or busy status via its gmi, so that the messages are filtered in only when the server signals a ready status. The content and the format of the messages exchanged between the member object and the GSM is application-specific and corresponds to the local 'member-GSM' protocol. For example, the values of the filter attributes can be notified through the `modify_attribute(attribute_name, attribute_value)` message, from the client to the F-Agent.

7.10.3 Group Application-10: A Printer-Pool

A set of printers with an associated print server is an example of *homogeneous service group*. The *printer group* is composed of different types of printers such as a dot-matrix printer, a laser (black and white) printer, color printer, etc. Although each printer provides the same service type, i.e., printing of text, there is a difference in the quality of service attributes such as the delay in handling the print request due to different queue lengths and printer speeds, and different quality of prints such as coarse and fine, etc.

The client broadcasts its print request “`print_request(data)`” to all print-servers, which are organised as a *printer group*. However, this request should be queued (or delivered) at the print server which satisfies the following client’s specifications:

1. smallest job queue,
2. color, laser print,
3. cost of printing less than 5 cents per page, and
4. printer speed greater than 5 pages per minute.

The print servers require that the users (clients) identify themselves with their user-ids, location, and their account numbers. Only the “print requests” of those users who are registered with them and who have sufficient funds available in their accounts are filtered in. Hence the attributes such as printer queue length, funds available in a user’s account are dynamic attributes which are communicated to the F-Agent by the print server whenever there is a change in their values, via the gmi and FMI (see figure 7.37).

Since the “print request” should be queued at only one print server, all the print servers (i.e., their F-Agents) who satisfy the client’s requirements and where the client also satisfies the print server’s requirements (i.e., registered as a user and have sufficient funds available) enter into an “m-out of-n selection” process (section 9.8). The selection process chooses the print server which best satisfies the client’s criterion and in case of a tie the printer closest to the client is selected. The selected print server informs the client where it is located and the number of pages that are printed, through a reply message to the client’s print request, such as “`collect_output(printer_id, number_of_pages_printed)`”.

7.10.4 Filtering Requirements & Policy Specification

The client’s requirements are specified as a boolean filtering criteria. The client’s attributes, such as ‘user_id’, ‘user_location’, etc. are specified as filtering attributes. The client’s filtering criterion is evaluated using the values of the server’s attributes and the server’s filtering criterion is evaluated using the values of client’s attributes, by the F-Agent associated with the server object. The number of servers required by the client to execute its service request is specified as filtering cardinality. If “m-out of-n selection” process is required, then identities of the members amongst whom this is to be carried out is specified in the ‘amongst’ clause. The following is the client’s filtering policy specification.

```
filtering_policy
for print_request()
amongst Printer_Group
filtering_cardinality ATLEAST(1),ATMOST(1)
filtering_criterion ((printer_type = laser) and (min(queue_length))
                     and (printer_quality = color) and
                     (cost_per_page <5 cents) and
                     (printer_speed > 5ppm))
filtering_properties (user_id = jim, user_location=computer_science_dept,
                      user_account=Ac11029)
end_policy
```

Fig. 7.35 Client’s Filtering Policy Specification

The print servers specify the value of their attributes and their filtering criterion in their filter policy specification, as shown below.

```
filtering_policy
for print_request()
filtering_criterion ((user_id in registered_user_list) and
                      user_account has sufficient_fund))
filtering_properties (printer_type = laser,
                      queue_length = 175 pages,
                      printer_quality = color,
                      cost_per_page=3 cents,
                      printer_speed=7ppm)
end_policy
```

Fig. 7.36 Server's Filtering Policy Specification

7.10.5 Interaction between GSM Agents to Support Filtered Message delivery to Server Object

The filtered delivery of messages to the server object is achieved through the cooperation of multiple GSM agents, such as the F-Agent, the D-Agent, and the C-Agent. These agents interact with each other locally via the inter-agent interfaces and remotely through the inter-GSM protocol, for the support of filtered message delivery, as explained below. We explain the interaction between these agents on the client (figure not given) and server side (figure 7.37) GSM.

1. *Interaction between G-Agent, D-Agent, and F-Agent* (*client side*): In the filtered message delivery applications, whenever an (operation | notification) message is received from the client, the G-Agent not only gives the message to the D-Agent, but also to the F-Agent, so that the latter can send the filtering constraints to be associated with the message to the D-Agent.
2. *Interaction between F-Agent and D-Agent* (*client side*):The client's filter attributes, filter criteria, and filter cardinality must be sent to the server side GSM along with the (operation | notification) message. These *filter constraints* are stored in the F-Policy Script (figure 7.35) in the F-Agent, as shown in figure 6.2. When a message is received from the client for distribution to the server group, the D-Agent does not distribute the message until the *filtering constraints*, which are to be included in the (OPR | NTF) GPDU are received from the local F-Agent. The F-Agent sends the filtering constraints to the D-Agent, as soon as the (operation | notification) message is received from the G-Agent.
3. *Interaction between D-Agent and C-Agent* (*client side*): After the D-Agent distributes the operation message to the server group members, as specified in the distribution policy, it informs the C-Agent to expect replies from those members. The D-Agent gives the identities of these members to the C-Agent. However, because of the filtering of these messages in the server group, only a sub set of the specified server group members receive the message and consequently only a sub set of them send the replies. In lieu of replies from the servers that were filtered out, the C-Agent receives "filter exception messages" from their C-Agents (see section 9.6.3), thereby avoiding any problem caused by the receipt of fewer than expected replies.
4. *Interaction between P-Agent, F-Agent, and C-Agent* (*server side*): The P-Agent always intercepts the GPDUs received from the network. Whenever an D-(OPR | NTF) GPDU (see section 9.6) is received with filtering constraints associated with it, the P-Agent extracts the filter constraints field from the GPDU and gives it to the F-Agent. The rest of the GPDU is given to the C-Agent. This is shown by arrows 1 and 2 in figure 7.37.
5. *Interaction between F-Agent and C-Agent* (*server side*): When an (operation | notification) message is received from the P-Agent for delivery to the server object, the C-Agent does not deliver the message until a *filtering message* is received from the F-Agent, authorizing it to deliver the message. The F-Agent

gives this authorisation only when the server satisfies the client's filtering criterion and the client satisfies the server's filtering criteria and if the server is finally selected in an "m-out of-n selection" process. This interaction is shown by arrow 4 in figure 7.37.

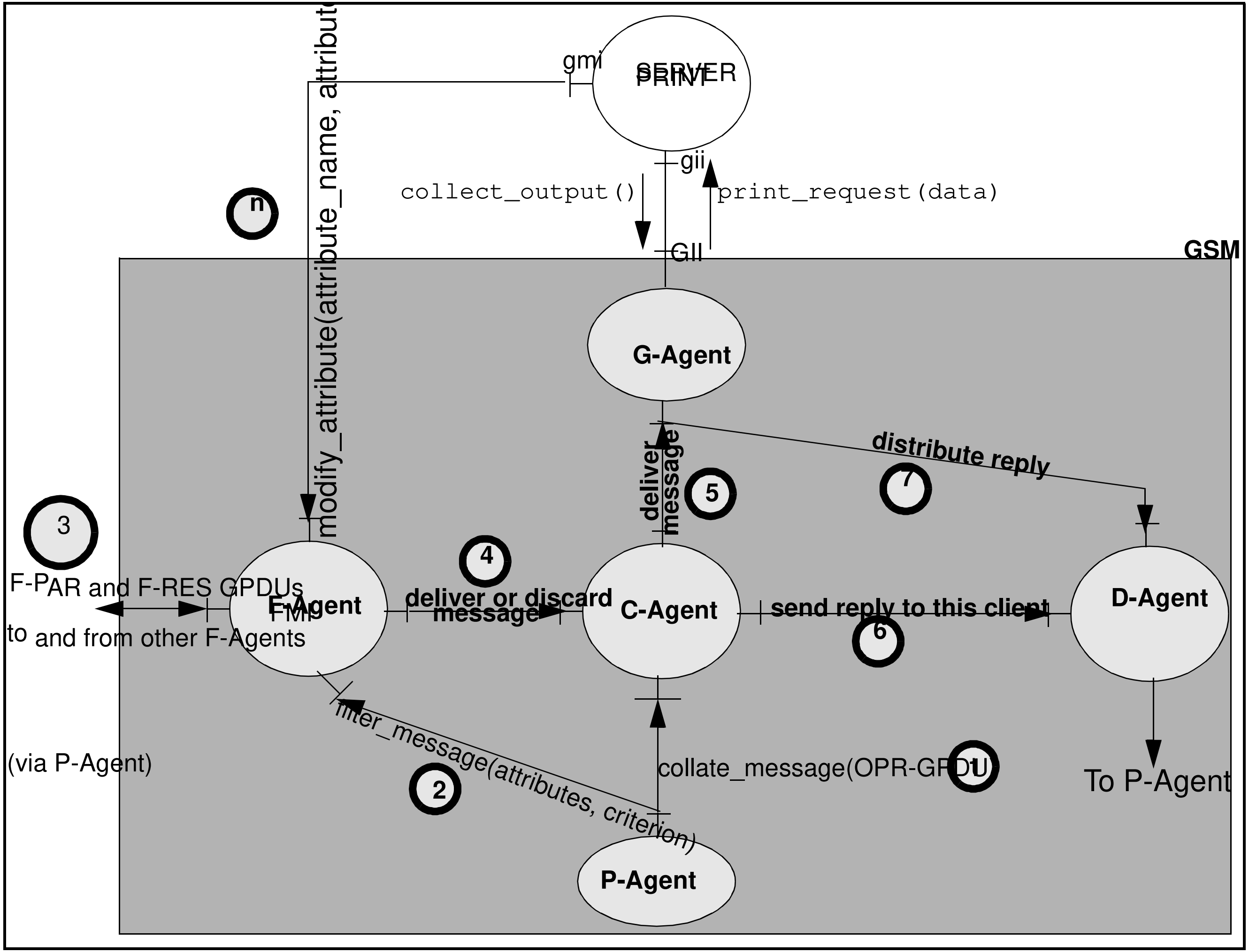

gmi
SERVER
PRINT
gii
collect_output()
print_request(data)
GII
GSM
n
modify_attribute(attribute_name, attribut
G-Agent
deliver message
5
distribute reply
7
3
F-PAR and F-RES GPDUs
to and from other F-Agents
F-Agent
FMI
4
deliver or discard message
C-Agent
send reply to this client
6
D-Agent
(via P-Agent)
filter_message(attributes, criterion)
2
collate_message(OPR-GPDU)
1
To P-Agent
P-Agent

Fig. 7.37 Coordination between GSM Agents to Support Filtered Message Delivery (Server Side)

4. *Inter-GSM communication between F-Agents* (*server side*): When the client's filtering criterion and filter attributes are received by the F-Agent on the server side, it evaluates both the client's and the server's filtering criterion using the server's and client's filter attributes respectively. When both the boolean conditions are evaluated as 'true', then the F-Agent enters into an "m-out of-n selection" process (arrow 3) with other such F-Agents through the inter-GSM communication protocol (see section 9.8).

5. *Interaction between C-Agent and D-Agent* (*server side*): When the C-Agent receives the permission to deliver the operation message from the F-Agent (arrow 4), it delivers the message to the server object via the G-Agent (arrow 5 in figure 7.37). The reply returned by the server object in response to this message must be sent to the correct client. The C-Agent has the knowledge of the identity of the client which is received in the D-OPR-GPDU. So, the C-Agent informs the client's identity to the D-Agent, as shown in arrow 6, so that the reply (arrow 7) is sent to the correct client.

6. *Interaction between server and F-Agent* (*server side*): In some applications, such as in group application-10, the server object (such as the print server) must inform the current values of the dynamically changing filter attributes (such as the queue length of the print server) to the F-Agent, so that filtering of the received messages is performed based upon the current values of the filter attributes. This information is communicated to the F-Agent, via its *filter management interface* (FMI), whenever there is any change in the values of the filter attributes (queue length of printer). Since this information can be received at any time, this interaction shown by an arrow labelled ‘n’ in figure 7.37.

7.10.6 Transparent & External Support for Filtered Invocation

As shown in the example above, the attributes and the criterion for filtered message delivery could be very complex. Moreover this criterion changes with different application requirements for the filtering of the same message in the server group. Hence it is desirable to keep the filter attributes and filtering criterion external to the application elements (client and server objects), so that the application elements do not have to be modified for different message filtering requirements.

In our model the message filtering constraints and the identities of the client and server group members are made transparent to the application elements and this information resides in the GSM (F-Agent) as programmable *message filtering policy.* The application elements only communicate certain application-specific information which is required for the filtering, such as the values of the dynamic filter attributes, to the GSM (F-Agent) and leave the rest of the message filtering functions to the GSM. The GSM transparently performs the following filtering functions for the application elements:

1. *Filtering constraint (policy) evaluation*: The F-Agent evaluates the client’s filtering criteria using the server’s filter attributes and the server’s filtering criteria using the client’s filtering attributes.
2. *m-out of-n selection*: When both the client’s and the server’s filtering criterion are evaluated as ‘true’, the F-Agent enters into an “m -out of-n selection” process to select the final winner(s) to whom the (operation | notification) message can be delivered.

7.11 Conclusion

The group support platform (GSP) provides the support for different group coordination patterns (or behaviors) between client and server group members. These coordination patterns can be realised through a combination of basic group support services, such as message distribution service, message collation service, message synchronisation service, and message filtering service. In our model, these coordination patterns can be programmed in the GSP by specifying appropriate message distribution policy, collation policy, synchronisation policy, and filtering policy in GPSL.

In this model, the GSAs manage the group communication and coordination patterns on behalf of the user applications, who influence the behavior of these agents by means of policy specifications. The idea is to describe the functionality required of the group support platform (GSP) in a declarative language, the *group policy specification language* (GPSL), in order to specify a rich set of application requirements with respect to different group support services such as message distribution, collation, synchronisation, filtering, etc. Therefore the GSP offers selective group transparency by allowing applications to specify group support policies.

CHAPTER 8 Group Policy Specification Language: An Introduction

Abstract

We have developed a language for the specification of message distribution, collation, synchronisation, and filtering requirements of an application, at a high-level independent of the mechanisms or protocols needed to implement them. This chapter describes the syntax and the semantics of the language.

8.1 Introduction

In existing group support systems, such as ISIS, Horus, Electra, etc., interactions between group members are expressed in terms of explicit communication protocols such as different types of multicast protocols. The inability to abstract over interaction between group members results in low-level specification and reasoning. As a consequence, it is not possible to describe and compose the interaction patterns between group members at a high-level without worrying about the low-level message multicasting protocols. Moreover, it is difficult to modify the interaction patterns between group members because the involved group support mechanisms are hardwired into applications.

Our approach is to express group coordination patterns as a combination of message distribution, collation, synchronisation and filtering constraints. They are specified abstractly as corresponding group support policies. We have developed a language framework for the specification of these group support policies. It is called Group Policy Specification Language (GPSL). In this chapter we describe the syntax and semantics of GPSL. The Backus Normal Form (BNF) of GPSL is given in appendix. Numerous examples of the use of this language were given in chapter 7.

8.2 Why Group Policy Specification Language

The GPSL is a special purpose language for the high-level specification of message distribution, collation, synchronisation and filtering requirements of an application. These group support requirements can also be viewed as group interaction constraints, such as message distribution constraints, collation constraints, etc. The design of GPSL is motivated by the following considerations:

1. *Separation of application concerns from group-coordination concerns*: There is a need to separate objects from inter-object interaction issues in an object group environment. The GPSL permits the separation of application logic from group-interaction issues. Group interaction constraints can be specified separately in policy scripts. The policy scripts are stored in the GSM and are interpreted by GSAs. Changes to group coordination behaviors are possible by modifying the relevant group support policies, without modifying the application. This enables a better description and modification of group coordina-

tion behaviors external to the applications. Moreover coordination patterns can be changed dynamically during an application session, without re-compiling the application.

2. *High-level specification of group interaction constraints*: GPSL permits a high-level and declarative specification of group-interaction constraints, independent of the engineering mechanisms or protocols needed to implement them. The high-level language abstracts away from any detailed behavior of group support agents.
3. *Policy-driven mechanisms*: Policies should be explicitly expressed rather than implicitly defined, in order to be able to represent and manipulate them within a computer system. Policies are specified using a policy definition notation. This notation is interpreted by the mechanisms which are required to execute the policies. The group communication and coordination is transparent to the programmer who specifies these aspects in a high-level and abstract way in the GSM using GPSL. The group interaction constraints are evaluated prior to the distribution or the delivery of each message.

8.3 Basic Elements of GPSL

The message distribution, collation, synchronisation and filtering functions have many aspects associated with them (section 4.4). These aspects show certain commonality over these group support functions. This commonality can be expressed clearly through a common language framework. The elements of the GPSL are based upon these fundamental aspects or issues. The relationship of the language elements to the group support services is summarized in table 8.1 .

1. *Message Specification*: Message specification is a basic requirement of a group policy specification language. We need to specify the *message signature* (see section 1.5.2) the instances of which are to be distributed, collated, synchronised, and filtered.
2. *Membership Specification*: We need to specify the members of the group to whom a message is to be distributed or from whom it is to be collated or from whom the synchronisation messages are required before the *distribution* or *delivery* of the message in consideration.
3. *Time Specification*: Collation and synchronisation functions are usually bounded by a time-limit in order to avoid an indefinite delay in the receipt of messages that are to be collated or of the receipt of synchronisation messages.
4. *Cardinality Specification*: We need to specify how many messages to include in the collation process for the construction of group (operation | termination | notification) message or how many messages are required for synchronisation or how many of the filtered servers should be finally selected to perform message processing.
5. *Combination-mode Specification*: In case of collation, the received messages need to be combined into a single group message, before invoking on the group member. Some basic message collation schemes are described in section 3.6.
6. *Ordering Specification*: The messages have to be distributed or delivered in a certain order to the group members.
7. *Attribute Specification*: The received messages are filtered based upon criterion of the client or of the server or of both, specified as a boolean attribute expression.

The message distribution, collation, synchronisation, and filtering policies are specified using a combination of these basic language elements. The combination of these elements associated with a message type is called a *group policy primitive* (GPP). Every message type, the instances of which are to be *distributed*, *collated*, *synchronised*, or *filtered*, is associated with one or multiple GPPs.

Table 8.1: Relationship between Basic issues of Group Support Services and Elements of GPSL

	Distribution	Collation	Synchronisation	Filtering
Message Specification	instances of what *message type* to distribute	instances of what *message type* to collate	instances of what *message type* to synchronise	instances of what *message type* to filter
Membership Specification	to whom to distribute the message	whose messages to collate	with whom to synchronise	whose messages to filter
Time Specification	NA	how long to wait to receive message before starting collation	how long to wait to receive synchronisation messages.	NA
Cardinality Specification	the minimum number of messages that must be delivered	how many messages to collate	how many synchronisation messages need to be received in order to schedule the message distribution	how many of the filtered server objects should be selected for message processing
Collation mode Specification	NA	how to combine received messages	NA	NA
Message Ordering Specification	what ordering guarantees are required for the distribution	in what order to deliver received messages	disabling of synchronisation	NA
Attribute Specification	NA	NA	NA	on what basis to filter the received messages

8.4 Syntax and Semantics of Group Policy Primitives

A group policy specification consists of one or more GPPs. In the following we present the syntax and the associated semantics of the basic GPPs used in the specification of *distribution policy*, *collation policy*, *synchronisation policy*, and *filtering policy.*

8.4.1 Distribution Policy Primitive

The GPP used in a distribution policy specification is called a *distribution policy primitive* (DPP). The following is the syntax and semantics of a DPP. The examples of DPPs are given in section 7.4.

8.4.1.1 DPP Syntax

```
distribution_policy
for message name
    [
     distribute message specification
     to membership specification
     distribution_cardinality cardinality specification
     using ordering specification
    ]
end_policy
```

8.4.1.2 DPP Semantics

Instances of the message specified in the **`distribute` clause**
are to be distributed to the members specified in the **`to` clause**
using the appropriate multicasting protocol which provides the ordering guarantees specified in the **`using` clause**;
in case of atomic ordered multicasting protocol, the message must be distributed to atleast the number of members specified in the **`distribution_cardinality` clause** or to none of them.

The `ordering specification` consists of ‘`UNORDERED`’, ‘`SOURCE_ORDERED`’, ‘`DESTINATION_ORDERED`’, ‘`ATOMIC_ORDERED`’ message ordering requirements.

8.4.2 Collation Policy Primitive

The GPP used in a collation policy specification is called a *collation policy primitive* (CPP). The following is the syntax and semantics of a CPP. The examples of CPPs are given in section 7.6 and in section 7.7.

8.4.2.1 CPP Syntax

```
collation_policy
for message name
    [
     deliver message specification
     from membership specification
     within | every time specification
     collation_cardinality cardinality specification
     collation_mode collation mode specification
    ]
end_policy
```

8.4.2.2 CPP Semantics

Instances of the message specified in the **`deliver` clause**
which are received from members specified in the **`from` clause**
within the time period specified in the **`within | every` clause**
are to be combined into a group message using the collation scheme specified in the **`collation_mode` clause**
only when their total number satisfies the **`collation_cardinality` clause**.

The CPP is composed of multiple GPSL elements enclosed within square brackets. As shown in examples in section 7.6, a reply collation policy may contain multiple CPPs which are joined together through *message ordering specifiers* described in section 8.5.7. The order in which (group) messages are delivered to the sink object is specified by these message ordering specifiers.

8.4.3 Synchronisation Policy Primitive

The GPP used in a synchronisation policy is called a *synchronisation policy primitive* (SPP). The following is the syntax and semantics of SPP. The examples of SPPs are given in section 7.9.

8.4.3.1 SPP Syntax

```
synchronisation_policy
for message name
```

```
sync message specification
with
    [
     (un |) solicited_reception_of message specification
     from membership specification
     within timing specification
     synchronisation_cardinality cardinality specification
    ]
notify
    [
     sync_events message specification
     to membership specification
    ]
end_policy
```

8.4.3.2 SPP Semantics

Synchronise the (*distribution* | *delivery*) of the message specified in the **`sync` clause**
(from | to) the group member
with the (un)solicited reception of the instances of synchronisation message specified in the
`(un |) solicited_reception_of` clause,
from the group members specified in the **`from` clause**,
only when the required number of synchronisation messages as specified in the **`synchronisation_cardinality` clause** are received within the time period specified in the **`within` clause**,
and notify the occurrence of the local synchronisation events specified in the "**`sync_events`**" clause to the group members specified in the **`to` clause**.

The SPP is composed of multiple GPSL elements enclosed within square brackets. As shown in examples in section 7.9, a synchronisation policy may contain multiple SPPs (one for each synchronisation message) which are joined together through *boolean operators* such as '**`and`**', '**`or`**', '**`not`**', '**`xor,`** etc.

8.4.4 Filtering Policy Primitive

The GPP used in the filtering policy specification is called the *filtering policy primitive* (FPP). The following is syntax and the semantics of the FPP. The examples of FPPs are given in section 7.10.

8.4.4.1 FPP Syntax

```
filtering_policy
for message name
    [
     filter message specification
     amongst membership specification
     filtering_cardinality cardinality specification
     filtering_criterion attribute expression
     filtering_properties attribute list
    ]
end_policy
```

8.4.4.2 FPP Semantics

Filter in the message specified in the `filter` **clause**
at m (specified in the client's `filter_cardinality` **clause**)
out of n (specified in the client's `amongst` **clause**) servers

1. which satisfy the client's filtering criterion (specified in the client's `filtering_criterion` **clause**) by their attributes (specified in the server's `filtering_properties` **clause**), at the time of evaluation
   (*I satisfy your criterion*), and
2. in which their own filtering criterion, if any (specified in the server's `filter_criterion` **clause**), is satisfied by the client's attributes (specified in the client's `filter_properties` **clause**),
   (*you satisfy my criterion*), and

if the required minimum number of servers (as specified in the `filtering_cardinality` **clause**) are not available, then do not filter in the message at any server.

8.5 Syntax and Semantics Of GPSL Elements

A policy specification in GPSL is a combination of basic language elements identified in section 8.3. The basic language elements are the message specifier, membership specifier, time specifier, cardinality specifier, combination mode specifier, attribute combination operators, and message ordering operators. In this section we present the syntax and the semantics of these language elements.

8.5.1 Message Specifier Elements

Message specification consist of the *message signature*. Every GPP specifies the (operation | notification | termination) signature, the instances of which are to be distributed, collated, synchronised or filtered.

8.5.2 Membership Specifier Elements

Membership specification is done in GPSL either by specifying member name, member role or group identifier. Member names, member roles and group identifiers are registered in the GSM. In case of termination distribution, the membership specification is done by the following specifiers:

a. **`SENDERS`**: If a single reply is generated by the server in response to a group operation, then this reply must be sent to those clients whose operation messages were included in the group operation.

b. **`SENDERS-IN-ROW-ORDER`**: If multiple replies are generated by the server object in response to a group operation, then the order in which the replies are generated is the same as the order in which the corresponding client operation messages were packed in the group message, hence the replies must be sent to the corresponding clients, in that order.

8.5.3 Cardinality Specifier Elements

The following cardinality specifiers are available in the GPSL for the specification of distribution cardinality, collation cardinality, synchronisation cardinality, and filtering cardinality:

a. **`ATLEAST(cardinal_expression)`**: This specifier means:
  - *distribution cardinality*: that the message be distributed to the specified minimum number of members or to none of the members of the sink group,
  - *collation cardinality*: that the instances of the message be received from specified minimum number

members before it can be collated into the group message for delivery to the sink object or the message be not delivered at all,

- *synchronisation cardinality*: that the instances of the message be received from the specified minimum number members before the message under synchronisation can be sent from the source object or delivered to the sink object.
- *filtering cardinality*: that the message be filtered in at the specified minimum number of members or be filtered at none of the members of the server group.

2. **`ATMOST(cardinal_expression)`**: This specifier means:

a. *distribution cardinality*: that the message be distributed to as many members as specified, but not exceeding the maximum specified.

b. *collation cardinality*: that the group message be collated from instances of the message received from as many members as specified, but not exceeding the maximum specified.

c. *synchronisation cardinality*: Not Applicable

d. *filtering cardinality*: that the message be filtered in at as many members as specified, but not exceeding the maximum specified.

where the,

cardinal_expression:= **`integer`** | **`POS(integer_list)`** | **`ANY(integer, role_name)`** | **`ANY(integer, POS(integer_list)`**), where

`POS(integer_list)`:= Pick the member(s) at the specified positions(s) in membership list specified in the (**to** | **from**) clause.

`ANY(integer, role_name)`:= Pick any ‘n’ members in the specified role.

`ANY(integer, POS(integer_list))`:= Pick any ‘n’ members from the specified positions.

3. **`UNSPECIFIED:`** The semantics of this specifier is equivalent to ATMOST(ALL), i.e., wait for the receipt of messages until the expiry of time specified in the “within” or “every” clause.

8.5.4 Time Specifier Elements

The following time specifiers are available in the GPSL for the specification of collation and synchronisation duration.

a. **`within`**: This specifier is used for specifying a maximum time limit for the receipt of termination messages in response to the operation message and for the receipt of synchronisation response messages in response to the synchronisation soliciting message. In case of termination collation, the timeout period starts after the *sending* of the corresponding operation message, and in case of synchronisation, it starts after the *sending* of the corresponding synchronisation soliciting message. The termination messages are delivered to the client object in the desired collation mode as soon as the required number of them as specified in the collation cardinality clause have been received, within the specified time limit.

2. **`every:`** This specifier is used for specifying the ‘periodic’ nature of collation of operation and notification messages required at the server object. The collation process starts and ends at the beginning and end of the collation period. The group (operation | notification) message is delivered to the server object only at the end of the collation period. Clocks are assumed synchronised to the precision required by the application.

The amount of time the C-Agent is required to wait before the delivery of the message to the sink object is based upon collation time and collation cardinality. This relationship is described in table 8.3 .

8.5.5 Combination Mode Specification Elements

The following collation operators are available in the GPSL corresponding to the *collation schemes* proposed in section 3.6. The "`SINGLETON`" mode corresponds to the delivery of individual messages without any collation. The semantics of these operators is specified in table 8.2 .

1. `MATRIX(SEQUENTIAL|ANY-ORDER, FIRST|RECENT|ALL)`
2. `LINEAR(FIRST|RECENT|ALL)`
3. `SINGLETON(SEQUENTIAL|ANY-ORDER, FIRST|RECENT|ALL)`

These operators specify the following aspects of the collation:

a. How to *construct* a group message from the component messages,

b. In what *order* to arrange the component messages in a group message, and

c. If multiple inputs are received from the same source object during a collation period, then how to handle the multiple inputs. The sink object may wish to include:

- **`All`** inputs received from a given source during the collation period in the group message.
- **`First`** input from a given source in the collated group message, and the subsequent inputs from that source are included in the subsequent collation periods for the construction of subsequent group messages.
- **`Recent`** input received from a given source during the collation period in the group message, while the earlier inputs are discarded. Other schemes are possible; they are not included.

8.5.6 Attribute Combination Specification Elements

The message filtering criterion is specified as an attribute expression, which is a boolean expression. The boolean operators, such as '**`and`**', '**`or`**', '**`xor`**', etc. are used in attribute expressions. The usual comparison operators such as '**<**', '**>**', '**==**', etc. are used for comparison between attributes. The boolean operators are also used for the conjunction of synchronisation policy primitives (see examples in section 7.9) in order to specify multiple synchronisation or alternative synchronisation requirements.

8.5.7 Message Ordering Specification Elements

Message ordering is a requirement in the *distribution* and *delivery* of messages. The distribution ordering requirements are specified as un-ordering, source ordering, atomic ordering, etc. They are satisfied by the choice of appropriate multicast protocols. The message delivery ordering requirements are specified using the following message ordering operators available in the GPSL to specify application-specific ordering of the delivery of received messages to the sink object.

a. **`followed_by`**: *previous message* **`followed_by`** *successor message*: The delivery of *previous message* is *followed by* the delivery of *successor message*, as soon as the required instances of the previous message is received from the specified source objects in the specified collation interval.

b. **`interleaved_with`**: *message-1* **`interleaved_with`** *message-2*: Two messages can be delivered in any order to the sink object as soon as they are scheduled for delivery (by the *collation mode*, *collation cardinality* or *collation duration* semantics).

c. **`disabled_by`**: *regular message* **`disabled_by`** *exception message*: The delivery of a *regular message* is disabled by the receipt of *exception message* once the required number of *exception messages* are received from the specified source objects in the specified collation interval, at which time only the *exception message* is delivered to the sink object, while the *regular message* discarded.

d. **`choice`**: *alternate-1* **`choice`** *alternate-2*: The *alternate-1* or *alternate-2* message is delivered to the sink object, whichever is scheduled first for delivery (using the collation mode, collation

Table 8.2: Semantics of Collation Operators

Collation mode	Component ordering	Number of components from a given source	Semantics
MATRIX	SEQUENTIAL	FIRST | RECENT | ALL	Collate the component messages in the matrix-mode in the order in which their respective source objects are specified in the **from** clause and include the (first|recent|all) message(s) from a given source object received during the collation period in the group message. In case of '**ALL**' constructor, all messages from a given source are combined in adjacent locations of the group message.
	ANY-ORDER	FIRST | RECENT | ALL	Collate the component messages from the source objects (specified in the **from clause**) in the order in which they are received, in the matrix-mode, and include the (**first|recent|all**) message(s) from a given **source object** received during the collation period in the group message.
LINEAR	Not Applicable	FIRST | RECENT	Collate the component messages from the source object (specified in the **from clause**) in the **linear mode** and include the (**first|recent**) message from a given source object received during the collation period in the group message. In *linear collation scheme*, the tuple parameters are arranged as specified in the *message signature*. '**ALL**' is an invalid constructor in the *linear collation scheme*.
SINGLE-TON	SEQUENTIAL	FIRST | RECENT | ALL	Do not combine the received messages. Deliver the (**first| recent| all**) message(s) received during a collation period from a given source object **individually** to the sink object, as **singleton** messages, in the order in which their respective source objects are specified in the **from clause**. In case of '**ALL**' and '**RECENT**' constructors, the message delivery to the **sink object** is delayed until the **collation_cardinality clause** is satisfied or until the collation duration expires (because the **C-Agent** does not know if more messages will arrive from a given **source object** while it is waiting for messages from other sources). In case of '**ALL**' constructor, all messages received from a given **source object** during a collation period are delivered sequentially to the sink object before starting the delivery of other messages from other source objects specified in the **from clause**.
	ANY-ORDER	FIRST | RECENT | ALL	Do not combine the received messages. Deliver the (**first| all**) message(s) received from a given source object **individually** to the sink object **as soon as it is received**. In case of '**RECENT**' constructor, the message delivery to the **sink object** is delayed until the **collation_cardinality clause** is satisfied or until the collation duration expires (because the **C-Agent** does not know if more messages will arrive from a given **source object** while it is waiting for messages from other sources). Once the **collation_cardinality** clause is satisfied or the collation duration expires, the most **recent** message received from each source object is delivered separately to the sink object in the order in which it arrived.

Table 8.3: Combined Semantics of Collation Time, Collation Cardinality, and Collation Mode

	ATLEAST	ATMOST	ATLEAST + ATMOST
within (used on the client side)	Wait until the receipt of specified minimum number of TER-messages or until the end of the collation period, whichever occurs first. If the specified minimum messages are not received during the collation period, give an **exception message** to the **client object**, otherwise give the specified minimum messages to the **client object** in the specified collation mode, **as soon as** they are received. Discard other messages. In case of **SINGLETON** delivery mode, the delivery of termination messages is delayed until the receipt of the specified minimum messages.	Wait until the receipt of specified maximum number of termination messages or until the end of the specified collation period, whichever occurs first. Give all messages received within the collation period to the client object in the specified collation mode. Discard other messages.	Wait until the specified minimum number of termination messages have been received, and if more collation period exists, then wait until the receipt of specified maximum number of messages or until the end of the collation period, whichever occurs first. If the required minimum messages are not received during the collation period, give an **exception message** to the **client object**, otherwise give the specified minimum and as many as received specified maximum messages to the **client object** in the specified collation mode. Discard other messages.
every (used on the server side)	Wait until the end of the specified collation period. If the specified minimum (OPR|NTF) messages are not received during the collation period, send an **exception message**[a] to the client objects from whom the messages were received, otherwise give all the received messages to the **server object** in the specified collation mode.	Wait until the end of the specified collation period, **and** give as many as received (OPR|NTF) messages to the **server object** in the specified collation mode. Discard other messages.	Wait until the end of the specified collation period. If the required minimum (OPR|NTF) messages are not received during the collation period, send an **exception message** to the client objects, otherwise give all messages received within the collation period to the **server object** in the specified collation mode.

---

[a] An **exception message** is not required if the received message is a **notification**.

cardinality and collation time semantics). In case of singleton delivery mode, the alternate message whose instance is received first is retained for delivery, while the other is dropped.

These operators are used to order the delivery of collated reply types to the client object, when multiple reply types or multiple instances of a given reply type are received in response to an operation invocation on the server group. A reply collation policy is composed of one or multiple collation policy primitives (CPPs) connected by these message ordering operators (see examples in section 7.6).

8.6 Conclusion

In this chapter we have introduced a language for expressing a broad family of group support policies. The language supports the specification of the requirements of message distribution, collation, synchronisation, and filtering.

CHAPTER 9 Inter-GSM Protocol

Abstract

The Group Support Machine (GSM) is a configuration of Group Support Agents (GSAs) which interact with each other locally via the inter-GSA interfaces and remotely through inter-GSM protocol for the provision of group support service. This chapter describes the remote communication between the peer GSAs located in different GSMs - the information that is exchanged between the GSAs, the format in which this information is exchanged, and the handshaking involved between the GSAs.

9.1 Introduction

The Group Support Machine (GSM) is a multi-agent machine. The Group Support Agents (GSAs) cannot offer their services independently in isolation. Instead, these agents need to communicate locally with other agents in the GSM, as well as remotely with their peers in other GSMs, in order to provide the required group support services to the application (client | server) components. In chapter 6, we have described the local interaction between the GSAs via the inter-GSA interfaces. This chapter describes the inter-GSM communication between the GSAs located in different machines using the *Inter-GSM Protocol* (IGP).

9.2 Why Protocol between GSMs

The first question that arises is why do we need a *communication protocol* between GSMs. The answer to this question is quite straight forward. The GSM is composed of different types of GSAs which perform specialised group support functions. The GSAs need to exchange their service specific information with their peer GSAs in order to perform their functions. The type of information that is exchanged between the GSAs depends upon the service offered by the GSAs, for example the filtering attributes and filtering criterion need to be exchanged between the F-Agents, the service-specific synchronisation signals need to be exchanged between the S-Agents, etc. Since this information is generated and consumed by the peer GSAs (and is not conveyed to the member), there is the need for *standardised interpretation* of information that is exchanged between them. A standardised syntax of the information and the associated semantics gives the standardised interpretation. Moreover the peer GSAs must have a consistent understanding of what information to expect in response to the information they have sent to their peers. This can be achieved by standardising the handshake procedure. These are the issues of a formal protocol between the GSMs.

9.3 Peer GSAs in Inter-GSM Protocol

Before we proceed to the description of the IGP, we need to identify which agent talks to which other agents through the IGP, in order to establish peer relationship between them.

1. *Peer of D-Agent and C-Agent*: The D-Agent performs the function of distributing (operation, notification | termination) messages on the (client | server) side, while the C-Agent performs the function of collecting these messages on the (server | client) side. Hence the C-Agent receives the messages sent by the D-Agent.
2. *Peer of S-Agent*: The S-Agents communicate with other S-Agents in the same group. The S-Agent plays one of two roles: the *synchronisation-seeker* or the *synchronisation-provider*. There is an exchange of message synchronisation related information between these two roles.
3. *Peer of F-Agent*: The F-Agent communicates with other F-Agents in the server group. The F-Agent plays one of two roles: the *contestant* or the *arbitrator*. There is an exchange of message filtering related information between these two roles.
4. *Peer of MM-Agent*: The MM-Agent communicates with other MM-Agents in the client and server groups. There is an exchange of group membership and member monitoring related information between the MM-Agents.

9.4 A General Format of the Inter-GSM Protocol Data Unit

The peer GSAs communicate with each other through the exchange of GSM protocol data units (GPDUs). Different types of information are exchanged between the peer agents. Before giving the details of the inter-GSM protocol, we start with identifying the type of information that needs to be exchanged between the peer GSAs in order to identify a general format of the GSM protocol data unit (GPDU), shown in figure 9.1.

Some types of information are always present in all GPDUs such as the:

1. *GPDU-type*: Every GPDU must be properly identified so that the P-Agent on the receiving side can give it to the appropriate GSA for processing.
2. *Sender-id*: The recipient GSA must know the identity of the sender of the GPDU in all peer-agent communication. The name of the sender of the GPDU is included in this field.
3. *Source-group-id*: The sender of the GPDU must also identify the group that it belongs to in order to uniquely identify a member in the client group and the server group.
4. *Message identifier*: Every operation and notification message invoked by the client object is uniquely identified locally by an *invocation instance identifier*, which is generated by the G-Agent. Every GPDU that carries an application message or the synchronisation and the filtering related information about that message carries the invocation instance identifier, as explained in following sections.

Some information types are exchanged in specific peer agent communication, such as:

1. *Information exchanged between D-Agent and C-Agent*: The D-Agent on the client side sends operation or notification messages and optionally filtering constraints associated with those messages. Optionally, the application-specific role of the sender of the GPDU is also sent to the recipient. Hence there is a need for a “payload” field, “group interaction constraints” field, and “membership-descriptor” field in the GPDUs exchanged between the D-Agent and the C-Agent.
2. *Information exchanged between S-Agents*: As shown in the example in section 7.9, the S-Agent either sends the message for which the synchronisation (permission) is being sought or the synchronisation-related information (in the form of a message) about that message which is identified by its invocation

instance identifier. In some cases, the members to whom the message can be distributed is also identified by the S-Agent. Hence there is a need for “payload” field to carry the message or the synchronisation-related information about that message, and “membership descriptor” field in the GPDUs exchanged between the S-Agents.

GPDU-Type	Sender-Group-Identifier	Sender-Identifier	Message Identifier	PAYLOAD	Group Membership-Descriptor	Group-Constraints

Fig. 9.1 A General Format of GSM Protocol Data Unit (GPDU)

3. *Information exchanged between F-Agents*: The contestant F-Agents send the filtering attributes and filtering criterion to the arbitrator F-Agent and the arbitrator F-Agent sends the outcome of the “m -out of-n selection process” in the form of a message (such as selected() | not-selected()) to the contestant agents. Hence there is a need for “payload” field to carry the result of selection process and “group constraints” field to carry filtering constraints in the GPDU.

9.5 Encoding of GPDUs

The GPDU is a place holder for the identification of information types that must be exchanged between the peer GSAs in the remote GSMs. The focus of the IGP is in the identification of the information types and the handshaking involved between the peer GSAs, and not on any specific encoding schemes for the reasons mentioned below. There are numerous encoding possible for the GPDU fields. One such scheme is employed in our Java implementation of IGP, and is described in section 10.2.4.

The first and the most important step in any protocol design is the identification of information types that need to be exchanged between the protocol partners and the handshaking (message exchanges) involved between them. The next step is the encoding of the information in the protocol data units. The type of encoding scheme to be employed for the coding of any protocol data unit is left open for bi-lateral agreements between the protocol partners and a specific choice to be made by protocol implementations. Moreover any explicit choice of the encoding scheme also restricts the usage of the IGP. For example, the “sender-id” field could be encoded as “octet” or an “integer” or a “fixed-size character string”, etc. The choice of one octet encoding would restrict the sender group size to 256. Hence, the choice of encoding scheme is left open for bi-lateral agreement between protocol partners.

The entire GPDU is essentially an application-level “complex data structure” consisting of multiple fields (just like a programming language data structure). The principles employed for the encoding of low-level protocol data units such as those of X.25 or ATM Frames, etc. with “byte-level” considerations and boundary alignments, etc. are not used at the application-level PDUs. The application-level protocol data units, such as X.400 messages, Corba Request Message, Reply Message, or the proposed GPDUs are specified using a high-level notation such as ASN.1, OMG Interface Definition Language (IDL), etc. There are compilers which translate the IDL specification of these high-level messages to “on the wire format” such as Common Data Representation (CDR) and then recover (unmarshal) the original message from the CDR format. These compilers take care of coding and de-coding issues such as byte alignments, variable field lengths, etc.

Here are some possible choices of IDL encoding that can be used for the GPDU fields:

1. *GPDU Type*: The GPDU type can be encoded as an IDL “octet” or “char” or an “enum” (i.e., enumerated data type.

2. *Sender Group Identifier*: This field can be encoded as IDL “octet”, “unsigned short”, or even as an “enum” (which will allow user defined restrictions on the choice of group identifiers).
3. *Sender Identifier*: The same choices as for “sender group identifier” field”. All the above mentioned fields are of fixed length.
4. *Payload*: This is a variable length field which contains application messages such as OPR-message, REP-message, etc. The first “l” bytes of this field define the total length of the rest of the GPDU and the next “p” bytes define the length of the rest of the payload field. The length of “p” and “l” is negotiated between protocol partners to accommodate the maximum possible required sizes of payload and of the GPDU. The payload itself contains programming language specific parameter data types such as boolean, integer, float, char, string, etc. They are encoded in the corresponding IDL types and are embedded in this field.
5. *Group Membership Descriptor*: describes the any “application-specific” role of the client or server group members, such as a group administrator, etc. This is again a variable length field, the first “m” bytes of the field define the length of the rest of this field. The length of “m” is negotiated between protocol partners to accommodate the maximum required size of this field. The role of client and server group members can be most naturally mapped into an IDL “enum” type or other possibilities such as an “octet” or “short” may also be used to represent a role.
6. *Group Constraints*: are described as a set of attribute-value pairs and/or a combination of those pairs. They are used as filter-constraints. Again, this is a variable length field; however, the length of this field can be deduced from the length of the previous fields. Each attribute is typed, such as an “integer” value attribute, a “float” value attribute, “boolean” value attribute, etc. They are encoded in the corresponding IDL types and are embedded in this field.

The entire IDL specification of the GPDU is then compiled or marshalled into “Common Data Representation”, which is the “on-the wire format”, and the individual fields are recovered into the original form at the receiving end using unmarshalling routines.

9.6 Inter-GSM Protocol between D-Agent and C-Agent

The D-Agent on the (client | server) side is responsible for the distribution of (operation, notification | termination) messages received from the (client | server) object. There is a unidirectional transfer of (operation, notification | termination) messages from the D-Agent on the (client | server) side to the C-Agent on the (server | client) side.

9.6.1 Application Message Communication between D-Agent & C-Agent

The G-Agent receives (operation, notification | termination) messages from the local (client | server) object. It adds “invocation instance identifier” to these messages in order to uniquely identify them. Then it gives the message and its identifier to the D-Agent. The D-Agent constructs the D-OPR-GPDU, the D-NTF-GPDU and the D-REP-GPDU, one each for the distribution of operation, notification, and termination message respectively (see figure 9.2). These GPDUs contain the respective message (in the payload field) and the invocation instance identifier associated with the message (in the message identifier field). The invocation instance identifier helps in the association of the reply messages with the corresponding operation messages, so that the C-Agent on the client side can separate the replies received in response to different operation messages from the server group.

The D-Agent also adds the (client | server) identifier in the “sender identifier” field and its group identifier in the “sender group identifier” field. If the (client | server) has an “application-specific” role, it is placed in the “group membership descriptor” field of the GPDU.

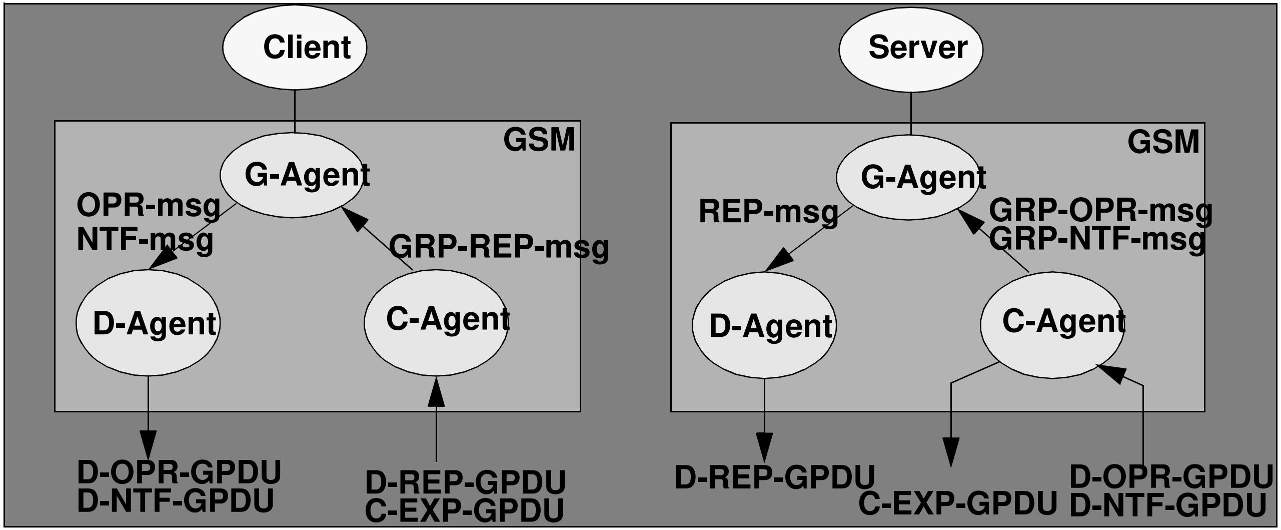


Fig. 9.2 Inter-GSM Protocol between D-Agent & C-Agent

The D-Agent on the client side also encapsulates the filtering attributes, the filtering criterion, and the filter cardinality, if any, in the D-(OPR | NTF)-GPDU. These filtering constraints are included in the GPDU in order to send both the message and the filtering constraints associated with it in the same GPDU. The filtering constraints are obtained from the local F-Agent, and are placed in the “group constraints” field of the GPDU. When filtering constraints are included, then these GPDUs are identified as “DF-OPR-GPDU” and “DF-NTF-GPDU”, in order to identify the presence of “group-interaction constraints” in the GPDU to the P-Agent on the server side. When this GPDU is received by the P-Agent on the server side, it strips the group constraint field, and gives the information contained in this field to the local F-Agent, along with a copy of the “message identifier” field. The rest of the GPDU is given to the local C-Agent for the unmarshalling of the encapsulated message and its subsequent collation.

9.6.2 Marshalling of Application Messages in GPDUs

The D-Agent receives (operation, notification | termination) messages from the local (client | server) object (via the G-Agent) as the message name followed by a series of zero or more parameter values. The D-Agent associates the received parameter values to their corresponding parameter names according to the message signature specified in the D-policy script, thereby constructing the <*parameter name*, *parameter value*> tuples. The message name along with these parameter tuples are carried in the “payload” field of the GPDU.

The application messages are marshalled (or encoded) as <parameter name, parameter value> tuples in order to handle the possible difference in message signatures on the client and server sides, and also to enable linear mode collation by the C-Agent on the other side. The C-Agent performs the unmarshalling of GPDUs sent by the D-Agent. The presence of <parameter name, parameter value> tuples in the GPDU allows the C-Agent on the receiving side to associate the parameter values to the corresponding parameter names before collating and invoking the message on the local (client | server) object.

9.6.3 Group Exception Handling Protocol Between C-Agents

The distribution of an (operation | notification) message from the client to the server group involves the activation of various GSAs on the client side and on the server side, in order to process the group-interaction constraints (such as distribution, collation, synchronisation, and filtering constraints), before the delivery of the message to the server objects. If the operation message is finally delivered to the server objects after successful group constraint processing by the GSAs, then multiple replies are received by the C-Agent on the client side from the D-Agents in the server group.

However when exception conditions arise in group constraint processing by the GSAs on the client side or on the server side, the message either cannot be distributed from the client's side or it cannot be delivered to the server objects. In case of group constraint processing exceptions on the client side such as synchronisation (or permission) to send a message not received, then the message is not distributed and a local exception termination message is returned to the client object by the involved GSA.

In case of group constraint processing exceptions on the server side, the operation message is not delivered to the server object and the appropriate group exception termination message is constructed by the C-Agent on the server side and it is sent to the C-Agent on the client side via the C-EXP-GPDU. The following group constraint processing exceptions are reported to the C-Agent on the client side by the C-Agents on the server side in the C-EXP-GPDU.

1. *Collation exceptions*: Many exceptions may be encountered during an operation collation process, such as an operation name or its parameter names are not recognised by the C-Agent, a minimum number of operation messages not received within the specified collation period, etc. The C-Agent on the server side constructs an exception termination message with appropriate reason parameters and sends it to the C-Agent on client side.
2. *Synchronisation exceptions*: If the synchronisation (or permission) to deliver a message is not received within the specified time interval from the specified group members, then the message cannot be delivered to the server object. The S-Agent on the server side informs the local C-Agent of the result of the synchronisation processing. The C-Agent discards the operation message and constructs an exception termination message with appropriate reason parameters and sends it to the C-Agent on client side. The reason parameters may convey application-specific details such as reason for disapproval, the identities of the group members who disapproved the delivery of the message, etc.
3. *Filtering exceptions*: Filter processing may encounter many exceptions such as a filtering criterion specified by the client or the server is not satisfied, client's filter attributes not recognised or insufficient on the server side, the server is not selected for service provision in "m -out of- n selection process", etc. In such cases, the message cannot be delivered to the server object. The F-Agent on the server side informs the local C-Agent of the result of the filter processing. The C-Agent discards the operation message and constructs an exception termination message with appropriate reason parameters and sends it to the C-Agent on client side.

The "payload" field of the C-EXP-GPDU contains the reason for the exception in the form of a message with appropriate reason parameters. The "message identifier" field contains the invocation instance identifier of the corresponding operation message, so that the C-Agent on the client side can associate the exception message with its corresponding operation message. All types of exceptions on the server side are delivered to the C-Agent on the client side by the C-Agent on the server side. This ensures that the C-Agent on the client side does not expect replies from the corresponding server objects.

9.7 Inter-GSM Protocol between Peer S-Agents

The S-Agents in the (client | server) group are responsible for synchronising the (distribution | delivery) of the message (from | to) the (client | server) object, with respect to other events in the group. The S-Agents in a group communicate with each other in order to perform this function. The S-Agent plays one of two roles: the *synchronisation-seeker* or the *synchronisation-provider*, as defined below.

1. *Synchronisation-seeker S-Agent*: It is the S-Agent associated with the (client | server) object which is required to seek the synchronisation or permission from other members of the (client | server) group in order to (send | receive) the message.

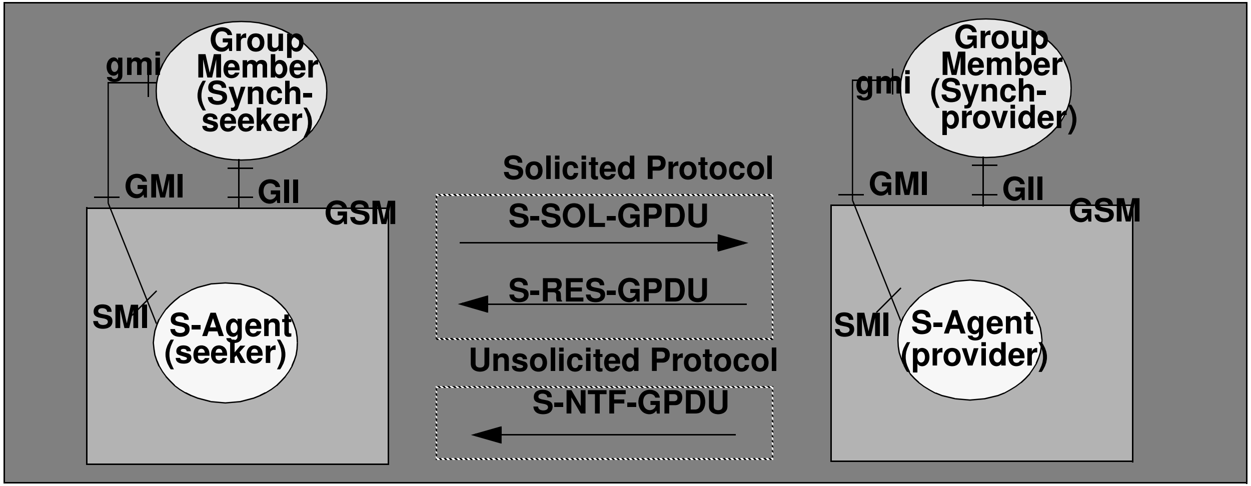


Fig. 9.3 Inter-GSM Protocol between S-Agents

2. *Synchronisation-provider S-Agent*: It is the S-Agent associated with the (client | server) object which provides the synchronisation-related information to the synchronisation (or permission) request from the synchronisation seeker (client | server) objects.

There is an exchange of synchronisation related information between these two roles, as shown in figure 9.3. The synchronisation-seeker S-Agent may either explicitly seek synchronisation (or permission) to distribute or deliver the message from the synchronisation-provider S-Agents or it may receive synchronisation-related information from the other synchronisation-provider S-Agents in an unsolicited manner. These two modes of synchronisation process are explained below.

9.7.1 Solicited Synchronisation Protocol

In applications which require *synchronised message distribution*, when an operation or notification message is received from the client object by the G-Agent, it gives the message both to the D-Agent and to the S-Agent. The D-Agent does not *distribute* the message until an appropriate synchronisation signal (permission) is received from the local S-Agent. Similarly, in applications which require *synchronised message delivery*, when a D-OPR-GPDU or a D-NTF-GPDU is received from the network by the P-Agent, it gives the GPDU both to the C-Agent as well as to the S-Agent. The C-Agent does not *deliver* the message to the server object, until an appropriate synchronisation signal (permission) is received from the local S-Agent.

In the solicited synchronisation protocol, the synchronisation-seeker S-Agent explicitly seeks the permission of the synchronisation-provider S-Agents, in order to authorize the distribution or delivery of the application message, by sending the S-SOL-GPDU to all the synchronisation provider S-Agents as specified in its S-policy script, and then waits for a specified time period for the receipt of S-RES-GPDUs from the synchronisation-provider S-Agents. On receipt of the S-SOL-GPDU, the synchronisation-provider S-Agents respond with their synchronisation-related information, such as the permission to (distributed | deliver) the message granted or denied, etc., by sending S-RES-GPDU to the synchronisation seeker S-Agent. The synchronisation-provider S-Agent may give the response to the synchronisation request based upon some policy or after consulting with the group member via the SMI and gmi, as shown in figure 9.3.

The S-SOL-GPDU contains the following information, apart from other sender identification information:

1. *payload-field*: a copy of the operation or notification message which is to be (distributed | delivered),
2. *message-identifier-field*: the invocation instance identifier associated with the message, and
3. *group-membership descriptor-field*: sender's application-specific role,
4. *group constraint field*: optionally, the identities of the (server group members | client object) (to | from) whom the message is (to be distributed | received).

This information is used by the synchronisation-provider S-Agent to decide whether to grant the requested synchronisation or not. The S-RES-GPDU contains the following information, apart from sender identification information:

1. *payload-field*: a response to the synchronisation request in the form of a messages, such as "synchronisation_request_approved()", "synchronisation_request_denied()", etc.
2. *message-identifier-field*: contains the invocation instance identifier which was present in the corresponding S-SOL-GPDU,
3. *group-membership-descriptor-field*: sender's application-specific role,
4. *group constraint field*: optionally, the identities of the server group members to whom the message can be distributed, this field is a sub-set of the corresponding field in the S-SOL-GPDU.

9.7.2 Unsolicited Synchronisation Protocol

In the unsolicited protocol, the synchronisation seeker S-Agent does not explicitly seek the permission of the synchronisation-provider S-Agents, instead it implicitly waits for the receipt of a synchronisation notification message from the synchronisation-provider S-Agents, in order to authorise the distribution of an (operation | notification) message from a client site.

In this protocol, the S-Agent switches its role between synchronisation-seeker and synchronisation-provider, depending upon whether it is required to wait for the receipt of synchronisation notification messages or whether it is eligible to send such messages.

An example of unsolicited synchronisation process in client group is given in the example in section 7.9.5 and illustrated in figure 7.25. In this application, the members of the client group (tester agents) send messages to the server group in a synchronised manner. Let us consider a subset of this application in order to demonstrate the unsolicited synchronisation protocol. The message "`test-A(`a_1`, `a_2`, `a_3`)`" from the tester agent TA-1 can be distributed to the server group, only when a permission is received from the TAdmin. This permission is received through a S-NTF-GPDU. Moreover this message should only be distributed to those server group members who have successfully passed the test "`init_test()`" of the TAdmin. This information is also present in the S-NTF-GPDU. In this instance, the S-Agent (in the GSM) associated with the TA-1 is the synchronisation seeker S-Agent whereas the one associated with the TAdmin is the synchronisation provider S-Agent.

When the replies corresponding to the "`init_test()`" operation message are received, the TAdmin analyses these replies and gives a "pass" or "fail" verdict for each reply, based upon its application-specific criterion, to its local S-Agent via the "gmi" and "SMI" as shown in figure 9.3. The S-Agent converts the reply identifiers into the corresponding server group member identifiers (see section 7.9.7) and constructs two messages, "passed(member_list)" and "failed(member_list)". These messages are sent in the "payload" field of the S-NTF-GPDU (see figure 9.4) to the synchronisation-seeker S-Agents as specified in its S-policy script.

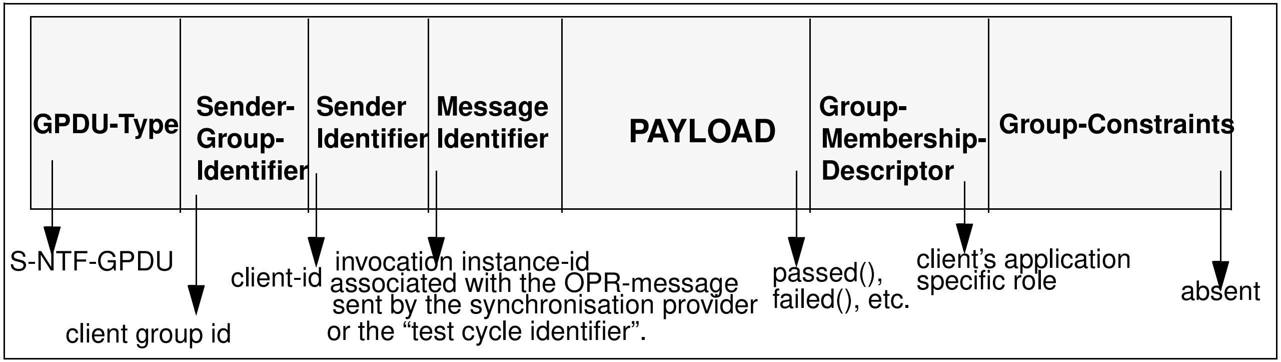

GPDU-Type
Sender-Group-Identifier
Sender Identifier
Message Identifier
PAYLOAD
Group-Membership-Descriptor
Group-Constraints
S-NTF-GPDU
client group id
client-id
invocation instance-id associated with the OPR-message sent by the synchronisation provider or the “test cycle identifier”.
passed(), failed(), etc.
client’s application specific role
absent

Fig. 9.4 S-NTF-GPDU Format

When the S-Agent associated with the TA-1 receives this S-NTF-GPDU, it informs the local D-Agent to distribute the message “test-A(a_1, a_2, a_3)” to the members who have passed the previous test, and then it assumes the role of synchronisation provider S-Agent (to communicate the pass and fail verdict from TA-1 to other members of the client group).

In general, the S-NTF-GPDU notifies the occurrence of certain synchronisation events at its sender, such as the delivery of an operation message to the server group, or the receipt of all the replies from server group, or the pass / fail verdict, to the synchronisation-seeker S-Agents. The notification of the occurrence of these events may be received from the local D-Agent, or local C-Agent, or from the associated member object, via the gmi and SMI as shown in figure 9.3. The S-NTF-GPDU contains the following information (apart from other standard fields):

1. *payload-field*: The payload field of the S-NTF-GPDU contains certain application specific synchronisation related information, such as “passed(member_list)”, “failed(member_list)”, “reply_not_received(member_list)”, etc. (see examples of these cases in section 7.9.6), which is required by the synchronisation seeker S-Agents in the group.
2. *message-identifier-field*: This field contains the invocation instance identifier of the operation or notification message which has been sent by the previous client (synchronisation provider). In the above example, this field represents a “test cycle identifier”.

9.8 Inter-GSM Protocol between Peer F-Agents

The F-Agents in the server group are responsible for filtering the (operation or notification) messages and performing “m- out of- n selection” of servers, before authorising the delivery of the message to the server objects. Filtering of messages based upon client and server’s filter attributes and filter criterion occurs locally, while “m- out of-n selection” requires inter-GSM protocol between F-Agents. It may be noted that there is no need for filtering of the termination messages in the client group. The F-Agent plays one of the two roles: the *contestant* or the *arbitrator*, as defined below.

1. *Contestant F-Agent*: It is the F-Agent associated with the server object which is required to satisfy the filtering criterion specified by the client object, before the delivery of message.
2. *Arbitrator F-Agent*: An arbitrator F-Agent may be chosen from amongst the contestant F-Agents or an F-Agent associated with a special member of the group, such as a Group Administrator, may be designated as an arbitrator F-Agent. The function of the arbitrator F-Agent, in an “m -out of- n selection” process is to select ‘m’ servers out of ‘n’ according to certain filtering criteria.

An example of the filtering process is given in section 7.10.3 and illustrated in figure 7.37. In this example a client sends it print request, “`print_request(data)`”, to all print-servers, which are organised as a *printer group*. This request is distributed to all the members of the printer group through the D-OPR-GPDU. The client’s filter criterion, filter attributes, and filter cardinality (specified in figure 7.35) are stored in the “group constraints” field of D-OPR-GPDU. However the print request should be delivered to only one member of the group which best satisfies the client’s filtering criterion.

When the P-Agents in the server group receive the D-OPR-GPDU, they strip the “group constraint” field and give it to the local F-Agent along with a copy of the message identifier field and give the rest of the GPDU to the local C-Agent. The C-Agent does not deliver the message in the payload field (“`print_request(data)`”) until the filtering process is completed by the F-Agent and the permission to deliver the message is received from it.

The filtering process consists of filter criterion evaluation which is performed locally and “m -out of -n selection” process which requires the inter-GSM protocol between F-Agents. The filter criterion evaluation ensures that the client’s filter criterion specified in figure 7.35 (for example cost per page < 5 cents, printer quality = color, etc.) is satisfied by the print server and the print server’s filter criterion specified in figure 7.36 (client is a registered user, client has sufficient funds in his account) is satisfied by the client.

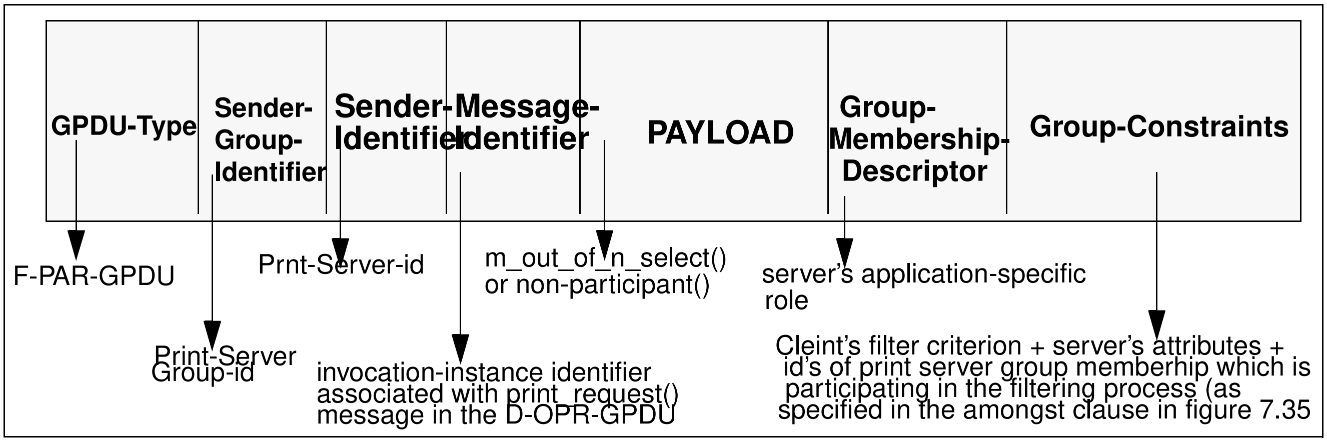

GPDU-Type
Sender-Group-Identifier
Sender-Identifier
Message-Identifier
PAYLOAD
Group-Membership-Descriptor
Group-Constraints
F-PAR-GPDU
Prnt-Server-id
m_out_of_n_select()
or non-participant()
server’s application-specific role
Print-Server Group-id
invocation-instance identifier associated with print_request() message in the D-OPR-GPDU
Cleint’s filter criterion + server’s attributes + id’s of print server group memberhip which is participating in the filtering process (as specified in the amongst clause in figure 7.35

Fig. 9.5 F-PAR-GPDU Format

When both the client’s and server’s filter criterion are satisfied, the F-Agents enter into an “m -out of -n selection” process in order to select a single best print server (the one with the minimum queue length, which offers service at lowest cost, etc.). Even if a server cannot satisfy the client’s filter criterion, it enters into an “m -out -of -n selection” process by sending a “non_participant()” message in the payload field of F-PAR-GPDU (see figure 9.5), in order to ensure that the arbitrator F-Agent receives bids from all the members of the print server group before it starts the “m -out of - n selection” process. These F-Agents are contestant F-Agents. A single F-Agent, usually associated with the (GSM of) Group Administrator is assigned the role of an arbitrator F-Agent. The “m -out of- n selection” of servers is usually required when a client specifies a certain (minimum or maximum) number of servers that must handle its service request.

The contestant F-Agents explicitly seek the arbitration of the arbitrator F-Agent by submitting their bids to it, in order to find if they are the losers or the winners in the “m -out of-n selection process”. So the contestant F-Agents participate in the “m -out of- n selection” process by sending the client’s filtering criterion and the associated server’s attributes in F-PAR-GPDU (see figure 9.5) to the arbitrator F-Agent. The identities of the print server group members amongst which the “m -out of- n selection” is to be done is also sent in the “group constraints” field of the F-PAR-GPDU. This information is obtained from the corresponding field in the D-OPR-GPDU in which the original “print_request()” was encapsulated.

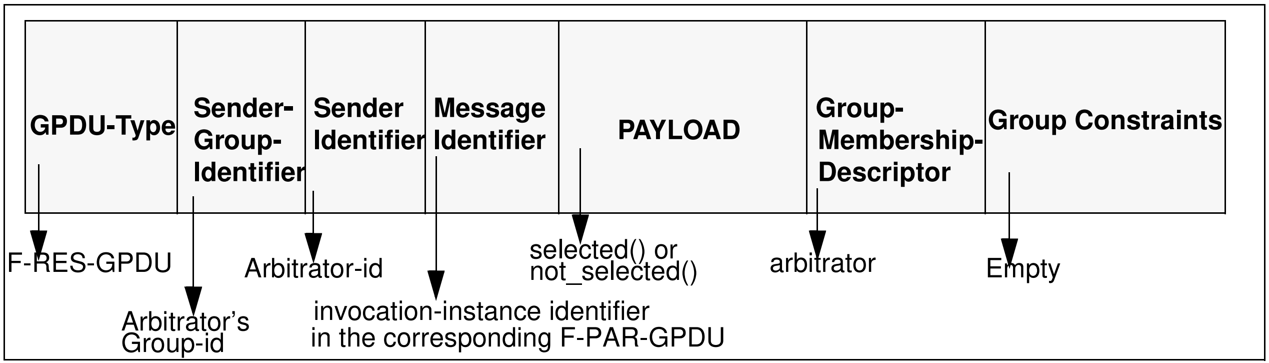

GPDU-Type
Sender-Group-Identifier
Sender Identifier
Message Identifier
PAYLOAD
Group-Membership-Descriptor
Group Constraints
F-RES-GPDU
Arbitrator's Group-id
Arbitrator-id
invocation-instance identifier in the corresponding F-PAR-GPDU
selected() or not_selected()
arbitrator
Empty

Fig. 9.6 F-RES-GPDU Format

The arbitrator F-Agent waits for a certain specified period of time for the receipt of F-PAR-GPDUs from all the F-Agents in the group (i.e., those specified in the “group constraint” field of the F-PAR-GPDU). If it receives the F-PAR-GPDUs from all the contestants within the specified time interval, then it starts the selection process immediately. It selects the ‘m’ servers that best satisfy the client’s filtering criterion, from amongst the bids that it has received, within a certain pre-specified time interval. Alternatively, in case of a complex filtering policy, the arbitrator F-Agent may consult its associated member, via the FMI and gmi, as shown in figure 9.7, in order for the member to perform the selection. Finally, the F-Agent informs the result of the selection process, such as “selected()” or “not_selected() to all the members of the group by sending the F-RES-GPDU (see figure 9.6). If an old F-PAR-GPDU is detected, it is ignored by the arbitrator F-Agent. If ‘m’ or no servers should do the job, then the F-RES-GPDU is sent to the selected contestants using an atomic broadcast protocol.

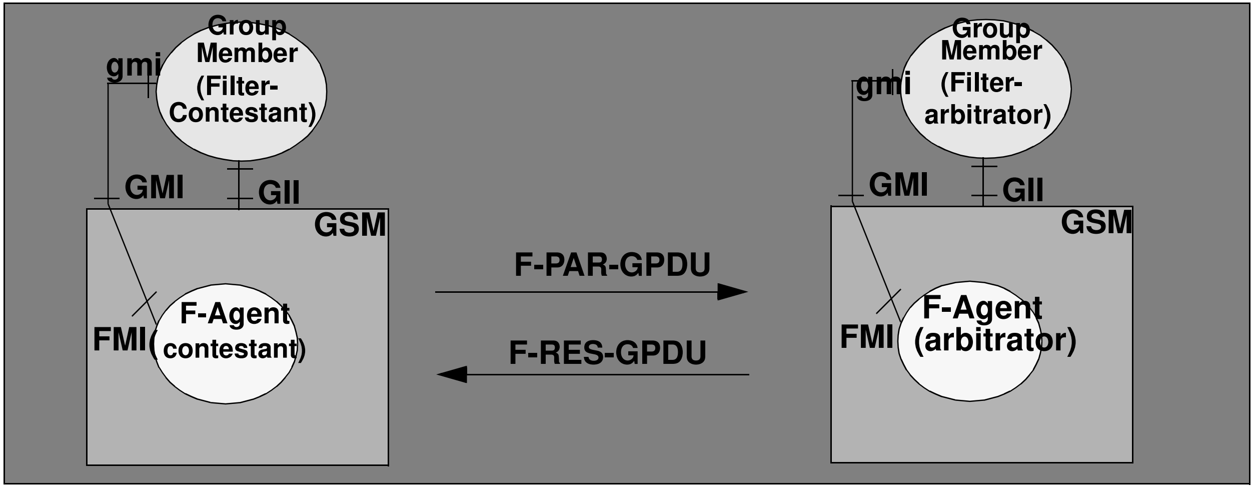

Group Member (Filter-Contestant)
gmi
GMI
GII
GSM
FMI
F-Agent (contestant)
F-PAR-GPDU
F-RES-GPDU
Group Member (Filter-arbitrator)
gmi
GMI
GII
GSM
FMI
F-Agent (arbitrator)

Fig. 9.7 Inter-GSM Protocol between F-Agents

9.9 Inter-GSM Protocol between Peer MM-Agents

An object group is a dynamic entity. New members may be added to the group, existing members may leave the group, or there may be link failures or member failures, leading to a change in the membership of the group. These events have an impact on the functioning of the GSAs, which depend upon the current group membership information in order to perform their respective functions. For example, if there is an addition to the membership of the server group, it should be notified to the D-Agents and C-Agents of the client group so that (operation | notification) messages are also sent to the new member, and reply messages expected from new members. Similarly if there is a removal or failure of an existing member of the server group, it should be notified to the C-Agent of the client group so that it does not expect replies from a deleted member. Any change in the membership of the (client | server) group should also be notified to the S-Agents of that group so that they may seek the synchronisation of the appropriate members of the group.

The group management aspects are beyond the scope of the thesis. It is not the intention of the thesis to present an elaborate membership management protocol. Such protocols have been extensively described in the literature, such as virtual synchrony [100 - 103], etc. Our aim is only to deal with those membership management aspects which have an impact on the functioning of the GSAs. The MM-Agent introduced in chapter 5, performs the *minimum* membership management functions required to support the other GSAs, such as monitoring the membership of the group, notifying the member failures, and receiving membership change notifications from other MM-Agents and the group administrator. The MM-Agent then notifies these membership changes to the local GSAs.

9.9.1 Distributed Membership Monitoring

Group membership needs to be continually monitored in order to detect member failures due to node or link failures. There are many schemes for monitoring the ‘liveliness’ of group members. They range from *centralised monitoring schemes* to *distributed monitoring schemes*. In centralised scheme, a special member of the group, usually in the role of a *group administrator*, is responsible for membership monitoring. The MM-Agent associated with the group administrator sends “probe messages” to the other MM-Agents (associated with other group members) which respond to these probes with “i am alive messages”. When a response is not received within a specific time period (or after specific number of retries), the administrator’s MM-Agent sends a “member failure(member_id)” notification to the rest of the MM-Agents. However, this scheme suffers the disadvantage of single point of failure. Hence we have adopted a distributed monitoring scheme.

In distributed monitoring scheme, the management responsibility is distributed amongst all MM-Agents. Each MM-Agent is responsible for sending the probe messages and notifying member failures to others. There are many distributed monitoring protocols, one of them is explained below. The MM-Agents of the object group are organised as a logical ring, with a predecessor and successor assigned to each member. Each MM-Agent is responsible for monitoring its predecessor and informing its own ‘liveness’ to its successor. Hence each member periodically sends an “i_am_alive()” message to its successor in the M-NTF-GPDU. If this message is not received from a predecessor, within a specified time interval, the member failure notification is sent to all the members of the group, including the suspected failed member using M-NTF-GPDU, with the “member_failure(group_id, member_id)” message in the payload field. When a member is restarted or receives its own failure notification, it seeks re-entry into the group through the *group administrator* (see next section).

9.9.2 Membership Change Notification

In many groups, the member addition and member removal is coordinated through a special group member in the role of a *group administrator*, which admits new members or gives permission to the existing members for leaving the group based upon application-specific membership management policies. Once it decides to join a new member to the group or to delete a member from the group, it notifies its local MM-

Agent. The MM-Agent then broadcasts any membership change notifications to all the MM-Agents of the group, through the MM-NTF-GPDU. The payload field of this message contains the “add_member(group_id, member_id, member_role, member_address, predecessor, successor)” or the “delete_member(group_id, member_id)” message, as appropriate. When a new member is added, its successor and predecessor are also assigned, so that membership monitoring can proceed, as discussed above.

The MM-Agent plays the role of monitoring, disseminating and receiving the management information pertaining to the group through M-NTF-GPDUs, and of informing any membership changes to the local GSAs. The M-NTF-GPDU contains the following information:

1. *message-identifier field*: This field identifies the number of invocation of the message of a given type, for example, “this is the 5th “i_am_alive()” message from me” or “this is the 7th “add_member()” message from me so that an MM-Agent knows if it has missed any message.
2. *payload field*: This field contains the message which is to be sent to other MM-Agent(s), such as
    a. i_am_alive(),
    b. member_failure(group_id, member_id),
    c. add_member(group_id, member_id, member_role, member_address, predecessor, successor)
    d. delete_member(group_id, member_id)
3. *membership-descriptor field*: This field contains the senders application specific role.
4. *group-constraints field*: This field is absent.

A summary of the GPDUs is given in table 9.1

Table 9.1: A Catalogue of GPDUs

GPDU Type	Sender	Receiver(s)	PAYLOAD	Membership Descriptor	Group Constraints
D-OPR-GPDU	D-Agent on client side	C-Agents in server group	OPR-message (received from client)	Client’s application-specific role	client’s filter attributes, criterion, and cardinality
D-NTF-GPDU	D-Agent on client side	C-Agents in server group	NTF-message (received from client)	Client’s application specific role	client’s filter attributes, criterion, and cardinality
D-REP-GPDU	D-Agent on server side	C-Agents in client group	REP-message (received from server)	Server’s application specific role	absent
C-EXP-GPDU	C-Agenton server side	C--Agent(s) in client group	Exception message	Server’s application specific role	absent
S-SOL-GPDU	Synchronisation seeker S-Agent	Synchronisation provider S-Agents	OPR | NTF-message	Sender’s application specific role	optional (see section 9.7.1)
S-RES-GPDU	Synchronisation provider S-Agent	Synchronisation seeker S-Agent	sync-request-approved or sync-request-denied	Sender’s application specific role	optional (see section 9.7.1)

Table 9.1: A Catalogue of GPDUs

GPDU Type	Sender	Receiver(s)	PAYLOAD	Membership Descriptor	Group Constraints
S-NTF-GPDU	Synchronisation provider S-Agent	Synchronisation seeker S-Agent(s)	synchronisation events, such as passed(member_list), failed(member_list).	Sender's application specific role	absent
F-PAR-GPDU	Contestant F-Agent	Arbitrator F-Agent	m_out_of_n_select()	Sender's application specific role	client's filter criterion + server's attributes + contestant id's.
F-RES-GPDU	Arbitrator F-Agent	Contestant F-Agents	selected() or not_selected()	arbitrator	absent
M-NTF-GPDU	MM-Agent	MM-Agent	i_am_alive(), member_failure(), add_member(), etc.	Sender's application specific role	absent

9.10 Inter-GSM Protocol over Multicasting Protocol

The Inter-GSM protocol (IGP) enables the peer GSAs to communicate with each other using standardised exchange formats, and it defines the handshaking between the peer GSAs. However, as shown in figure 9.8, the IGP is supported by the underlying multicasting protocols in the Group Communication Layer (GCL), i.e., the GPDUs that are exchanged between the GSAs are actually carried by the underlying multicasting protocols. These GPDUs are encapsulated in the "payload" field of the multicast protocols and transparently carried to the other GSMs.

9.10.1 Group Communication Layer

As shown in figure 9.8, the GSM is supported by Group Communication Layer (GCL). The GCL is composed of different types of Multicast Protocol-Objects or MP-Objects. Each MP-Object supports a different class of multicast protocol, such as unordered multicast protocol, source-ordered multicast protocol, causal-ordered multicast protocol, atomic-ordered multicast protocol, etc. The GCL provides the following services to the IGP:

1. *Message delivery service*: The multicast protocols provide the low-level message delivery services and takes care of failure handling, retransmissions, etc.
2. *Message ordering service*: The multicast protocols provides different types of message delivery ordering such as source-ordering, destination-ordering, casual-ordering, etc.
3. *Resilience guarantees*: The multicast protocols provide message delivery guarantees such as atomic delivery (including same ordered delivery) at all destinations.

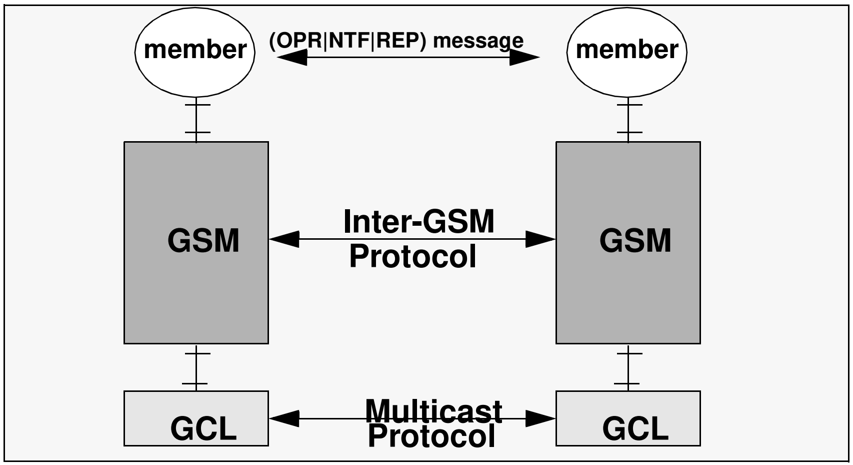


Fig. 9.8 Inter-GSM Protocol over Multicast Protocol

9.10.2 GSM - GCL Interface

The P-Agent of the GSM interfaces with the GCL. The P-Agent receives the GPDUs from the local GSAs and gives them to the appropriate MP-Objects for delivery to other GSMs. Similarly it receives the GPDUs from the MP-Objects and gives them to the appropriate local GSA based upon the information in the "GPDU type" field.

The P-Agent gives the GPDU and the list of its recipients to the appropriate MP-Object, by invoking a standardised service primitive: "`multicast(this_GPDU, to_these_members)`". Similarly, the MP-Object delivers the GPDU to the P-Agent by invoking a standardised primitive on the local P-Agent: "`receive(this_GPDU)`".

9.11 Conclusion

This chapter has described the Inter-GSM protocol (IGP) between the peer GSAs. This protocol enables the communication between the peer GSAs using a standardised format. The information content of the GPDUs and the handshaking involved between the peer GSAs in order to perform their respective functions is described. Some possible encoding schemes for GPDU fields are outlined. The IGP is supported by a set of underlying multicast protocols which provide different message delivery and ordering guarantees.

CHAPTER 10

Group Support Platform: Implementation and Performance

Abstract

This chapter describes the implementation details and performance aspects of the Group Support Platform. A partial model of the Group Support Machine involving the G-Agent, D-Agent, C-Agent, and P-Agent was implemented in the Java programming language. The aim of this implementation exercise is to validate the abstract model and the protocol, proposed in the previous chapters, experimentally and to gain insight into the performance aspects of the model.

10.1 Introduction

The proposed model of the Group Support Machine (GSM) is implemented in the object-oriented programming language Java. The object-oriented features of Java map naturally into the object-oriented basis of the proposed model. The language also supports some of the advanced object-oriented features such as, *classes*, *inheritance*, *exceptions*, *method overloading*, *multi-threaded programming*, *object serialization*, *remote method invocation*, *thread synchronization* and *buffered input/output* facilities, required for the straightforward implementation of the model.

The aim of this exercise is to widen the scope of the thesis in order to implement the abstract models presented previously.

10.2 Implementation Details

The implementation reported in this chapter is carried out in Symantec's VisualCafe Professional Java Development Environment (Ver. 3.0) running on Microsoft Windows 95. VisualCafe supports JDK 1.1.

We have implemented the G-Agent, D-Agent, C-Agent, and P-Agent of the GSM, as highlighted in figure 10.1. The implementation consists of approximately 1400 lines of Java code. In the following sub-sections we describe in detail the implementation of each aspect of the model. We describe implementation requirements for each aspect, the corresponding Java language features, and how the requirements can be supported by one or a combination of Java features.

10.2.1 Implementation of GSM Agents

Each GSM Agent, such as the D-Agent, C-Agent, etc., encapsulates a distinct functionality and as such corresponds to a separate "functional module" or an "object" in an object-oriented paradigm. Hence the GSM agents are implemented as an *instance* of a separate Java *class*. Therefore, the GSM consists of a *G_Agent class*, *D-Agent class*, *C_Agent class* and a *P_Agent class,* as shown in figure 10.2.

10.2.1.1 GSM Class

The GSM class serves as a GSM agents instantiation and initialization class. It creates the instances of GSM Agent classes such as the *D_Agent class*, *G_Agent class*, etc. It contains the Java *main()* method, the starting point of any program execution.

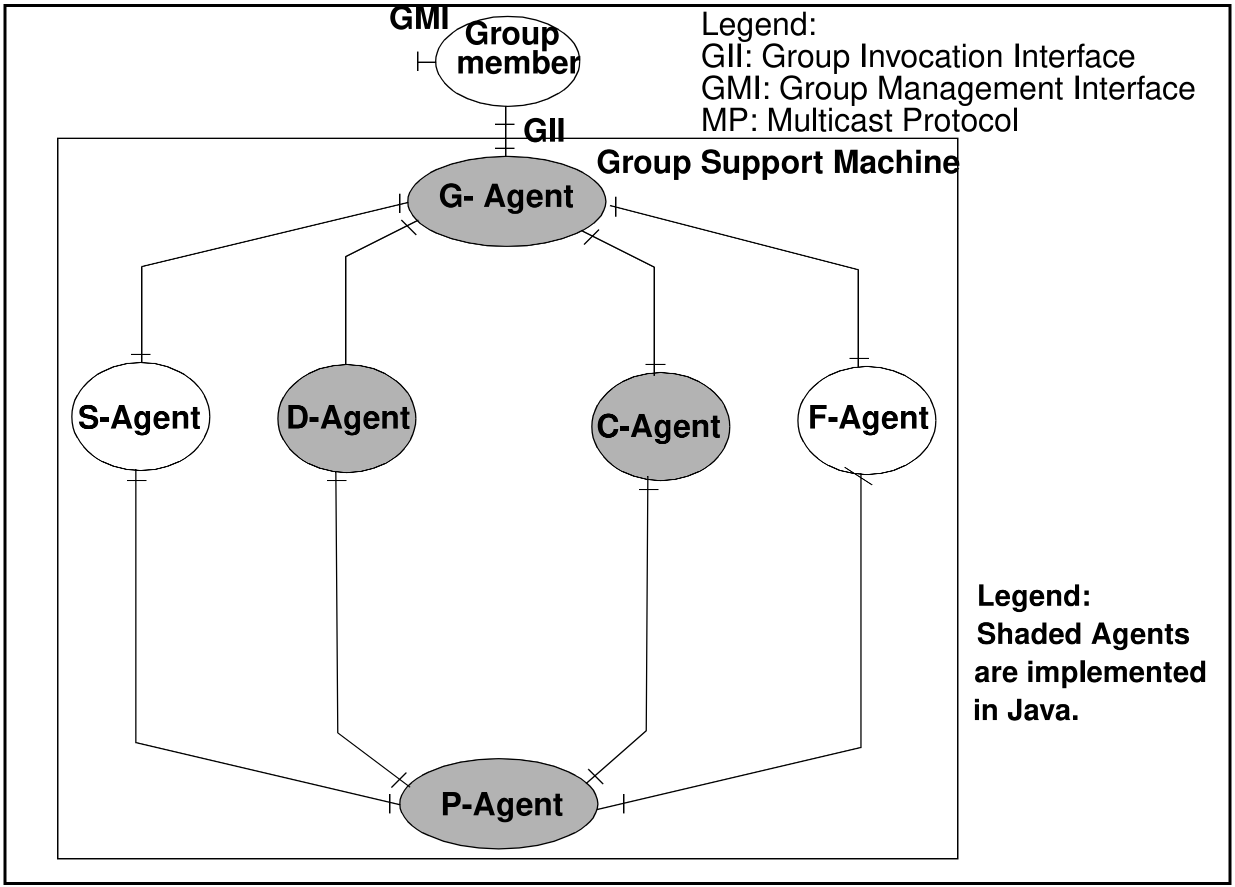

GMI
Group member
GII
Legend:
GII: Group Invocation Interface
GMI: Group Management Interface
MP: Multicast Protocol
Group Support Machine
G- Agent
S-Agent
D-Agent
C-Agent
F-Agent
P-Agent
Legend:
Shaded Agents are implemented in Java.

Fig. 10.1 Which Agents are implemented

The GSM agents need to communicate with each other in order to perform their respective functions. Hence they should know the identity of other local GSM agent in order to invoke methods on them. The *GSM class* passes the *object reference* of the instance of each GSM agent to every GSM agent. This is done by invoking the method `gsmObjectRefs(G_Agent, D_Agent, C_Agent, P_Agent)` on each GSM agent.

The GSM class inputs the group-id and member-id of the associated group member (yet another class) and unitizes the local GSM agents with this information.

The GSM class also invokes the policy programming interfaces of the D-Agent and the C-Agent, by invoking `init()` method on them, in order to input the respective distribution and collation policy from the user.

10.2.1.2 G_Agent Class

The *G_Agent class* implements the equivalent of the client-side stubs and the server side skeletons found in the conventional middleware platforms such as Corba. In the current implementation, these stubs and skeleton are hard-coded with the appropriate invocation handlers to handle a selected set of application messages, due to the lack of automatic stub and skeleton generation capability.

On the client side, the *G_Agent class* contains a hard coded stub for each application message (an OPR-message). The stub accepts the OPR-message invocations from local client application, generates a unique invocation instance identifier, and gives the message along with the invocation instance identifier to the *D_Agent class* for distribution, by invoking `gd_distribute_message(message, messageId)` on D_Agent. If the G_Agent is programmed for *solicited reply delivery*, the stub returns a *reply handle* immediately to the client, thereby unblocking the client. (The client can then issue *poll_reply()* later on whenever a reply is required, see section 10.2.6). If the G_Agent is programmed for an *unsolicited reply delivery*, the stub executes a java thread synchronization method, *wait()*, thereby blocking the client until the receipt of a group reply from the *C_Agent class*.

On the server side, the `cg_deliver_message()` method of the *G_Agent class* acts as a skeleton. It contains the appropriate invocation handlers for each OPR-message supported by the local server object. These invocation handlers receive the OPR-message and the associated invocation instance identifier from the local *C_Agent class*. They withhold the invocation instance identifier and deliver the OPR-message to the local server application and are blocked until the receipt of the reply from the application. When a reply is received from the server application, the invocation handlers send the reply along with the corresponding invocation instance identifier to the *D_Agent class* for distribution to the appropriate client(s), by invoking `gd_distribute_message(message, messageId)` on D_Agent.

The G_Agent class supports the GSM Invocation Interface (GII) and hence supports a group interrogation API with *solicited*, *unsolicited,* and *terminable* reply delivery semantics. This is discussed in detail in section 10.2.6.

10.2.1.3 D_Agent Class

The *D_Agent class* implements the policy-based distribution of the OPR and REP-messages received from the *G_Agent class.* It contains methods to input and store the message distribution policy from the user, to perform GPDU coding and construction, and its delivery to the *P_Agent class* for distribution using an appropriate low-level protocol. As mentioned in section 10.2.3, we use Java RMI to transport the GPDU payload between the GSMs.

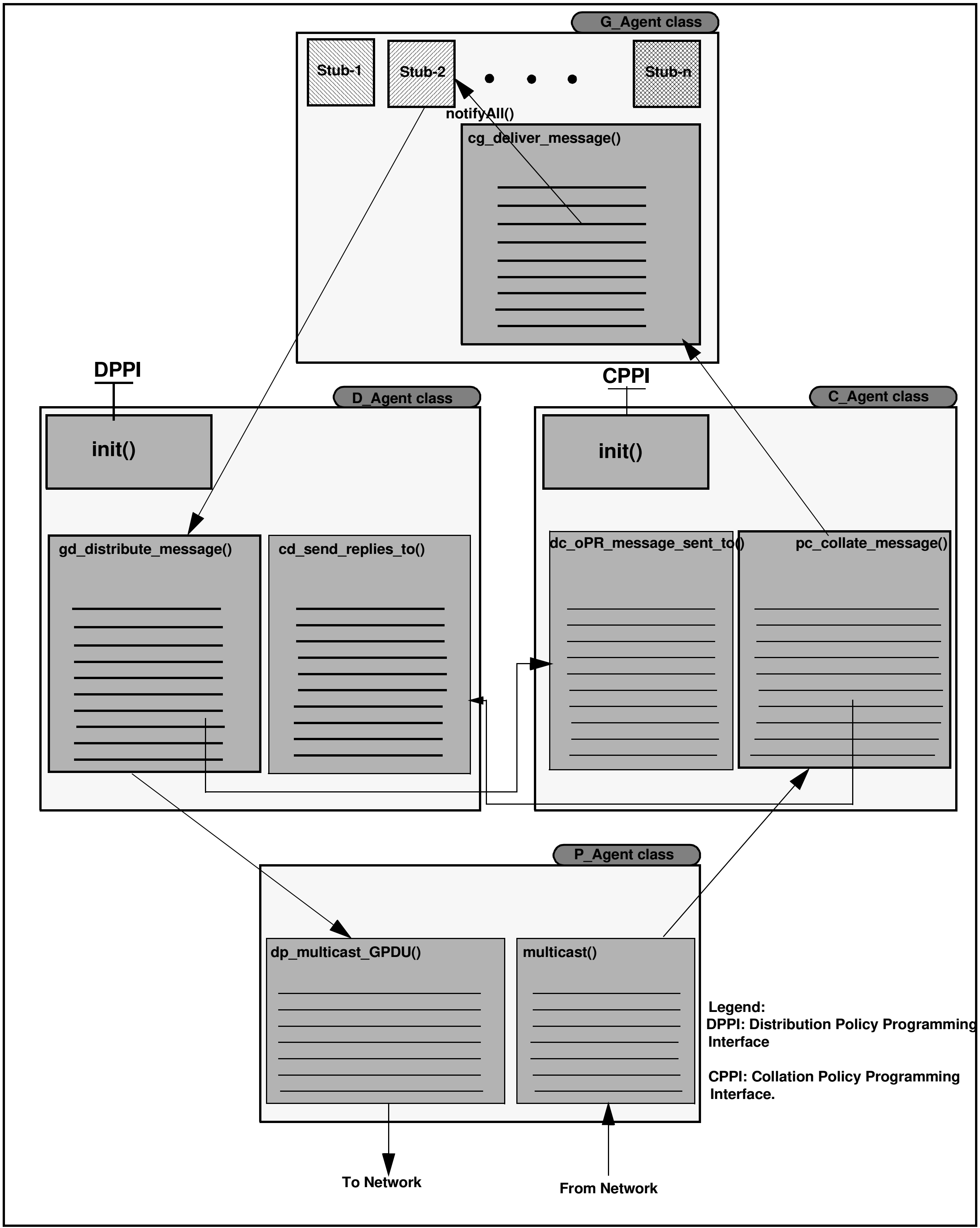

G_Agent class
Stub-1
Stub-2
Stub-n
notifyAll()
cg_deliver_message()
DPPI
D_Agent class
init()
gd_distribute_message()
cd_send_replies_to()
CPPI
C_Agent class
init()
dc_oPR_message_sent_to()
pc_collate_message()
P_Agent class
dp_multicast_GPDU()
multicast()
Legend:
DPPI: Distribution Policy Programming Interface
CPPI: Collation Policy Programming Interface.
To Network
From Network

Fig. 10.2 GSM Implementation: GSM Agents and their Interaction

The *D_Agent class* contains an important method, `gd_distribute_message(message, messageId)`, which is invoked by the stubs and the skeletons of the *G_Agent class* to request the distribution of OPR and REP-messages respectively. This method performs application message marshalling, GPDU coding and construction. A detailed description of these functions is given in section 10.2.4. Message marshalling is done based upon the message signature, hence there is a separate marshalling routine for each message type. Once a GPDU is constructed, this method reads the distribution policy from the "`distributionPolicyObject`" to find out the destination group membership and the protocol to be used for message distribution. The GPDU, the destination group membership and the protocol information is given to the *P_Agent class* by invoking `dp_multicast_GPDU(thid_GPDU, to_these_members, using_this_protocol)` on it. Finally, on the client side, it invokes a method, `dc_oPR_message_sent_to(oPRName, oPRId, sentToList)` on the local *C_Agent class* to inform it about the invocation instance identifier that will be associated with the corresponding REP-messages (only those REP-messages which are identified properly and are sent by the members in the "`sentTOList`" are accepted for collation by C-Agent).

10.2.1.4 C_Agent Class

The *C_Agent class* implements the policy-based collation of the OPR and REP-messages which are encapsulated in the corresponding GPDUs received from the *P_Agent class.* It contains methods to input and store the message collation policy from the user, to perform GPDU de-coding, application message un-marshalling (message reconstruction), message collation and its delivery to the *G_Agent class* for final delivery to the client or server application using the appropriate delivery semantics (*solicited*, *unsolicited, terminable*).

The *C_Agent class* contains an important method, `pc_collate_message(GPDU)`, which is invoked by the *P_Agent class* to request the collation of OPR and REP-messages contained in the corresponding GPDUs. This method performs GPDU decoding, payload un-marshalling to recover the original message, and message collation. A detailed description of these functions is given in section 10.2.4. Message un-marshalling is done based upon the message signature, hence there is a separate un-marshalling routine for each message type. Before performing collation on the recovered message, this method checks if the received (OPR | REP) message is acceptable to the (server | client) application (i.e., if the message type is included in the *message specification* of the "`collationPolicyObject`"), if the message sender is authorized (i.e., if the message sender is included in the *membership specification* of the "`collationPolicyObject`"), and in case of a REP-message, if the invocation instance identifier associated with the message is valid (i.e., whether an OPR-message with an identical invocation instance identifier was sent by the D-Agent).

Once the above mentioned steps are performed, the `pc_collate_message(GPDU)` method collates the recovered message based upon the collation policy specified for the corresponding message in the "`collationPolicyObject`". When the required number of messages of a given type are received (i.e., *collation cardinality* is satisfied), a group message is constructed based on the specified *collation mode* (matrix or linear or singleton) and delivered to the *G_Agent class* by invoking cg_deliver_message(message, messageId) on it. On the server-side, when a group-OPR message is delivered to the *G_Agent class*, a `cd_send_replies_to(messageId, componentIIIds, replyReceivers)` method is invoked on the *D_Agent class*, so that the reply received from the server in response to the group OPR-message is sent by the D-Agent to only those members of the client group from whom the component OPR-messages were received.

10.2.1.5 P_Agent Class

The *P_Agent class* performs the inter-GSM communication. This is described in detail in section 10.2.3. The *P_Agent class* contains a `dp_multicast_GPDU(this_GPDU, to_these_members, using_this_protocol)` method which is invoked by the *D_Agent class* to transport the GPDUs from the D-Agent to the C-Agent. This method transports the GPDUs to the peer P-Agents associated with the GSMs of the specified group members using the Java RMI protocol. When a GPDU is received, the P-Agent gives it to the local C-Agent by invoking pc_collate_message(GPDU) on it.

10.2.2 Implementation of Inter-Agent Invocations

The GSM agents communicate by invoking methods on each other. In Java, as in any other programming language, the method invocations are *blocking,* i.e., the caller is blocked until the receipt of the reply (or the completion of method, if the result is void). The use of this type of *caller-blocking* communication mechanism poses certain performance and architectural problems in a multi-agent software such as Group Support Platform (GSP) which consists of multiple GSMs (and their component agents) communicating with each other.

In the realization of the Group Support Platform (GSP), this type of blocking overhead is extremely serious. It results in a *chain of blocked methods*. For example, when a client's OPR-message is invoked on a GSM (i.e., its G-Agent), this message is invoked by the G-Agent on the D-Agent which in turn invokes the corresponding OPR-GPDU on the P-Agent. The source P-Agent invokes the GPDU on the destination P-Agents in the server group. The destination P-Agent invokes the received GPDU on the local C-Agent which in turn invokes the collated group message on the local G-Agent and finally on the server objects in the server group. Due to the blocking nature of method invocations, the first method invoker is not released until all the successor invokers are released (i.e., have received their replies). This results in all the agents in the GSM being tied up to handle a single client request.

There is another architectural requirement in the GSM, the path traced by the reply is not the same as the one traced by the original client's request. The replies are distributed by the D-Agent on the server side, and collated by the C-Agent on the client side. Hence there is a need to de-couple the request and reply path and to unblock the method call chain.

A simple and elegant solution to the problem mentioned above exists in Java. This is offered by the Java *threads*. The thread gives a way to implement concurrency or, in the case of a single processor environment, interleaved execution. The ability to create multiple threads and to embed the inter-agent method invocations within the body of the threads is the key to the solution. In our implementation whenever a class (or a method) needs to invoke another class (or a method) it instantiates an instance of a Java thread referred to as a `NonBlockingInvoker`, using a thread's *start()* method. The body of the `NonBlockingInvoker` thread contains a single method called *run()* which contains the appropriate inter-agent method invocation. So a method invocation is handled by an independent and concurrent thread of execution, thereby unblocking the original thread, the caller. Using this mechanism we have realized a *non-blocking invocation mechanism,* thereby avoiding the chain of blocked invocations. We now have a parallel architecture for Group Support Platform in which each GSM (and the component agents within the GSM) can execute concurrently with other GSMs.

In our implementation, the inter-agent invocations follow a certain naming convention. For example, the method called `gd_distribute_message()` is invoked (in an independent thread) by the G-Agent on the D-Agent to request the distribution of the message.

10.2.3 Implementation of Inter-GSM Communication

Inter-GSM communication occurs via the P-Agent. In a real environment, the P-Agent uses an appropriate multicast protocol to distribute the GPDUs to the (P-Agents of the) destination GSMs. Our implementation is carried out in a simulated environment on a single machine. We have implemented a four member group, each of them supported by an individual instance of GSM, identified as GSM1, GSM2, GSM3, and GSM4. All these GSMs run in separate address spaces on a single machine. So we chose Java's native inter-process communication protocol to transport the GPDUs between the GSMs. In our implementation P-Agents communicate with each other via Java's Remote Method Invocation (RMI) protocol.

After the instantiation of GSM agents, each P-Agent is registered in the Java RMI registry using the Java's *bind()* or *rebind()* method. Before making an invocation, the P_Agent is located using the "*lookup()*" operation of the Java RMI registry. These invocations between P-Agents are also executed in Java's *thread* and hence are non-blocking.

10.2.4 Implementation of Inter-GSM Protocol

The Inter-GSM Protocol (IGP) is described in chapter 9. This protocol consists of a set of GPDUs which describe how an agent within a GSM communicates with its peer agent in a remote GSM. The emphasis of that chapter is on the identification of the different fields required for inter GSM communication and their information content. There are several aspects of the protocol that are not defined in that chapter, such as marshalling of the message parameters and the detailed coding of the GPDU fields. These aspects pertain to the implementation of the protocol, and are decided based upon bi-lateral agreement between protocol partners. They are described in this section.

As mentioned in section 9.5, there are many possibilities with respect to the format and the coding of the GPDU fields. In our Java implementation of IGP, we have chosen a simple and straightforward coding for the GPDU fields. This is described below for each of the GPDU fields.

The *GPDU Type* field identifies the type of the GPDU. This field takes a limited set of values, such as "D-OPR-GPDU", "D-REP-GPDU", etc. There are numerous encodings possible for this field, such as a "byte", "integer", "enumerated types", "character strings", etc. In our implementation this is encoded as a character *String* in Java.

The *Sender Group Identifier* and the *Sender Identifier* fields of the GPDU identify the source group and the sender of the GPDU. Again there are many possibilities for the encoding of these fields, as outlined in section 9.5. The simplest being the representation of these fields as character *String* which we have chosen in our implementation.

The *Message Identifier* field contains the "invocation instance identifier" associated with an OPR, REP, or an NTF-message. It is a unique identifier and it could be represented as an integer, character string, or some combination thereof. In our implementation we encoded it as an *integer*. The *Group Membership Descriptor* and *Group Constraints* fields are not used in our implementation.

The encoding (or marshalling) of the *Payload* field of the GPDU deserves careful attention. This field contains the application message (i.e., an OPR, REP, or an NTF message) and its parameters. In high-level programming languages, such as Java, C++, C, etc., these messages have their corresponding *message signatures* and they are typed. Each message contains zero or more parameters and each parameter is an instance of a basic language type such as an *integer*, *boolean, character, float,* etc. or of a constructed type. There are many encoding schemes possible for these typed messages. In Corba's General Inter-Orb Protocol (GIOP), these typed messages are encoded using a Common Data Representation (CDR).

In our implementation we have chosen a simple message encoding scheme which is native to the Java language. All basic and constructed types in Java are an extension of the *Object class*. The *Object* class is at the root of the Java class hierarchy. Moreover, an *Object* class is *serializable*, a requirement for Java RMI protocol which is used to transport the GPDUs between the GSMs. The *Payload* field of the GPDU is implemented as an array of *Object* class. This allows all basic and constructed Java types to be marshalled as an *Object* type, the base type.

The marshalling of the OPR and REP messages is carried out based upon the *message specification* (message signature) in the `distributionPolicyObject`. Each message parameter is marshalled into its base type, the *Object* type, using an appropriate marshalling function for that type. The payload consists of a sequence of parameter names, implemented as a string, and the corresponding marshalled parameter value.

10.2.5 Implementation Distribution and Collation Policies

The Group Policy Specification Language (GPSL), introduced in chapter 8, is essentially a framework which identifies the basic elements of message distribution policy, collation policy, etc. The emphasis of GPSL is to give a language framework rather than any specific notation. Being a language framework, it can be implemented in a variety of mechanisms.

In our implementation, the distribution policy and the collation policy are realized as Java classes identified as `DistributionPolicy` and `CollationPolicy` respectively. These classes contain the corresponding elements of the policy such as *message specification*, *membership specification*, *cardinality specification*, etc. The *message specification* consists of the message name, followed by a sequence of parameter name and parameter type specification. This is implemented as a character *String*. The *membership specification* consists of a comma separated list of group member names. This is again a *String*. The *time specification* specifies the maximum collation waiting period. This is implemented as an *integer*. The *cardinality specification* specifies the minimum or maximum number of messages required for collation. This is naturally mapped into an *integer*. The *collation mode* could be “matrix”, “linear” or “singleton”. It can be represented either as an enumerated type or as a string. The latter is chosen.

Distribution policy and collation policy are programmed in the `distributionPolicyObject` and the `collationPolicyObject` respectively. These objects are implemented as an array of `DistributionPolicy` class and `CollationPolicy` classes respectively. The size of the array is equal to the number of message types supported by the group members.

The distribution and collation policies are solicited from the user in the `init()` method of the *D_Agent class* and *C_Agent class* respectively. This method serves as the Group Policy Programming Interface of the GSM (see figure 10.2).

10.2.6 Implementation of an API for Group Interrogation Primitive

The group interrogation primitive proposed in chapter 3 essentially defines an application-level API which is offered by the GSM and used by client components of a group-based application (see definition in section 3.10) for invoking OPR-messages on the server group and for receiving multiple replies in a solicited or unsolicited manner from the server group. This communication primitive is offered as an API to the application components by the G-Agent of the GSM and it is referred to as Group Interrogation Interface (GII) in section 6.2.1.1.

The group interrogation primitive gives the client the capability to receive a single group reply or multiple individual replies in a solicited or an unsolicited manner. In the following sub-sections we describe how these capabilities are supported in our API implementation using the Java’s thread synchronization primitives, *wait()* and *notifyAll()* which are used by the G-Agent to wait for the replies and to be notified when the replies arrive.

10.2.6.1 Implementation of Unsolicited Group Reply Delivery - API

This API supports the most simple semantics of the group interrogation, i.e., the client invokes an OPR-message on the GII and it is blocked until the receipt of a single group reply.

The client's request (OPR-message) invocation is intercepted by the *G_Agent class* and is handled by the appropriate stub. The stub generates a new invocation instance identifier (iiid) and sends the message along with its iiid to the D-Agent for distribution to the server group. The stub then waits for the group reply to be received from the C-Agent by executing the *wait()* Java synchronization primitive.

Due to the non-blocking nature of inter-agent invocations (as discussed in section 10.2.2), and due to the replies following a path different from the one followed by the request message, the stubs have to execute a *wait()* method.

When the C-Agent has received all the expected replies from the server group members, as specified in the reply collation policy, it gives a single group reply to the G-Agent by invoking `cg_deliver_message()` on the G-Agent. The `cg_deliver_message()` method of the G-Agent stores the received group reply in a local buffer and sends a notification to the waiting stub by executing another Java synchronization primitive, the *notifyAll()*. The stub then fetches the group reply from the buffer and returns it to the client, thereby unblocking it.

10.2.6.2 Implementation of Solicited Multiple Reply Delivery - API

The solicited reply delivery semantics of the group interrogation imply that the individual replies received from the server group in response to an OPR-message are delivered to the client by the GSM *only* when explicitly requested by the client, the client is unblocked immediately after invoking the OPR-message.

As in the previous case, the client's request (OPR-message) invocation is intercepted by the *G_Agent class* and is handled by the appropriate stub. The stub generates a new invocation instance identifier (iiid) and gives the message along with its iiid to the D-Agent (using the non blocking invocation, described in section 10.2.2) for distribution to the server group. However, in this case, the stub returns the invocation instance identifier to the client as a *reply handle*, thereby unblocking the client immediately. The client may now engage in other processing. The iiid is also locally stored in the G-Agent as an index for reply storage (when it is received) and retrieval (when requested by client).

Incoming replies are identified with the iiids that were associated with the corresponding OPR-message. In case of singleton and solicited reply delivery, the C-Agent sends the individual replies along with their invocation instance identifiers to the G-Agent as soon as they are received from the server group (without collation). These replies are buffered within the G-Agent and indexed with the corresponding invocation instance identifier.

Whenever a new reply is required, the client invokes a *poll_reply(reply_handle)* on the GII of the G-Agent. This invocation is handled by the stub for the poll_reply. If the reply corresponding to the reply handle is available in the local buffer, the stub returns the oldest reply in the buffer. If no reply is available, the stub may either block the client, waiting for the receipt of a new reply or it may give a "reply not available" reply and unblock the client (the client may try later). In our implementation the former option is chosen. In the former case, the stub executes a *wait()* primitive waiting for the receipt of reply available notification from the `cg_deliver_message()` method. When a reply is received from the C-Agent, the `cg_deliver_message()` method executes the Java synchronization primitive *notifyAll()* to notify the waiting stub about the receipt of the reply. The stub then returns this reply to the client, thereby unblocking it. This API can be further enhanced to handle the delivery of different types of replies by including a reply type argument in the poll_reply(reply_handle).

10.2.6.3 Implementation of Unsolicited, Multiple and Terminable Reply Delivery - API

This type of reply delivery semantics imply that the individual replies from the server group are delivered to the client by the GSM in an unsolicited manner as soon as they are received from the server group. The terminable reply delivery capability gives the client the control to terminate the delivery of subsequent replies when it does not want them any more. The terminable reply delivery capability can also be combined with the solicited reply semantics, although it adds little advantage.

In Java, as in any other programming language, the clients are unblocked after the delivery of the first reply, whether it is a singleton reply or a group reply or a reply handle as in the previous case. The implementation of unsolicited delivery of multiple replies individually (as and when they are received) using existing languages requires the client to support a *call back interface* to receive individual and unsolicited replies from the GSM. The clients register their callback interface with their local GSM. The clients may either support a single callback interface or multiple call back interfaces in order to support the receipt of different types of replies. So, the GSM knows which client's interface(s) to invoke to deliver the replies.

As in the previous case, the client's request (OPR-message) invocation is intercepted by the *G_Agent class* and is handled by the appropriate stub. The stub generates a new invocation instance identifier (iiid) and sends the message along with its iiid to the D-Agent (using the non blocking invocation) for distribution to the server group. Then the stub returns the invocation instance identifier to the client as a *reply handle*, thereby unblocking the client immediately. The client may now engage in other processing. In this case each reply handle (iiid) is associated with a corresponding reply delivery flag which is set true by the stub before returning the reply handle to the client.

When individual replies are received from the local C-Agent, the G-Agent invokes these replies on the appropriate callback interface of the client if the reply delivery flag associated with the reply handle is set to true, otherwise the reply is discarded. (The client is internally notified of the reply receipt by their callback methods).

When the client wants to terminate the reply delivery for a particular OPR-message, it invokes a special message *terminate_replies(reply_handle)* on the Group Invocation Interface (GII) of the G-Agent. This message is handled by the corresponding stub, which turns off the corresponding reply delivery flag, thereby disabling the delivery of subsequent replies.

10.3 Performance Aspects

Middleware is an entity that lies between the applications and the low-level communication infrastructure. Therefore the performance of middleware platform is an important aspect to the applications that use it. Middleware platforms perform some useful functions in order to make distributed computing transparent to the applications that use them. Hence, in general, there is some performance overhead associated with these platforms.

In this section we discuss performance of the Group Support Platform (GSP), a middleware for the support of group-based distributed applications. It makes *group-based distributed communication* aspects transparent to the applications.

Some of the desirable properties of the middleware platforms are high *throughput*, low *latency*, *flexibility*, *scalability*, *reliability*, and *ease of use*. In a group-based distributed environment, there are many factors that have an effect on these desirable properties. In the following sub-sections we examine these factors and evaluate the performance of GSP using the *performance metrics* [149 - 153] used for the evaluation of middleware platforms.

10.3.1 Message count

The number of messages exchanged between the middleware entities is a basic performance metric. It has a direct effect on the network load and the application throughput and reply-reception latency. In case of GSP, the basic data unit that is exchanged between the middleware entities (i.e., the GSMs) is the Group Protocol Data Unit (GPDU). As shown below, this message count is directly proportional to the size of the group, i.e., number of group members. For example, an OPR-GPDU has to be distributed to all the server group members and the originating GSM has to wait for the reception of all the REP-GPDUs before a group reply can be sent to the client. Hence group size increases the message count.

The reply reception latency, ideally, is not a function of message count (or rather the group size). In a environment composed of *concurrent* server group and communication links, it takes the same amount of time to receive a “single” reply or “n” number of replies from the group (given a negligible collation processing time to construct a group reply in the client’s GSM). This is due to the fact that the REP-GPDUs from all the server group members are received simultaneously by the client’s GSM in response to an OPR-GPDU (because all OPR-messages are processed and the corresponding REP-GPDUs are generated simultaneously because of the *concurrency* in the server group). However the distribution of the members of the server group from the client object and the characteristics of the communication links between them (and the dynamic network load) affects the latency of the reception of replies (REP-GPDUs) from the individual members of the server group.

Here is a quantitative evaluation of the message count. Consider a client object interacting with a server group of size “n”. The client’s GSM sends “n” number of OPR-GPDUs, one to each member of the server group. Similarly, it receives “n” number of REP-GPDUs, one from each member of the server group. Therefore the total message count is “2n”.

Now let’s evaluate the message count if a “filtering requirement” (see example in section 7.10) is imposed in the server group. This means that “m out of n” servers in the server group are filtered (selected) to perform (execute) the client’s request (OPR-message)[1]. Again, the client’s GSM sends “n” number of OPR-GPDUs, one to each member of the server group. The OPR-GPDUs contain the filtering constraints specified by the client. An OPR-message can only be given to “m” members of the server group which satisfy the client’s filtering criterion. In order to do this, the contestant F-Agent in each GSM sends an F-PAR-GPDU (see section 9.8) to an arbitrator F-Agent, a total of “n” messages. In response to this, the arbitrator F-Agent sends its “selected” or “not selected” reply in F-RES-GPDUs, one to each member of the server group, another “n” messages. The set of “m” filtered (selected) servers send their replies in REP-GPDUs. Therefore the total message count in this case is “3n + m”.

10.3.2 Message Complexity

The size of the messages exchanged between the middleware entities is another performance metric. It is usually measured in “bytes”. It depends upon the amount of control information carried in the GPDUs (in addition to the usual payload size) and the type of encoding scheme employed. As described in chapter 9, the GSMs need a minimal control information to be exchanged between them in order to communicate with each other and to perform their function.

Here’s a quantitative evaluation of the size of the GPDUs which carry application messages, such as

---

[1] It may be noted that filtering does not always imply “m out of n” selection. If a client wants all servers in the group that satisfy the client’s filtering criterion to be selected, then only local filtering is done, “m out of n” selection is not required.

D-OPR-GPDU, D-REP-GPDU and D-NTF-GPDU. The size of any PDU depends upon the encoding scheme. Here we illustrate the size using most simplest scheme. The minimum size of the GPDU fields are:

1. *GPDU Type*: using an “octet” IDL encoding = 1 byte,
2. *Sender Group Id*: using an “octet” IDL encoding = 1 byte,
3. *Sender Id*: using an “octet” IDL encoding = 1 byte,
4. *Message Identifier*: using a “short” IDL encoding = 2 bytes,
5. *Payload*: This a variable length field which contains application message. The first 2 bytes of this field specify the rest of the payload length. This field is encoded as parameter name and parameter value tuples. The parameter names are required in a GPDU to support the “linear-mode message collation” on the receiver side. The inclusion of parameter names increases the payload size. The parameter names are “IDL stings”. Therefore the encoding of an OPR-message (with a variable number of parameters) requires “q” bytes to encode parameter names and “p” bytes to encode the actual parameter values.

Therefore the total message size is “p + q + 7” bytes. The overhead is q+7 bytes. If only matrix mode collation is supported, then there is no need to encode parameter names in the payload field. In this case message size shrinks to “p + 7” bytes. One possibility of minimizing the payload complexity is to use optimal coding techniques, which generates the least number of bytes.

Apart from message counts and message complexity, we present some other factors that also have an impact on the performance of a “middleware” system.

10.3.3 Communication Network Speed

The response time or the reply reception delay is primarily dependent upon the characteristics of the communication network and individual links that connect the client object to the server group, and on the physical distribution of individual group members from the client. The speed of the individual communication links between the client and the server group members, is the primary factor that affect the response time.

Moreover, the response time is also affected by such varying and dynamically changing network traffic conditions arising due to congestion, link failures, re-routing, etc.

10.3.4 Message Marshalling and Un-marshalling Overhead

The encoding and decoding of the GPDU on the sender and receiver side also adversely affect the overall performance of the GSP. It decreases the throughput and increases the latency of reply reception by the client. The encoding of all the GPDU fields, except the payload, incurs a fixed amount of processing overhead. The marshalling and un-marshalling of the payload field involves variable processing overhead. The payload consists of variable number and types of parameters. The use of complex data structures for parameter types requires excessive marshalling and un-marshalling overhead. Latency also increases linearly with the size of the request.

In general long operations with many parameters and complex parameter types take longer for marshalling and un-marshalling. This is due to the fact that marshalling routines convert the complex data types into most basic data representation forms (flattening). These transformation functions, such as message marshalling, buffering, data copying, etc. are the primary areas that must be optimized to achieve higher throughputs.

10.3.5 Intra-GSM Invocations Overhead

Conventional middleware platforms such as Corba, DCOM, etc. suffer from excessive intra-ORB function calls. One of the reasons for latency in these platforms is the long chain of intra-ORB function calls. These intra-ORB invocations and the internal data copying between buffers consumes a significant amount of CPU, memory and I/O bus resources and they affect the efficiency of the model.

In our implementation of GSMs we make special consideration to minimize this overhead. Due to the use of non-blocking invocations (see section 10.2.2) for inter-agent invocation within the GSM, this type of delay is almost non-existent in our implementation. In particular, this type of architecture avoids the long chain of inter-agent calls. After making an invocation on another agent, the calling agents are released immediately to accept another request (either from the client or another agent) because inter-agent invocations are performed in an independent and newly spawned thread. Also the design of the GSM minimizes the need for invocations between the GSM agents.

10.3.6 Internal Buffer Sizes and Queue Lengths Considerations

Any implementation of middleware platform requires the use of internal buffers and queues to store and process messages until they are ready for delivery to the client or server applications. The size and the allocation of these system resources is also a performance consideration. A small number (and size) of these buffers would result in the user requests being queued, thereby affecting throughput, and a large number of them would result in waste of system resources. An optimal and dynamic allocation strategy is always a requirement.

In our implementation, the most prominent buffers are the collation buffers required to store messages (OPR or REP) until a group message is constructed. The size of each collation buffer is known because the group size and message size are known. It is the number of these buffers that are needed that is unknown. However this is not a problem in Java because Java permits dynamic allocation of buffers.

10.3.7 Concurrency and Multi-threaded architecture aspects

The concurrency in middleware is dependent upon the use of multi-threaded design techniques. Multi-threading is in general a good design performance optimization technique. It minimizes latency, increases throughput and ensures predictability in middleware platforms.

The proposed GSM model, can be implemented in many ways and by using any combination of multi-threading techniques [154] such as *thread-per-request architecture*, *thread-per-connection architecture*, *thread-per-object architecture, thread-pool architecture*, etc. The choice of one or the other architecture depends upon the nature of applications supported. It is a trade-off between concurrency and the host system resources that are consumed. For example the *thread-per-request architecture* is very costly in terms of consuming system resources, but is useful to handle long duration requests such as database queries. In our sample GSM implementation we use multi-threading mainly to avoid performance bottlenecks such as in inter-agent invocations (which may result in long-chain of inter-agent calls within GSM) and to avoid client requests starvation.

Multi-threading allows requests to execute simultaneously without impeding the progress of other requests. If properly used, it can ensure that client requests can be handled quickly enough and new requests are not starved or unduly delayed. With multiple threads, each request can be serviced in its own thread, independent of other requests. Likewise system resources are are also conserved, since creating a new thread is typically much less expensive than creating a new process. Moreover the thread dies after the completion of the request, thereby releasing the resources. However, even with multi-threaded architectures, it may be noted that the performance is dependent upon the number of CPUs and the thread scheduling policies employed by the native host environment.

10.3.8 Timers

The use of timers in the middleware platforms allows users to impose a maximum upper bound on the time spent waiting for the receipt of replies. Timers are an integral part of middleware platforms. In the GSP, the the upper limit on the reply reception delay is determined by the collation timers (which are programmed as part of the reply collation policy).

10.3.9 Collation Processing Overhead

So far we have evaluated the GSP based upon the metrics that are relevant both to the "single client - single server" middleware platform such as Corba and to the group-support middleware such as GSP. In this and in the subsequent sub-section we will evaluate GSP based upon the considerations that are specific to the group support middleware.

In a group-based environment, there are many factors that contribute to the latency and delays in receiving responses. One of them is the requirement for delivery of group reply to the client. The construction of the group reply involves message collation mechanisms. Reply collation can only be completed and a group reply constructed when all the replies are received from the server group. Hence collation introduces its own reply reception delay.

10.3.10 Other Group Processing Overhead

Other group processing functions such as synchronization and filtering discussed in chapter 4, also have a major impact on the throughput and latency of the GSP. The synchronization of client's (OPR-message) invocations in the client group requires the execution of a solicited or an unsolicited synchronization protocol as described in section 9.7. This protocol involves the exchange of S-GPDUs which introduces further delay in the distribution of OPR-message to the server group, and consequently the reply reception latency.

Similarly, the filtered delivery of client's invocation in the server group requires the execution of an "m-out of-n selection" protocol as described in section 9.8. This protocol involves the exchange of F-GPDUs which introduces further delay in the delivery of OPR-message to the server group, and consequently the reply reception latency.

10.3.11 Reliability and Robustness

So far we have focussed our attention on the throughput and latency characteristics of the GSP. In this and the following sub-sections we evaluate the other performance aspects of the GSP.

The GSP is a replicated architecture. Due to the presence of GSM entity at every member node, the failures are localized. Failure of one node or GSM does not affect the overall performance of the rest of the GSP (except for example the unavailability of reply from one group member). Moreover the management protocol discussed in section 9.9, detects failures and provides failure notifications to the rest of the GSMs in the GSP.

Robustness of the middleware platform is determined by the upper limits of the entities handled, e.g., the maximum size of the request, the maximum number of client and server objects, etc. In a multi-threaded implementation of GSP, each client request is handled by a separately spawned thread. Hence the robustness of the GSP is limited by the availability of the underlying system resources such as the maximum number of threads, buffers, etc. that can be obtained from the host system.

10.3.12 Scalability

Scalability is the ability to handle the increasing number of objects in the end systems and in the distributed system. Scalability is important for large scale applications that handle large number of objects on each network node as well as large number of nodes throughout a distributed computing environment.

There are two aspects to the scalability of the GSP, the scalability of the GSMs and the scalability of the Inter GSM Protocol (IGP). The former determines the ability to handle the increasing number of requests in the end systems and the latter determines the ability to add more nodes in the distributed system.

In a multi-threaded GSM implementation, each client's request is handled by a separate thread. The ability of the GSM to handle an increasing number of clients and their requests in the end systems is limited by the maximum number of threads and the buffers that can be obtained from the host. Therefore, the maximum limit on the scalability is set by the underlying host resources.

The scalability of the IGP is determined by the scalability of the GPDU fields. The fields that have a direct impact on the scalability are the "Sender-Group-Id", "Sender-Id", and the "Message-Id". The encoding of these fields affects the scalability. For example, if "Sender-Id" field is encoded as a "byte", then a maximum of 256 objects can be supported by the GSP. However, if this field is encoded as a variable length "String" (as is done in our Java implementation), then an arbitrarily large number of objects can be supported by GSP. None of the GPDU fields constraint the expansion of the group size. The IGP itself does not limit the scalability of the middleware. A similar consideration applies to the scalability of the underlying multicast protocols which transport the GPDUs between the GSMs.

10.3.13 Ease of Use

The ease of use of a middleware platform is an important criterion for the applications. In this sub-section we discuss how easy it is for the client and server components of the group-based application to use the GSP. As mentioned in the previous chapters, the GSM offers two interfaces to the user application. These are the GSM Invocation Interface (GII) and GSM Policy Programming Interface (GPPI). These are the only interfaces that are accessible to the applications and we will describe in the following sub-sections the ease of use of these interfaces. These interfaces are available as high-level APIs in our Java implementation of GSM.

10.3.13.1 Ease of use of GSM Invocation (GII) Interface and Group Interrogation primitive

The GII is used by the group-oriented client and server components. It's an API that supports group interrogation primitive and its associated semantics, such as solicited reply delivery, terminable reply delivery, etc.

This is a simple and easy to use interface. In the simplest case, the client invokes request (OPR-message) on the GII, as it would on any other middleware such as Corba, and is blocked until the receipt of a single group reply. All aspects involved in giving a group reply, such as request distribution, reply collation, etc. are handled transparently by the GSM.

The GII also supports sophisticated reply delivery semantics. The GII supports some simple primitives to enable clients to control the reply delivery. The client is given the control to receive individual replies as when they are required and to terminate the reply delivery when they are no longer required (in an unsolicited delivery semantics). This control is again simple. The client simply has to invoke "*poll_reply()*" in order to solicit a reply and "*terminate_replies()"* in order to stop the flow of reply delivery.

Similarly, the group-oriented server object receives the "group request" from the client group via a *single invocation* from the GII, as a "single request" is delivered to a singleton server by Corba. The request collation is done transparently by the GSM. Similarly, the group-oriented server returns either a single reply or group reply in a *single invocation* of GII. The dis-aggregation of group reply and the reply distribution, collation, etc., is all handled transparently by the GSM.

The group oriented clients and server use the GII with almost the same flexibility and ease of use as the singleton clients and server use the Corba invocation interface. GSM allows all the group coordination aspects to be modeled and executed external to the applications.

10.3.13.2 Ease of use of Group Policy Programming Interface

Unlike in conventional middleware platforms such as Corba, there are many aspects in a group support middleware which must be programmed by the user. As discussed in detail in the previous chapters, these aspects describe how to distribute the messages, how to collate and deliver the replies, how are client requests synchronized, how are they filtered before delivery to the server group. These pertain to the message distribution, collation, synchronization, and filtering policies. The GPPI offers an interface for the programming of these policies. As shown in our implementation, the GPPI can be realized as a simple user interface through which all aspects of distribution, collation, etc. policies are input from the user. In a more sophisticated implementation, the GPPI can be realized as a Graphical User Interface.

10.4 Comparison of Group Support Platform with CORBA Middleware

The GSP is an enhancement of the currently available middleware solutions in order to provide support to a special category of distributed applications, the *group-based distributed applications*, which are composed of a client group interacting with a server group. In this section we compare the support of the group-based applications in GSP with the support of same applications in the currently available middleware solutions such as CORBA. The idea is to compare

1. the ease of use of the proposed *group interrogation* primitive with the *traditional interrogation* (or *remote procedure call*), and
2. the *group support platform* with traditional middleware approaches such as *Corba*.

A detailed comparison between Corba and GSP w.r.t. the performance metrics such as message count, message complexity, response times, and a discussion of trade-offs is given in table 10.1 and table 10.2

10.4.1 Comparison at Programming-Level

In this section we will evaluate the programming effort required by the client application in invoking a server group and receiving the replies using both the traditional Corba approach and the proposed GSP approach. Similar effort on the server side using both these approaches is also compared.

10.4.1.1 Group Interrogation vs. Remote Procedure Call

Using the traditional *remote procedure call* mechanism of Corba, the client has to invoke each server in the server group separately and due to the blocking semantics of the remote procedure call, it has to wait until the reply is received before it can invoke the next server. This introduces considerable latency and also puts the responsibility of knowing the server group membership and any changes in the membership on the client application. The proposed group interrogation primitive along with the underlying GSP gives the client the capability to invoke multiple servers simultaneously and to receive a single group reply without knowing the membership of the server group.

Table 10.1: Corba vs. GSP: How do they compare w.r.t. Crucial Performance Metrics

GSP	Corba
Message Count A detailed analysis of the message count in GSP is given in section 10.3.1. As shown, the message count is directly proportional to the size of the group, and to be precise it is equal to twice the size of the group. Message count = 2n, where, "n" is the size of the server group, with which the client is interacting.	**Message Count** In case of Corba, the basic data unit that carries the request / replies between the middleware entities (i.e., the ORBs) is the "Request Message / "Reply Message". Because corba does not provide "group support", the client has to separately invoke each member of the server group, one after another (see discussion in row 3 of this table). For each invocation a "Request Message" is sent from client ORB to the server ORB and a "Reply Message" is sent in the reverse direction. Thus 2 messages are exchanged between ORBs for each invocation. Therefore for a server group of size "n", a total of 2n messages are exchanged between the client object and the server group. **Conclusion**: The message count in both the platforms is identical. The GSP does not incur any extra overhead over Corba with respect to this metric. Moreover, the GSP gives a lot more functionality and group support to the applications, than is available in Corba, such as collation of replies, solicited delivery of replies, and other functions described earlier in the thesis.

Table 10.1: Corba vs. GSP: How do they compare w.r.t. Crucial Performance Metrics

GSP	Corba
Message Complexity: The IGP is the Inter-GSM communication protocol in GSP. The request/replies are carried in the D-OPR-GPDU/D-REP-GPDU. A detailed analysis of the message complexity in GSP is given in section 10.3.2. It was found that the message size is (7 + q + p) bytes long, where q is the number of bytes required to encode parameter names, and p is the number of bytes required to encode the parameter values. If only matrix-mode collation is required, then there is no need to send parameter names in the payload field, so the total message size in this case is 7 + p bytes.	**Message Complexity:** The GIOP is the General Inter-ORB communication protocol in Corba. The application messages (request/replies) are carried in the GIOP “Request Message”/ “Reply message[a]”. As described below, there is some additional complexity in Corba “Request Messages” due to different types of control information that are carried in this message. Apart from the usual “Message Type” field (1 byte long) and the “Message Size” field (4 bytes long), the Corba “Request Message” also contains the following control information in its header: 1. *magic*: This field identifies the “GIOP” protocol itself. This field is used to identify the GIOP from other possible inter-ORB communication protocols. It is 1 byte long. 2. *GIOP-version*: contains the version number of the GIOP protocol being used. This is used to ensure inter-operability between ORBs. This field is 2 bytes long. 3. *Request-Id*: This is similar in function to the “Message-Id” field in GPDUs. It is used to associate reply messages with the corresponding request messages. This field is 4 bytes long. 4. *Response Expected*: This field is used to indicate if a reply is expected to the enclosed request message. This field is unnecessary in our case, because notification messages (which do not have a reply) are sent in a separate GPDU, the D-NTF-GPDU. This field is 1 byte long. 5. *Object_key*: This field identifies the target object. It is identified as a “sequence of octets”. It’s length is negotiated between protocol partners, and is usually between 4 to 16 bytes. However, this field is redundant and actually not required. A message need only identify its sender (such as “Sender-Id” in GPDU) . The target object reference need only be given to the lower layer protocol which carries the message to the target object. 6. *Requesting_principal*: This field identifies the message sender. This is similar in function to the “Sender-Id” field in GPDU. Again its length is not specified, it is negotiated between protocol partners and varies from 4 to 16 bytes. 7. *Operation*: Identifies the name of the operation being invoked. It is identified as a *string*. It is a variable length field. The operation name is included as part of payload in GPDU and not in a separate field. 8. *Request body*: This is the payload field. Its length is variable, say “p” bytes. So, the **total size** of the Corba “Request Message” (taking only minimal field length, wherever unspecified) = “21 + p” bytes **Conclusion**: In general the Corba Request message carries a lot of control information than that required in GPDUs. The control information in the corresponding GPDU is *minimal*. In any protocol design, it is always desired to keep the control information minimal in the protocol data unit. However, in the OPR-GPDU the payload field is much longer than the corresponding field in Corba due to the need to carry parameter names along with parameter values. The parameter names are required to assist the linear-mode collation on the receiver side. If only matrix-mode collation is supported, then parameter names are no more required in the GPDU payload and the payload fields in both GSM and Corba are identical in length (i.e., p bytes).

Table 10.1: Corba vs. GSP: How do they compare w.r.t. Crucial Performance Metrics

GSP	Corba
Response Time In case of GSP, the client invokes a single OPR-message on the local GSM which is then simultaneously distributed (multicast) by the GSM to the server group and hence the server group members are invoked simultaneously. Therefore, the servers respond with their replies simultaneously and the client receives multiple replies (as a single group reply) in one “invocation time delay”, say “d” units of time.	**Response Time (Reply Reception Delay)** In case of Corba, the client has to invoke a *remote procedure call* on each member of the server group separately, and due to the blocking semantics of the remote procedure call, it has to wait until the reply is received before it can invoke the next server in the group. If “d” is the single “invocation time delay” or the round trip delay in receiving a single reply, then to invoke a server group of size “n” and get back their replies requires “nd” units of time. Response Time = (nd) units of time, where, “d” is the single invocation round trip delay and “n” is the size of the server group. **Conclusion**: The response time is far better in GSP compared to Corba. This is because the GSP is designed exclusively for the support of group-based applications. So it is capable of simultaneous distribution OPR-messages and of collating the received replies into a group reply. The group-based application experience a much lower reply reception delay when used on GSP. They receive all the replies from the server group in a single round trip delay. Corba is not well suited for use by group-based applications.

---

a. PDUs are referred to as Messages in GIOP.

10.4.1.2 Ease of group request invocation

Using the *remote procedure call* mechanism of Corba, there is no way for the server to receive multiple requests (OPR-messages) from the client group as a single “group request” via a *single invocation* on the server. In order to receive and process a “group request”, the server application has to receive individual client requests from the client group. Moreover the server has to do all the housekeeping of tracking the individual received requests. It must also know the membership of the client group and also changes in the group membership. The proposed group interrogation primitive along with the support of GSP solves this problem. It gives the server the facility to receive the multiple client requests in a *single invocation* and to respond to the group request with single or multiple replies. Moreover the GSP takes the responsibility of sending the replies to the appropriate clients.

10.4.1.3 Support for Advanced Programming-level facilities in GSP vs. Corba

The support for advanced programming-level facilities such as multiple and variable reply delivery to the client (in response to a request invocation on the server group), solicited reply delivery, terminable reply delivery, is non-existent in the current *remote procedure call* mechanism of Corba. The proposed group interrogation primitive with the support of the GSP gives the client the capability to receive multiple and variable number of replies in a solicited or unsolicited manner and the ability to request the termination of reply delivery, when they are no more required. For example to support the solicited reply delivery, the GSP stores the replies and gives them to the client on an explicit request. Such as support is non-existent in Corba.

Table 10.2: GSP vs. Corba: What are the Other Trade-Offs

	GSP	Corba
Need for sophisticated Client and server applications	While there are major gains achieved by the use of GSP (see section 5.6) and its group interrogation capability (see section 3.3) as outlined previously, they also require some amount of sophistication in the (client | server) applications that use them. Essentially this complexity arises due to the need for clients and servers to be partially *group-aware*. The clients should be capable of handling not only multiple and variable number of replies, but also capable of processing group replies, and of invoking "poll_reply()" and "terminate_reply()". Similarly, the servers should be capable of processing group operation messages. So, there is an underlying requirement for the client and server applications to be designed "*group-aware*". However, this is also the underlying basis of any group-based application.	Corba applications are based upon "singleton client" and "singleton server" communication assumption. Hence they cannot be used as components of the "group-based applications" without at least being designed partially group-aware.
Code Complexity	In case of GSP, there is a need for some sophisticated programming language facilities such as multi-threading, thread synchronisation facilities, etc. to implement advanced features such as non-blocking invocation, solicited reply delivery semantics, etc. Code complexity also increases with the addition of other group support functions such as Synchronisation and Filtering, which are required in some group-based applications.	Simple client-server based applications of Corba do not demand the sophistication required for the support of group based applications. So the code for the basic Corba infrastructure is quite straightforward. However, the addition of other Corba Services such as Naming Service, Transaction Service, Security Service, etc. increases the code complexity.

10.4.2 Comparison at Platform-Level

A comparison of the Corba and the GSP at the platform-level reveals the difference in the scope of these platforms. They address different types of distributed applications and as such encapsulate different sets of functionalities.

10.4.2.1 Middleware functions of GSP vs. Corba

The Corba and the GSP cater to different types of distributed applications. While Corba supports single client and single server type interactions, the GSP is targeted exclusively at the support of group-based distributed applications. Hence at the platform-level we see different sets of "middleware functions" in Corba and GSP. Corba automates common distributed computing tasks such as object registration, location, and activation; request de-multiplexing; parameter marshalling and un-marshalling; and operation dispatching. The GSP automates common group communication tasks such as message distribution, collation, synchronization, filtering etc. It also provides a framework for identification and placement of other group support services.

10.4.2.2 Platform programmability Capability in GSP vs. Corba

Corba, like other middleware platforms does not define APIs that allow applications to specify their end-to-end QoS requirements. Similarly it does not provide support for end-to-end QoS enforcement between applications across a network. For instance, Corba provides no standard way for clients to indicate the relative priorities of their requests to an ORB. The GSP is a programmable and policy driven middleware platform. It provides an explicit API and language framework for the programming of the middleware functions according to the user requirements.

10.5 Case Studies

In our Java implementation of Group Support Platform, we used four case studies to demonstrate some of the key capabilities of the GSP, such as
1. Group Reply Delivery, Matrix mode collation (Singleton client interacting with a server group),
2. Group Reply Delivery, Linear mode collation (Singleton client interacting with server group),
3. Solicited Reply Delivery (Singleton client interacting with a server group),
4. Group Request Delivery and Reply Distribution (Client group interacting with a singleton server)

The case studies were chosen from the examples given in chapter 7. In the following subsections we describe *briefly* each case study that was chosen for the demonstration of the above mentioned capabilities. Simple examples were chosen in order to demonstrate the concepts. The reader is referred to the corresponding examples in chapter 7 for details about those examples.

Each case study involved either a client object interacting with a server group or a client group interacting with a server object. The size of the (client | server) group is chosen to be three members. Each member of the (client | server) group is supported by the GSM. Hence the execution of each case study involved four GSMs, identified as GSM1, GSM2, GSM3, and GSM4. The case studies were carried out in a single machine. Each client and server object is resident in a separate Java process, and supported by its own GSM.

10.5.1 Case Study-1: Group Reply Delivery, Matrix-Mode Collation

This case study is based upon the example given in section 7.6.2. This example is developed to demonstrate the distribution of operation message by the client's GSM to the GSMs in the server group and the matrix-mode collation of replies by the client's GSM into a single *group reply* before its delivery to the client object.

In this example, the client object sends a "query_sale_status()" operation to the members of the server group (consumer group in the example), each of which responds with a "sale_status(reatiler_id, merchandise_1, ..., merchandise_5)[1] reply. A single group reply is constructed by the client's GSM based upon reply collation policy as specified by the client.

Our implementation supports the specification of message distribution and reply collation policies by the user before the client and server applications are triggered.

10.5.2 Case Study-2: Group Reply Delivery, Linear-Mode Collation

This case study is based upon the example given in section 7.6.3. This example is developed to demonstrate the linear-mode collation of replies. Each server sends part of the reply expected by the client, and the client's GSM constructs a total reply using linear-mode collation principle, before its delivery to the client object.

In this example, the client object sends a "compute(p1, p1, ..., p5)" request to the server group (an arithmetic group), each member of the server group does part of the computation, i.e., addition, multiplication, and arithmetic mean, and therefore sends part of the reply as result(sum), result(product), result(am). The client's GSM performs a linear mode collation of these replies, and then gives the total reply, i.e., result(sum, product, am), to the client. With the linear mode collation process, the client is unaware if it is interacting with a singleton server or a server group.

---

[1] Assuming each server sells 5 types of merchandise).

10.5.3 Case Study-3: Solicited Reply Delivery

This case study is based upon the example given in section 7.6.4. This case study is developed to demonstrate the "Solicited Reply Delivery API" of group interrogation. This API gives the client the capability to receive replies only when it explicitly asks for them from the local GSM, using "poll_reply()". This allows the client to control the reply delivery from underlying GSM and not to be overwhelmed by a flow of replies from the server group. The client gets the replies as and when it needs them

In this example, the client object sends a "query_merchandise_availability()" request to the the server group (the merchandise supplier group) through its GSM. Each member of server group responds to this request with its reply, "merchandise_availability(supplier_id, merchandise_id1, quantity_1, merchandise_id2, quantity_2).

When the "query_merchandise_availability()" request is received from the client, the local GSM immediately unblocks the client by returning it an "invocation instance identifier" (see section 3.12.1) which serves as a reply handle. The client can now perform other computation or processing. When it needs a reply from the server group, it invokes "poll_reply(reply_handle)" on the local GSM. If a reply is available corresponding to the reply handle, the GSM returns this reply, otherwise the client is blocked until the receipt of a reply from a member of the server group.

10.5.4 Case Study-4: Group Request Deliver and Reply Distribution

This case study is based upon the example given in section 7.7.4. This case study involves a client group (sonar group) interacting with a server object (tracking object). This case study demonstrates the construction of *group service request* from the partial service requests sent by the individual members of the client group, which is invoked on the server object, and the distribution of replies to the client group.

In this example the client group consists of three members. Each member of the client group sends a partial service request "target_distance(d_i)", which is the distance of a "target object" from the client, and expects to receive the "x", "y", and "z" coordinates of the target (so that it can fire that target). The server object (i.e., the tracking system) cannot compute the coordinates of the target until it receives the distance of that target from three different locations, which is the location of the clients in the client group. The GSM on the server side collates the partial service requests, i.e., target_distance(d_i), into a complete service request, i.e., target_distance(d_1, d_2, d_3), which is then invoked on the server object. The reply received from the server, "target_location(x_coord, y_coord, z_coord)", is then distributed by the GSM to all the members of the client group.

CHAPTER 11 Conclusions and Directions for Future Work

Abstract

In this chapter we summarize the work presented in the thesis and highlight its main contributions. Some suggestions for future work are also given.

11.1 Conclusion and Contribution of Thesis

To date most of the research in this area was focussed on low-level aspects of group communication, such as various types of message multicasting protocols and membership management protocols. Our thesis has extended the benefits of group communication to the application level. We have identified a new paradigm of distributed computing - the *group-based distributed computing paradigm*. This paradigm is a synergy of *group communication model* and other *distributed object models* such as the *client-server model* and *object group model*.

Consequently, this thesis represents a shift of research focus from low-level issues of group communication to the high-level issues of an overall *distributed environment* required for the support of group-based distributed computing applications. Our thesis fills a void that exists at this high-level. The thesis addresses dual levels of support for group-based distributed computing applications. These are the *distributed programming-level support* (computational support) and the *distributed platform level support* (engineering support).

11.1.1 Contribution at the Programming-Level

A major contribution of the thesis towards the *programming-level* support for group-based distributed applications is the notion of *group interrogation* and its associated semantics. This communication primitive takes into consideration the most general requirements of group communication, at the application level, between the client and server components of a group-based application. It is analogous to the *remote procedure call* primitive which is used for communication between a singleton client and a singleton server.

The proposed *group interrogation* primitive enables *one-to-many* and *many-to-one* communication between the client group and the server group. It allows a singleton client to access a server group in one call, through the mediation of the group proxy object (i.e., the *Group Support Machine*), and to receive multiple and variable number of replies in a controlled manner in response to that call. Similarly, it allows a singleton server to receive multiple service requests from the client group as a single group service request and to issue multiple replies, one for each client, in response to the group service request. Therefore, interrogating (or remote invoking) an object group is as natural as interrogating a singleton object.

Our work is notable particularly with respect to giving generality to the group communication primitive. It goes beyond the requirements of communicating with the replicated server groups, in which all replies returned from the group are identical, and hence a single reply can be returned to the client. We also take into consideration the general requirements of communicating with homogeneous and heterogeneous server groups, in which multiple, variable, and different types of replies need to be returned to the client, in a controlled manner either as singleton replies or group replies (see chapter 2). Similarly, we also take into consideration the requirements of client group invoking a *group service request* on a singleton server object. Hence the proposed group communication primitive possesses *single request - multiple reply* (client side), *group request - single reply* (server side), and *group request - multiple reply* (server side) semantics, apart from other features outlined below.

Our work recognizes that reply handling transparency is sometimes impossible, and in many cases undesirable by the client applications. So clients must have access to multiple and different types of replies in a controlled manner.

The proposed group interrogation primitive is versatile and useful in a variety of application domains. The flexibility and the power of group interrogation is obtained through the following features (see chapter 3).

1. *multiple reply delivery capability*, which allows the same or different types of replies to be delivered to the client one after another,
2. *variable reply delivery capability*, which allows the client to receive an end of reply notification from the underlying group support platform,
3. *group reply delivery capability*, which permits both the generic *signature-based* reply collation semantics as well as the application-specific *content-based* reply collation semantics,
4. *controlled reply delivery capability*, which allows the client to explicitly solicit the replies as and when required by the client,
5. *terminable reply delivery capability*, which allows the client to terminate the delivery of the rest of the replies if it has received a sufficient number of them or the desired ones.
6. *ordered reply delivery capability*, which allows the client to receive replies of different types or replies from different sources in a desired order; reply delivery ordering is specified in our model as part of collation policy.

Transparency is an important aspect of a programming primitive. Our model allows the programmer to configure the level of group transparency by specifying different message distribution and collation policies (see section 7.4 to section 7.8). A notable feature of our model is that we have proposed a generic message collation scheme, which we call *signature-based* collation scheme (see section 3.6). This scheme allows a (client | server) object to receive a group (reply | service request) without modification of its contents. Hence, the client applications can process the group message in an application-specific manner.

The semantics of the group interrogation primitive has an impact on the message invocation, reception, and processing requirements of the client and server objects. We describe the characteristics of the group-oriented client and server objects which are capable of invoking, receiving and processing group interrogation messages (see chapter 4).

The proposed group interrogation primitive is a powerful and flexible programming language level communication primitive which supports ‘*request-response*’ style communication between client group and server group.

11.1.2 Contribution at the Platform-Level

The focus of our thesis is on *middleware-level support* for group-based distributed applications. This middleware resides on top of the existing low-level group communication protocols. The main contribution of the thesis at the *distributed platform level* is the software architecture of an *agent-based and policy-driven group support middleware platform*. This is an extensible, configurable and programmable architecture which permits the separation of group coordination aspects from application issues. The group support platform is composed of many components. These are summarised below.

1. *Group support agents*: We have identified a set of middleware-level *group support services* (GSSs*),* required by many applications (see chapter 4). Some of the commonly required group support services are message *distribution service*, message *collation se*rvice, message *filtering service*, message invocation and delivery *synchronisation service*, etc. These services are offered by the corresponding *group support agents* (GSAs). We identify the different aspects of each group support service which form the basis of the design of the group policy specification language.
2. *Group support machine*: Group-based applications usually need a combination of group support services, rather than individual ones. These services need to interact with each other in order to support diverse application requirements. An important contribution of this thesis is the design of an architectural framework for the organisation and configuration of these group support services in the group support platform (see chapter 5). This framework is called the *group support machine* (GSM). GSM is composed of a configuration of GSAs which interact with each other locally via the inter-agent interfaces. We describe how the components of this software machine work together in the provision of group support services to the applications (see chapter 6). Each member (or component) of the group-based application is supported by a GSM. The set of GSMs communicating with each other through an inter-GSM protocol (IGP) constitutes a *group support platform* (GSP).
3. *Inter-GSM Protocol*: The GSM is a multi-agent machine. The GSAs cannot offer their services independently or in isolation. Instead these agents need to communicate locally with other agents in the GSM, as well as remotely with their peers in other remote GSMs, in order to provide the required group support services to the applications. The remote communication between the peer GSAs in different GSMs occurs through the inter-GSM protocol (IGP) (see chapter 9). The IGP describes the information that is exchanged between the peer GSAs, the format in which this information is exchanged, and the handshaking involved between the GSAs.
4. *Programmable and Policy-Driven Group Support Platform*: We distinguish agents from policies which determine what those agents will do (see chapter 6). The GSP is composed of generic and policy-neutral GSAs, which can be programmed to offer their services according to application requirements for message distribution, collation, synchronisation, filtering, etc. (see chapter 7). These requirements are specified using group policy specification language. The GSP is driven by these policies which are specified as *policy scripts* and stored in the GSM.
5. *Group Policy Specification language*: Each group support service has different aspects to it. These aspects correspond to different application requirements with respect to that service. They have a certain commonality over different group support services. They can be modeled as elements of a group policy specification language. We have developed a language framework, based upon these elements, for the specification of group support policies (see chapter 8). The language permits the high-level and declarative specification of message distribution, collation, synchronisation and filtering requirements of an application, in an abstract manner independent of the mechanisms or protocols needed to implement them. It permits the separation of application concerns from group coordination concerns which are specified external to the applications, i.e., inside the GSM. Changes to group coordination behaviors are possible by modifying relevant group support policies in the GSM, without any change to the application code.

6. *Group coordination models*: Many existing distributed applications, in different domains, consist of multiple client and server components, which interact with one another on a one-to-one basis, thereby sacrificing the parallelism and performance inherent in these applications. One of the main contributions of our work is the visualization of these applications as group-based applications and casting (or modeling) them as a client group interacting with a server group. The intra-group and the inter-group interactions between the members of the client and server group can be viewed at a high-level as *group coordination patterns* or *group coordination behavior*. A *group coordination model* is a combination of *group coordination behavior* within a given *group organisation*. In this thesis we have shown how different group coordination patterns (or behaviors) can be obtained by composing the basic group support services, such as message distribution, collation, synchronisation, filtering service, etc., in different combinations (see chapter 7). The combination of these basic services in different group organisations yields different group coordination models. We have represented group coordination patterns as programmable coordination behaviors which can be specified as message distribution, collation, synchronisation, and filtering policies.
7. *Performance*: The performance of the proposed group support platform is comparable to and in some aspects better than the performance of conventional client-server based middleware platforms such as Corba. The implementation of the proposed model and an evaluation of the performance characteristics of the model reveal that the message count is the same in both the platforms. The message complexity is much less in GSP. The messages (i.e., the GPDUs) exchanged between the GSMs have fewer control information than the messages exchanged between the ORBs in the Corba platform. The response time is much better in GSP because the client gets a group reply from the server group in a single "invocation round trip time delay". Additionally, the implementation of the model was carried out using advance Java multi-threading and thread synchronisation techniques, thereby allowing the support of non-blocking invocations and solicited reply delivery semantics.

11.2 Directions for Future Work

The work presented in this thesis provides a basis for many future research and development activities. As the new group-based distributed applications emerge in different application domains and proliferate in the commercial arena, the need to provide a *programming-level* support and *platform-level* support in commercially available distributed systems will be increasingly recognised. We believe that the model we have presented in the thesis for the *programming-level* and *distributed platform level* support for group-based distributed applications provides a good starting point for the design of such systems.

11.2.1 Research on Group-Oriented Programming Languages & Systems

The group communication primitive, *group interrogation*, proposed in the thesis is a candidate for implementation in programming languages for use in real applications. The semantics of this primitive has an impact on the client and server applications, as well as on the programming languages used in the development of these applications. Both the applications and the programming languages need to be *group-oriented*. This can be very easily seen as explained below.

When a client invokes a group interrogation on the server group, the client application and hence the programming language should be capable of handling multiple replies, group replies, and of receiving end of reply notifications from the underlying group support platform. Similarly when the server receives a group interrogation from the client group, it should be capable of handling group service requests and of generating multiple replies. These and other aspects of group interrogation, as explained below, require enhancements to the current programming languages. This represents a fertile area of research in programming languages.

1. *Multiple reply semantics*: The programming languages need to be capable of handling multiple and different types of replies in response to group interrogation. Current programming languages return the thread of control to the client as soon as a single reply is received. In server applications, the server objects should be capable of generating multiple and different types of replies in response to a group service request. Current server applications return a single reply only.
2. *Non-blocking semantics*: The non-blocking invocation semantics imply that a client can invoke multiple group interrogations without waiting for replies of the previous invocation. This calls for multi-threaded clients and need for multiple termination (or reply) handlers corresponding to different types of expected replies. This semantics also imply that the invocation mechanism must be capable of accepting an "invocation instance identifier" in response to the invocation of a group interrogation.
3. *Controlled reply semantics*: This semantics imply that the programming languages provide a handle, such as "poll_reply()", as suggested in the thesis or employ some other mechanism to give the client the control to receive the replies as and when it wants them, at the desired pace.
4. *Terminable reply semantics*: This semantics imply that the programming languages provide a handle, such as "terminate_replies()", as suggested in the thesis to give the client the control to terminate the replies when it does not want any more replies.
5. *Variable reply semantics*: This implies that the programming languages provide a special termination signature, such as "end_of_reply()", or some other mechanism to indicate the last reply.
6. *Group reply or group request semantics*: This semantics has very little impact on existing client and server applications. A group message (reply or request) can be constructed using existing programming language structures such as an array or a linked list.

11.2.2 Integration of GSM Model in CORBA

An important future activity is the integration of the proposed model of GSM in popular distributed platforms such as CORBA. Adding GSM support in a CORBA environment is a challenging task that requires various considerations. Whether the GSM can be integrated within the Object Request Broker (ORB) or placed outside the ORB as an external *object service* is an issue which requires further consideration.

The D-Agent and the C-Agent of the GSM model perform, amongst other functions, the message marshalling (encoding) and unmarshalling (decoding) functions. They also encode and decode the group protocol data units (GPDUs) which are the units of exchange in the inter-GSM protocol (IGP). Similar functions are performed by the *stubs* and *skeletons* of the CORBA model. The integration of the GSM model in CORBA need to take into consideration these common issues.

While IIOP is an inter-ORB protocol, the IGP is an inter-GSM protocol. Therefore, the synergy of these protocols depends upon whether the GSM is integrated within the ORB or outside of it. The proposed model of group support platform is intended to enhance the middleware support available from currently available distributed platforms, such as CORBA, in order to support an important class of distributed applications - the group-based distributed applications.

11.2.3 Extension of GSM Model

We have proposed a model of GSM which contains a sub-set of group support services commonly required by many group-based applications. However the GSM serves as a framework within which new group support services can be identified and their interaction with the existing ones defined. Group support services tend to be domain-specific and hence new services may be identified as new application domains are considered. In such cases, it is required to identify the relationship of the new services with the existing ones and to define a new configuration of group support services. This also has an impact on the existing inter-GSM protocol.

11.2.4 Extension to Group Policy Specification Language

The group policy specification language (GPSL), as defined in the thesis, is capable of specifying message distribution policy, collation policy, synchronisation policy, and filtering policy. In other words, the language is capable of specifying the application's requirements with respect to a limited set of group support services. It will be interesting to explore other group support services and find out the different issues involved in the provision of these services, as well as to see if the existing GPSL language framework can specify the new service's requirements or if an extension to the language framework is required.

List of References

[1] International Standard ITU-T X.901 / ISO 10746-1: Basic Reference Model of Open Distributed Processing - Part-1: Overview.

[2] International Standard ITU-T X.902 / ISO 10746-2: Basic Reference Model of Open Distributed Processing - Part-2: Descriptive Model

[3] International Standard ITU-T X.903 / ISO 10746-3: Basic Reference Model of Open Distributed Processing - Part-3: Prescriptive Model

[4] Draft International Standard ITU-T X.904 / ISO 10746-4: Basic Reference Model of Open Distributed Processing - Part-4: Architectural Semantics

[5] Farooqui, K., Logrippo, L., and de Meer, J. The ISO Reference Model for Open Distributed Processing: An Introduction. *Computer Networks and ISDN Systems*, Vol. 27, 1995, 1215-1229.

[6] Farooqui, K., and Logrippo, L. Architecture for Open Distributed Software Systems. In Zomaya, A.Y.H. eds. *Parallel and Distributed Computing Handbook*, McGraw-Hill, 1996, 303-329.

[7] Linington, P.F., Introduction to the Open Distributed Processing Basic Reference Model, *Proceedings of the IFIP International Workshop on Open Distributed Processing*, Berlin October 1991, North-Holland, (1992), 3 - 14.

[8] Herbert, A. The Challenge of ODP, *Proceedings of the IFIP International Workshop on Open Distributed Processing*, Berlin October 1991, North-Holland, (1992), 15 - 28.

[9] ANSA Reference Manual, Volume A, B, C., Release 01.01, Architecture Projects Management Limited, Cambridge, U.K., July 1989.

[10] ANSA Computational Model, AR.001.01, Architecture Projects Management Limited, Cambridge, U.K., February 1993.

[11] ANSA: An Application Programmer's Introduction to the Architecture, TR.017.00, Architecture Projects Management Limited, Cambridge, U.K., November 1991.

[12] ANSA: An Engineer's Introduction to the Architecture. TR.03.02, Architecture Projects Management Limited, Cambridge, U.K., November 1989.

[13] ANSA: A System Designer's Introduction to the Architecture. RC.253.00, Architecture Projects Management Limited, Cambridge, U.K., April 1991.

[14] ANSA Technical Report, Management in Object-Based Federated Distributed Systems, TR.39.00, Architecture Projects Management Limited, Cambridge, U.K., February 1993.

[15] Edwards, N. Open Dependable Distributed Systems, ANSA Phase 3 Technical Report APM. 1145.01, Architecture Projects Management Limited, Cambridge, U.K., March 1994.

[16] RACE Open Service Architecture, 13th Deliverable, The ROSA Handbook, Release 2, RACE Project R1093. December 1992.

[17] Schoo, P., and Tonnby, I. The ROSA Object Model, *Proceedings of the IFIP International Workshop on Open Distributed Processing*, Berlin October 1991, North-Holland, (1992), 291-300.

[18] Object Management Group, "The Common Object Request Broker: Architecture and Specification", Rev. 2.0, 1995.

[19] Open Software Foundation. (1994) OSF DCE Application Development Guide. Open Software Foundation, revision 1.0, edition 1994.

[20] The Component Object Model Specification Microsoft (1995).

[21] Open Services Architecture within Integrated Services Engineering, CASSIOPEIA RACE Ref: R2049, External Deliverable, R2049/CRA/SAR/DS/P/014/b1, February 1994.

[22] Iggulden, D., Rees, O., and van der Linden, R. Architecture & Frameworks, ANSA Phase 3 Technical Report, APM.1017.00.03, Architecture Projects Management Limited, Cambridge, U.K., June 1993.

[23] Taylor, C.J. Object-Oriented Concepts in Distributed Systems, *Computer Standards and Interfaces*, Vol. 15, No. 2/3, 1993.

[24] Jul, E. Separation of Distribution and Objects, *Proceedings of the Workshop on Object-Based Distributed Programming in conjunction with 7th ECOOP'93* in *Lecture Notes in Computer Science 791*, Springer-Verlag, 1994, 47-54.

[25] TINA-C Deliverable, Overall Concepts and Principles of TINA, Version 1.0, Document Label: TB_MDC.018_1.0_94, Telecommunication Information Networking Architecture Consortium, February 1995.

[26] TINA-C Deliverable, Service Architecture Version 2.0, Document Label: TB_MDC.012_2.0_95, Telecommunication Information Networking Architecture Consortium, March 1995.

[27] TINA-C Deliverable, TINA-C Service Design Guidelines, Version 1.0, Document Label: TP_JS_001_0.1_95, Telecommunication Information Networking Architecture Consortium, March 1995.

[28] TINA-C Deliverable, Computational Modeling Concepts, Version 2.0, Document Label: TB_A2.HC.012_1.2_94, February 1995.

[29] TINA-C Deliverable, Engineering Modeling Concepts (DPE Architecture), Version 2.0, Document Label: TB_NS.005_2.0_94, December 1994.

[30] TINA-C Deliverable, Management Architecture, Version 2.0, Document Label: TB_GN.010_2.0_94, Telecommunication Information Networking Architecture Consortium, December 1994.

[31] TINA-C Deliverable, Connection Management Architecture, Document Label: TB_JJB.005_1.5_94, Telecommunication Information Networking Architecture Consortium, March 1995.

[32] TINA-C Deliverable, TINA Object Definition Language (TINA-ODL) Manual, Version 1.3, Archiving Label: TR_NM.002_1.3_95, Telecommunication Information Networking Architecture Consortium, June 1995.

[33] ISO/IEC 13244 / ITU-T Draft Rec. X.703, Open Distributed Management Architecture, 1997.

[34] Oskiewicz, E., and Edwards, N. A Model for Interface Groups. ANSA Phase 3 Technical Report, APM. 1002.01, APM Limited, Cambridge, U.K., May 1994.

[35] Achmatowicz, R. Object Groups For Groupware Applications: Application Requirements and Design Issues. Technical Report No. 685, Queen Mary and Westfield College, Department of Computer Science, London, U.K., September 1994.

[36] Watanabe, T., and Yonezawa, A. An actor-based meta-level architecture for group-wide reflection. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages, Noordwijkerhout, Netherlands, May 1990, Lecture Notes in Computer Science 489*, Springer-Verlag, 1991, 405-425.

[37] Matsuoka, S., Watnabe, T., and Yonezawa, A. Hybrid group reflective architecture for object-oriented concurrent reflective programming. *European Conference on Object Oriented Programming*, 1991, 231-250.

[38] Zweiacker, M. The Persistent Object Group Service-An approach to fault tolerance of open distributed applications. *Proceedings of the IFIP/IEEE International Conference on Open Distributed Processing and Distributed Platforms*, May 1997, Toronto, Chapman & Hall (1997), 224-235.

[39] Pardyak, P. Group Communication in an Object-Based Environment. *Proceedings of the 2nd International Workshop on Object-Orientation in Operating Systems IWOOS'92*, September 1992, 106-116.

[40] Murata, S., Shionozaki, A., Tokoro, M. A Network Architecture for Relaible Process Group Communication. *Proceedings of the 14th International Conference on Distributed Computing Systems*, 1994, 66-73.

[41] Glade, B.B., Birman, K.P., Cooper, R.C., and van Renesse, R. Lightweight process groups. *Proceedings of the OpenForum'92 Technical Conference*, Utrecht, The Netherlands, November 1992, 323-336.

[42] Maffeis, S. The Object Group Design Pattern. *Proceedings of the 1996 USENIX Conference on Object-Oriented Technologies*, Toronto, Canada, June 1996.

[43] Versimmo, P. and Rodrigues, L. Group Orientation: A Paradigm for Distributed Systems of the Nineties. In *Proceedings of the 3rd Workshop on Future Trends of Distributed Computing Systems*, IEEE Computer Society Press, April 1992, 57-63.

[44] Powell, D., ed. *Delta-4: A Generic Architecture for Dependable Distributed Computing* (1991) Springer-Verlag, Berlin.

[45] Birman, K.P. The process group approach to reliable distributed computing. *Communication of ACM* (December 1993), Vol. 36, No. 12, 36-53.

[46] Liang, L, Chanson, S.T., and Neufeld, G.W. Process groups and group communications: classification and requirements. *IEEE Computer*, Vol. 23, No. 2, (February 1990), 56-66.

[47] Achmatowicz, R. Object Groups For Groupware Applications: Application Requirements and Design Issues. Technical Report No. 685, Queen Mary and Westfield College, Department of Computer Science, London, U.K., September 1994.

[48] Benford, S. and Palme, J. A Standard for OSI Group Communication. *Computer Networks and ISDN Systems*, Vol. 25, (1993), 933-946.

[49] Cosquer, F.J.N. and Versimmo, P. The Impact of Group Communication Paradigms on Groupware Support. In *Proceedings of the 6th Workshop on Future Trends of Distributed Computing Systems*, Korea, IEEE Computer Society Press, 1995, 207-214.

[50] Rodrigues, L., and Versimmo, P. Replicated Object Management using Group Technology, In *Proceedings of the 4th Workshop on Future Trends of Distributed Computing Systems*, Lisbon, Portugal, September 1993, IEEE Computer Society Press, 54-61.

[51] Babaoglu, O., and Schiper, A. On Group Communication in Large-Scale Distributed Systems. In *ACM SIGOPS Operating Systems Review*, July 1993, 62-67.

[52] Prinz, W. Survey of Group Communication Models and Systems. In *Computer Based Group Communication, the AMIGO Activity Model*, Ellis Horwood, 1989.

[53] Szyperski, C., and Ventre, G. Efficient Group Communication with Guaranteed Quality of Service. *Proceedings of the 4th Workshop on Future Trends of Distributed Computing Systems*, Lisbon, Portugal, September 1993, IEEE Computer Society Press, 150-156.

[54] Chanson, S.T., Neufeldm G.W., and Liang, L. A Bibliography on Multicast and Group Communication. *Operating Systems Review*, Vol. 23, No. 4, October 1989, 20-25.

[55] Navarathnam, S., Chanson, S., and Neufeld, G. Reliable Group Communication in Distributed Systems. *Proceedings of the 8th International Conference on Distributed Computing Systems*, San Jose, California, June 1988, 439-446.

[56] Birrel, A.D., and Nelson, B.J. Implementing remote procedure calls, *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1984, 39-59.

[57] Ananda, A.L., and Tay, B.H. An Asynchronous Remote Procedure Call Facility. *Proceedings of 11th International Conference on Distributed Computing Systems*, May 1991, 172-179.

[58] Bershad, B.N., Anderson, T. E., Lazowska, E.D., and Levy, H. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 1990, 37-55.

[59] Martin, B., Bergan, C., and Russ, B. Parpc: A system for parallel procedure calls. *ICPP*, 1988, 449-452.

[60] Wilbur, S. and Bacarisse, B. Building distributed systems with remote procedure calls. *Software engineering journal*, September 1987, 148-159.

[61] Yap, K.S., Jalote, P., and Tripathi, S. Fault tolerant remote procedure call. *International Conference on Distributed Computing Systems*, 1988, 48-54.

[62] Johnson, D., and Zwaenepoel, W. The Peregrine high-performance RPC system. *Software Practice & Experience*, Vol. 23, No. 2, 1993, 201-222.

[63] Liskov, B., and Shrira, L. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. *ACM SIGPLAN Notices*, Vol. 23, No. 7, July 1988.

[64] Birman, K.P., and van Renesse, R. RPC Considered Inadequate. In *Birman, K. and van Renesse, R. eds. Reliable Distributed Computing with ISIS Toolkit*, IEEE Computer Society Press, 1994, 68-78.

[65] Ramakrishna, S., Prasad, B., Thenmozhi, A., Samdarshi, S., Velaga, K., Shah, K., and Ravindran, K. Design of broadcast programming primitives for distributed systems. *Computer Communications*, Vol. 16, No. 9, September 1993, 557-567.

[66] Hughes, L. A Multicast Response-Handling Taxonomy, *Computer Communications*, Vol. 12, No. 1, February 1989, 39-46.

[67] Maffeis, S. Distributed Programming Using Object Groups, IFI TR 93.38, Department of Computer Science, University of Zurich, Zurich, Switzerland, September 1993.

[68] Maffeis, S. A Flexible System Design to Support Object-Groups and Object-Oriented Distributed Programming. *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, Lecture Notes in Computer Science 791, Springer-Verlag 1994, 213-224.

[69] Birman, K.P., Cooper, R., Gresman, B. Programming with Process Groups: Group and Multicast Semantics. TR-91-1185, Cornell University, Ithaca, USA, January 1991.

[70] Cheriton, D.R. Request-response and multicast interprocess communication in the V kernel. *Lecture Notes in Computer Science 248,* Springer-Verlag, 1986.

[71] Hagsand, O., Herzog, H., Birman, K., and Cooper, R. Object-Oriented Relaible Distributed Programming. *IEEE Workshop on Object-Orientation in Operating Systems*, September 1992.

[72] van Renesse, R., and Birman, K.P., Fault-Tolerant Programming using Process Groups. In F. Brazier and D. Johansen, eds., *Distributed Open Systems*, IEEE Computer Society Press, 1994.

[73] Zhou, W. A Fault-Tolerant Remote Procedure Call System for Open Distributed Processing. *Proceedings of the International Conference on Open Distributed Processing*, Brisbane, Australia, February 1995.

[74] Wood, M.D. Replicated RPC using Amoeba closed group communication. *Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, 1993.

[75] Cooper, E.C. Programming Language Support for Multicast Communication in Distributed Systems. *Proceedings of the International Conference on Distributed Computing Systems*, 1990, 450-457.

[76] Welling, G., and Badrinath, B.R. An Architecture of a Threaded Many-to-Many Remote Procedure Call. *Proceedings of the 12th International Conference on Distributed Computing Systems*. 1992, 504-511.

[77] Pardyak, P., and Bershad, B.N. A Group Structuring Mechanism for a Distributed Object-Oriented Language. *Proceedings of the 14th International Conference on DIstributed Computing Systems*, 1994, 312-319.

[78] Hiltunen, M.A., and Schlichting, R.D. Constructing a Configurable Group RPC Service. *Proceedings of the 15th International Conference on Distributed Computing Systems*, 1995, 288-295.

[79] Wang, X., Zhao, H., and Zhu, J. GRPC: A Communication Cooperation Mechanism in Distributed Systems. *ACM SIGOPS*, January 1995, 75-86.

[80] Farooqui, K., and Logrippo, L. Group Interrogation: A Group Programming Primitive. *Proceedings of the IFIP/IEEE International Conference on Open Distributed Processing and Distributed Platforms*, May 1997, Toronto, Chapman & Hall (1997), 34-47.

[81] Melliar-Smith, P.M., Moser, L.E., and Agrawala, V. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1, January 1990, 17-25.

[82] Reiter, M.K. Secure agreement protocols: Relaible and atomic group multicast in Rampart. In *Proceedings of 2nd ACM Conference on Computer and Communications Security* (Fairfax, November 1994), 68-80.

[83] Whetten, B. A reliable multicast protocol, In *Theory and Practice of Distributed Systems*. K.P. Birman, F. Mattern, and A.Schiper, eds., *Lecture Notes in Computer Science 938*, Springer-Verlag.

[84] Birman, K., Schiper, A., and Stephenson, P. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, Vol. 9, No. 3, August 1991, 272-314.

[85] Hadzilacos, V., and Toueg, S. Fault-tolerant broadcasts and related problems. In S. Mullender, ed., *Distributed Systems*, Addison-Wesley, Reading, Mass., 1993.

[86] Birman, K., Schiper, A., and Sephenson, P. Lightweight causal and atomic group multicast. *ACM Transactions of the ACM*, Vol. 36, No. 12, 37-53.

[87] Navaratnam, S., Chanson, S.T., and Neufeld, G. Reliable group communication in Distributed Systems, In *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988, CS Press, Los Alamitos, California, 439-446.

[88] Schiper, A. and Sandoz, A. Uniform Relaible Multicast in a Virtually Synchronous Environment. In *Proceedings of 13th International Conference on Distributed Computing Systems*, May 1993, 501-568.

[89] Nakamura, A., and Takizawa, M. Priority-Based Total and Semi-Total Ordering Broadcast Protocols. *Proceedings of the International Conference on Distributed Computing Systems*, 1992, 178-185.

[90] Luan, S.W. and Gilgor, V.D. A Fault-Tolerant Protocol for Atomic Broadcast. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 3, 1990, 271-285.

[91] Kaashoek, M.F., Tanenbaum, A.S., Hummel, S.F., and Bal, H.E. An Efficient Relaible Broadcast Protocol, *ACM Operating Systems Review*, Vol. 23, No. 4, 1989, 5-19.

[92] Ezhilchelvan, P.D., Macedo, R.A., and Shrivastava, S.K. Newtop: A Fault-Tolerant Group Communication Protocol. *Proceedings of the International Conference on Distributed Computing Systems*, 1995, 296-306.

[93] Nakamura, A., and Takizawa, M. Causally Ordering Broadcast Protocol. *Proceedings of the International Conference on Distributed Computing Systems*, 1994, 48-55.

[94] Anceaume, E. A Comparison of Fault-Tolerant Atomic Broadcast Protocols. *Proceedings of the 4th Workshop on Future Trends of Distributed Computing Systems*, Lisbon, Portugal, September 1993, IEEE Computer Society Press, 1993, 166-172.

[95] Reiter, M.K. A secure group membership protocol. In *IEEE Transactions on Software Engineering*, Vol. 22, No. 1, (January 1996), 31-42.

[96] Jahanian, F., Fakhouri, S., and Rajkumar, R. Processor group membership protocols: Specification, design and implementation. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, October 1993.

[97] Diaz, M. and Villemur, T. Membership services and protocols for cooperative frameworks of processes. *Computer Communications*, Vol. 16, No. 9, September 1993, 548-556.

[98] Amir, Y., Dolev, D., Kramer, S., and Malki, D. Membership Algorithm for Multicast Communication Groups. *Proceedings of 6th International Workshop on Distributed Algorithms*, November 1992, 292-312.

[99] Melliar-Smith, P.M., Moser, L.E., and Agarwala, V. Membership Algorithms for Asynchronous Distributed Systems. *Proceedings of International Conference on Distributed Computing Systems*, May 1991, 480-488.

[100] Birman, K.P. and Joseph, T.A. Exploiting Virtual Synchrony in Distributed Systems. In *11th Symposium on Operating Systems Principles*, November 1987, 123-138.

[101] Moser, L.E., Amir, Y., Melliar-Smith, M., and Agarwal, D.A. Extended Virtual Synchrony. *Proceedings of 14th International Conference on Distributed Computing Systems*, 1994, 56-65.

[102] Schiper, A., and Sandoz, A. Uniform Relaible Multicast in a Virtually Synchronous Environment. *Proceedings of the 13th International Conference on Distributed Computing Systems*, 1993, 561-568.

[103] Schiper, A. and Ricciardi, A. Virtually Synchronous Communication based on a weak failure suspector. In *Proceedings of the 23rd International Conference on Fault Tolerant Computing Systems*, June 1993.]

[104] Birman, K. and Cooper, R. The ISIS Project: Real Experience with a Fault-Tolerant Programming System. *ACM SIGOPS Operating Systems Review*, Vol. 25, No. 2, April 1991, 103-107.

[105] Birman, K.P., and van Renesse, R. *Reliable Distributed Computing with the ISIS Tool Kit*. IEEE Computer Society Press, Los Alamitos, California, ISBN 0-8186-5342-6, 1994.

[106] Orbix + ISIS Programmer's Guide, Document D071-00, ISIS Distributed Systems Inc., IONA Technologies Limited, 1995.

[107] van Renesse, R., Birman, K.P., and Maffeis, S. Horus: A flexible group communication system. *Communications of ACM, Vol. 39, No. 4,* (April 1996), 76-83.

[108] van Renesse, R., Birman, K.P., Friedman, R., Hayden, M., and Karr, D.A. A framework for protocol composition in Horus. In *Proceedings of the 14th Symposium on the Principles of Distributed Computing ACM* (Ottawa, August 1995), 80-89.

[109] Maffeis, S. Adding group communication and fault-tolerance to CORBA. In *Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies* (Monterey California, USA, June 1995).

[110] Maffeis, S., and Schmidt, D.C. Constructing Reliable Distributed Communication Systems with CORBA. *IEEE Communications*, Vol. 35, No. 2, February 1997, 56-61.

[111] Kaashoek, M.F., and Tanenbaum, A.S. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th IEEE International Conference on Distributed Computing Systems*, May 1991, 222-230.

[112] Mullender, S., van Rossum, G., Tannenbaum, A., van Renesse, R., van Staveren, H. Amoeba - A Distributed Operating System for the 1990's. *IEEE Computer*, May 1990.

[113] Dolev, D., and Malki, D. The Transis Approach to High Availability Cluster Communication, *Communications of ACM, Vol. 39, No. 4,* (April 1996), 64-70

[114] Amir, Y., Dolev, D., Kramer, S., and Malki, D. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing* (July 1992), 76-84.

[115] Reiter, M.K. Distributing Trust with Rampart Toolkit, *Communications of ACM, Vol. 39, No. 4,* (April 1996), 71-74.

[116] Reiter, M.K. The Rampart toolkit for building high-integrity services. In K.P. Birman, F. Mattern, and A. Schiper, eds., *Theory and Practice in Distributed Systems* (Lecture Notes in Computer Science 938), 99-110. Springer-Verlag, 1995.

[117] Moser, L. E., Melliar-Smith, P. M., Agarwal, D.A., Budhia, R.K., Langley-Papadopoulos, C.A., Totem: A Fault-Tolerant Multicast Group Communication System, Communications of the ACM, April 1996, Vol.39, No. 4, 54-63.

[118] Babaoglu, O., Davoli, R., Giachini, L.A., and Baker, M.G. Relacs: A Communication Infrastructure for Constructing Relaible Applications in Large-Scale Distributed Systems. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, January 1995, 612-621.

[119] Cheriton, D.R., and Zwaenepoel. Distributed Process Groups in V Kernel. *ACM Transactions on Computing Systems*, Vol. 3, No. 2, May 1985, 77-107.

[120] Mishra, S., Peterson, L., and Schlichting, T. Consul: A Communications Substrate for Fault-Tolerant Distributed Programs, *Distributed Systems Engineering*, Vol. 1, 1993, 87-103.

[121] Powell, D., ed. *Delta-4: A Generic Architecture for Dependable Distributed Computing* (1991). ESPIRIT Research Reports, Springer-Verlag, Berlin.

[122] Costa, F.M., and Madeira, E.R.M. An object group model and its implementation to support cooperative applications on CORBA. In *A. Schill, C. Mittasch, O. Spaniol, and C. Popien, eds., Distributed Platforms*, Chapman & Hill (Publishers), Proceedings of the IFIP/IEEE International COnference on Distributed Platforms, 213-229.

[123] Bakker, H., and ter Hofte, G.H. MORB, a Multicast Object Request Broker for a CSCW software platform, Internal Paper, Telematics Research Centre, P.O. Box 589, 7500 AN Enschede, The Netherlands, 1997.

[124] Farooqui, K. ODP-Based Distributed Platform: Policy-Driven Engineering Support for Mobile and Group-Oriented Distributed Computing, *Proceedings of the IFIP/IEEE International Conference on Distributed Platforms - Industrial Session*, Dresden 1996, 290-297.

[125] Farooqui, K. and Logrippo, L. Group Communication Models, Computer Communications, Vol. 19, 1996, 1276 - 1288.

[126] Farooqui, K. and Logrippo, L. Group Support Platform: Middleware Support for Group-Based Distributed Applications, Submitted to Middleware'98 - IFIP International Conference on Distributed Platforms and Open Distributed Processing, The Lake District, England, September 1998.

[127] Moffet, J. and Sloman, M. Representation of Policies as System Objects. In *Proceedings of the Conference on Organizational Computer Systems*, Atlanta, Georgia, November 1991. SIGOIS Bulletin, Vol. 12, Nos. 2&3, 171-184.

[128] Dean, G., Rodden, T., Sommerville, I., and Hutchinson, D. Distributed Systems Management as a Group Activity, Technical Report, Department of Computing, Lancaster University, LA1 4YR, U.K.

[129] Koch, T. Policy-Based Management of Distributed Systems. Internal Paper, FernUniversitat, 58084 Hagen, Germany, 1996.

[130] Sloman, M. Policy Driven Management for Distributed Systems, *Journal of Network and Systems Management*, Plenum Press, Vol. 2, No. 4, 1994.

[131] Roos, J., Putter, P., and Bekker, C. Modeling Management Policy Using Enriched Managed Objects. *Integrated Network Management*, Vol. 3, North-Holland, 1993, 207-215.

[132] Alpers, B. and Plansky, H. Domain and Policy Based Management: Concepts and Implementation Architecture, *IFIP/IEEE Workshop on Distributed Systems Operations and Management*, Toulouse, October 1994.

[133] Meyer, B., and Popien, C. Defining Policies for Performance Management in Open Distributed Systems, *IFIP/IEEE Workshop on Distributed Systems Operations and Management*, Toulouse, October 1994.

[134] Popien, C. and Meyer, B. Service Request Description Language, FORTE'95.

[135] Trevor, J., Rodden, T., and Blair, G. COLA: A Lightweight Platform for CSCW. In *Proceedings of the European Conference on Computer Supported Cooperative Work*, September 1993, Milan, Italy, 15-30.

[136] Shenker, S., Weinrib, A., and Schooler, E. Managing Shared Ephemeral Teleconferencing State: Policy and Mechanism. Lecture Notes in Computer Science, Springer Verlag, 69 - 88.

[137] Bentely, R., and Dourish, P. Medium versus mechanism: Supporting collaboration through customisation. *4th European Conference on Computer Supported Cooperative Work*, Kluwer-Academic Publishers, 133-148.

[138] *Coordination Languages and Models*, Lecture Notes in Computer Science 1061, Springer-Verlag, 1996.

[139] Berry, A. and Kaplan, S. Language Support for Distribution in CSCW Systems. In *Proceedings of the International Workshop on Object Oriented Groupware Platforms*, (Part of ECSCW'97), Lancaster, U.K., September 1997, 61-67.

[140] Putter, P., and Roos, J.D. From Policy to Specification. *Proceedings of the IFIP International Workshop on Open Distributed Processing*, Berlin October 1991, North-Holland, (1992), 441-448.

[141] Cortes, M. and Mishra, P. DWCPL: A programming language for describing collaboration. In *ACM 1996 Conference on Computer Supported Cooperative Work*, November 1996, ACM Press.

[142] Papazoglou, M.P., Delis, A., Haghjoo, M., Bouguettaya, A. Language Support for Long-lived Concurrent Activities, *International Conference on Distributed Systems*.

[143] Frolund, S., and Agha, G. A Language Framework for Multi-Object Coordination. *7th European Conference on Object-Oriented Programming (ECOOP'93)* in *Lecture Notes in Computer Science 707*, Springer-Verlag, 1993, 346-359.

[144] Richard M. Adler, Distributed Coordination Models for Client/Server Computing, Computer, April 1995, Vol. 28, No. 4, 14-22.

[145] Nehmer, J., and Mattern, F. Framework for the organisation of cooperative services in distributed client-server systems. *Computer Communications*, Vol. 15, No. 4, May 1992, 261-269.

[146] Diaz, M. A logical model of cooperation. *Proceedings of the 3rd IEEE Workshop on Future Trends of Distributed Computing Systems*, April 1992, 64-70.

[147] Kreifelts, T., Pankoke-Babatz, U., Victor, F. A Model for the Coordination of Cooperative Activities. In *Proceedings of the International Workshop on CSCW*, Berlin 1991, 85-100.

[148] Kirsche, T., Lenz, R., Luhrsen, H., Meyer-Wegener, K., Wedekind, H., Bever, M., Schaffer, U., and Schottmuller, C. Communication support for cooperative work. *Computer Communications*, Vol. 16, No. 9, September 1993, 594-602.

[149] Gokhale, A.S., and Schmidt, D.C. Measuring the Performance of Communication Middleware on High-Speed Networks, ACM *SIGCOMM Conference*, 1996.

[150] Gokhale, A.S., and Schmidt, D.C. Measuring and Optimizing Corba Latency and Scalability Over High-Speed Networks, *IEEE Transactions on Computers*, Vol. 47, No.4, April 1998.

[151] Gokhale, A.S., and Schmidt, D.C. Optimizing a Corba Inter-ORB Protocol Engine for Minimal Footprint Embedded Multimedia Systems, *IEEE Journal on Selected Areas in Communications*, September 1999.

[152] Schmidt, D.C., Levine, D.L., Cleeland, C. Architectures and Patterns for High-Performance, Real-time ORB Endsystems, Advances in Computers, Academic Press, Ed., Zelkowitz, M. (to appear).

[153] Schmidt, D.C. and Gokhale, A. Techniques for Optimizing Corba Middleware for Distributed Embedded Systems, *Proceedings of INFOCOM'99*, March 1999.

[154] Schmidt, D.C. Evaluating Architectures for Mult-threaded Corba Object Request Brokers, *Communications of ACM*, Special Issue on Corba, Vol.41, No.10, October 1998.

Glossary of Abbreviations

1. OPR-message: Operation message, section 1.5.2
2. REP-message: Reply (or termination) message, section 1.5.2
3. NTF-message: Notification message, section 1.5.2
4. GI: Group Interrogation, section 3.2.
5. GSS: Group Support Service, section 4.3
6. GSA: Group Support Agent, section 5.2
7. GSM: Group Support Machine, section 5.3
8. GSP: Group Support Platform, section 5.4
9. GII: GSM Invocation Interface, section 6.2.1.1
10. GMI: GSM Management Interface, section 6.2.1.2
11. GNI: GSM Network Interface, section 6.2.2
12. DMI: Distributor Management Interface, section 6.2.1.2
13. CMI: Collator Management Interface, section 6.2.1.2
14. SMI: Synchroniser Management Interface, section 6.2.1.2
15. FMI: Filter Management Interface, section 6.2.1.2
16. GPSL: Group Policy Specification Language, chapter 8
17. DPP: Distribution Policy Primitive, section 8.4.1,
18. CPP: Collation Policy Primitive, section 8.4.2
19. SPP: Synchronisation Policy Primitive, section 8.4.3
20. FPP: Filter Policy Primitive, section 8.4.4
21. IGP: Inter-GSM Protocol, chapter 9
22. GPDU: Group Protocol Data Unit, section 9.4.

APPENDIX

BNF of Group Policy Specification Language (GPSL)

```
Group_Support_Policy_Specification::= Distribution_Policy_Specification
                                      | Collation_Policy_Specification
                                      | Synchronisation_Policy_Specfcation
                                      | Filtering_Policy_Specification

Distribution_Policy_Specfcation::= distribution_policy_specfcation_symbol
                                   for_specification
                                   distribution_policy
                                   end_policy_symbol

Collation_Policy_Specfcation::= collation_policy_specfcation_symbol
                                for_specification
                                collation_policy
                                end_policy_symbol

Synchronisation_Policy_Specfcation::= synchronisation_policy_specification_symbol
                                      for_specification
                                      sync_specification
                                      synchronisation_policy
                                      event_notification_policy
                                      end_policy_symbol

Filtering_Policy_Specfcation::= filtering_policy_specification_symbol
                                for_specification
                                filtering_policy
                                end_policy_symbol

distribution_policy::= DPP                          (* Distribution Policy Primitive *)

collation_policy::= CPP                              (* Collation Policy Primitive *)
                  | “(” collation_policy “followed_by” collation_policy “)”
                  | “(” collation_policy “interleaved_with” collation_policy “)”
                  | “(” collation_policy “disabled_by” collation_policy “)”
                  | “(” collation_policy “choice” collation_policy “)”
```

synchronisation_policy::= SPP (* Synchronisation Policy Primitive *)
| “(” synchronisation_policy “**and**” synchronisation_policy “)”
| “(” synchronisation_policy “**or**” synchronisation_policy “)”
| “(” synchronisation_policy “**xor**” synchronisation_policy “)”

filtering_policy::= FPP (* Filtering Policy Primitive *)

DPP::= “[”
“**distribute**” message_specification
“**to**” membership_specification
“**distribution_cardinality**” cardinality_specification
“**using**” ordering_specification
“]”

CPP::= “[”
“**deliver**” message_specification
“**from**” membership_specification
“**within**” | “**every**” time_specification
“**collation_cardinality**” cardinality_specification
“**collation_mode**” collation_mode_specification
“]”

SPP::= “[”
“**solicited_reception_of**” | “**unsolicited_reception_of**” message_specification
“**from**” membership_specification
“**within**” time_specification
“**sync_cardinality**” cardinality_specification
“]”

FPP::= “[”
“**amongst**” membership_specification
“**filtering_cardinality**” cardinality_specification
“**filtering_criterion**” filtering_criterion_specification
“**filtering_properties**” filtering_properties_specification
“]”

sync_specification::= “**sync**” message_specification “**with**”

event_notification_policy::= “**notify**” NPP (* Notification Policy Primitive *)

NPP::= “[”
“**sync_events**” message_specification_list
“**to**” membership_list
“]”
filtering_properties_specification::= “(” attribute_name_value_pair_list “)”

attribute_name_value_pair_list::= attribute_name_value_pair
| attribute_name_value_pair “,” attribute_name_value_pair_list

attribute_name_value_pair::= attribute_name “**=**” attribute_value

filtering_criterion_specification::=
filter_clause
| “(” filtering_criterion_specification “**and**” filtering_criterion_specification “)”
| “(” filtering_criterion_specification “**or**” filtering_criterion_specification “)”
| “(” filtering_criterion_specification “**xor**” filtering_criterion_specification “)”
| “(” “**not**” filtering_criterion_specification “)”

filter_clause::= “(” attribute_name **comparison_operator** attribute_value “)”

distribution_policy_specification_symbol::= “**operation_distribution_policy**”
| “**notification_distribution_policy**”
| “**termination_distribution_policy**”

collation_policy_specification_symbol::= “**operation_collation_policy**”
| “**notification_collation_policy**”
| “**termination_collation_policy**”

synchronisation_policy_specification_symbol::= “**synchronisation_policy**”

filtering_policy_specification_symbol::= “**filtering_policy**”

for_specification::= “**for**” message_name

message_specification::= message_signature
| “_REPLY_”
| “_REPLIES_”

membership_specification::= group_identifier
| member_name_list
| member_role_list

cardinality_specification::= “ATLEAST” “(” cardinal_expression “)”
| “ATMOST” “(” cardinal_expression “)”
| “UNSPECIFIED”

cardinal_expression::= integer
| “POS” “(” integer_list “)”
| “ANY” “(” integer, member_role “)”
| “ANY” “(” integer, “POS” “(” integer_list “)”

ordering_specification::= “UNORDERED_MULTICAST”
| “SOURCE_ORDERED_MULTICAST”
| “DESTINATION_ORDERED_MULTICAST”
| “ATOMIC_ORDERED_MULTICAST”

time_specification::= time_units

collation_mode_specification::= “MATRIX” “(” “ORDERED” | “ANY-ORDER” “,”
“FIRST” | “RECENT” | “ALL” “)”
| “LINEAR” “(” “FIRST” | “RECENT” | “ALL” “)”
| “SINGLETON” “(” “ORDERED” | “ANY-ORDER” “,”
“FIRST” | “RECENT” | “ALL” “)”

comparison_operator::= “**==**” | “**<**” | “**>**” | “**<=**” | “**>=**”

end_policy_symbol::= “**end_policy**”