JMSGROUPS: JMS COMPLIANT GROUP COMMUNICATION

THÈSE N^{O} 3341 (2005)

PRÉSENTÉE À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Institut d'informatique fondamentale

SECTION DES SYSTÈMES DE COMMUNICATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Arnas KUPŠYS

Bachelor of Science in Informatics, Kaunas University of Technology, Lituanie
et de nationalité lituanienne

acceptée sur proposition du jury:

Prof. A. Schiper, directeur de thèse
Prof. P. Felber, rapporteur
Prof. P. Sens, rapporteur
Prof. A. Wegmann, rapporteur

Lausanne, EPFL
2005

Abstract

Nowadays, computers are the indispensable part of our life. They evolve rapidly and are more and more versatile. Computer networks made the remote corners of the world just a click away. But unavoidably, any software and hardware component is subject to failure. Distributed systems spread on tens or hundreds of machines are particularly vulnerable to failures. Consequently, high availability and fault tolerance became a "must have" feature for such systems.

Software fault tolerance is achieved through the technique called replication. In replication several software replicas are executed at the same time. If one or several of them fail, other still provide the service. Software replication is often implemented using group communication, which provides communication primitives with various semantics and greatly simplifies the development of highly available and fault tolerant services.

However, despite tremendous advances in research and numerous prototypes, group communication stays confined to small niches and academic prototypes. In contrast, other technology, called message-oriented middleware such as the Java Message Service (JMS) is widely used in distributed systems, and has become a de-facto standard. We believe that the lack of a well-defined and easily understandable standard is the reason that hinders the deployment of group communication systems.

Since JMS is a well-established technology, we propose to extend JMS adding group communication primitives to it. Foremost, this requires to extend the traditional semantics of group communication in order to take into account various features of JMS, e.g., durable/non-durable subscriptions and persistent/non-persistent messages. The resulting new group communication specification, together with the corresponding API, defines group communication primitives compatible with JMS, that we call JMSGroups. To validate the specification and API we provide a prototype implementation of JMSGroups. As such, we believe it facilitates the acceptance of group communication by a larger community and provides a powerful environment for building fault-tolerant applications.

Résumé

De nos jours, les ordinateurs jouent un rôle indispensable dans notre quotidien. Ils évoluent rapidement et sont de plus en plus versatiles. Les réseaux informatiques rendent accessibles en un clic les coins les plus retirés du monde. Immanquablement, tout logiciel ou composant matériel est sujet à des pannes. Les systèmes distribués qui s'étendent sur des dizaines ou des centaines de machines sont particulièrement vulnérables aux fautes. En conséquence, haute disponibilité et tolérance aux fautes sont devenues, pour ces systèmes, une caractéristique essentielle. La tolérance aux fautes des logiciels est obtenue par une technique appelée réplication. Grâce à la réplication plusieurs copies d'un même logiciel sont exécutées simultanément. Si l'une d'entre elles tombe en panne, d'autres pourront assurer la disponibilité du service. La réplication de logiciel est souvent réalisée en utilisant la communication de groupe, qui definit des primitives de communication avec diverses sémantiques et simplifie beaucoup le développement du services hautement disponibles et tolérants aux fautes.

Malgré de grands progrès et la réalisation de nombreux prototypes, la communication de groupe demeure confinée dans des domaines très particuliers. Par comparaison, une autre technologie appelée, en anglais, *Message Oriented Middleware* (*MOM*), comme Java Message Service (JMS), est très utilisée dans les systèmes répartis et est devenue un standard de fait. Nous pensons que l'absence de standards freine le déploiement d'applications utilisant la communication de groupe.

Puisque JMS est une technologie bien établie, nous proposons d'étendre sa spécification en y ajoutant des primitives de communication de groupe. Tout d'abord, cela nécessite l'extension de la sémantique de la communication de groupe afin d'y inclure les divers aspects de JMS tels que, par exemple, les souscriptions durables/non-durables et les messages persistants/non-persistants. Cette nouvelle spécification, avec l'API correspondante, définit des primitives de communication de groupe compatibles avec JMS que nous appelons JMSGroups. Pour valider la spécification et l'API, nous avons implémenté un prototype de JMSGroups. Nous espérons que JMSGroups favorisera l'acceptation des communications de groupe dans une plus grande communauté et constituera une infrastructure robuste pour construire des applications réparties tolérantes aux fautes.

To my mother
Skiriu savo mamai

Acknowledgments

First of all, I would like to thank my supervisor, Prof. André Schiper for his guidance and support throughout my Ph.D., and for the time and effort he put into each paper we wrote together.

I also would like to express my gratitude to the members of the jury Prof. Pascal Felber, Prof. Pierre Sense and Prof. Alain Wegmann for the time invested in reading my thesis and for the comments they provided. Also I would like to thank Prof. Karl Aberer who kindly agreed to be the president of the jury.

Many thanks to the members of the Distributed Systems Laboratory (LSR) David Cavin, Allan Coignet, Richard Ekwall, Sergio Mena, Stefan Pleisch, Olivier Rutti, Yoav Sasson, Péter Urbán, Matthias Wiesmann and Paweł Wojciechowski for their inspiring discussions and helpful suggestions. I'm especially thankful to Stefan Pleisch for his collaboration on Chapters 4 and 7 and Matthias Wiesmann for his collaboration on Chapter 7. I also want to express my gratitude to our secretary France Faille for her care and help with all administrative work. I was very happy to work in the friendly atmosphere created by all these people.

I would like thank the reviewers of the conference papers for their time and valuable remarks. Special thanks to Sam Toueg for his comments and discussions related to group communication.

I express my gratitude to all my friends and colleagues in EPFL and especially to Petr Kouznetsov and Sidath Handurukande for their friendship and support. Also I'm very grateful to all my teachers at school and university, that I had honor to learn from.

Last but not least, I am specifically grateful to my wife Valdonė for her comprehension, support and for the patience during the long evenings when I was absent, and to my mother, who made a lot of sacrifices for me and always believed in me.

Table of Contents

1 Introduction **1**
- 1.1 Contributions . . . 2
- 1.2 Thesis Roadmap . . . 3

2 Group Communication and its Toolkits **5**
- 2.1 Group Communication . . . 5
  - 2.1.1 Agreement problems . . . 7
- 2.2 Group Communication Toolkits . . . 9
  - 2.2.1 Isis . . . 9
  - 2.2.2 Transis . . . 10
  - 2.2.3 Phoenix . . . 11
  - 2.2.4 JGroups . . . 13
  - 2.2.5 Object Group Service . . . 14
  - 2.2.6 Eternal System . . . 16
  - 2.2.7 Interoperable Replication Logic . . . 17
- 2.3 Summary . . . 19

3 Java Message Service **21**
- 3.1 The architecture . . . 23
- 3.2 JMS publish-subscribe . . . 24
- 3.3 JMS point-to-point . . . 25
- 3.4 JMS message delivery requirements . . . 26
- 3.5 JMS interfaces . . . 26
- 3.6 Summary . . . 28

4 JMS Compliant Group Communication: Semantics and API Mapping **29**
- 4.1 GC and JMS: a semantic mapping . . . 29
  - 4.1.1 The problem: the semantic gap . . . 29
  - 4.1.2 Bridging the gap . . . 30
- 4.2 JMSGroups specification . . . 32

4.2.1 Definitions . . . 32
4.2.2 Reliability guarantees of the broadcast primitive . . . 33
4.2.3 Ordering guarantees of the broadcast primitive . . . 35
4.3 Mapping GC primitives to JMS API . . . 35
4.3.1 Relevant JMS classes and methods . . . 36
4.3.2 JMSGroups API . . . 37
4.4 Summary . . . 41

5 Architectural issues **43**
5.1 Notation . . . 44
5.2 Fault tolerant JMS server architecture . . . 44
5.2.1 Non-replicated context . . . 46
5.2.2 Replicated context . . . 47
5.2.3 Comparison of the JMS server replication options . . . 49
5.3 JMSGroups based on JMS server . . . 50
5.4 Non-centralized architecture . . . 53
5.5 Related work . . . 54
5.6 Summary . . . 55

6 JMSGroups Implementation **57**
6.1 Fortika . . . 57
6.2 JORAM based implementation . . . 58
6.2.1 ScalAgent asynchronous agent platform . . . 59
6.2.2 Clustered topics in JORAM . . . 60
6.2.3 Integrating Fortika ABcast stack into JORAM . . . 61
6.3 Component Chain based implementation . . . 62
6.3.1 Framework architecture . . . 63
6.3.2 JMS wrapping . . . 65
6.3.3 CCB server replication . . . 65
6.4 Performance . . . 66
6.5 Use case - A Replicated Table . . . 69
6.5.1 Table group . . . 70
6.5.2 Member suspicions and removal . . . 71
6.6 Summary . . . 73

7 Replicated Invocation **75**
7.1 Replicated Invocation . . . 77
7.1.1 Deterministic servers . . . 78
7.1.2 Non-Deterministic servers . . . 78
7.2 Specification with Non-Deterministic Servers . . . 79

7.2.1 Invocation $C \longleftrightarrow R$ . . . 80
7.2.2 Invocation $R \longleftrightarrow S$ . . . 82
7.3 The Problem of Orphan Subtransactions with Replicated Invocation . . . 82
7.3.1 Pessimistic *vs*. optimistic server S . . . 83
7.4 Replicated Invocation Protocol . . . 84
7.4.1 Basic idea: sharing undo information among replicas of R . . . 84
7.4.2 The protocol . . . 85
7.4.3 Correctness Issues . . . 88
7.4.4 Evaluation . . . 89
7.5 Related Work . . . 90
7.6 Summary . . . 91

8 Conclusion **93**
8.1 Research Assessment . . . 93
8.2 Open Questions and Future Research Directions . . . 94

A Group Communication Toolkits API **101**
A.1 Isis . . . 101
A.2 Transis . . . 103
A.3 Phoenix . . . 103
A.4 JGroups . . . 104
A.5 Object Group Service . . . 105
A.6 Eternal System . . . 106
A.7 Interoperable Replication Logic . . . 107

B Replicated Table Source Code **109**
B.1 ReplTable.java . . . 109
B.2 ReplTableMsgListener.java . . . 113
B.3 Client.java . . . 116

List of Figures

2.1 The architecture of JGroups. . . . 14

3.1 Basic JMS architecture. . . . 23
3.2 JMS client-server communication. . . . 24
3.3 JMS publish-subscribe messaging model. . . . 25
3.4 JMS point-to-point messaging model. . . . 26
3.5 JMS object relationship. . . . 27

4.1 Conflicts between two messages that are reliably or atomically broadcast. . . . 36
4.2 JMS classes (the arrow shows the logical flow of the messages). . . . 37

5.1 Replicated JMS server. . . . 45
5.2 Different JMS server replication options. . . . 46
5.3 Client reconnection scenario. . . . 49
5.4 Two replication levels in JMSGroups. . . . 51
5.5 JMSGroups member's composite total order communication primitives. . . . 53
5.6 Non-centralized JMSGroups architecture. . . . 54

6.1 General component architecture. . . . 58
6.2 ScalAgent architecture. . . . 59
6.3 Clustered topic in JORAM. . . . 60
6.4 Network agent structure. . . . 62
6.5 The structure of a component. . . . 63
6.6 Component chain. . . . 64
6.7 ABcast early latency vs. group size (replicated context). . . . 67
6.8 ABcast average throughput vs. group size (replicated context). . . . 67
6.9 ABcast early latency vs. group size (non-replicated context). . . . 68
6.10 ABcast average throughput vs. group size (non-replicated context). . . . 68
6.11 Replicated table. . . . 70
6.12 Replicated table architecture. . . . 71

7.1 Nested request invocations (the processing of the request is shown by gray bars). 76
7.2 Deterministic server *R* is replicated using active replication. 78
7.3 Non-deterministic server *R* is replicated using passive replication (with a failure of the primary). 79
7.4 Representation in terms of (sub)transactions (invocations between *C*, *R*, and *S*). 80
7.5 An orphan subtransaction est_0 on pessimistic server *S*. 83
7.6 An orphan subtransaction est_0 on optimistic server *S*. 84
7.7 Message type declaration and predicate definition. 86
7.8 Replicated invocation protocol. 87
7.9 Primary R_0's failure after invoking pessimistic server *S* (the primary fails before backups are updated). 88

List of Tables

2.1 Isis system group API. 10
2.2 Transis system group API. 12
2.3 Phoenix object oriented API. 13
2.4 JGroups API. 15
2.5 OGS API. 16
2.6 Eternal ReplicationManager API. 17
2.7 IRL ReplicationManager API. 19
2.8 Group communication toolkits. 19
2.9 API comparison for two methods: *Join Group* and *Multicast*. 20

3.1 JMS interfaces. 27

4.1 The Mapping from GC primitives to JMSGgroups API (communication methods). . . . 38
4.2 The Mapping from GC primitives to JMSGgroups API (administrative methods). 42

5.1 Replicated JMS server: comparison between replicated and non-replicated context. . . . 50

6.1 Tested implementations. 66
6.2 The structure of a view message. 72
6.3 View table example. 72
6.4 Suspicion message structure. 72
6.5 Exclude message structure. 73

Chapter 1

Introduction

In the middle of the last century, when the first computers accessible for common people just started appearing, the main challenge for the developers writing the software for these machines, was to design an optimal algorithms for a given problem. This was done in order to use efficiently the limited resources (CPU power, memory, storage space, processor time, etc.) that these machines provided. At that time computer programs were executed on a single machine and the failure of hardware or software terminated the application completely.

To the present days the computer hardware was rapidly evolving, its computational power dramatically increasing and the price constantly in decline. Software also became much more diverse and complicated, but even more important with the rise of computer networks and the Internet it became distributed. Applications no more execute on a single processor, but instead are distributed on the machines across the network. They communicate to achieve the defined task and/or provide the service for the users. In such a system the failure of a component does not necessary terminate the whole application, but can put it in an unexpected/undefined state, which can be hard to detect and correct. This diversity of distributed environment and the not necessary predictable failure semantics makes distributed systems more failure-prone.

To make computer systems more robust and fault tolerant various techniques can be used, such as: self-checking/repairing, replication. Self-checking is usually used in hardware to detect faulty transistor states. Self-repairing is also used in hardware and is based on the idea that some general purpose devices can be dynamically reprogrammed to substitute failed hardware components. For software fault tolerance usually the replication technique is used, i.e., several software component replicas are executed at the same time (normally on different machines), so that in the case of a failure there are always enough replicas to provide the service. Even if such a technique seems intuitive and natural, it hides some fundamental issues, such as: replica state consistency, failure detection, agreement between the replicas. These and other issues are addressed by the computer science branch called *group communication*, which is at the heart of software replication for fault tolerance.

Group communication (also denoted by GC hereafter) has been an active area of research for more than a decade. The notion of process groups, with the possibility to multicast messages to the members

of a group, was initially proposed in the context of the V System [11], and later extended by the Isis system in the context of failures [7]. GC systems provide *one-to-many* communication primitives with various semantics (e.g., reliable delivery of messages and/or delivery of messages in total order). Such high-level communication abstractions among groups of processes greatly simplifies the development of highly available services. Yet, despite tremendous advances in research and numerous prototypes, e.g., [8, 43, 51, 15, 9], GC stays confined to small niches and to academic prototypes. One reason for this might be the lack of flexibility of existing GC prototypes. However, recent GC toolkits can be tailored to the application's needs [31, 50, 32], but still are not widely adopted in industry. Performance is also sometimes mentioned as a reason for the lack of acceptance of GC. Clearly, increased quality of service with respect to message delivery has a trade-off in performance. However, at the same time middleware systems that suffer from significant performance problems [25] have seen wide industry acceptance.

In contrast to GC, another communication technology has attracted considerable interest: the so-called message oriented middlewares (MOMs) such as MQSeries [33], Tuxedo [5], and Rendezvous [73]. This technology, which provides abstractions for asynchronous message sending, is increasingly used in industry and is now considered to be an integral part of an enterprise computing infrastructure. One of the key reasons for the success of MOMs (the Java based ones) has been the wide adoption of the Java Message Service (JMS) interface standard [30]. As can be seen with other technologies (e.g., HTTP and Java in the context of the World Wide Web), standards can be the driving force for the distribution and acceptance of a technology. Unfortunately, no widely accepted standard exists for GC. We believe that the lack of such a standard is one major reason for the limited distribution and acceptance of group communication in enterprise computing infrastructure. Thus, it is crucial to define a standard for GC that drives the acceptance of GC in industry.

In this thesis, we propose a standard interface (Application Programing Interface or API) for GC. Instead of specifying yet another GC API with probably low chances of becoming a standard (similarly as the existing GC APIs), we take advantage of the widespread acceptance of JMS and propose to extend it with GC functionality. Currently, JMS supports two paradigms: *point-to-point* and *publish-subscribe*. We add group communication as the third paradigm to JMS. The resulting specification and interface is called JMSGroups and should be easily understandable by both the GC community and developers familiar with JMS. As such, it facilitates the acceptance of group communication by a larger community and provides a powerful environment for building fault-tolerant applications.

1.1 Contributions

The main contributions of the thesis are the following:

Mapping and specification. JMSGroups is defined in the spirit of JMS; this requires to clearly identify the semantic differences between the two technologies and to "bridge" the semantic gap. As a "bridge" we present a semantic mapping between JMS and traditional GC. The result of this mapping provides a GC specification. We also extend the JMS API to incorporate GC features, while trying to keep the minimal deviation from the standard JMS specification.

Architecture and implementation. For the implementation of JMSGroups different architecture options were considered, namely *distributed* and *centralized*. A distributed architecture is used in most group communication toolkits. Inside an architecture there is no central entity and group members communicate directly with each other. This architecture is easy to implement given a GC toolkit (only a JMSGroups API adapter layer need to be added between the GC stack and the application). Unfortunately such an implementation inherits semantic inconsistencies between JMS and GC.

A centralized JMSGroups architecture is based on a central entity (server) which provides group communication as a service. Such architecture is compatible with the JMS specification, but is more complicated to implement. Moreover, the server must be made fault tolerant to eliminate a single point of failure.

We have chosen the centralized architecture for our JMSGroups implementation. Together with its advantage of JMS compatibility, this architecture allows us to explore different approaches to the implementation of group communication (when GC is provided as a service). Additionally, such architecture is easier to understand and to use for the developers familiar with JMS. At the same time, it is less expensive to integrate into existing enterprise systems.

Replicated invocation. In the scope of this work we analyzed the replicated invocation problem. The problem occurs when a replicated server calls another replicated server. We formalized the problem in the transactional environment and presented a solution.

1.2 Thesis Roadmap

The thesis is organized in eight chapters:

Basics. Chapter 2 introduces the basics of group communication and presents the most known group communication toolkits together with their interfaces. The introduction to the Java Message Service is given in Chapter 3.

Semantics and Mapping. Chapter 4 presents a semantic comparison between group communication and JMS, as well as a semantic and API mapping between the two. The mapping and API define *JMSGroups*. The specification of JMSGroups is also presented in the chapter.

Architecture. Chapter 5 analyzes the architectural issues of JMSGroups and presents the possible options for its implementation.

Implementation. Chapter 6 presents the implementation of JMSGroups together with performance evaluation and a use case example.

Replicated Invocation. Chapter 7 analyzes the problem of replicated invocation and proposes a solution.

Conclusion. And finally, Chapter 8 summarizes and concludes the thesis.

Chapter 2

Group Communication and its Toolkits

This chapter presents the basic definitions and models used in group communication, as well as the specifications of commonly used communication primitives. At the same time it points out the system models we consider for JMSGroups.

The second part of the chapter presents the survey of popular group communication toolkits with the description of their basic API. We include this survey to demonstrate the diversity of the existing GC toolkits and their interfaces.

2.1 Group Communication

Basic definitions. Distributed applications are composed from various components distributed accross the computer network. These components in group communication are called *processes*. A process is a dedicated unit of binary code that executes its tasks and communicates with other processes by sending and receiving *messages* on the network.

Processes are managed by groups. A process *group* is a set of processes identified by a group identifier. Group communication provides one-to-many and many-to-many communication primitives to the processes in a group. The processes that belong to a group are called *members* of that group. The set of members in the group at a given moment t in time is called a *group view* or just a *view*. The evolution of the group view during the system execution is managed by a service called the *group membership service*.

Group membership. Two group membership models are used in group communication: *static* and *dynamic*. In the static group membership, all processes are started at the system initialization time and the group membership remains the same during the lifetime of the system, i.e., the group view does not change. In this model the process that fails remains in the group and cannot be replaced by a new one.

In the dynamic group membership model the group view evolves during the lifetime of the system, i.e., processes can join and leave the group. In this model process can start at any time (i.e. even after the system initialization) and invoke a *join* operation to join the group. Similarly process can leave the group

by invoking a *leave* operation. In the dynamic membership model, the process that fails is removed from the group and can be replaced by a new one.

Group access. As mentioned above, processes communicate by exchanging messages. According to who can multicast a message to the group, the groups are divided into two types: *closed* and *open* groups. If the group is closed only the members can send messages to the group, i.e., processes which do not belong to the group (are not in the view) are not allowed to do so. On the contrary, open groups do not have this restriction, i.e., processes that are not members of a group can multicast messages to it.

Process failures. Processes can fail due to the software, hardware or design errors. Usually when a process fails it stops the execution and does not accept or produce any new messages. Such failure model is called *crash-stop*. The failed processes in this model stop their execution "forever". In this model a process that never fails during the system execution is called a *correct* process; a process that does fail is called a *faulty* process.

Another failure model is called *crash-recovery*. In the crash-recovery model a process also stops its execution in the case of a failure, but only temporary. Later it recovers and moreover partially preserves the state it had before the crash. State recovery implies the use of persistent storage.

There is a third failure model used in group communication, called *Byzantine* failures [38]. In the byzantine failure model a failed process does not stop its execution, but produces incorrect output: its behavior becomes inconsistent with the specification that it implements.

In the thesis we consider the first two failure models: crash-stop and crash-recovery, but not byzantine failures.

Communication channels. Usually two kinds of communication channels are considered between group members:

> *Reliable channel*: a reliable channel between two processes p and q ensures the following: if p executes $send(m)$ and q is correct, then q eventually receives m.
>
> *Quasi-reliable channel*: a quasi-reliable channel between two processes p and q ensures the following: if p and q are correct and p executes $send(m)$, then q eventually receives m.

Reliable channels are considered to derive impossibility results; quasi-reliable channels better model communication links in real networks.

Another property of communication channels is related to the message transmission time. If there exist a bound on a message transmission delay and this bound is known, then the channels are called *synchronous*. If such bound does not exist, the channels are called *asynchronous*. The physical communication networks can be modeled as *quasi-synchronous* channels, for which the bound on the message transmission delay exists, but is not known. Unless indicated, in the thesis we assume quasi-reliable and quasi-synchronous communication channels.

Partitions. Process crash is not the only reason that makes a process inaccessible. The failure of communication channels can split the group into two or more isolated parts, which cannot communicate. Such splitting is called a *partition*. Partitions are not trivial to handle, since each part of the partitioned group is not sure if the other one failed or not.

The problem with partitioned groups is that the state of the group members can evolve in each partition independently from each other. Then the clients accessing different partitions of the same group will see the different state of the group. One solution to the problem is the so called *primary partition* model. It allows the group state to evolve only in the partition which has the majority of the group members.

The alternative, *partitionable* model, allows the group state to evolve in the different partitions, but requires complicated state merge algorithms once the partitions "heal". Also in the partitionable model is hard to ensure that the inconsistent state of the different partitions will not propagate further into the system (e.g., to the clients using the services provided by the group). The primary partition model does not have these drawbacks and ensures group state consistency. In the thesis we only assume the primary partition model.

Replication types. Depending on how replicas process the client requests and keep their state consistent there are two replication types: *active replication* and *passive replication*.

Active replication also called *state machine replication* is a replication type when all replicas receive requests and process them, this way all replicas are doing the same task, i.e., are active. Active replication provides a quick reaction to failures, but requires more resources and can be used only with deterministic replicas.

If in active replication all the replicas are equal, in passive replication there are two replica types: *primary* and *backups*. Only the primary receives and processes the requests, the backups do not do the processing. The primary periodically multicast the updates to the backups to keep their state up to date. If a backup replica fails it is simply excluded from the group. When the primary fails, the new primary must be elected among the backups. Primary election takes more time than the member exclusion, this is reflected on the system response time to failures. However, as only the primary processes the requests, passive replication does not require deterministic replicas.

2.1.1 Agreement problems

The main challenge when using process groups for software replication is to keep the state consistent between the group members. For that, group members have to agree on various decisions, e.g., the set of delivered messages, message delivery order, group view, etc. In group communication, various communication primitives are specified that help to reach agreement between the group members. Here we introduce some of them (as defined and specified in [29, 12]): *Consensus*, *Reliable Broadcast*, *Reliable FIFO Broadcast*, *Reliable Causal Broadcast*, *Atomic Broadcast*.

Consensus. Consensus is the fundamental agreement problem. It allows group members to agree on a common value, that is the value proposed by one of the processes. Instances of this problem occur to

deliver messages in total order, to agree on group views, etc. Consensus is defined by two primitives: *propose(v)* by which a process proposes value *v*, and *decide(v)* by which it decides value *v*. Consensus is specified by the following properties:

Termination Every correct process eventually decides.

Validity If a process decides *v*, then *v* was proposed by some process.

Agreement Two correct processes cannot decide differently.

The Agreement property sometimes is replaced by the following *Uniform Agreement* property:

Uniform Agreement Two processes (correct or not) cannot decide differently.

The *uniformity* condition is frequently used for the group communication primitives and it strengthens the specification properties by applying them not only to the correct processes, but to the faulty ones as well. The uniform properties can be more costly to implement.

Reliable Broadcast. Group communication provides one-to-many and many-to-many communication means to the group members. Such communication has different semantics and can impose stronger properties than a simple one-to-one communication. *Reliable Broadcast* or *RBcast* provides reliable one-to-many communication. It is defined by the primitives *R-broadcast(m)* and *R-deliver(m)*, where *m* is a message. RBcast satisfies the following properties:

Validity If a correct process R-broadcasts message *m*, then it eventually R-delivers *m*.

Agreement If a correct process R-delivers message *m*, then all correct processes eventually R-deliver *m*.

Integrity For any message *m*, every correct process R-delivers *m* at most once, and only if *m* was previously R-broadcast.

The *Uniform-RBcast* is defined by applying the above specification (except validity property) not only to the correct processes but to all (correct and faulty ones).

Reliable FIFO Broadcast. *Reliable FIFO Broadcast* adds some ordering guarantees to the above specified Reliable Broadcast. It basically ensures that the messages are delivered in the order in which they were sent. The specification of the Reliable FIFO Broadcast adds the following property to the Reliable Broadcast specification:

FIFO delivery If a correct process *p* sends two messages, then these messages are delivered in the order in which they were sent.

Reliable Causal Broadcast. *Reliable Causal Broadcast* similarly to Reliable FIFO broadcast adds some order guarantees to the delivered messages, namely messages are delivered according to their causal relation. The specification of the Reliable Causal Broadcast adds the following property to the Reliable Broadcast specification:

> **Causal delivery** If two messages m and m' are sent so that m causally precedes m', then every correct process delivers m before m'.

Atomic Broadcast. *Atomic Broadcast (ABcast)* also called *Total Order Broadcast* adds to Reliable Broadcast the guarantee that messages are delivered by every process in the same order. It is defined by two primitives: *A-broadcast(m)* and *A-deliver(m)*, where m is a message. Atomic Broadcast is specified as Uniform Reliable Broadcast, with an additional property:

> **Uniform Total Order** For any two processes p and q, if p A-delivers message m' after message m, then q A-delivers m' only after A-delivering m.

2.2 Group Communication Toolkits

This section presents a small survey of group communication toolkits and aims to present their variety and diversity. A short description and the basic API are presented for each toolkit. The survey includes the toolkits frequently used in academic community, but the list is not complete.

2.2.1 Isis

Isis distributed toolkit [34], developed in Cornell University, USA, was the first to implement *virtual synchrony* [7]. Virtual synchrony is a programming model, which assumes that the events happening in the distributed system and their order are the same on each individual process. If the processes are deterministic, the state of each of them will be the same after the equivalent sequence of events. The term *virtual* is added because the events are not synchronous in physical time.

The Isis system consists of a collection of programs that are started on each machine where Isis facilities will be accessed directly; these machines are called `sites`. Isis data structures, such as group view, are kept and updated on the sites. Not all machines in the system have to be Isis sites. Applications running on the other machines can access sites remotely. Such machines do not have to run Isis core, but have to support the Isis client library.

Communication unit in Isis is a process group. Isis supports different kind of groups, but the main of them are four: *peer groups*, *client/server groups*, *diffusion groups* and *hierarchical groups*. The simplest of them are the peer groups, where all processes are equal members and cooperate between themselves to get the task done. In the client/server groups, a peer group of processes acts as a server to the clients of that group. Clients are not the full members and interact with the group in the request/reply manner. There is a possibility for the client to chose one particular member to interact with, or multicast

the requests for the whole group. The diffusion groups are the special type of client/server groups, where members of the group multicast the messages for the both sets: members and clients. Hierarchical groups have a tree structure that is constructed at the application connection time by the root group. Once connected the application always interacts with its "leaf" subgroup and the data is partitioned between different subgroups.

For the members of the group, Isis provides *view synchronous* multicast with FIFO, causal and total order protocols [67].

Programming model. Groups in Isis are identified by the group address, the address can be retrieved by the group name. Isis group membership uses a dynamic model, i.e., processes can join and leave the group during the execution. Groups are open, processes do not have to be members of the group to multicast a message to it. The membership is based on primary partition, i.e., only one set of processes stays in the group in the case of partition, others are excluded from the membership.

Interface. Isis interface is given in Table 2.1 (for full signatures and detailed method description see Appendix A.1).

Table 2.1: Isis system group API.

Interface	Description
`isis_remote_init(...)`	Connect to the Isis backbone
`site_getview()`	Get the view of the site
`bcast(...)`	Broadcast a message and collect the replies
`pg_join(...)`	Join a process group
`pg_subgroup(...)`	Create a process group with prespecified initial membership
`pg_leave(...)`	Leave a process group
`pg_client(...)`	Become a client of a process group
`pg_getview(...)`	Get information about a process group
`sv_monitor(...)`	Monitor changes to the site view
`sv_watch(...)`	Watch for a site to fail or recover
`pg_monitor(...)`	Monitor changes to the membership of a process group

2.2.2 Transis

Transis [41] is a transport level group communication service developed in the Hebrew University of Jerusalem, Israel. It employs a multicast protocol based on Trans protocol [46], that uses hardware multicast. Message flow control mechanism called network sliding window helps to achieve high throughput

for multicast messages. The main feature of Transis, in the context of group communication toolkits, is its support for partitionable service and the means for consistent merging of the partitioned components.

Transis group service manages the messages sent to the group and group views. It supports several types of message multicast: FIFO multicast provides sender-based FIFO message delivery order, CAUSAL multicast preserves the causal order among the delivered messages, AGREED or ATOMIC multicast enforces a unique delivery order among every pair of messages in all their destinations and SAFE multicast guarantees a unique order of message delivery, and in addition, delivery atomicity in case of communication failures. Group view management preserves so called virtual synchrony [7] between the members of the group. Intuitively, the virtual synchrony guarantees that local view history of the group is consistent on all group members, unless they crash, and between consecutive view changes, the same set of messages is delivered by all overlapping members. As an important extension to the virtual synchrony model, Transis allows partitionable operation: if a group partitions into two components, which do not communicate, then each component continues observing the virtual synchrony model separately. Furthermore, upon re-merging, the merged set will be virtually synchronized starting with the membership change that denotes the merge.

The systems in Transis are composed from a collection of multicast clusters. Each multicast cluster consists of a set of machines communicating using hardware multicast. Such clusters can be located on a local area network (LAN), or multiple LANs interconnected by transparent gateways or bridges.

Programming model. Process group is the basic unit of communication in Transis. To send a message to a single process, one must send a message to a group consisting only of that process. Groups in Transis are identified by the group name. There are no specific create group call, groups are automatically created when the first members joins. Transis uses dynamic group model, i.e., the members can join and leave the group during the system execution. In addition Transis groups are open, that means the process do not have to be a member of the group to multicast a message to the group members. Group membership model is consistent with virtual synchrony and is extended to partitionable operation, it supports consistent merging upon the partition healing.

Interface. Transis group interface is given in Table 2.2 (for full signatures and detailed method description see Appendix A.2).

2.2.3 Phoenix

Phoenix [43] is a toolkit for programming fault-tolerant large scale applications and was developed at École Politechnique Fédérale de Lausanne (EPFL). It provides view synchronous communication in large scale distributed systems, addressing such problems as network partitioning due to the link failures. Phoenix has a layered structure, where each layer provides a service used by the upper layers. The upper layers communicate with the lower ones by procedure calls, and the lower ones use callbacks to notify the upper layers.

Table 2.2: Transis system group API.

Interface	Description
`zzz_Connect(...)`	Connect to Transis
`zzz_ Join(...)`	Join the group
`zzz_Leave(...)`	Leave the group
`zzz_Send(...)`	Send a message to the group
`zzz_Receive(...)`	Receive messages from the group
`zzz_Add_Upcall(...)`	Add an upcall event

Every application using Phoenix requires the set of protocol layers, but the layers are not dependent on the application, i.e., the same layers can be used by several applications. Therefore there is a possibility to separate core layers from the application and use them as a service. This is implemented using Phoenix daemons. The daemons are composed of the core Phoenix protocol layers and allow the applications to connect and use these layers as a service. In order to do that, the application only needs a thin Phoenix client layer. Consequently, Phoenix daemons and the applications using them do not necessary need to be located on the same machine.

The layers composing Phoenix are the following (from bottom up): *socket interface layer* uses UDP for point to point communication, *routing layer* improves the reliability of the socket interface layer by rerouting the messages through the other hosts if the UDP point-to-point link breaks, *reliable communication layer* implements reliable communication channels, *view synchronous communication layer* implements view synchronous message broadcast, *ordered multicast communication layer* orders the application messages, it defines different primitives for ordering (FIFO, weak total order, strong total order, uniform multicast and global order multicast), the most upper layer *application programing interface* provides the application with the primitives of all underlying layers.

Programming model. Phoenix groups use dynamic group model in which processes can join and leave the group during the execution. The groups in Phoenix are open, the processes not members of the group can multicast messages to the group. Process can be a member only of one group. Communication channels are asynchronous and unreliable. Phoenix groups implement the primary partition model.

Interface. Phoenix has two API options for the application to use: the first is the mentioned application programing interface layer, which provides the procedure calls of the underlying layers, the second is the object oriented programing interface. The latter provides classes for three Phoenix process types: *members, clients* and *sinks*. These types form a hierarchical structure with the sink class type on top, the client class inheriting from the sink class, and the member class inheriting from the client class. The sink class inherits from the basic class called *team*. The difference between sink, client and member classes is the set of Phoenix methods the objects of these classes can use. Only the processes of type `member`

form the groups.

The object oriented programming interface is more complete and convenient to use for the application, therefore we present only this interface (see Table 2.3, for full signatures and detailed method description see Appendix A.3).

Table 2.3: Phoenix object oriented API.

Interface	Description
Sink class	
`SinkSubscribe(...)`	Subscribes an object as a sink to the group
`SinkUnsubscribe(...)`	Unsubscribes an object as a sink
Client class	
`ClientSubscribe(...)`	Subscribes an object as a client to the group
`ClientUnsubscribe(...)`	Unsubscribes an object as a client
`ViewChangeDeliveryCB(...)`	View change notification callback
Member class	
`Join(...)`	Adds the object to the group
`Leave()`	Makes the object leave the group
`Multicast(...)`	Multicast a message to the group
`IntermediateViewDeliveryCB(...)`	Intermediate view delivery callback

2.2.4 JGroups

JGroups (formerly JavaGroups) [2] is a toolkit for reliable group communication developed at Cornell University, USA. JGroups is a rewrite of Ensemble [31] in Java™ language, Ensemble itself is a rewrite of Horus [74] in ML language. JGroups uses IP multicast as a communication basis, on top of it JGroups provide reliability for message delivery, duplicate detection, message ordering, group membership and fault detection services. JGroups, like Ensemble, is based on the partitionable membership model, i.e, the group can split into partitions during the execution and concurrent views are allowed in these partitions.

The architecture of JGroups is shown in Figure 2.1 and consists of three parts: *protocol stack*, *channel* and *building blocks*. The protocol stack implements the specified communication properties, the channel API is used by the application to connect to the protocol stack, and the building blocks provide the higher abstraction layer for the applications using channel.

The communication protocol stack in JGroups has a layered architecture. The desired communication properties can be chosen composing various protocol layers at the stack creation time. This provides the flexibility for the application designer, as the desired properties can be chosen depending on their need.

The channel represents a group and is the access proxy to the group communication facilities on the application side. Whenever the application sends a message, the channel passes it on to the protocol

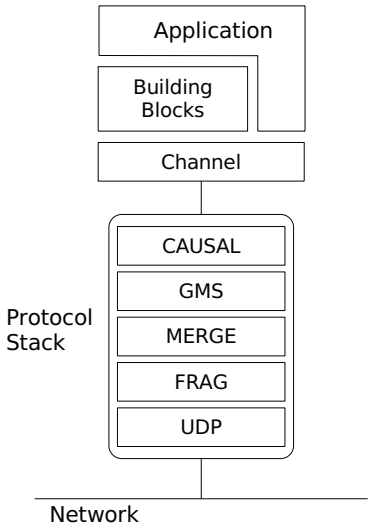


Figure 2.1: The architecture of JGroups.

stack, which passes it to the topmost protocol layer. After the layer processes the message, it is passed to the layer below and etc., until it reaches the network. The same, just in the reverse order, happens when the protocol stack receives the message. It is passed up through all the layers, until delivered to the application. Creating a channel also creates the protocol stack associated with it; connecting to the channel automatically joins the connecting application to the group. To leave the group it is enough to disconnect from the channel.

Programming model. Each group has a name associated with it and the channel connects (joins) to the group providing its name. The group is created by the first member joining it. JGroups are dynamic, as the members can join and leave at runtime. Groups are closed: the client must be a member to be able to send messages to the group. As already mentioned, JGroups uses partitionable group membership.

Interface. Group representative in JGroups is the `Channel` class. It provides the API for the application to use the JGroups. The summary of the API is given in Table 2.4, full documentation is in [3]. JGroups provide some event listener interfaces, which application have to implement in order to automatically receive the notifications. Table 2.4 also includes some methods from `MessageListener` and `MembershipListener` interfaces (for full signatures and detailed method description see Appendix A.4).

2.2.5 Object Group Service

Object Group Service (OGS) [18] developed at École Politechnique Fédérale de Lausanne (EPFL) provides a group communication service for the Common Object Request Broker Architecture (CORBA).

Table 2.4: JGroups API.

Interface	Description
Channel class	
`connect(...)`	Connects the application to the channel (joins the group)
`getView()`	Gets the current view of the group
`send(...)`	Sends a message
`receive(...)`	Receives messages, views and suspicions
`disconnect()`	Disconnects from the channel (leaves the group)
MessageListener interface	
`receive(...)`	Receives messages
MembershipListener interface	
`viewAccepted(...)`	Receives membership views
`suspect(...)`	Receives member suspicions

OGS has a modular, component-oriented architecture, where each module can be modified without the knowledge of the application using it. OGS adds the following notions to CORBA objects: *object group*, *group behavior* and *group reference*. These notions help to make the groups transparent for the clients addressing them, and make dealing with a group appear as dealing with a single CORBA object. OGS does not change the CORBA specification [56] to support object groups, but provides group communication as a service. This approach in theory makes OGS portable to any CORBA compliant ORB.

OGS uses the so called service approach to provide fault tolerance to CORBA objects. In the service approach the fault tolerance mechanisms are implemented above the ORB, i.e., themselves act as a CORBA objects providing dedicated service. OGS consists of several such services. The core *group service* implements two functionalities: *group membership* manages the life cycle of the group, and *group multicast* provides the communication primitives, on a per message basis. The *consensus service* allows a set of CORBA objects to solve the distributed consensus problem, and is used to implement group communication protocols. The *monitoring service* defines the interfaces for detecting remote component failures, and supports several modes of monitoring. Finally, the *message service* provides the remote communication service.

Programming model. The group in OGS is addressed by the object groups reference and managed by the group membership service. OGS supports dynamic groups, i.e., the membership of the groups can change over time. New members can explicitly join or leave the group, or can be implicitly removed from the group because of failure. Naturally CORBA object groups in OGS are open, i.e., non-member objects can issue multicasts to the group. The OGS group membership service implements the primary partition model.

Interface. Each OGS service has an interface defined in the Interface Definition Language (IDL). IDL is the part of the CORBA specification and has mappings to the main programing languages. Even the clients implemented in the different language than OGS can access its services using the IDL mapping. Table 2.5 summarizes the interface of the main OGS service, i.e., the group service (for full signatures and detailed method description see Appendix A.5).

Table 2.5: OGS API.

Interface	Description
Server side	
`join_group(...)`	Adds the object to a group
`leave_group(...)`	Removes the objects from a group
`view_change(...)`	Notifies the member about the view change
`deliver(...)`	Delivers the message
Client side	
`multicast(...)`	Issues a multicast to the group
`get_view(...)`	Gets the latest group view

2.2.6 Eternal System

The Eternal System [52] developed at the University of California, Santa Barbara, uses the interception approach to provide fault tolerance to CORBA. The interception mechanism is inserted underneath the ORB and involves "capturing" and modifying specific system calls used by the application and the ORB. The calls are transparently redirected to the Eternal replication mechanism. The advantage of this approach is that the application is not aware of its calls being "intercepted" and modified; for this reason the approach also is called "transparent". Moreover, neither the application, nor the ORB have to be modified or even recompiled to provide fault tolerance.

The Eternal system mechanisms underneath the ORB include the Interceptor, the Replication Mechanisms and the Logging-Recovery Mechanisms. The Interceptor transparently captures IIOP messages generated by the application and diverts them to the Replication Mechanisms, which maintains strong replica consistency. To preserve strong consistency among the object replicas, the intercepted requests are conveyed using the reliable totally ordered multicast provided by the underlying Totem system [51]. The Logging-Recovery Mechanisms detect faults and manages replica recovery.

The Eternal system underlying Totem communication toolkit provides reliable totally ordered multicast of messages over the local-area network (LAN) or over multiple LANs interconnected by gateways. It exploits the hardware broadcasts of such networks to achieve high performance. Totem provides message delivery service in the presence of various types of communication and processor faults, including message loss, network partitioning, and processor crash, omission and timing faults. In addition to the faults, the Eternal system supports byzantine faults. This is provided by the Immune system [55].

Programming Model. In Eternal system groups are addressed by the Interoperable Group References, which usually are provided by the naming service. Group membership is dynamic, i.e., the group members can join and leave the group after it is created. Groups are open, the client does not have to be member to send messages to the group. The Eternal system supports group partitioning.

Interface. The interception approach provides transparent fault tolerance for the application. Consequently, the replication and group interfaces in Eternal system are not exposed to the application, except for some limited application control methods.

For application-level control, Eternal provides the services implemented above the ORB. These services, include the Replication Manager that replicates each application object (according to user specified fault tolerance properties) and distributes the replicas across the system. The Replication Manager interface provided to the application, allows the creation/deletion of object groups, and also of the individual object replicas on specific locations. The references returned by the Replication Manager interface are references to the object groups and not to the individual objects. The Replication Manager methods related to the object group control are given in Table 2.6 (for full signatures and detailed method description see Appendix A.6).

Table 2.6: Eternal ReplicationManager API.

Interface	Description
`create_object(...)`	Creates an object group
`delete_object(...)`	Deletes an object group
`create_member(...)`	Creates a member of an object group
`add_member(...)`	Adds an existing member to an object group
`remove_member(...)`	Removes a member from an object group

2.2.7 Interoperable Replication Logic

The Interoperable Replication Logic (IRL) [62] developed at University of Rome ”La Sapienza” uses a three-tier approach to provide fault tolerance to CORBA objects. In a three-tier architecture the clients do not access an object group (the third-tier) directly; the middle-tier is used to transparently redirect the requests to the object group. The architecture used by IRL is the fusion of the interception and the service approach to provide fault-tolerance to CORBA. Portable CORBA request interceptors are used to transparently intercept the client requests and to redirect them to the IRL management objects. Each stateful object group member is transparently wrapped by the IRL Incoming Request Gateway Component (IRGW), which adopts the same interface as the object. The IRGW intercepts all requests directed to the group member and provides duplicate filtering, request/reply logging and garbage collects the expired request/reply pairs. The group members must implement Fault Tolerant CORBA (FT-CORBA) [56]

`Checkpointable` interface to enable the logging of its state, but there is no special interface related to the group. Groups are managed by the middle-tier components.

IRL middle-tier consists of three main entities: `ReplicationManager`, `ObjectGroupHandler` and `FaultNotifier`. To eliminate the single point of failure, the middle-tier entities are replicated. The IRL ReplicationManager represents the management interface of the IRL infrastructure. It exports the `ReplicationManager` FT-CORBA interface and allows object group creation/disposal, and membership modification. The IRL ObjectGroupHandler (OGH) component is in charge of maintaining consistency among the state of the members of a stateful object group. In particular, an OGH component is associated to each stateful object group. It receives client requests and transforms them in a set of requests addressed to the object group members. OGH knows the composition of its group and can therefore create the real request to all server objects belonging to the group. The IRL FaultNotifier (FN) implements a publish and subscribe engine to provide subscribers with fault notifications. It essentially detects host failures and propagates object and host fault reports to every object that subscribed for a report of the failure event.

In order to let client applications benefit from transparent client invocation even on non FT-CORBA compliant client ORBs, client applications are augmented with the IRL Object Request Gateway (ORGW) component. In short, ORGW is a CORBA Client Request Portable Interceptor that (i) intercepts requests addressed to object groups, (ii) uniquely identifies them as the FT-CORBA standard prescribes, and (iii) iteratively tries to send the request to a correct member, until either it receives a reply (that it returns to the client application) or it has tried all members in the group without receiving a reply.

Programming Model. In IRL a group of objects is addressed by Interoperable Object Group Reference (IOGR), which contains the references to all group members. IRL uses the open group model: the clients do not have to be members of the group to issue a request to the group. Group membership in IRL is dynamic and is managed by the Object Group Handler entity. IRL uses the primary partition model.

Interface. In IRL client or group members do not have direct access to the group communication interface. Group management is done by the IRL entity ReplicationManager; part of its interface related to the group and membership management is given in Table 2.7 (for full signatures and detailed method description see Appendix A.7). As the reader can notice there is no multicast primitive in the table. The multicast in IRL is done by OGH entity, which receives the membership of the group from the replication manager and invokes the requests on each member using their interface obtained from CORBA Interface Repository.

Table 2.7: IRL ReplicationManager API.

Interface	Description
`create_object(...)`	Creates an object group
`delete_object(...)`	Deletes an object group
`add_member(...)`	Adds a member to an object group
`remove_member(...)`	Removes a member from an object group

2.3 Summary

This chapters presented group communication and some of the group communication toolkits with their interfaces. The properties of the toolkits are summarized in Table 2.8. The API diversity and differences of the presented toolkits (see Table 2.9) make it difficult for GC to become really popular and widely used. The ad-hoc interface is common obstacle when integrating GC into the existing enterprise systems. Consequently, in this thesis we propose to standardize group communication interface. To achieve that we do not propose a new standard, but we suggest the use of an already existing and accepted standard. We have chosen JMS API, which is a de facto standard for the Java based message oriented middleware. JMS and its API are presented in the next chapter.

Table 2.8: Group communication toolkits.

Toolkit	Group Membership	Group Type	Primary Partition
Isis	Dynamic	Open	Yes
Transis	Dynamic	Open	No
Phoenix	Dynamic	Open	Yes
JGroups	Dynamic	Closed	Yes
OGS	Dynamic	Open	Yes
Eternal	Dynamic	Open	No
IRL	Dynamic	Open	Yes

Table 2.9: API comparison for two methods: *Join Group* and *Multicast*.

Toolkit	Join Group	Multicast
Isis	`pg_join(...)`	`bcast(...)`
Transis	`zzz_Join(...)`	`zzz_Send(...)`
Phoenix	`Join(...)`	`Multicast(...)`
JGroups	`connect(...)`	`send(...)`
OGS	`join_group(...)`	`multicast(...)`
Eternal	`add_member(...)`	CORBA object invocation
IRL	`add_member(...)`	CORBA object invocation

Chapter 3

Java Message Service

The distributed nature of todays computing environments requires the mean for communication between the users and between the applications. The usual communication mechanism used by the users around the world is e-mail. It is simple, and for people usually is enough to exchange the short text messages, occasionally attaching files to them. But take an enterprise system which components might be distributed all over the world and communicate using complicated semantics. E-mail is too limited for such systems and instead application-to-application messaging systems are used. They provide much richer message and communication semantics, and when used in business systems are generally referred as enterprise messaging systems, or Message Oriented Middleware (MOM).

There are a lot of vendors who develop their own MOM systems ([33], [5], [73]), but the semantics of sending and receiving messages are similar for all of them. The message must be created, then the payload (application data) must be attached to it, and the message must be sent using the specific routing information. Then upon the reception of the message on the destination, the payload must be extracted. The MOMs are much more complex than that, but the basic principle is the same for almost all the messaging systems on the market.

All MOM vendors provide the developers with the API for sending and receiving messages. While the internals differ from implementation to implementation the basic API provided to developers are closely similar. This similarity in APIs makes the Java Message Service possible.

Java Message Service (JMS) is vendor-independent Java API that can be used to access MOMs from many different vendors, which support this API. It was developed by Sun Microsystems [49] with collaboration from numerous MOM vendors and is a part of the Sun's Java 2 Enterprise Edition (J2EE) [71]. JMS is not a messaging system itself, rather it is a specification of interfaces and classes for a client accessing a messaging system [30]. Moreover JMS provides more than just an API, it additionally includes a rich set of message delivery semantics. Today most of the messaging systems vendors on the market comply with the JMS specification.

This chapter provides the introduction to JMS; for more in depth information please refer to [30]. The structure of the chapter is the following: Section 3.1 presents the architecture of JMS, Sections 3.2 and 3.3 talk about JMS messaging models, and finally Section 3.5 presents the main JMS interfaces.

But before let us present some notation used in the JMS specification, and introduce some basic entities, properties and messaging models.

Basic notation. In the JMS specification, messaging clients are called *JMS clients*, and the messaging system is called *JMS provider* or *JMS server*.[1] In addition, a JMS client that produces and sends a message is called a *producer*, while the client that receives a message is called a *consumer*. A single JMS client can be both a producer and a consumer.

JMS messages. The main communication entity in JMS, as in all messaging systems, is a message. A JMS message is an information unit consisting of the message headers, properties and the payload. JMS message headers contain various metadata related to the individual message, such as message ID, time to live, delivery mode, priority, etc. JMS message properties are like extended headers. They provide the possibility to the developer to assign optional name-value properties to each message. The properties can be of various types, from a simple boolean to an object type. Finally, the JMS message payload (body) contains the main portion of the application data and also supports various data types.

JMS message delivery mode. Each message sent by a JMS producer has a *message delivery mode* specified, which can be *persistent* or *non-persistent*. Persistent messages are stored by the JMS server on a stable storage, and provide delivery guarantees if the server crashes. When a JMS server receives a persistent message, it acknowledges the reception to the producer only after having stored the message on persistent storage. If the server later crashes before delivering the message to the receiver, persistent messages are not lost, and will be delivered upon the server's recovery. In contrast, non-persistent messages are not saved on persistent storage, and can thus be lost if the JMS server crashes.

The delivery requirement for non-persistent messages is *at-most-once*, which allows message loss. Messages are lost if the server crashes before delivering them to the destination. Their recovery is not possible together with the server, because non-persistent messages are not saved to the stable storage. In contrary, for persistent messages the delivery requirement is *once-and-only-once*, which does not allow message loss. Duplicate delivery is not allowed for the both types of messages.

JMS messaging models. JMS defines two messaging models: *publish-subscribe* and *point-to-point*. The specification requires the vendor to implement at least one of them, but usually both are supported. In simple terms, publish-subscribe defines one-to-many broadcast of messages, and point-to-point is intended to one-to-one delivery of messages. The two models in the JMS specification are referred as *messaging domains*. We will present each model in more details later in the chapter.

[1] We will use the notation *JMS server* throughout this thesis.

3.1 The architecture

The basic architecture of JMS is shown in Figure 3.1. JMS assumes a central JMS server, which generally acts as the hub for all communications, and has access to stable storage. The server is transparent to the application, composed of the JMS clients (message producers and consumers) and a set of application-defined messages. The JMS specification does not define how the server is implemented. It only defines the interfaces and the services that the JMS infrastructure must provide.

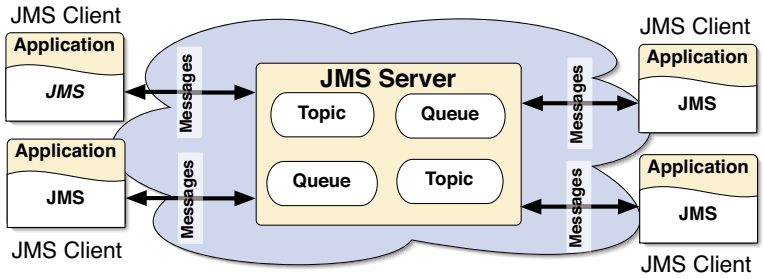


Figure 3.1: Basic JMS architecture.

Although JMS specification focuses on the central server design, it does not impose it. The central server architecture is the easiest to implement and usually is chosen by the developers. But it has a drawback: the central server is a single point of failure. If it fails the whole messaging systems becomes nonoperational. To prevent this from happening the JMS server can be replicated. Other architectures also can be chosen for JMS, e.g., non-centralized architecture, where each client has a layer representing the server, but there is no central server in the system. Hybrid architectures are possible as well.

The basic communication schema between the JMS client and server is shown in Figure 3.2. The JMS client usually consists of two layers: the *application layer* and the *JMS client-side layer*. The application layer is implemented by the user. It uses the JMS client-side layer to communicate with the JMS server and receive the messaging service. The client-side layer is provided by the JMS implementation and manages the client's interaction with the JMS server.

The client side communication entities are strictly defined in the JMS specification. This is however not the case for the communication entities on the server side. The JMS specification does not define how the server should be implemented, but rather defines the interfaces and services that the JMS infrastructure must provide. The JMS server providers thus have a large freedom in implementing the server. To generalize, we however assume that the server side communication entity can be represented as a single *client context* entity or simply a *context* (see Figure 3.2). For every client connected to the server, an individual context is created. It contains all the necessary information about the client's communication with the server, such as the queues of messages received from and to be sent to the client, as well as other connection related information. The major part of the JMS server state consists of the clients' contexts, and the major part of the processing the server does is spent managing these contexts.

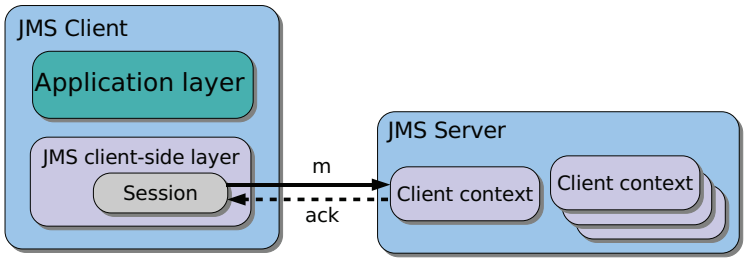


Figure 3.2: JMS client-server communication.

Figure 3.2 shows the very basic client-server communication. In JMS only the clients are message producers and consumers, i.e., a JMS server does not produce or consume messages.[2] Furthermore, sending messages in JMS is blocking: whenever the client application sends a message, the application is blocked until the message is received by the server and an acknowledgement is sent back (dashed line in Figure 3.2).

3.2 JMS publish-subscribe

As noted previously JMS publish-subscribe messaging model is used for one-to-many message communication (see Figure 3.3). The messages are broadcast through the virtual channel called *topic*, which is identified by a name. JMS clients have to connect to the specific topic to send or receive messages. There are two different connection modes: *publisher* and *subscriber*. If a JMS client connects as a publisher it can publish (send) messages to the topic, e.g., the JMS client on the left in Figure 3.3. If a JMS client connects as a subscriber, it will receive the messages published to the topic (by the JMS clients on the right in Figure 3.3). A JMS client can be a message publisher and subscriber at the same time (this is not illustrated in Figure 3.3). If there are no subscribers, the topic is not obliged to keep the messages, except for durable subscriptions defined in the next paragraph.

Durable subscriptions. There are two types of topic subscribers specified in JMS: *non-durable* and *durable*. Non-durable subscribers receive the messages published to the topic since their subscription time, and as long as the physical connection between the subscriber and the topic is active. The connection can break (i.e., become inactive) for example because of a link failure, or because of a crash of the client, or of the server. Messages published after the connection is broken are not guaranteed to be received by the client.[3] On the contrary, durable subscriptions enable the client to get the messages even if its physical connection to the topic is not always active. If the physical connection is not active,

[2] Here we mean the application level messages.

[3] If the connection is broken, the client can try to subscribe again to the topic. Let us assume that the connection was broken at time t_1, and that a new subscription is received by the JMS server at time t_2. With non-durable subscriptions, the messages published in the interval $[t_1, t_2]$ may not be received by the client.

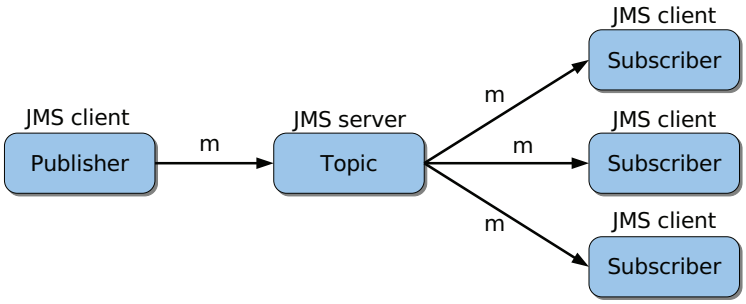


Figure 3.3: JMS publish-subscribe messaging model.

the JMS server keeps the messages for the durable subscriber, and delivers them as soon as the client reconnects again. Each durable subscriber is assigned a unique ID, which helps the JMS sever to keep track of its connection status, and respectively, deliver or store the messages.

The durability property does not formally specify the message type (persistent or non-persistent) to be used. But if non-persistent messages are used, the durability properties cannot be fully satisfied. Take a simple example: non-persistent messages kept on the server for a durable subscriber will be lost in the case of the server crash. When the server recovers and the subscriber connects again, the messages for it cannot be recovered. Indeed, for durable subscriptions persistent messages must be used to avoid message loss in the case of a JMS server failure.

Assume that all subscribers to a particular topic have the same type of subscription and only persistent messages are used, also assume that the messages are published to the topic during the time interval $[t1,t2]$ and the subscribers can crash or loose the connection to the server during that interval; all crashed subscribers later recover. Then the durability property can be specified as following:

- for non-durable subscriptions the messages are guaranteed to be delivered only to those subscribers which are not crashed and have physical connection to the server during the time $[t1,t2]$;
- for durable subscriptions messages are guaranteed to be delivered to all topic subscribers.

3.3 JMS point-to-point

On the contrary to the publish-subscribe model, JMS point-to-point messaging is used for one-to-one communication. The messages in the point-to-point model are sent through the virtual channel called *queue*, which is identified by a name. JMS clients have to connect to the queue in order to send and receive messages (see Figure 3.4). A JMS client sending messages connect as a queue *sender*, and the one receiving messages connect as a queue *receiver*. A queue can have several senders and several receivers, but every message sent to the queue is delivered only to a single receiver. Its for the JMS server to decide to which one. A JMS queue holds the messages sent to it until they are consumed by a

receiver. In contrary to JMS topic, if the queue has no receivers and gets the message, it will not discard the message, but will keep it until a receiver connects.

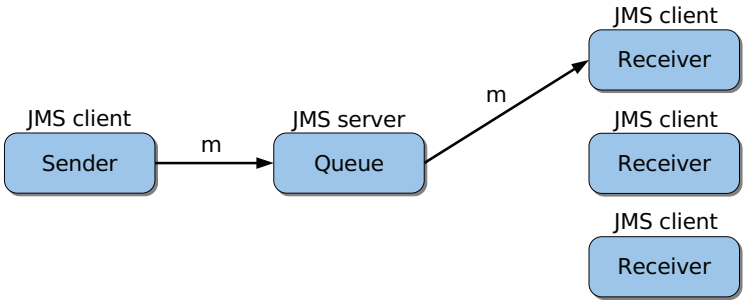


Figure 3.4: JMS point-to-point messaging model.

3.4 JMS message delivery requirements

The JMS specification [30] also defines the order of message delivery on the subscribers. Essentially, JMS guarantees FIFO ordering of the messages that are sent between two client sessions (see the next section for the definition of a session). Messages that are sent by a session must be received in the order in which they were sent.[4] However, JMS does not define message delivery order across the subscribers, when the messages are sent by the different sessions (the message delivery order can be m_1, m_2 for one subscriber and m_2, m_1 for another, if the sender of m_1 and the sender of m_2 aren't the same sessions).

The above presented message order requirements are for the publish-subscribe messaging. Point-to-point messaging is required to provide FIFO message delivery order.

Finally, the JMS specification does not allow duplicate delivery of the acknowledged messages, with one exception: if a failure occurs between sending a message to a consumer and receiving the acknowledgment from it, the message can be redelivered (as it is not clear if the consumer delivered the message or not). Only the last message delivered by a consumer is subject to this ambiguity.

3.5 JMS interfaces

Each JMS messaging model defines a set of interfaces (API) that provide access to the messaging middleware. Besides that, JMS defines common interfaces that give the model independent (point-to-point or publish-subscribe) access to the messaging system. These interfaces are the superinterfaces of each messaging model and are used to increase the portability. The common JMS interfaces together with the corresponding messaging model interfaces are given in Table 3.1. The relationship among the objects

[4] JMS also provides options such as message types and priorities that can alter the delivery order, but we only consider messages of the same type and priority here.

implementing the common JMS messaging interface is given in Figure 3.5. The following is the brief definition of the common JMS interfaces:

Table 3.1: JMS interfaces.

Common Interfaces	Publish-subscribe Interfaces	Point-to-point Interfaces
`ConnectionFactory`	`TopicConnectionFactory`	`QueueConnectionFactory`
`Connection`	`TopicConnection`	`QueueConnection`
`Destination`	`Topic`	`Queue`
`Session`	`TopicSession`	`QueueSession`
`MessageProducer`	`TopicPublisher`	`QueueSender`
`MessageConsumer`	`TopicSubscriber`	`QueueReceiver`

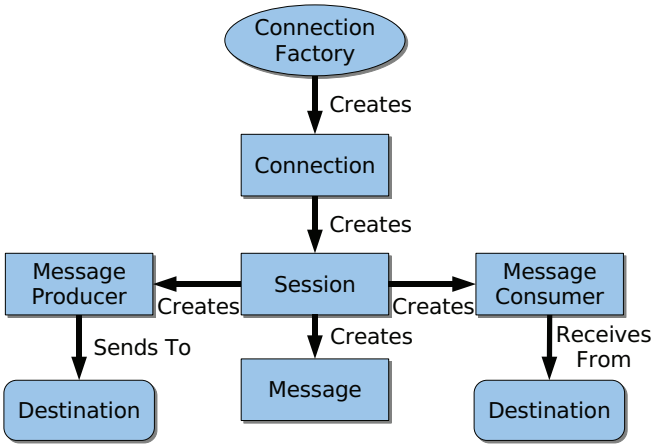


Figure 3.5: JMS object relationship.

- `ConnectionFactory` - encapsulates a set of connection configuration parameters. A client uses it to establish (create) a `Connection` to a JMS server. `ConnectionFactory` together with `Destination` are called administered interfaces. They are used to provide JMS configuration information to the clients, and are initiated by an administrator. The instances of the administered interfaces are not the part of the client and usually are accessed remotely. The JMS specification does not strictly define the means to access them, but establishes a convention that the clients find them using `JNDI`. The `ConnectionFactory` is a superinterface of the `TopicConnectionFactory` and `QueueConnectionFactory` interfaces.
- `Connection` – encapsulates a client's active connection to a JMS server. Usually creating a connection involves resource allocation outside the client's virtual machine (opening a TCP/IP socket);

also creating a connection usually encapsulates the client's authentication protocol. `Connection` is a superinterface of the `TopicConnection` and `QueueConection` interfaces.

- `Destination` – is also an administered interface, it encapsulates the identity of the message destination together with the provider specific address information. It is a superinterface of the `Topic` and `Queue` interfaces.
- `Session` – a context for sending and receiving messages. A physical connection between the JMS client and the server is represented by the `Connection` interface; similarly the `Session` represents a logical connection. One `Connection` can be a factory for many `Sessions`; similarly the `Session` is used as a factory for JMS messages. It is a superinterface of the `TopicSession` and `QueueSession` interfaces.
- `MessageProducer` – is used for sending messages to a `Destination`. The instance of `MessageProducer` is created by a `Session` and associated with a single `Destination`. It is a superinterface of the `TopicPublisher` and `QueueSender` interfaces.
- `MessageConsumer` – is used for receiving messages from a `Destination`. The instance of `MessageConsusmer` is also created by a `Session` and associated with a single `Destination`. It is a superinterface of the `TopicSubscriber` and `QueueReceiver` interfaces.

3.6 Summary

This section introduced the basics of the Java Message Service. Its central server based architecture, publish-subscribe and point-to-point messaging models, persistent and non-persistent message delivery properties, as well as message delivery order defined by JMS. As well, the main user interfaces for the JMS messaging objects were presented.

Chapter 4

JMS Compliant Group Communication: Semantics and API Mapping

In this chapter, we propose a standard specification and interface for group communication (GC). Instead of specifying yet another GC API with similarly low chances of becoming a standard as existing GC APIs, we take advantage of the widespread acceptance of JMS and propose to extend the JMS specification with GC. Currently, JMS supports two paradigms: *point-to-point* and the *publish-subscribe*. We add group communication as the third paradigm to JMS. The resulting specification and interface is called JMSGroups and should be easily understandable by both the GC community and developers familiar with JMS. As such, it facilitates the acceptance of group communication by a larger community and provides a powerful environment for building fault-tolerant applications. JMSGroups is defined in the spirit of JMS; this forces us to clearly identify the semantic differences between the two technologies and to bridge this semantic gap. As a consequence, we need to address some of the shortcomings of traditional GC over MOMs that have been highlighted in [4], such as process recovery after a crash.

Chapter Roadmap. Section 4.1 discusses the semantic mapping of GC to JMS, in Section 4.2, we give a formal specification of the mapping, Section 4.3 presents a JMS compliant API for GC and finally Section 4.4 summarizes the chapter.

4.1 GC and JMS: a semantic mapping

4.1.1 The problem: the semantic gap

JMS and GC have been developed by different research communities, have different semantics and use different APIs. While the problem of mapping one API to the other is not too difficult and will be addressed in Section 4.3, the semantical mapping raises more difficult and interesting issues. For example, JMS uses notions that are not present in GC, such as durability and persistence. Moreover, JMS assumes a model with process recovery, which is not the case for existing GC systems (where crashed processes recover with a new identity). Consider, for instance, the case of durable subscriptions. Clearly, durable

subscriptions only make sense if processes can recover after a crash; a process that does not recover needs no durable subscriptions. As a consequence, in order to support connection durability in GC, we need to change the failure model that underlies GC specifications.

4.1.2 Bridging the gap

We start the discussion of the semantic mapping between JMS and GC by the key (and simple) idea, which is to represent *process groups* as *JMS topics*.

Mapping groups to JMS topics

We map a group g to a JMS topic t as follows: (i) members of a group g map to the subscribers of the corresponding topic t, and (ii) broadcasting a message to the members of g corresponds to publishing a message to the topic t.

The idea of representing a group as a topic is quite natural, since JMS uses the notion of a topic to indirectly address a set of JMS clients. Note that representing the group as a JMS queue is less natural. It raises the following semantic issue: while multiple clients can read from the same queue, only *one* client gets a particular message (i.e., if client c reads message m, then client c' cannot read m). Queues are therefore not suited to express the multicast semantics of GC.

In the context of GC, it is sometimes required that the process that broadcasts a message to group g is a member of g. This is called the *closed* group model. In the *open* group model, no such restriction exists. In JMS a publisher does not have to be a subscriber to publish to the topic. This corresponds to the open group model. Since the open group model is more general, it seems natural to adopt this model for GC based on JMS.

GC message delivery guarantees *vs.* JMS message persistence

In JMS, QoS is defined by *persistence/non-persistence* with respect to messages, and by *durability/non-durability* with respect to subscriptions (see Chapter 3). Although GC does not consider these properties, we show how they can be incorporated in a GC specification. We first address message persistence. Durability is the topic of the next section.

Consider a JMS publisher that publishes message m to topic t. If m is persistent, and the publisher has received an acknowledgment from the JMS server, then the publisher has the guarantee that message m will not be lost, even in case of the JMS server crash. In contrast, if message m is non-persistent, then m may be lost if the JMS server crashes. Note that the loss of m may occur although the publisher does not crash.

If we transpose the non-persistent case to the context of GC, we have the case of a process that broadcasts some message m to group g with no guarantee that m is ever delivered by any process, even if p does not crash. The message loss does not happen if the message is persistent. In other words, non-persistent messages provide what is usually called *best-effort* guarantees, while persistent messages can be seen as providing the *strong* guarantees of a reliable (logical) channel between the sender and the

group. As GC traditionally provides more than best-effort guarantees, we assume persistent messages further in the chapter.

GC crash model *vs.* JMS subscription durability

The mapping of durable and non-durable subscriptions is more difficult to address than the question of persistence/non-persistence. The issue cannot be discussed without referring to what happens to the processes that are members of a group and crash.

In one commonly adopted GC model, processes that crash are eventually removed from the group. Upon recovery, these processes adopt a new identity before joining again the group. This model is sometimes called the *crash-no recovery* model: processes that crash seem not to recover, since they recover under a new identity. This model is for example the one of the Isis system [8]. Note that, if p crashes and later rejoins g with a new identity p', the GC system has no obligation to deliver to p/p' messages broadcast while the process is down.

If we transpose this in terms of type of subscriptions, we see that the crash-no recovery model can very nicely be mapped to non-durable subscriptions, in which the JMS server stops to have any obligation toward a subscriber with respect to message delivery if the connection is broken.

If the crash-no recovery model can be mapped to non-durable subscriptions, what is the GC model that corresponds to durable subscriptions? With durable subscriptions, even if the connection to a subscriber is broken, the JMS server has the obligation to deliver messages to that subscriber. This can be interpreted in the following way in terms of GC. Let p be a process member of group g, and let p crash at time t_1, and later recover at time t_2. Despite of being down during the interval $[t_1, t_2]$, process p delivers all the messages broadcast to the group g. In other words, although p crashes, it is not removed from the group. This means that the GC system has the obligation to deliver to p *all* messages broadcast to g, after p has become a member of g. This model is sometimes called the *crash-recovery* model [1, 63].

GC service *vs.* JMS server

The JMS architecture distinguishes between the JMS server and JMS clients (see Fig. 3.1). In the context of GC, this distinction is rather unusual. GC specifications usually refers only to what JMS calls *clients*. However, servers cannot be ignored in the context of JMS. Even if this is unusual, it has a positive consequence: it decouples explicitly the *server(s)* that provide the GC service, from the *clients* that use this service. Note that this decoupling does not prevent a process, in some implementation, to be at the same time a client and a server. This special case is the standard case in the context of GC. However, an implementation is not forced to adopt this solution (e.g., [43, 15, 62]). Moreover, an implementation of GC could be based on one single (JMS) server. Of course, such an implementation is not fault-tolerant. Another implementation could be based on multiple (JMS) servers, and so be fault-tolerant. Yet, in another implementation, the same process could be both a (JMS) server and a (JMS) client.

It is important to have the decoupling between clients and servers clear in mind, in order to avoid misunderstanding of some issues discussed below. For example, the distinction between crash-recovery and crash-no recovery can apply both to (JMS) clients, and to (JMS) servers. However, if one model

is chosen for (JMS) clients, this does not impose the same model on (JMS) servers. Moreover, in the context of GC specifications, the model issue (crash-recovery *vs*. crash-no recovery) refers only to (JMS) clients.

4.2 JMSGroups specification

The specification in [29] is usually considered to be a standard specification for GC. However, as discussed in the previous section, JMS introduces features that impact the specification of GC, e.g., the use of stable storage (e.g., message persistence) and the potential recovery of crashed processes (e.g., in the context of durable subscriptions). In this section, we explain how these features impact the specification of GC. We first recall some definitions. We then split the specifications of GC into two parts: (1) the specification of the *reliability guarantees* provided by the broadcast primitive, and (2) the additional *ordering guarantees* that can be superimposed on top of the reliability guarantees provided by the broadcast primitive. Since these two issues are orthogonal, we discuss them separately.

4.2.1 Definitions

Correct and good processes

In this section, we use the term *process* as a synonym for *JMS client*. A process can be *up* or *down*. A process is up if it is operational, and down if it has crashed. A crashed process, after recovery, is again up. However, the specification of GC is not given in terms of the status up/down of processes at a given time. Instead, the specification refers to the status of processes *over their entire execution*. In this context, many specifications of GC consider that processes do not recover after a crash.[1] In this model, a process that never crashes is said to be *correct* and a process that crashes is said to be *faulty*.

However, because of durable subscriptions, the distinction between correct and faulty processes is not enough. We have to include in our specification the case of processes that crash and later recover. As in [1, 63], we say that a process is *good* if it is eventually always up, i.e., if there is a time t such that after t the process is always up.[2] So, a process that crashes only a finite number of times, and recovers after each crash, is a good process. Trivially, a process that never crashes (i.e., a correct process) is also a good process. Processes that are not good are said to be *bad*.

Membership views

A process group corresponds to a JMS topic. Processes can join a group by subscribing to the corresponding JMS topic; they can leave the group by unsubscribing from the corresponding JMS topic. So, the membership of a group changes over time. In GC, the current group membership is provided to the

[1] As already mentioned, this does not prevent a process from recovering after a crash. However, the consequence is that a process that crashes must recover under a new identity.

[2] It is usual in a specification to have properties that are eventually true forever. Actually, from a pragmatic point of view, it is sufficient that the property holds "long enough", where "long enough" depends on the application.

current group members. The information about the current membership of the group is called the group's *view*. We do not discuss here the precise specification, we only assume that for every group g, its successive views are totally ordered. Note that this specification is called *primary partition membership* [12].[3]

Broadcast vs. partial broadcast

A process that broadcasts a message can crash during the execution of the broadcast primitive. This is usually not a problem for specifications. If the broadcast has started, it is considered to be executed; if the sender crashes (during the broadcast or later) there is no obligation for the message to be delivered.

In the context of JMS, the situation is different. This is related to the acknowledgment mechanism provided by JMS. With persistent messages,[4] when some publisher process p (or JMS client) has received an acknowledgment from the JMS server, we have the guarantee that the message will be delivered by the destination processes, *even if p later crashes*. This leads us to distinguish *broadcast* from *partial broadcast*. Consider some process p that broadcasts (i.e., publishes) message m. If p receives the acknowledgment from the JMS server, we say that p has *broadcast* message m. If p crashes before having received the acknowledgment, we say that p has *partially* broadcast message m. Indeed, if no acknowledgment is received by p before the crash, there is no guarantee that the message is received by the JMS server.

4.2.2 Reliability guarantees of the broadcast primitive

We now formally define the guarantees provided by the broadcast primitive. The properties are expressed in terms of *broadcast* or *partial broadcast*, and *deliver*. [5] Delivery of some message m is the event by which a message is provided to a process (JMS client). We first discuss the case of non-durable subscriptions, and then the case of durable subscriptions.[6] These specifications are adapted from those in [66], which extends the specification in [29] to the case of dynamic groups.

Non-durable subscriptions

In the case of non-durable subscriptions (see Section 4.1.2), the specification distinguishes between correct and faulty processes:

- (P1) *Uniform Validity*: If a process broadcasts message m to the group g, then some *correct* process in g eventually delivers m, or no process in g is *correct*.

[3] In the specification we assume primary partition membership, but the API presented can also be applied to partitionable groups.

[4] Recall that we have excluded non-persistent messages from our discussion (Sect. 4.1.2).

[5] We could define *partial deliver* as well, but it does not influence the specification.

[6] To simplify the specifications, we assume here that all members of some group g have the same QoS for the subscription: either all have durable subscriptions, or all have non-durable subscriptions. However, note that in practice different subscription types for the members of the same group are possible.

- (P2) *Uniform Agreement*: If a process p delivers message m in view v, then all processes that are *correct* in v eventually *deliver* m.[7]
- (P3) *Uniform Integrity*: For any message m, every process in g delivers m at most once, and only if m was previously broadcast to g.
- (P4) *Uniform Same View Delivery*: If two processes p and q deliver m, in view v_i for p, and in view v_j for q, then $i = j$.[8]

The Uniform Validity property (P1) is similar to the one in [29]. It is the property we need in the open group model (Sect. 4.1.2), i.e., the model in which the process broadcasting a message to group g does not need to be a member of g. Note that the property is uniform, which means that the delivery is also ensured if the sender crashes after the broadcast has been executed (see the discussion in Sect. 4.2.1).

The Uniform Agreement property (P2) requires agreement on message delivery. While (P1) requires that some correct process delivers the message, (P2) requires that if some process (correct or not) delivers message m, then all correct processes also deliver m.

The Uniform Integrity property (P3) prevents the delivery of duplicate messages. It also requires that the delivery of message m is justified by a corresponding broadcast of m. Note that partial broadcast of m is not enough to guarantee the delivery of m, and duplicate delivery of partially broadcast messages is allowed by this property[9].

The Uniform Same View Delivery property (P4) requires that all processes deliver message m in the same view. This is a standard property in the context of GC. The property is sometimes replaced by a stronger property, called *Sending View Delivery* [12]. However, sending view delivery does not make sense in the open group model.

Durable subscriptions

In Section 4.1.2 we have discussed the link between durable subscriptions and the crash/recovery model. In the case of durable subscriptions, a process p that crashes at time t_1 and recovers at time t_2, after recovery is expected to deliver all messages it has missed in the interval $[t_1, t_2]$. This requirement can only be expressed if the specification distinguishes between good and bad processes, and not only between correct and faulty processes, as for non-durable subscriptions (see Section 4.2.1). Hence, for durable subscriptions, we simply replace *correct* by *good* in the properties (P1)-(P4) above (actually only in (P1) and (P2), since (P3) and (P4) do not refer to correct processes).

A comment is needed here for the reader familiar with the GC literature. In most existing GC systems, if process p crashes while in some view v_i, then p is removed from the group. This means that a new view v_{i+1} is defined, from which p is excluded. If p later recovers, and requests to join again,

[7] The notion of *correct in a view* or *v-correct* is defined as follows [66]: process p is *v-correct* if: (i) p installs view v, and (ii) p does not crash while its view is v, and (iii) if v is not the last view of some process in v, then $\exists$ view v' installed immediately after v by some process in v such that p is in v'.

[8] We say that process p delivers message m in view v_i, if the current view of p is v_i when m is delivered.

[9] This complies with the JMS specification, but as described there the potential duplicate messages resulting from partial broadcast must be marked with *JMSRedelivered* header field (we also keep this requirement for JMSGroups).

then a new view v_{i+2} is defined, which includes p again. In this case, all messages delivered in view v_{i+1}, will *not* be delivered by p. JMS forces us to handle the case differently with durable subscriptions: *a process p that crashes and later recovers, remains a member of the group, even while being down*. A process is removed from the group only as a result of an *explicit* request to leave the group (i.e., unsubscription from the corresponding topic). This is the behavior that users familiar with JMS expect from a durable subscription, and would be surprised not to have similar guarantees in the context of GC. Our specification enforces this behavior.

4.2.3 Ordering guarantees of the broadcast primitive

After the specification of the reliability guarantees, we now specify additional ordering guarantees for the delivery of messages. Traditionally, the choice is between no ordering requirement (called *reliable broadcast*), and total order (called *atomic broadcast*).[10]

There is however a more general and elegant solution; the solution consists in using the GC primitive called *generic broadcast* [58]. Generic broadcast orders messages according to a *conflict relation*. Generic broadcast ensures that two messages that conflict are delivered in the same order everywhere. Two messages that do not conflict do not need to be ordered.

For example, we can define that all messages tagged "reliable broadcast" conflict with all messages tagged "atomic broadcast". This ordering guarantee, which is very useful as illustrated in [48, 58], is not provided by the traditional approach. In our specification it can be adapted from [66] as follows:

- (P5) *Uniform Generic Order:* If some process delivers message m in view v before it delivers message m', and the two messages m, m' conflict, then every process p that is in view v delivers m' only after it has delivered m.

Note that specification (P5) is the same for non-durable and durable subscriptions.

For a process p that broadcasts a message to the group g, the "generic broadcast" approach has the following consequence. Instead of choosing a broadcast primitive (reliable broadcast or atomic broadcast), process p simply tags its message with one of the tags (e.g., tag=ReliableBroadcast or tag=AtomicBroadcast) defined for group g (there can be more than just two tags). The corresponding conflict relation (see Figure 4.1) is attached to the group, and defined at group creation time.

4.3 Mapping GC primitives to JMS API

The previous section has specified JMSGroups semantics. In this section, we present the JMSGroups API, more specifically, we show the mapping of traditional GC primitives onto the JMS methods related to the publish-subscribe paradigm. Clearly, a direct mapping is not always possible, as some GC concepts do not exist in JMS.

[10] We do not discuss causal order here.

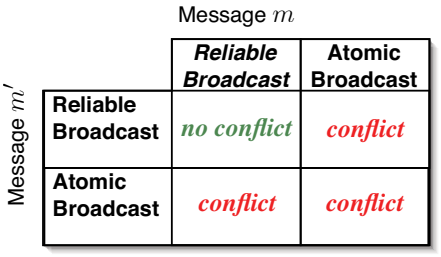


Figure 4.1: Conflicts between two messages that are reliably or atomically broadcast.

There are two possible approaches here: (1) rely strictly on the interfaces and standard mechanisms offered by JMS, or (2) add new interfaces to JMS where needed (e.g., for functionality specific to GC). Both approaches have advantages and drawbacks. Approach (1) has the important advantage not to modify the existing JMS API, whereas approach (2) violates JMS compatibility and thus might confuse developers familiar with JMS. On the other hand, approach (1) might, for some features, not be very natural from the perspective of GC. Approach (2) does not have this problem.

We have chosen approach (1). By not extending the JMS API for GC, we believe that we increase the acceptance of our proposal. In the following, we show how the additional GC functionality can be mapped onto existing JMS interface methods.

We have to devise a mapping for the GC functionality such as providing views in JMS to group members or issuing the requests to add and remove a process from a group. Moreover, while in GC systems, a process p can usually issue a request to add another process q to the group, the JMS API does not allow a client p to request a subscription for another client q. A similar issue arises for leaving the group, in the case of non-durable subscriptions (JMS provides the interface for topic subscribers to remove a durable subscriber from the topic subscriber list). So we have to provide a mechanism for client p to add another client q, and in the case of non-durable subscription to remove client q.

For this purpose, we use an extension mechanism provided by JMS. Indeed, JMS allows to attach arbitrary *properties* to messages. Using these properties, we can attach membership information to messages and construct special messages to add/remove other members to/from the group. With the same technique, we can map all the GC primitives to the existing JMS API, and remain fully compliant with the JMS API. In the next section we briefly present the JMS classes whose interfaces are relevant in this context.

4.3.1 Relevant JMS classes and methods

We represent groups as JMS topics and group members as the subscribers to these topics. So, in terms of JMS API, each client is connected to the JMS server using the `TopicConnection` class. The creation of a `TopicConnection` instance is expensive (it opens a TCP socket), thus the client usually uses one `TopicConnection` and from it creates different `TopicSession` class instances, usually one for each

topic it wants to subscribe and/or to publish. Once instantiated, the `TopicSession` class is used to create instances of `TopicPublisher` and `TopicSubscriber` classes, which are used to publish and to receive messages, respectively. When creating `TopicPublisher` and `TopicSubscriber`, the client has to provide a reference to the instance of the `Topic` class that represents the topic the client is interested in or wants to publish to. Topics are located on the JMS server and their creation is not defined by the JMS specification, i.e., different JMS implementations have their own ways to create topics. Providing the `Topic` instance reference to the client is also outside the scope of the JMS specification; most implementations for this purpose use the Java Naming and Directory Service (JNDI).

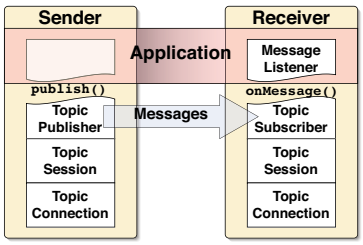


Figure 4.2: JMS classes (the arrow shows the logical flow of the messages).

To publish a message to a particular topic the client uses method `publish` of the `TopicPublisher` class instance. The reception is done either by (1) calling a method `receive` on the class `TopicSubscriber` instance (the call can be blocking if no message is available, or can return immediately), or (2) using a callback. If the client wants to use a callback, it has to provide `TopicSubscriber` with a reference to a class instance that implements `MessageListener` interface. Upon message arrival, the method `onMessage` of this interface is called. The messages are created by the `TopicSession` instance and are represented by instances of the class `Message`.

Figure 4.2 depicts the client side classes just presented. We rely on these classes to present the JMSGroups API in the next section.

4.3.2 JMSGroups API

The JMSGroups methods can be separated into two basic categories: *communication methods* and *administrative methods*. The latter are used to set up and maintain the group, while the former represent the interface used for the actual communication, i.e., for message broadcasts and delivery. Administrative methods and communication methods can further be characterized as *down calls* or *up calls*. Down calls correspond to usual method calls, and up calls correspond to callbacks. Tables 4.1 and 4.2 summarize the API mapping, which we now discuss in more details. Where needed, we indicate whether the primitive is a down or up call.

Representing GC's communication primitives using JMS API methods

The GC's communication primitives in the JMSGroups API are represented by the following methods (see Table 4.1):

Table 4.1: The Mapping from GC primitives to JMSGgroups API (communication methods).

Communication methods			
GC Primitive	**JMSGroups API**	**Call**	**Description**
broadcast(g,m)	`TopicPublisher.publish(m)`	down	Broadcasts a message to a group.
deliver(g,m)	`m=TopicSubscriber.receive()`, or `m=TopicSubscriber.receiveNoWait()`	down	Delivers a message.
	`MessageListener.onMessage(m)`	up	
viewChange(g,m)	`m=TopicSubscriber.receive()`, with the property *"JMS_view"* containing a new view.	down	Notifies about a view change.
	`MessageListener.onMessage(m)` with the property *"JMS_view"* containing a new view.	up	
getGroupView(m)	`m.getStringProperty("JMS_view")`	down	Returns the view in which message *m* was delivered.
suspect(g, processName)	Deliver a special JMS message with the property *"JMS_suspect"* containing a suspected process name.	up	Delivers a suspicion notification.

- *broadcast(g,m).* The *broadcast* primitive sends a message to all members of a group. In order to broadcast a message *m* to some group *g*, a client simply calls the method `publish(m)` on the instance of the JMS `TopicPublisher` class that corresponds to *g*. The client uses this method to send all messages, regardless of the type of ordering properties he expects (order or no order). The ordering constraints are defined by the message conflict relation (see Sect. 4.2.3), and the client just needs to attach the appropriate tag to each message.

- *deliver(g,m) — down call.* In order to deliver a message broadcast to group *g*, a client simply calls the method `receive()` on the instance of the JMS `TopicSubscriber` class that corresponds to *g*. The call is blocking if no message is available. Note that a non-blocking JMS method, called `receiveNoWait()`, is also available.

- *deliver(g,m) — up call.* To enable the delivery of a message broadcast to group *g*, a client can also register a callback provided by the JMS interface `MessageListener`.

- *viewChange(g,m).* Traditionally GC systems have a special call to notify group members about a view change. However, JMS has no such interface, but allows the attachment of *properties* to messages. So, a simple solution is to consider that delivering a new view *v* for group *g* is like delivering a message *m* for group *g*. A *view message* is distinguished from a *normal message* by its *"JMS_view"* property with a value equal to the new view. Similarly to normal messages, a view change message can be received either by a down call or an up call (callback).

- *getGroupView(m).* Traditionally, GC systems provide the means to query the current membership (i.e., view) of the group. JMS does not provide an interface method for this. As for view messages, we propose to attach a property *"JMS_view"* to the ordinary messages, whose value is the view in which the message was delivered. So, calling the method `m.getStringProperty(''JMS_view'')` returns the view in which message *m* was delivered. To get the current view, the client must call this method on the last message delivered, where the last message is either a normal message, or a view message.

- *suspect(g, processName).* The application can have their own fault detection mechanism or use a separate service. But as we mentioned in Section 4.1.2, the JMS server can be seen as an entity providing a GC service for the clients, in some implementation it can be used to provide a fault notification service as well. For such cases we introduce an interface for the suspicion delivery. As we did for the view change, here we also rely on a message with the dedicated properties. If the infrastructure suspects a process in the group *g*, then a special JMS message must be constructed with a property *"JMS_suspect"* containing a suspected process name as a value, and broadcast to the group (i.e., published to the topic). The client's reaction to the suspicion is application dependent.

Representing GC's administrative primitives using JMS API methods

The GC's administrative primitives in the JMSGroups API are represented by the following methods (see Table 4.2; the "Impacts" column in the table marks what process(es) the given method affects):

- *createGroup(g).* Creating a new group corresponds to creating a new JMS topic. This is done with `Session.createTopic(topicName)` method, where the argument `topicName` is the name for the newly created topic. The JMS specification notes that this method is provided for rare cases and clients that depend on this method are not portable. We provide this method for those applications

that need to create new groups dynamically. For those that do not need the dynamic group creation, to achieve better portability we suggest that groups are created by the administrator at the system initialization time, using proprietary server methods or from the configuration files.

- *setMessageConflictRelation(g, conflict)*. Message conflict relation can be set with a special message that contains a conflict description as a payload and is tagged with a property *"JMS_set_conflict"*. The server upon the reception of such message would set the conflict table for the corresponding topic and would not multicast the message to the subscribers. Similarly, as for the previous method, if the conflict relation does not need to be set dynamically, this can be done at the server initialization time with proprietary methods or configuration files.

- *joinGroup(g) — non-durable subscription – self*. As explained before, JMS API provides only the interface for a client to subscribe itself to a topic (the subscription of another client is discussed below). In the case of non-durable subscriptions, the client calls the JMS method `TopicSession.createSubscriber(g)`, where *g* is the topic representing a group.

- *joinGroup(g, processName) — durable subscription – self*. Joining a group with a durable subscription requires an additional parameter, namely the process name (provided by *processName*). In JMS, this parameter is used to uniquely identify a durable subscription. To join a group using durable subscription, the client thus calls the JMS method `TopicSession.createDurableSubscriber(g, processName)`, where *g* is the topic representing the group. Note that in this case *processName* corresponds to the name of the process executing this primitive.

- *joinGroup(g, processName) — non-durable and durable subscription – other*. The subscription of a client to a topic by another client is not specified in JMS. If GC needs this option, we propose the following solution using JMS properties. A special message containing two properties can be used to tell the JMS server that it should invoke a mechanism to subscribe another client. These properties are: *"JMS_join_process"*, containing process name, and *"JMS_subscription"*, containing subscription type for the newly joined process.

- *leaveGroup(g) — non-durable subscription – self*. The JMS API provides an interface for the client to unsubscribe itself from the topic. With non-durable subscriptions the client calls the method `close()` of the `TopicSubscriber` class associated with group *g* in order to leave it.

- *leaveGroup(g, processName) — non-durable subscription – other*. In GC, members can generally remove other members from the group, but JMS does not provide a mechanism to remove other subscribers from the topic, if they have non-durable subscription. Similarly as to *joinGroup(g, processName)* method we thus define a special message with a property *"JMS_remove"* containing the name of the process to be removed. When the JMS server gets such message, it removes the subscription for the specified client.

- *leaveGroup(g, processName) — durable subscription – other*. For durable subscriptions, JMS allows a client to unsubscribe another client. To remove a client from the group, a member client

calls the JMS method `TopicSession.unsubscribe(processName)`. Note that the JMS specification requires the process name to be unique not only in the group, but in the entire system.

4.4 Summary

In this chapter we have presented JMSGroups, a novel specification and API for group communication. JMSGroups adds GC as a third paradigm, besides point-to-point and publish-subscribe, to the JMS standard. For this purpose, we have shown how the features provided by GC can be mapped onto the JMS standard and thus bridge the semantic gap between GC and JMS. In particular, we have addressed the issue of durable/non-durable subscriptions and persistant messages, concepts that are not available in GC.

Table 4.2: The Mapping from GC primitives to JMSGgroups API (administrative methods).

Administrative methods				
GC Primitive	**JMSGroups API**	**Call**	**Impacts**	**Description**
createGroup(g)	`Session.createTopic(topicName)`	down	all	Creates a new group.
setMessageConflictRelation(g, conflict)	`TopicPublisher.publish(m)`, where *m* has the property *"JMS_set_conflict"* and its body contains a conflict definition.	down	all	Defines message conflict relation for group *g*.
joinGroup(g)	`TopicSession.createSubscriber(g)`	down	self	Adds the calling process to the group (non-durable subscription).
joinGroup(g, processName)	`TopicSession.createDurableSubscriber(g, processName)`	down	self	Adds the calling process to the group (durable subscription).
	`TopicPublisher.publish(m)`, where *m* has the properties *"JMS_join_process"* (containing process name) and *"JMS_subscription"* (containing subscription type).	down	other	Adds other process to the group.
leaveGroup(g)	`TopicSubscriber.close()`	down	self	Removes the calling process from the group (non-durable subscription).
leaveGroup(g, processName)	`TopicSession.unsubscribe(processName)`	down	other	Removes other process from the group (durable subscription).
	`TopicPublisher.publish(m)`, where *m* has the property *"JMS_remove"*, indicating the name of the process to be removed.	down	other	Removes other process from the group (non-durable subscription).

Chapter 5

Architectural issues

In Chapter 4 we proposed a standard specification and interface for GC. Instead of specifying yet another GC API, we took advantage of the widespread acceptance of the Java Message Service (JMS) and presented a GC API that was extended from the JMS API. The resulting specification and interface is called JMSGroups and is easily understandable both by the GC community and by developers familiar with JMS.

In this chapter we focus on the architectural issues for the JMSGroups implementation. Two main architecture types might be considered for JMSGroups: 1) a centralized server architecture and 2) a non-centralized architecture. The centralized server architecture is similar to the one used by JMS. In such architecture the GC service is provided by a separate middleware entity (the server). The group members are the clients communicating with each other through the server. The second architecture type is the classical GC model, without a central entity. Each group member has a GC layer which is responsible for group communication. Such architecture has been well studied in the group communication context [8, 57, 43, 75, 51, 15, 9] and will be only briefly presented in this chapter. Our discussion will focus on the centralized server architecture.

The JMSGroups centralized server architecture can be implemented by modifying the existing JMS server, i.e., by extending it to provide a GC service in addition to JMS. The specification of such a modification was presented in the previous chapter. However, since the server is a communication hub for all its clients, it becomes a single point of failure in the system. The crash of the server blocks the entire system. And since GC is used to provide fault-tolerance to the application, a single point of failure in its architecture is not acceptable. To avoid that, the JMS server used for the implementation of JMSGroups must be fault tolerant, i.e., replicated. Therefore, the presentation of the JMSGroups central server architecture issues in this chapter is divided into two parts. The first part focuses on the architecture of a replicated JMS server, in order to remove a single point of failure in the system (but without providing a group communication service yet). Different replicated architecture options are presented and compared. Then, by using the replicated JMS server architecture as a base, the second part presents the modifications that are needed to implement the JMSGroups server (and thus provide a group communication service in the JMS-based system).

Chapter Roadmap. Section 5.1 gives a brief explanation of the terms we use in this chapter, Section 5.2 presents the architecture options for the fault tolerant JMS server, whereas Section 5.3 analyzes how the replicated JMS server should be modified to provide a group communication service. Section 5.4 briefly presents the non-centralized server architecture for JMSGroups. Related work is then presented in Section 5.5 and finally Section 5.6 summarizes the chapter.

5.1 Notation

To avoid the confusion for the reader let us give the explanation of the terms we use in this chapter:

1. *Single JMS server* or just *JMS server* - a server that complies with the Sun^{tm} JMS specification and provides the JMS service for its clients.
2. *JMS client* - a stand alone application which uses the service provided by the JMS server.
3. *JMSGroups server* - a server which provides a JMS compliant group communication service (it can provide the JMS service as well).
4. *Replicated JMS/JMSGroups server* - a JMS/JMSGroups server replicated to provide fault tolerance.
5. *JMSGroups member* - a stand alone application that uses the group communication service provided by JMSGroups server.
6. *JMSGroup* - a set of JMSGroups members belonging to the same group.
7. *JMSGroups client* - a stand alone application that uses the service provided by a JMSGroup.
8. *Server communication channel* - the communication channel used by a replicated JMS or JMSGroups server for communication between the server replicas.
9. *Client communication channel* - the communication channel used between the server and the clients. This channel is different from the one between the replicated servers.

5.2 Fault tolerant JMS server architecture

As already stated, our goal is to build a JMS compliant group communication service. The service must be as close as possible semantically and in terms of the interface to JMS. If a server is used to provide such a service, clearly the server itself must be fault tolerant.

Fault tolerance is achieved through replication and in this section we present the architecture for a replicated JMS server. Note that here we talk about “pure” JMS, i.e., without a group communication service. The architecture we present below can be applied to any JMS server to make it fault tolerant.

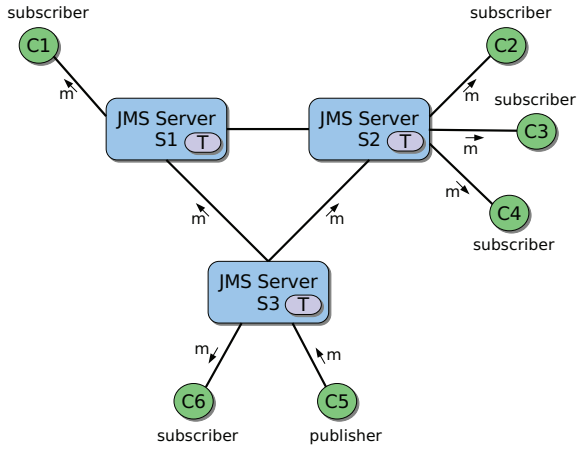


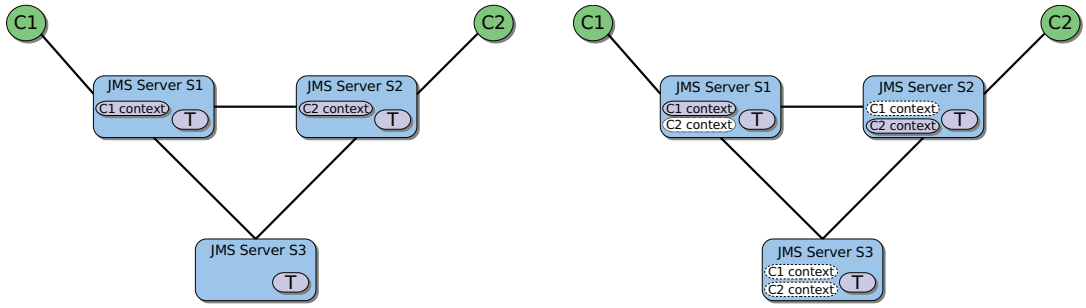
Figure 5.1: Replicated JMS server.

Understanding the replicated JMS server architecture will allow us to introduce the changes needed to provide a group communication service; we discuss these changes in Section 5.3.

A typical example of a replicated JMS server architecture is shown in Figure 5.1. The JMS server consists of three replicas $\{S1, S2, S3\}$. Six clients $\{C1, C2, C3, C4, C5, C6\}$ are connected to the different server replicas. The server contains a replicated topic T, i.e., each server replica hosts a replica of T. The clients can connect to the topic as publishers or subscribers, or both. In our example client $C5$ is a publisher, and the rest are subscribers to T. When $C5$ publishes a message m to the topic T, the message is first received by the server replica $S3$. $S3$ then sends (broadcasts) the message to the server replicas, so that every replica receives it. After that m can be dispatched to the subscribers of T. On their behalf, server replicas $S1$ and $S2$ also dispatch m to the subscribers of T, if any.

The JMS server replication does not influence the properties of the client communication channel (the channel between the server replica and the clients connected to it). We assume that this channel satisfies reliable FIFO message delivery. We also assume that server replicas process the messages in sequential order, i.e., do not reorder them. These assumptions will remain valid throughout the chapter.

As already mentioned in Section 3.1, when the JMS client connects to the server, a client context for that connection is created on the server (see Figure 3.2). Depending on how the client context is managed on the replicated server we distinguish two JMS server replication options: (1) server replication with *non-replicated context* and (2) server replication with *replicated context*. In case (1), which is illustrated in Figure 5.2(a), a single client context is created on the server replica when the client connects to it, i.e., this context is not replicated on the other server replicas. Thus the server replicas do not hold any state related to the contexts managed by the other replicas. On the contrary, in case (2), illustrated in Figure 5.2(b), each client's context is replicated on all server replicas and their state is kept consistent. These two JMS server replication options are presented in detail in the following sections.



(a) Non-replicated client context (b) Replicated client context

Figure 5.2: Different JMS server replication options.

5.2.1 Non-replicated context

In JMS server replication with non-replicated context, each client chooses one server replica to connect to and receives the requested messaging service from it. The client context is created only on the server replica to which the client connects, and it is not shared between the other server replicas. Thus each server replica hosts only a subset of the client contexts in the system (see Figure 5.2(a)).

The problem of such an architecture is that, in the case of a server replica crash, the clients of the crashed replica cannot connect to the other server replicas as those do not have the sufficient information to restore the clients' context. Therefore, the clients of the crashed server replica are isolated from the whole system. This situation lasts until the recovery of the crashed server replica. After the recovery, the clients can reconnect to the same server replica and continue to receive messages. Stable storage must be used on the server replica to prevent the context loss in case of a crash. Recovery and stable storage are not specific requirements for the replicated JMS server as they are defined in the JMS specification. Because the crashed server replicas recover there is no need to remove them from the group (they recover with the same identity). Therefore, the static group membership model suits the best for the JMS server replication with non-replicated context.

As stated above, in server replication with non-replicated context a server replica is responsible only for the subset of the clients connected to it. In the case of a server replica crash this subset is isolated from the rest of the system. However, the clients connected to the non-crashed server replicas still have access to the service, i.e., only part of the system is not functioning. The other part is still operational and can produce and consume messages (which is not the case in a single server architecture). Therefore, the overall server state changes even when there is a crashed replica. This poses a problem to the durable subscribers.

Durable subscription requires to deliver even those messages which were produced when the subscriber was not connected to the server. Consequently, in server replication with non-replicated context, the durable subscribers connected to the server replica which crashed and recovered, must also receive the messages produced during the down time of the replica. To solve this problem, the non-crashed repli-

cas have to store the part of the system state on behalf of the crashed replicas until they recover. This state consists of the messages addressed to the crashed replica, which were produced between the crash and the recovery.

The other problem is the message delivery order between the clients of a replicated JMS server. As mentioned in Section 3.4, JMS requires FIFO message delivery. We will show that to ensure FIFO order between the clients, a reliable FIFO broadcast primitive is sufficient for communication between the server replicas (the server communication channel).

Lemma 1. *For server replication with non-replicated context, reliable FIFO message delivery is sufficient between the server replicas to provide the reliable FIFO message delivery order between the clients.*

Proof sketch. To deliver the messages between the clients both communication channels in the system are used: the client channel and the server channel (the communication channel between the server replicas). For each client, a separate client communication channel connects the client to a server replica and as defined earlier, this channel satisfies reliable FIFO message delivery property. Also, we assume that server replicas do not lose messages and process them in sequential order, i.e., do not reorder them. Thus, the communication primitive between the server replicas must preserve the FIFO message order it receives from the client communication channel. For that, a FIFO communication primitive is enough. In addition, this primitive must be reliable in order not to lose any messages between the server replicas. □

5.2.2 Replicated context

In server replication with replicated context architecture, each client also connects to one of the server replicas and receives the messaging service it requests from that replica. Here, in contrast to the previous solution, each client context is replicated on all server replicas and the state of the context replicas is kept consistent during the system execution (see Figure 5.2(b)). As the JMS client connects and interacts with a single server replica, its context on that replica is called *active* (shown with color filling and shadow in Figure 5.2(b)). The same client's contexts on the other server replicas are not active (shown in white in Figure 5.2(b)), but their state is kept up to date with the active one. This is similar to primary-backup replication.

In the case of a server replica failure, all the clients connected to that replica are automatically reconnected to another non-failed server replica and continue getting the messaging service. When the client reconnects to another server replica, its context on that replica becomes active. The reconnection is done automatically without any user intervention.

Due to the replicated context and the client reconnection mechanism, the clients connected to a given server replica do not lose the service when the replica crashes. In other words, unlike in the non-replicated context solution, a server replica crash does not render part of the system non-operational. Therefore, the recovery of a crashed server replica is not as crucial as before. Moreover, if there are enough non-crashed server replicas, the service continues to be provided to the whole system, even if the crashed replicas do not recover. This allows a different failure model to be used for the server. Instead of the crash-recovery

model required by the non-replicated context, the crash-stop model can be used for the server replicas. In the crash-stop model the crashed replicas do not recover and are removed from the group.[1]

To keep the desired number of server replicas, new replicas can be created and added to the group dynamically. A state transfer mechanism must be provided to synchronize the state of the added replicas with the rest of the system. The removal and addition of members during the runtime corresponds to the dynamic group membership model for the JMS server replicas.

With the replicated context and the assumption that the majority of the server replicas do not crash, there is no need for stable storage on the server replicas. The client contexts (including messages) are replicated on the server and are not lost. Without such assumption, stable storage and the crash-recovery failure model must be used for the server replicas.

For the replicated context, the states of client' context on the different server replicas must be consistent to satisfy the JMS message delivery requirements. The message delivery properties must be satisfied even when a server replica failure occurs and the affected clients reconnect to another server replica. We show that reliable FIFO for the server communication channel is still enough to satisfy the JMS message delivery requirements.

Lemma 2. *For server replication with replicated context, reliable FIFO message delivery is sufficient between the server replicas to provide the reliable FIFO message delivery order between the clients.*

Proof sketch. The proof is similar to the one of Lemma 1. The difference with the replicated context is that the state of the context is present on each server replica. That state consists of the message queues which contain the messages from/to the client. To comply with the JMS specification, the order in which messages are received in the queues for different context replicas can be different, but must satisfy FIFO. As the client communication channel satisfies the reliable FIFO message order and the server replicas do not reorder messages and process them sequentially, the reliable FIFO message communication primitive is sufficient between the server replicas to keep the FIFO message delivery order to the client context replicas. □

Client reconnection example. For a better understanding of the client reconnection mechanism, let us additionally illustrate it by an example and show how the FIFO order is preserved on the JMS clients during the reconnection. Let's take an example of a replicated JMS server with replicated context shown in Figure 5.3. The JMS server consists of three replicas $S1$, $S2$ and $S3$ connected with reliable FIFO communication channel. Client $C2$ is connected to the replica $S2$. Assume that $C2$'s context on each replica contains two messages produced by different producers (the producers are also the clients, but are not shown in Figure 5.3): on replica $S1$, the message order is $\{m1, m2\}$, on replica $S2$ it is $\{m2, m1\}$ and on replica $S3$, the order is $\{m1, m2\}$. The order on $S2$ is different, because the messages are produced by different producers and FIFO channels guarantee the same message order only for messages produced by the same producer. Assume that the first message in the queue on $S2$ (message $m2$) is delivered to $C2$ and that after that $S2$ crashes. After the crash, $C2$ reconnects to $S1$. Here, depending on $C2$'s context

[1] In fact, a server replica can recover, but must join the group as a new member, i.e., with a new identity.

state there are two possible scenarios: a) message $m2$ was acknowledged by $C2$ and garbage collected on the server replicas before the crash of $S2$, and b) message $m2$ was acknowledged by $C2$, but not garbage collected on server replicas.

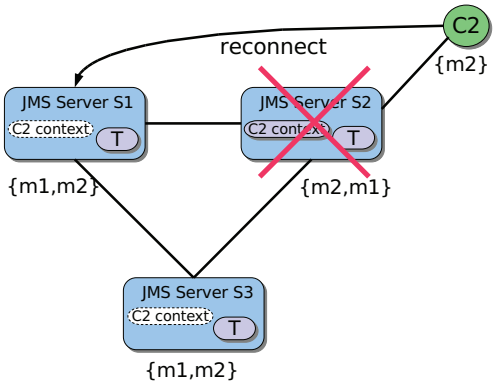


Figure 5.3: Client reconnection scenario.

In case (a), message $m2$ will have been garbage collected before $C2$ reconnects to $S1$. For $C2$ there is therefore no risk that $m2$ will be redelivered. After the reconnection $S1$ will send $m1$ to $C2$ and the message delivery order on $C2$ will thus be $\{m2, m1\}$. If there were other message consumers for $m1$ and $m2$ on $S1$, they would deliver the messages in the order they were delivered on $S1$, i.e., $\{m1, m2\}$. Thus, after the reconnection, the order of message delivery can differ on the clients connected to the same server replica, but this does not violate the JMS specification as the FIFO order is preserved. Since the client context on each server replica receives the messages in FIFO order and processes them sequentially, the FIFO order for the messages will be preserved even in the case of a reconnection.

Case (b) is more complicated because of a possible message duplicate, since message $m2$ is delivered to $C2$, but not garbage collected by the server replicas before the client reconnection. Thus after the reconnection, $S1$ will send message $m1$ to $C2$, as it is the first in its queue. When $m1$ is acknowledged and garbage collected $S1$ will send $m2$ to $C2$, which would be a duplicate of the one received by $C1$ from $S2$ before the crash of $S2$. However, this still satisfies the JMS specification for duplicate of the last delivered message in the case of a JMS server crash (see Section 3.4). $C2$ must be ready to handle the duplicate of the last delivered message (in our case $m2$) after the reconnection. Except for this, the message delivery order issue is the same as in case (a), i.e., the reconnection to another server replica won't violate FIFO order on the client.

5.2.3 Comparison of the JMS server replication options

Table 5.1 presents the comparison between the non-replicated context and replicated context options. The replicated context uses the simpler crash-stop failure model, has an option not to use the stable storage and most importantly does not isolate the clients in the case of a server replica crash, which

greatly improves system liveness. But on the other hand, every server replica keeps the client context of the whole system, which can cause a resource problem with a large number of clients, and makes the system less scalable.

On the contrary, the non-replicated context, for a large number of clients, can fully exploit load balancing by distributing the clients between the server replicas, whereas with the replicated context load balancing is less efficient. Moreover, the static group membership model used by the non-replicated context is simpler and easier to implement than the dynamic one used by the replicated context. However, the non-replicated context option requires server replica recovery, which in general is more difficult to implement, but is required in the JMS specification and is already implemented in most of the non-replicated JMS servers. Also the reconnection protocol to the same recovered server replica is simpler than the one required by the replicated context, when the client reconnects to a different server replica.

While the replicated context option seems to be a more attractive choice for the replicated JMS server, it is hard to draw a strict line between the two. The choice depends on the needs and properties of the particular application that uses the replicated JMS server.

Table 5.1: Replicated JMS server: comparison between replicated and non-replicated context.

	Non-replicated context	**Replicated context**
Failure model	Crash-recovery	Crash-stop
Group membership model	Static membership	Dynamic membership
Communication primitive	Reliable FIFO	Reliable FIFO
Client behavior for failures	Wait until the replica recovers	Reconnect to an available replica
Stable storage needed	YES	NO, if a majority of replicas do not crash
Number of client contexts in the system	$\|Clients\|$	$\|Clients\| \times \|Server\ replicas\|$

5.3 JMSGroups based on JMS server

Our JMSGroups specification can be implemented using an existing JMS server. For that the JMS server must be changed internally to provide group communication as a service to its clients. Let us remind, that such a modified JMS server is called a JMSGroups server. A JMSGroups server contains special topics called *group topics*. The clients form a group by subscribing to the corresponding group topic. Compared to JMS, JMSGroups provides additional group communication services to the clients subscribed to the group topics (group members): group membership information, member suspicions, etc. It can also optionally provide JMS service to the clients that do not need GC.

As discussed before, for the JMSGroups implementation based on a JMS server, a replicated JMS server must be used, such as the one presented in the previous section. As such, JMSGroups adds some

additional requirements to the replicated JMS server architecture. We will present these requirements in the following paragraphs, but first, we define the two replication levels that exist in JMSGroups centralized server architecture.

Two level replication in JMSGroups. The replicated JMSGroups server provides GC as a service used for replication by its clients. This forms a system with two replication levels: (1) a *server replication level* and (2) an *application replication level* (see Figure 5.4). The server replication level is responsible for the server replication and the application level for the application replication. Each level uses a distinct group communication infrastructure to provide the replication. The server level uses GC provided to the server replicas by a group communication toolkit. The application level on the other hand uses the GC provided by the JMSGroups server. The server level consists of a single group (the one of the server replicas), whereas the application level supports many groups, as well as standalone clients (see Figure 5.4).

Both levels define distinct communication primitives to provide message delivery guarantees. Message delivery at the application level depends on the primitives at the server level, but not vice versa. To distinguish to which level the primitive belongs we add the corresponding prefix to the name of the primitive, e.g., *S-ABcast* is a server level ABcast and *A-ABcast* is an application level ABcast.

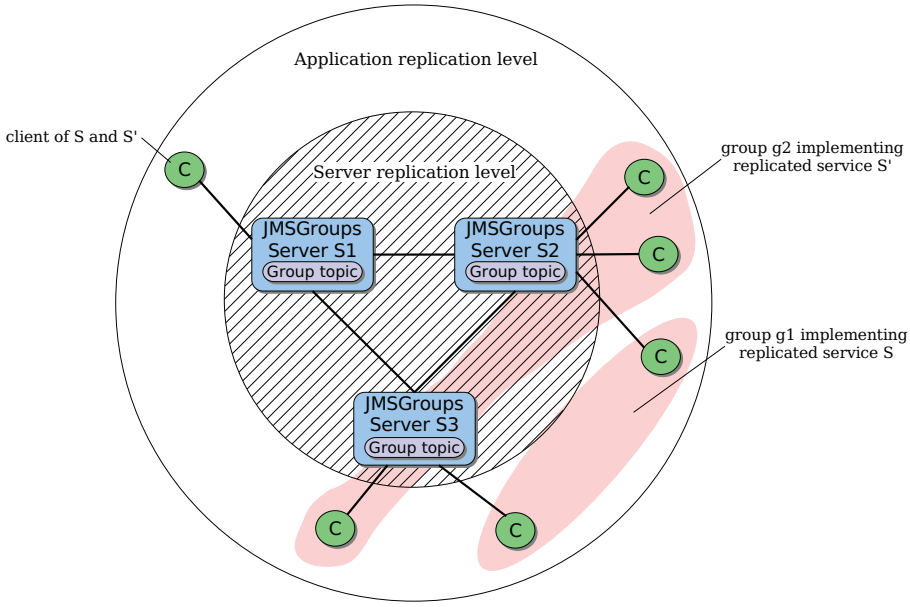


Figure 5.4: Two replication levels in JMSGroups.

Server replication options in JMSGroups. In Section 5.2 we presented two different options for replicated JMS server architectures: replication with non-replicated context and replication with replicated context. They differ in the way the client context is kept on the server and in the behavior of the

clients when the server replica fails. The same two solutions apply to the JMSGroups architecture, more precisely to the server replication level. If the non-replicated context is used at the server replication level, a failure of a server replica will not be transparent at the application replication level. Indeed, the clients connected to the failed replica lose the connection and are isolated from the rest of the system until the server replica recovers. Depending on the application requirements, the time until the server replica recovers can be too long for the group members to wait for the reconnection. In such a case, the replicated context is an alternative. With replicated context, the clients do not wait for the failed replica recovery, but instead reconnect to another available one. The time taken by the clients to reconnect to another server replica is much shorter than the server replica recovery delay, and therefore the clients are not isolated from the rest of the system. Unfortunately, replicated context has a higher communication cost between the server replicas and uses more resources on the server.

The choice between the non-replicated context and replicated context must be done considering the nature and the requirements of the application using JMSGroups: a server with non-replicated context will perform better than the one with replicated context as long as no failures occur. If a failure occurs, a system using a non-replicated context will however isolate a number of clients until their server replica recovers.

Message delivery order in JMSGroups. In JMSGroups, as in JMS, all communication between the group members goes through the JMSGroups server. This implies that the application level primitives A-ABcast and A-ADeliver are composite primitives, i.e., they are implemented using primitives of the server level (see Figure 5.5). Indeed, A-ABcast consists of reliable FIFO between the JMSGroups client and the server, plus S-ABcast between the server replicas. Similarly, A-ADeliver consists of a S-ADeliver between the JMSGroups server replicas, plus a reliable FIFO delivery between the server and the JMSGroups client.

In Section 5.2 we showed that reliable FIFO message delivery is enough for the server communication channel in order to comply with the JMS specification. Group members in JMSGroups require stronger message delivery guarantees than JMS clients, and consequently reliable FIFO is not enough for the server communication channel. A common GC requirement is the total message delivery order for the group members. To provide total order in JMSGroups application replication level, stronger message delivery properties (e.g., ABcast) must be used for the server replication level.

Lemma 3. *For the JMSGroups server replication level, a total message order primitive (S-ABcast) allows us to provide the total order message delivery to the group members at the application replication level.*

Proof sketch. The S-ABcast primitive used by the JMSGroups server replicas guarantees the total order of message delivery between the server replicas. As there is a FIFO communication link between the server replica and each client, and the server replicas do not reorder or lose the messages, the delivery order of messages on the server replicas will be preserved on the clients as well. Total order between the server replicas ensures total message delivery order for the group members. □

```
A-ABcast(m) {
   Member: FIFO send m to the Server;
   Member: wait for the Ack(m) from the Server;

   Server: receive(m) from a member;
   Server: S-ABcast(m) between the Server replicas;
   Server: S-ADeliver(m);
   Server: send Ack(m) to the Member;
 }
A-ADeliver(m) {
   Server: S-ADeliver(m);
   Server: FIFO send m to the Member;
   Server: wait for the Ack(m) from the Member;

   Member: receive m from the Server;
   Member: send Ack(m) to the Server;

 }
```

Figure 5.5: JMSGroups member's composite total order communication primitives.

5.4 Non-centralized architecture

The alternative architecture to the centralized server based JMSGroups is the non-centralized architecture. An example of the non-centralized JMSGroups architecture is shown in Figure 5.6. In this architecture clients do not rely on the server for communication, but each client has an independent communication unit with the underlying protocol stack, which is responsible for group communication. In other words, each JMSGroups member or client corresponds to a group member. This is exactly the same architecture as in most of the GC toolkits. The only difference is the additional *JMSGroups adapter* layer between the GC protocol stack and application layer, which is responsible to provide JMS compatible interface to the application and convert its JMSGroups calls to GC and vice versa.

Having a GC toolkit, the non-centralized architecture is relatively easy to implement: basically the JMSGroups layer must be adapted for the given GC toolkit interface. Unfortunately, such architecture suffers from the semantic inconsistencies between JMS and GC described in Section 4.1. Namely JMS durable subscription, which in GC maps to the static group membership model with process recovery (see Section 4.1.2) is imposed for all group members, because the whole group must use the same group membership and failure model. This differs from the standard JMS, where durable subscription can be chosen or not by each subscriber individually. As a result subscription durability which is a member property in JMS, in non-centralized JMSGroups architecture becomes a topic (group) property, which is not compatible with the JMS specification.

The centralized server architecture, on the contrary, does not suffer from this problem as it does not have one-to-one mapping between the underlying GC and JMSGroups members.

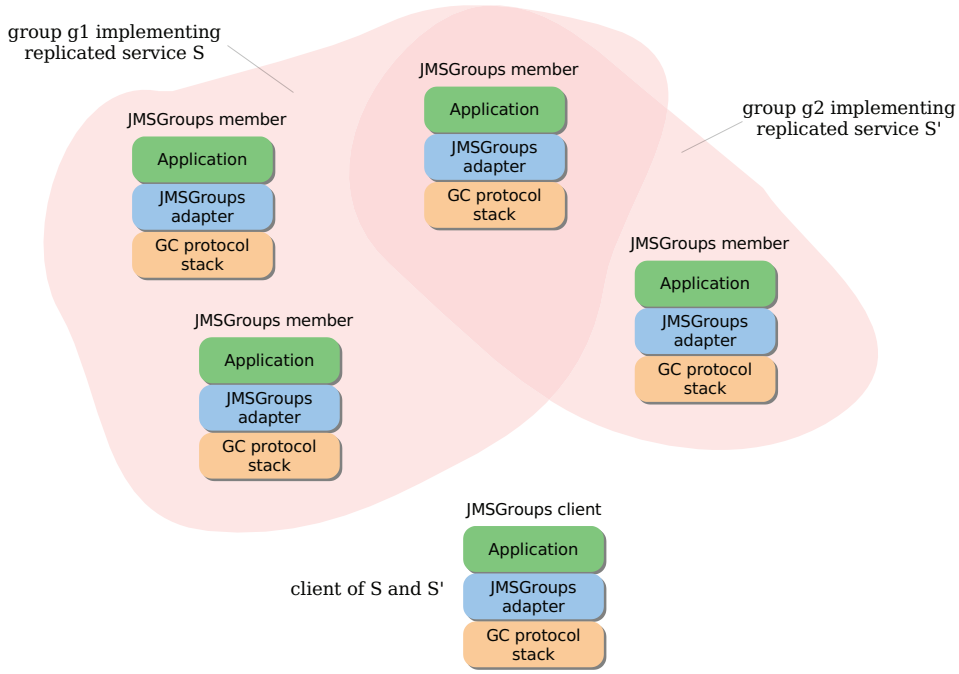


Figure 5.6: Non-centralized JMSGroups architecture.

5.5 Related work

The open source JMS implementation called JORAM [64] as an option provides high availability for the JMS server by replication. JGroups [2], a Java group communication toolkit, is used for the communication between the server replicas. A replicated JORAM server uses primary backup replication, i.e., only one server replica is communicating with all the clients, and its state change is synchronously propagated to the backup replicas. In the case of a failure of the primary replica, a new primary is elected among the backup replicas and the clients reconnect to it. The client reconnection mechanism preserves JMS message delivery properties for the clients. This architecture is similar to the JMS server replication with replicated context described in Section 5.2.2. However, in our proposed architecture primary-backup replication is used only for the client context and not for the server replicas. The advantage is that the clients can connect to any of the server replicas, not only to the primary as in the case of JORAM.

Another replication mode provided by JORAM is called *collocated client mode*. In this mode JMS clients are collocated and replicated together with the server replicas. However, only stateless clients can be used in this mode and only one client replica (the one located on the primary server replica) is receiving and processing the messages. For the other client replicas the communication with the server is blocked. This can be compared with the non-replicated context described in Section 5.2.1. However, in our architecture the clients do not have to be collocated together with the server and can contain a state.

Although JORAM's collocated client mode deals with client replicas, it is too constrained and cannot be compared with the GC service provided by JMSGroups

Another JMS implementation called SonicMQ, which is a commercial product from the Sonic Software Corporation, also provides a replicated JMS service [72]. JMS topics in SonicMQ are replicated together with the server, which allows to apply a load balancing mechanism for the connecting clients. The replicated server uses non-replicated client contexts. Additionally, for durable subscriptions, the client contexts are replicated on the server and a similar client reconnection technique to the one of the replicated context described in Section 5.2.2 can be used. However, unlike in the replicated context, the state of these client contexts on the server replicas are not kept consistent with the replica communicating with the client. The risk therefore exists to lose messages after the reconnection to a different server replica. If the option not to lose the messages is chosen, the client is required to reconnect to the same replica and a part of the system is blocked until the failed server replica recovers.

5.6 Summary

JMSGroups provides a JMS compliant group communication, whose specification and API were defined in Chapter 4. As a follow-up, this chapter focused on the architectural issues related to the JMSGroups implementation.

We have chosen to implement JMSGroups by internally modifying the existing JMS server and adding group communication service to it. It is clear that such a service itself must be tolerant to failures. Therefore, JMSGroups must be based on the replicated JMS server. We proposed two different approaches for replicating the JMS server: non-replicated context and replicated context. In the first approach, each server replica contains only the contexts of the clients connected to it. In such a system load balancing between the server replicas can be used more efficiently. But in the case of a crash, clients connected to the crashed server replica are isolated from the system until the replica recovers. Furthermore, since recovery is needed, the server replicas need access to stable storage in order to periodically save their state. In the second approach each server replica stores the contexts of all clients connected to the system. This allows the clients to reconnect to the other server replica, when the one they are connected to crashes. Moreover, server replicas do not need stable storage anymore (as long as the majority of replicas do not crash), since each replica has a copy of all client contexts. The drawbacks of this approach are: a bigger resource requirements by the server replicas and a higher network communication cost, since the server replicas need to exchange more information to keep the client contexts' states consistent.

The second part of the chapter addressed the issue of providing a group communication service on top of the replicated JMS server. To provide a group communication service, we proposed the JMSGroups server architecture defining two levels of replication: the server level and the application level. For the server replication level the same replication approaches as for the JMS server are used, but with the stronger communication primitives. At the same time, the service provided to the application level enables the clients to delegate the complicated and expensive communication primitives (e.g., total message

order) to the server, and still profit from the group communication to create fault tolerant applications.

Non-centralized JMSGroups architecture was presented in brief, which is identical to the architectures used by the GC toolkits. With this architecture, JMS durable subscriptions do not comply with the JMS specification any more.

Chapter 6

JMSGroups Implementation

In the previous chapters we presented the JMSGroups specification and the possible architectures for the replicated JMS server. This chapter presents the JMSGroups implementation. We have done two implementations: one is based on the modified JORAM server [64] (version 3.6), the other based on *Component Chain* and is built from scratch. The first implementation was done like a proof of concept, but because of poor documentation provided for JORAM it was hard to track all the details of the server internals and to extend it. For that reason we decided to make our own implementation. The two implementations are presented at a high level, more details can be found in the manuals attached to the each of them.

Both implementations are based on the component architecture which consists of components and the container. The components are small objects providing basic functions, they are created, held and managed by the container. Inside the container the components exchange messages and collaborate to provide the service. Some dedicated components communicate with clients that are outside the container and/or other components on the different containers. The general scheme of such architecture is shown in Figure 6.1.

Chapter Roadmap. Before presenting JMSGroups implementation Section 6.1 introduces the Fortika group communication toolkit, that is used in both implementations. The JORAM based implementation is presented in Section 6.2 and the Component Chain based implementation in Section 6.3. Performance evaluation and a use case example are given in Section 6.4 and Section 6.5 respectively.

6.1 Fortika

The Fortika group communication toolkit [48, 47] developed at École Politechnique Fédérale de Lausanne (EPFL) provides a new architecture for the group communication protocol stack. The protocol stacks, used in the popular group communication toolkits rely on the architecture historically influenced by the pioneer in this domain namely the Isis toolkit [8]. In the "traditional architecture" group membership and view synchrony services are the basic components in the system [12]. Other group communi-

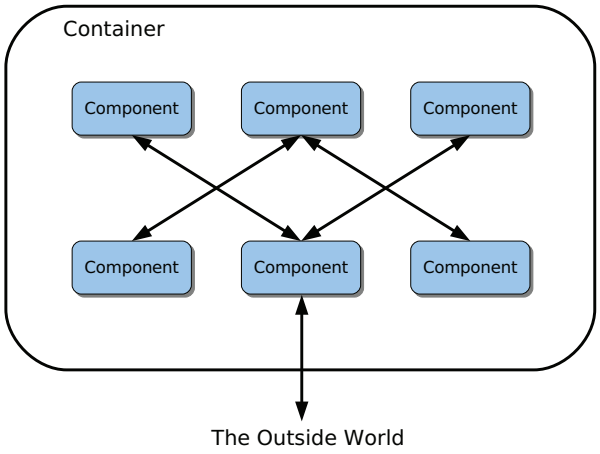


Figure 6.1: General component architecture.

cation services, e.g., atomic broadcast, are implemented using these services.

The Fortika framework architecture adopts a new approach to the group communication protocol stack architecture [66]. The first feature of the new architecture is atomic broadcast as a basic component, instead of group membership and view synchrony. Atomic broadcast is used to build other group services on top of it, e.g., group membership, and not vice versa. The second feature of the architecture is the absence of the view synchrony component, which is replaced by more general and powerful abstraction, called generic broadcast [58]. The third feature is the decoupling of the group membership and failure detection services. Such decoupling allows the system to distinguish between failure suspicion and membership exclusion. As a result, a failure detection does not necessarily lead to the exclusion of the suspected process.

The benefits of the Fortika new architecture are: (1) less complex group protocol stack thanks to the relocated atomic broadcast component, which enables us to use the same ordering for messages and views; (2) more flexibility and power, thanks to generic broadcast, which gives more control over message order; (3) higher responsiveness, thanks to the decoupling of group membership and failure detection, which allows to reduce failure detection timeouts.

The Fortika architecture is implemented using two different protocol composition frameworks: Appia [50] and Cactus [32]. We used the Fortika implementation based on the Cactus framework.

6.2 JORAM based implementation

This section presents JMSGroups implementation based on the modified JORAM server (v3.6) [64]. It gives an introduction to the JORAM architecture and explains the changes we made to it to implement JMSGroups specification.

JORAM is the open-source JMS implementation based on the ScalAgent distributed asynchronous

agent platform [65]. ScalAgent is a proprietary message oriented middleware (MOM), with its own defined message types and API. JORAM is the JMS API wrapping around the ScalAgent MOM. First we introduce the basic notions of the ScalAgent/JORAM architecture.

6.2.1 ScalAgent asynchronous agent platform

ScalAgent is an asynchronous distributed agent platform which provides a framework for distributed systems development. The communication between its agents is based on asynchronous message passing [6]. The ScalAgent architecture (see Figure 6.2) contains three main types of components: *agents*, *agent servers* and *channels*. Agents are "passive" units of execution, that are "woken up" for a particular event (message) to process. They have persistent state, i.e., in the case of a crash the state of the agent is preserved. Agent's event processing is atomic, i.e., the event is either fully processed, or the processing is aborted without any impact on the agent's state. Agents are managed by the agent server (container), a single process that contains a set of agents. An agent server has an entity called `Engine` running inside it. The `Engine` is responsible for finding the corresponding agent, when given its ID, "waking it up" when the event of interest happens, transferring the message to it and making sure that agent's processing is transactional. The third type of components, channels, are message queues which receive, hold and dispatch the events (messages) in the ScalAgent system. The agents residing on the same or on the different agent servers do not communicate directly with each other, but rather they use channels to exchange the messages. This decouples the agents and provides asynchrony, i.e., if agent A sends a message to agent B, there is no need that both of them are active at the same time. A puts a message in the channel and B takes it when ready to process it. ScalAgent channels are also transactional and can provide causal message ordering.

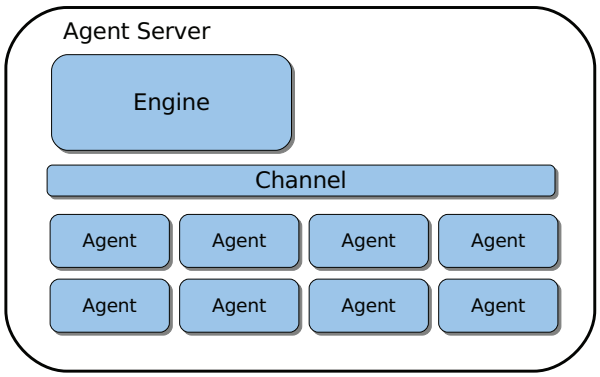


Figure 6.2: ScalAgent architecture.

6.2.2 Clustered topics in JORAM

As already mentioned, JORAM provides JMS wrapping of the ScalAgent framework. It consists of a set of specialized agents which provide JMS functionality and the entities which expose the JMS API to the client. The main JMS entity in JMSGroups, the topic, is represented in JORAM by a `Topic` agent.

In JMS a topic is represented as a single entity created and residing on the server and dispatching the messages to its subscribers. The users connect to the server in order to subscribe to the topic. If one server fails the topic becomes unavailable for all subscribers. To avoid this problem, JORAM provides the possibility to introduce several servers, and to cluster JMS topics residing on the different servers to a ”single” topic. The client subscribing to such topic can choose to which JMS server to connect. When a message is published to such topic on one server, it is received by the subscribers on all the servers (see Figure 6.3). If the server fails, the subscribers connected to that server loose the connection, but subscribers on the functioning servers are still able to publish and receive the messages. Clustered JORAM topics correspond to the JMS server replication with non-replicated context presented in Section 5.2.1. To implement the JMSGroups specification presented in Chapter 4, we implemented a new agent called `GroupTopic`. The main features added to the `GroupTopic` are the group view and suspicion notifications dispatched to the group members.

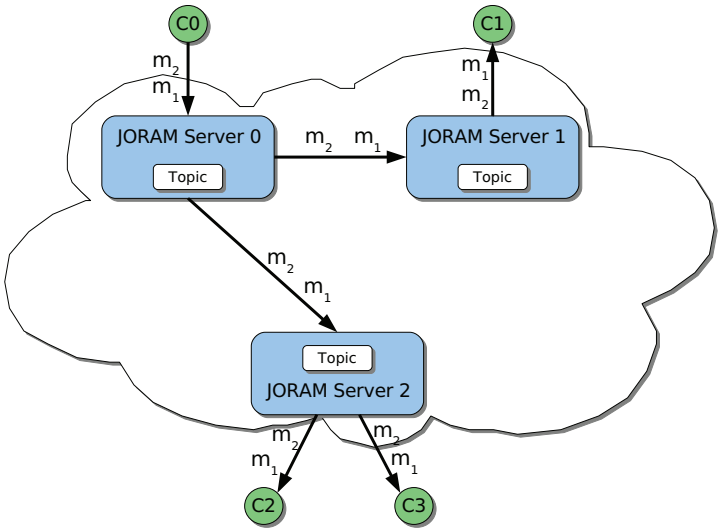


Figure 6.3: Clustered topic in JORAM.

The communication protocol used between JORAM servers hosting clustered topic is TCP/IP. Nevertheless, the delivery order of messages produced on different servers can differ between the servers since each pair of them have a separate TCP/IP connection for communication. However, as described in Chapter 5 this complies with the JMS specification which only requires the FIFO order (ensured by the TCP/IP protocol). But for JMSGroups stronger order requirements are needed, i.e., total order. To

provide total message order for clustered JORAM topics, we integrated Fortika ABcast protocol stack into the JORAM architecture. The integration is described in the next section.

6.2.3 Integrating Fortika ABcast stack into JORAM

As mentioned above the JORAM JMS server is based on the ScalAgent messaging middleware, where the processing of messages is done by entities called agents, each agent being responsible for processing certain types of messages. In JORAM the messages coming from and going to the network are processed by `Network` agents. Depending on the communication properties chosen by the system user (e.g., TCP/IP communication channels, causal message ordering, etc.), the server is initialized with the `Network` agent type providing these communication properties.

For the total order communication protocol between the JORAM servers we implemented an `Abcast-Network` agent, which wraps the Fortika ABcast protocol stack in the `Network` agent interface. It takes the same input as the `Network` agent and returns the same output. Internally agent notifications are wrapped to/from ABcast messages.

The following two paragraphs go more into details describing the integration of the Fortika ABcast protocol stack into the JORAM JMS server. The first paragraph shortly gives a more detailed overview of the actual JORAM `Network` agent structure, and the second informally describes the changes made for the integration.

JORAM AgentServer structure

A `Network` agent in JORAM has three components running in separate threads: `NetServerOut`, `Net-ServerIn` and `WatchDog` (see Figure 6.4).

`NetServerOut` is responsible for sending the messages to other AgentServers through the physical network. It waits on the `Network` message queue and when there is a message, it checks for which server it is, opens a socket connection to that server and sends the message. `NetServerOut` then waits for the acknowledgment and if everything went well, commits the action. The messages to the `Network` queue are put by the Engine when it receives the notification of the type `TopicForwardNot` from the `Topic` agent.

`NetServerIn` thread is waiting on the network socket and if the connection is established to this socket, reads the message, passes it to the destination agent for processing and sends an acknowledgment.

`WatchDog` is responsible for the messages whose destination server is not available at the time when the `NetServerOut` thread sends them. Those messages are put in the list of delayed messages and the `WatchDog` thread periodically tries to reach the server.

Different `Network` implementations can be chosen at the `AgentServer` initialization time to provide, for instance, causal ordering of the messages or no ordering at all. By default JORAM uses the `SimpleNetwork` implementation, which provides causal message order.

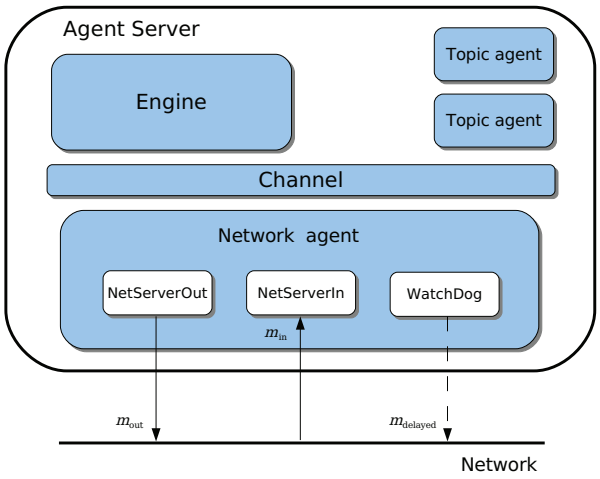


Figure 6.4: Network agent structure.

AbcastNetwork

To integrate the ABcast stack into a JORAM server, a `SimpleNetwork` component was replaced with the one using the Fortika ABcast for the inter server communication instead of a simple TCP/IP protocol. This component is called `AbcastNetwork`. The architecture of `AbcastNetwork` is similar to the one of the `Network` agent (at least `NetServerOut` and `NetServerIn` threads remain), but the functionality of these threads is changed. `NetServerOut` uses the ABcast stack instead of a simple socket to send the messages to the other servers. `NetServerIn` accordingly handles message A-delivery. In Fortika ABcast stack A-delivery is done using a callback, which was not designed to perform complicated actions. We solved the problem by adding a message queue for the delivered messages and connected `NetServerIn` to it as a consumer. The ABcast callback puts the messages into the queue and the `NetServerIn` thread reads the messages from there. The `WatchDog` thread is no more needed. If some server is not available for some time, the ABcast stack keeps the messages in its buffers.

6.3 Component Chain based implementation

The Component Chain based implementation (CCB) does not provide the full JMS API, but only a part of it, which is needed to implement the JMSGroups specification (given in Chapter 4). This section shortly presents the architecture of the framework on which the implementation is based and explains how the JMS interface and server replication is provided.

6.3.1 Framework architecture

We developed a small component based framework which is equivalent to the ScalAgent framework. Our framework architecture consists of the three main parts: *components*, *component chain* and *container*.

Components. As already mentioned, in our framework the main building block is a `Component` object. A component receives the incoming messages, processes them and optionally produces the output messages. A single component usually implements a limited set of functions. More complicated service is implemented by using various types of components connected together. Components, similarly to the ScalAgent framework, communicate with each other using asynchronous messaging, but the way the components communicate in our framework is different from ScalAgent (we describe these differences in the next paragraph).

Figure 6.5 shows the internals of the basic `Component` type object. A single component can be seen as a box with `in` and `out` interfaces for the messages. The processing inside the component depends on the input message type. Clearly different types of components process messages differently. Inside the component there are two message queues, "`input`" and "`output`", one for each interface. For example, the messages received on `in` interface are put to the "`input`" queue. These queues are blocking, i.e., if the queue is empty, taking a message from the queue is blocking. Putting a message to the queue is non-blocking and returns immediately. The messages in the queues preserve FIFO order. Optionally the queues can be automatically saved (persisted) to the stable storage.

Together with the message queues, each component has two threads, one for each queue. The "`in`" thread takes the messages one by one from the "`input`" queue and processes them using the component's `process(m)` method. This method is not implemented in the basic class: different component types provide their own implementation for it. The processing can optionally produce output messages. If this is the case, these messages are put in the "`output`" queue. The "`out`" thread reads the messages from the "`output`" queue and sends them to the component's `out` interface.

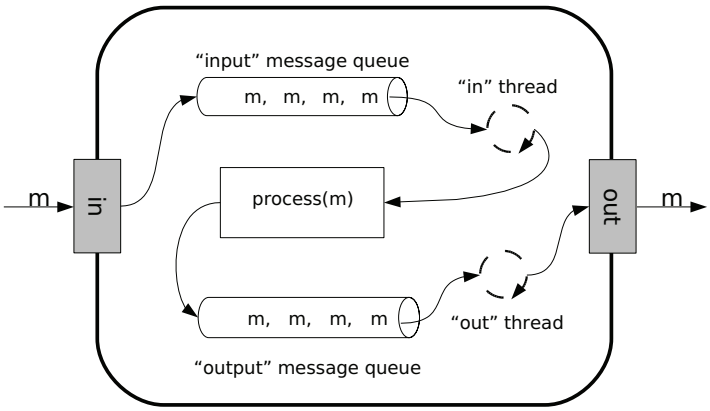


Figure 6.5: The structure of a component.

Component chain. The key feature of our framework is the way the communication is done between the components, which we call a "component chain" (or just a "chain"). The "component chain" is constructed by connecting the component's `out` interface to the other component's `in` interface. Then, if the first component produces a message, it will be automatically forwarded to the connected component. Moreover, component's `out` interface can be connected to several components. In that case, the message is sent to all of them. An example of "component chain" is shown in Figure 6.6.

The "chain link" between two components is established using component's `connect` method. It connects the calling component's `out` interface to the specified component's `in` interface. The method has an option to define the type of the messages to be forwarded. This enables to filter the messages that are forwarded through the "chain link" to the connected component. If the option is "ALL_TYPE", the filtering is not done and all messages are forwarded. The filtering options are valid on a per link basis.

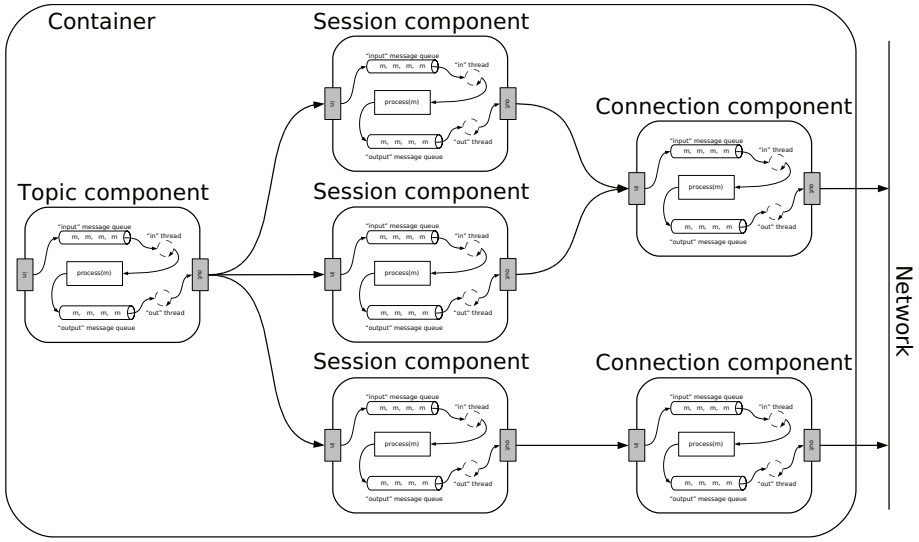


Figure 6.6: Component chain.

The way the communication between the components is implemented is the main difference between our framework and ScalAgent. In the ScalAgent architecture the messages are sent to the central channel and each message contains the ID of the destination agent. The channel finds an agent with the specified destination ID and forwards the message to it. In our framework sending the messages directly to the component using its ID is also possible, but the main communication is done using the "component chain". The drawback of the "chain" is that it needs to be established during the initialization phase, i.e., no communication is done before the "chain" is ready. The advantage is that using the "chain", the source component requires no knowledge of the component to whom it has to send messages (this knowledge is encoded in the "chain" structure). The "chain" also provides easier means to send messages to several components at a time and it makes the developing process more structured and less error-prone.

Container. The third entity in our framework is `Container`. Its role is to initialize and manage the components, so it keeps the references to all the components it contains. The `Container` has a mechanism to send a message to the `input` interface of any of its components using the component ID. But it does not have the control of the message flow between the components once the "chain" is constructed.

6.3.2 JMS wrapping

When implementing JMSGroups we used the *object adapter* design pattern [23] to provide the JMS interface for our framework. The object adapter pattern provides an interface wrapper around the original object. The adapter implements the required interface and wraps it around the object; it also has a reference to the wrapped object. In this way the adapter method calls can be easily translated to the wrapped object calls. In Component Chain based (CBB) implementation the adapters wrap the JMS interface around the corresponding component type.

Two containers with the corresponding component sets were derived from the framework: *server* side and *client* side. Both sets comply with the JMSGroups specification given in Chapter 4. Naturally the JMS wrapping was provided only to the client side components, as the server side interface is not directly accessed by the user.

In CCB the main server and client side components are: `ServerConnection` / `ClientConnection`, `ServerSession` / `ClientSession` and `ServerTopic`. The components `ServerConnection` and `ClientConnection` provide the implementation for the JMS connection interface. These component are responsible for the network communication between the CCB server and clients. The `ServerSession` and `ClientSession` represent a JMS session: they are responsible for managing message producers and consumers for the topics. JMS message acknowledgment is also implemented by these components. JMS topics are implemented by the `ServerTopic` components. These components manage the list of subscribers (in the case of `GroupTopics` - group membership) and are also responsible for message persistence.

6.3.3 CCB server replication

As described in Chapter 5 the JMSGroups server has to be replicated and strong message delivery order (total order) between the server replicas must be provided. As for the JORAM based implementation, we again used the Fortika ABcast protocol stack for the CCB server replication.

The Fortika protocol stack integration into our component architecture was similar to the one done for JORAM architecture. Special components (`ABcastNetwork`) were developed and integrated into the CCB server. These components communicate using the ABcast protocol stack instead of usual TCP/IP. The `ABcastNetwork` components are only used when sending the messages between the server replicas.

6.4 Performance

For performance measurements we used a replicated server providing the JMSGroups service and a process group. Group members were distributed equally between the server replicas. We used one client (not a member) as a message producer to ABcast messages to the group. The members were just message consumers and did not produce messages. The producer was publishing messages at a full speed. Each message contained a single string of text. The experiments were run for different client group sizes (from 1 to 30) and for different number of server replicas (3 and 5).

Table 6.1 visualizes the set of implementations we used to measure the performance. For each option we used two performance benchmarks: *ABcast early latency*[1] *vs. group size*, and *ABcast average throughput*.

Performance measurements were done on a local area network. We used one machine per server replica, group member, or client. The machines used for the tests were Sun Microsystems Sun Ultra 5/10 workstations, with system clock frequency 110 MHz, each equipped with 256MB of RAM and one Sun UltraSPARC-IIi CPU running at 440MHz. Machines were connected with a 100BaseT local area network. They were running SunOS v5.8 and Java 2 Standard Edition (build 1.5.0-b64).

	Replicated Context	**Non-Replicated Context**
JORAM	✓	✓
CCB	✓	✓

Table 6.1: Tested implementations.

Figures 6.7 and 6.8 show early latency and average throughput benchmarks for the JORAM and the CCB *replicated context* implementations (the first column in Table 6.1). Note that with the replicated context option the servers do not use stable storage. For both server sizes (3 and 5 replicas) the CCB implementation performs better that the one based on JORAM (approx. 55% and 50% better for the early latency and approx. 60% and 55% better for the average throughput).

Figures 6.9 and 6.10 show early latency and average throughput benchmarks for the JORAM and the CCB *non-replicated* context implementations (the second column in Table 6.1). Note that with the non-replicated context option the servers must be able to recover after the crash, so they use stable storage to log their state. For the JORAM implementation the early latency stabilizes only for the larger group sizes (approx. larger than 15 members) and stays relatively very high (approx. 10 times higher than CCB). The reason for this is the acknowledgment mechanism used in the JORAM non-replicated context implementation: the published messages are acknowledged immediately when they are received on the server replica where the publisher is connected, and only then they are sent to the other server replicas. This means that the publisher is capable to publish quickly a large number of messages, but later they stay in the server buffers, which gives high latency for the messages. Differently, the CCB implementation has flow control at the publisher level. It delays the acknowledgment for the publisher

[1] "Early latency" is the latency for a message from its A-broadcast to the first A-deliver in the group.

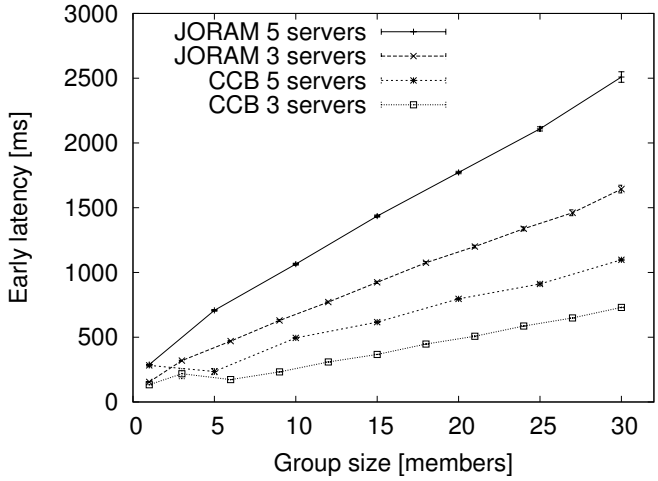


Figure 6.7: ABcast early latency vs. group size (replicated context).

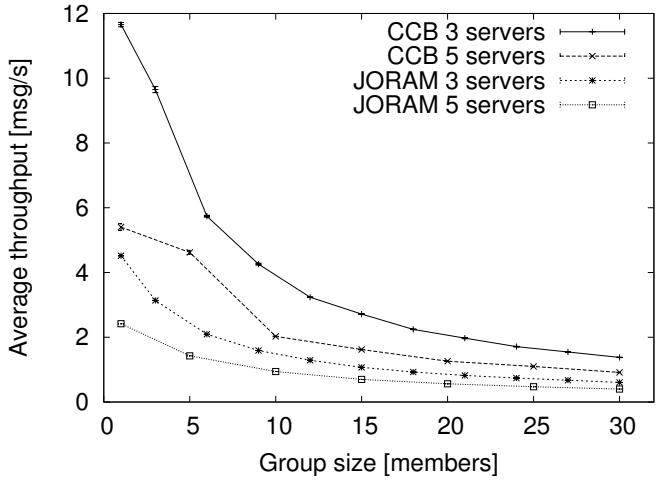


Figure 6.8: ABcast average throughput vs. group size (replicated context).

until the message reach all server replicas. This keeps the server less loaded and the messages spend less time in the buffers, which gives better latency.

On the contrary the average throughput for the JORAM non-replicated context implementation is better compared to CCB as shown by Figure 6.10 (approx. 65% for 3 server replicas and even approx. 80% better for 5 server replicas). This shows that the messages from the JORAM server buffer are dispatched much more efficiently than it is done with CCB publisher flow control.

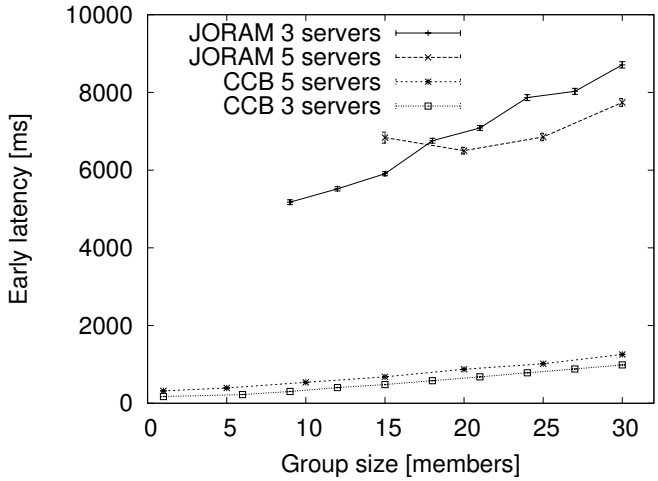


Figure 6.9: ABcast early latency vs. group size (non-replicated context).

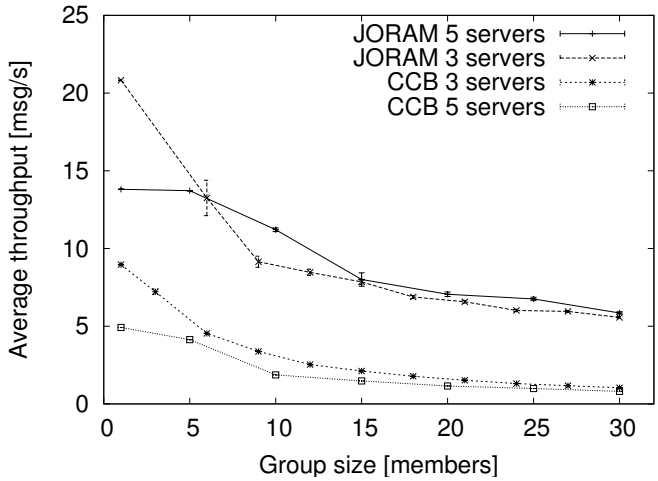


Figure 6.10: ABcast average throughput vs. group size (non-replicated context).

It is interesting to observe that for large groups the JORAM server shows better performance for 5 replicas than for 3 replicas, see Figures 6.9 and 6.10. The reason for that is that with larger groups the server with smaller replica number handles more clients (group members) per single server replica. This means more logging which is costly and slows down the server.

The given performance figures compare JORAM and CCB implementations for each replication option. The different replication options for the same implementation can be compared relating the Figures 6.7 and 6.9, and the Figures 6.8 and 6.10. For the CCB implementation the replicated context option performs slightly better than the non-replicated context one (approx. 20% for the early latency and approx. 17% for the average throughput), because the stable storage is not used to save the servers state. For the JORAM implementation the replicated context option shows better early latency, but worse throughput compared to the non-replicated context option. But because of different acknowledgment mechanisms the comparison of these two options for the JORAM implementation is not accurate.

6.5 Use case - A Replicated Table

As a use case example of JMSGroups we present a simplified version of a replicated table. A table is a popular data structure containing a set of key-value pairs. A key-value pair can be added to the table as well as removed. The values from the table are retrieved by their keys.

The table is replicated for fault tolerance using JORAM server based implementation (see Section 6.2). Active replication is used, i.e., all replicas of the table are identical, receive identical requests in the same order and perform the same operations. The replicated table is used by the clients. The logical view of the application is shown in Figure 6.11.

The replicated table implementation consists of two Java class files: `ReplTable` and `ReplTableMsgListener`. The `ReplTable` class is the implementation of the table replica: it keeps its state and contains the methods for the client to invoke. The `ReplTableMsgListener` class implement the JMS `MessageListener` interface and provides the table replica with the "reaction" to the incoming messages. The source code for these classes is given in Appendix B.1 and Appendix B.2 respectively. The third `Client` class implements the client of the replicated table; its source code is given in Appendix B.3.

To the clients, the table provides four methods: `put`, `get`, `containsKey` and `remove`. The methods are similar to those of the `java.util.Hashtable`. We use this class as a base for your replicated table. The interface of the methods is the following:

1. `Object put(Object key, Object value)` - adds a key-value pair to the table.
2. `Object get(Object key)` - returns the value from the table associated with a given key. If the table does not contain the given key, `null` is returned.
3. `boolean containsKey(Object key)` - returns `true` if the key is associated with a value in the table, `false` otherwise.
4. `void remove(Object key)` - removes the key and its corresponding value from the table. If the key does not exist, the method does nothing.

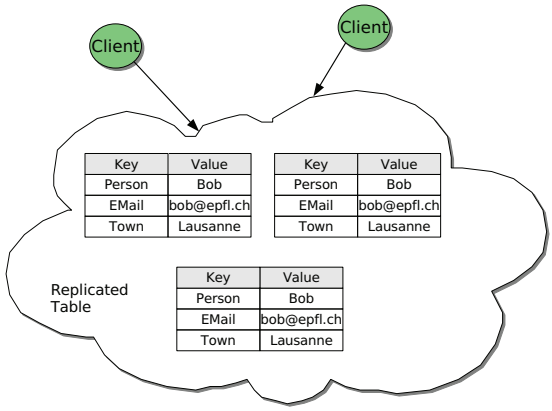


Figure 6.11: Replicated table.

Clients invoke the method on the replicated table by sending the predefined JMS message to the replica group. The structure of the message for each method invocation and its result are defined. To form a group each table replica subscribes to the group topic on a replicated JMSGroups server. Messages published to that topic are broadcast to the group members. The answers are not returned through the group topic, but rather through the client's private destination, which is specified in the request messages (using JMS temporary queues). Table replicas also handle the following group events:

1. Joining of the new members, which includes the state transfer.
2. Suspicion and removal of the crashed replicas.

The replicated table is functional if at least one replica is correct. The sample architecture of the system is shown in Figure 6.12. Solid lines show the requests and messages related to the group communication, dashed lines show the replies to the client.

6.5.1 Table group

As mentioned above, JMSGroups are based on the JMS publish-subscribe paradigm, i.e., a group is represented by a topic. But in standard JMS, topics do not provide group information, such as group view, for the subscribers. Therefore JMSGroups has a special topic called `GroupTopic`, which has the same interface as a standard JMS topic and provides the subscribers with the group information. `GroupTopic` is a server side object and is created by the JMSGroups administrator upon the start of the server.

The subscription to the group topic is equivalent to the subscription to a standard JMS topic. The main difference is that the user subscribed to the group topic receives the view of the group, i.e., the IDs of the members already in the group. A view is delivered as a JMS message to all subscribers of the group topic. The message is of the type `ObjectMessage`; its structure is given in Table 6.2. It has two `String` type properties. One has a name *"JMS_view"*, it contains an empty string and is used to

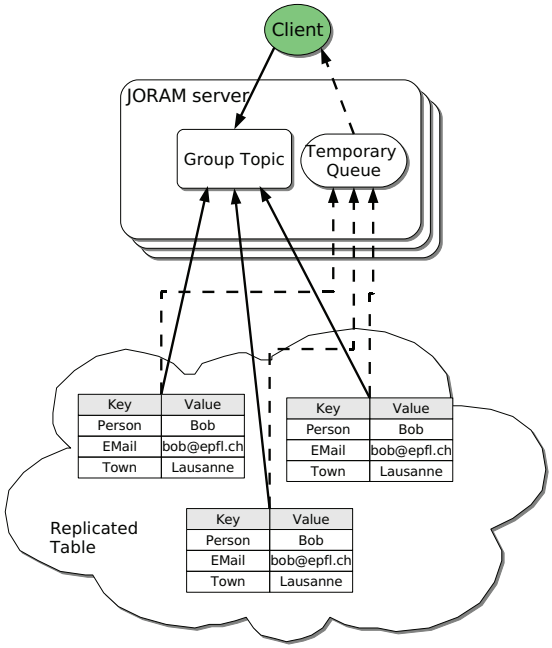


Figure 6.12: Replicated table architecture.

tag the view messages. The second property name depends on the event which triggered the sending of the view. If the view was broadcast to the group because some new member(s) have joined, the view message contains property with name *"JMS_join"*. The value of this property contains the IDs of the joined processes in the `String` format. If the view was broadcast because some member(s) have left the group, the view message contains property with name *"JMS_leave"*. The `String` value of this property contains the IDs of the processes, that have left.

The object in the view message's body contains the current view of the group; it is of the type `View`. The class `View` extends `java.util.Hashtable` and has two additional fields: (1) `int ID` - view sequence number and (2) `String procID` - the member ID of the process. Because the member IDs are assigned on the server, initially a process does not know its ID; it finds it out upon the first view delivery. The table of the view stores the IDs of the members of the group. The IDs correspond to the keys in the table; the value fields are empty strings (see Table 6.3).

6.5.2 Member suspicions and removal

In JMSGroups the crashed group members are not removed from the group automatically: only a suspicion is returned to the other members. Once a member is suspected, all correct group members get the suspicion message. Then it is up to the group members to exclude the suspected member from the group. In our case the server acts as a failure detector and sends the suspicion message to the correct

Property Name	Property Type	Explanation
"JMS_view"	String	*Contains an empty string*
"JMS_join"	String	*This property is present if some process(es) have joined the group. Contains the ID(s) of the process(es), that have joined.*
"JMS_leave"	String	*This property is present if some process(es) have left the group. Contains the ID(s) of the process(es), that have left.*
Message Body	**Type**	**Explanation**
View	Object	*Contains the group view.*

Table 6.2: The structure of a view message.

Key	Value
#0.0.1032-c1sub1	""
#0.0.1032-c2sub1	""
#0.0.1029-c1sub1	""

Table 6.3: View table example.

group members. The suspicion message is of the type `javax.jms.Message`. It contains a property of type `String` with name *"JMS_suspect"*: its value contains the ID(s) of the suspected members.[2] The structure of the suspicion message is shown in Table 6.4.

Property Name	Property Type	Explanation
"JMS_suspect"	String	*Contains the ID(s) of the suspected group member(s).*

Table 6.4: Suspicion message structure.

Group members can exclude other members from the group, e.g., members can decide to exclude the suspected member. Member exclusion from the group is done by publishing an exclude message to the group topic. The message must contain the property of type `String` with name *"JMS_remove"*, which value contains the ID(s) of the members to be excluded. The members of the group get a new view after the exclusion. The structure of exclude message is shown in Table 6.5.

[2]Because of the JORAM architecture, two identical suspect messages are sent upon a suspicion.

Property Name	Property Type	Explanation
"JMS_remove"	String	*Contains the suspected group member(s) ID(s) to be excluded.*

Table 6.5: Exclude message structure.

6.6 Summary

This chapter presented two centralized server based implementations of JMSGroups: the JORAM server based and the Component Chain based. Preliminary performance measurements were presented; more detailed performance study will be presented in the final version of the thesis. Also a use case example of a replicated table implemented using JMSGroups was presented.

Chapter 7

Replicated Invocation

In today's systems, applications are composed from various components that can be collocated but may also be located on different machines (e.g., in J2EE [71], CORBA [56]). These components collaborate in order to service a client request. More specifically, a client request executed in one component may trigger a request to another component. While acting as a server component[1] to the client, the component at the same time assumes the role of a client, by invoking a service on another server.

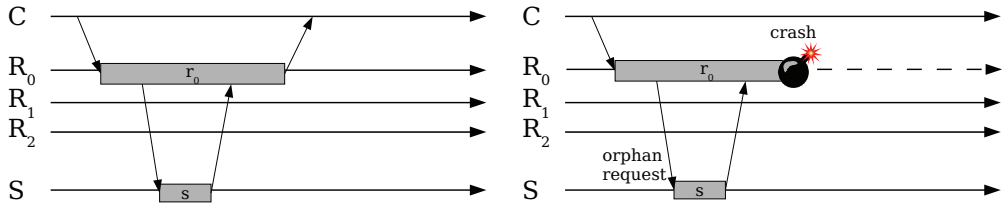
To ensure that applications work even in the face of failures, replication is generally used within the components. While the problem of replicating a server and invoking a replicated server has been thoroughly studied [10, 28, 61], the problem of a replicated server invoking another (replicated) server has not been addressed in a satisfactory manner. We call the latter invocation a *replicated invocation*. Replicated invocation in the context of deterministic servers causes a problem of *duplicate requests*. This problem is addressed in [45], where proxies are presented to filter the requests using their ID numbers. However, the proxy solution assumes deterministic replicas. Hence, it is not applicable for non-deterministic servers, because the requests sent by the replicas of non-deterministic servers may not be identical [60]. In the context of non-deterministic servers, replicated invocation causes a different problem: the problem of *orphan requests*.

The problem of orphan requests described in this chapter was first mentioned in [59] in the context of transactional agents. There the specification and the solution of the problem were presented in terms of invocations between the clients and the servers. Here we present a new specification of the problem and a new protocol to solve it, both in terms of transactions.

Informally, an orphan request occurs if a server processes a request from another server, but this request is not valid any more. Consider, for instance, a system where client C invokes replica R_i of replicated server R (see Figure 7.1(a), in our case $i = 0$). To process $C's$ request, R_i invokes another server S, i.e., R_i itself acts as a client to server S. We denote by r_i (resp. s) the processing on R_i (resp. S). We say that s is a subinvocation or *nested invocation* of r_i. If no failures occur the servers update their states and R_i sends the reply to the client. However, a component may be subject to a failure. If R_i fails before sending the reply to C (see Figure 7.1(b)), C will eventually notice the failure, but not S (since S

[1] In the following, a server component is called a *server*.

already finished the processing). The state of S will reflect invocation r_i, which has not finished properly. Hence, the state of S is inconsistent. In this case we call s an orphan request. If at this point some other client accesses S, there is a danger that the inconsistent state of S will propagate in the system. Note that the failure of S causes a different problem. Indeed, it does not result in an orphan request; rather, the state of S is no longer available.



(a) Without failures. (b) With a failure.

Figure 7.1: Nested request invocations (the processing of the request is shown by gray bars).

As mentioned above, and as described in details further in the chapter, the orphan request problem can occur when the invoked replicated server is non-deterministic. The work in [36, 54] provides mechanisms to enforce deterministic execution. In contrast, our approach supports non-determinism and at the same time prevents orphan requests. It is based on the idea of exchanging sufficient undo information prior to the server invocation to allow other client replicas to undo the requests of failed client replicas. In contrast to [21], we do not limit our approach to three-tier architectures (consisting of presentation, application logic, and data tier) [39] and stateless clients. Rather, we assume that replicas R_j do maintain their own state. Moreover, we show that our approach allows us to prevent blocking when the server uses locking to ensure concurrency control [27]. Indeed, a failure of R_i in such a scenario may prevent the termination of the processing on server S, and thus no other client can access the locked data items.

JMSGroups context. In the JMSGroups central server architecture presented in Chapter 5 the server is replicated in order to be fault tolerant. In providing a group communication service it acts as a middleware for its clients, either groups of clients implementing replicated services or single clients accessing these services (see Figure 5.4). In other words, the JMSGroups server replication level in Figure 5.4 corresponds to replicated server R in Figure 7.1, the services implemented by JMSGroups correspond to server S and the client of JMSGroups services corresponds to client C. As such, we have the architecture where the client invokes a replicated server, which to process the client's request invokes another service. As described above, this architecture is prone to the orphan request problem.

In both our implementations presented in Chapter 6 the replicated servers providing JMSGroups service are deterministic and are not subject to the orphan request problem. But non-deterministic JMSGroups server is also possible. For example, if JMS message expiration time is used for messages, the messages are held on the server until they expire. In such case server replica state is dependent on time, which can lead to non-determinism. For the non-deterministic JMSGroups server the orphan request

problem should be taken into account.

Chapter Roadmap. We first introduce replicated invocation in Section 7.1. In Section 7.2 we specify the problem of replicated invocations in terms of transactions. Using the transaction model, an orphan request becomes an orphan subtransaction (Section 7.3). The core contribution of the chapter is presented in Section 7.4. In this section, we present an orphan-subtransaction-free replicated invocation protocol in the context of non-deterministic execution. Finally, we relate our solution to the existing work in Section 7.5 and summarize the chapter in Section 7.6.

7.1 Replicated Invocation

We assume an asynchronous system which has no bounds on communication delays nor relative processing speeds. Processes can crash and do not recover.[2] In such a system, accurate failure detection is impossible and consensus cannot be solved [20]. However this issue is orthogonal to the problem addressed in the chapter. Processes communicate via quasi-reliable channels: if processes p and q are correct (i.e., do not crash) and p send message m to q, then q eventually receives m. Note that quasi-reliable channels provide a more accurate model for TCP connections than reliable communication channels, which do not require p to be correct.

Replication is a widely used technique to address failures of a server. Instead of relying only on a single server, the service is provided by multiple server replicas. If a failure of one server replica occurs, another replica takes over and services the clients' requests. As a consequence, the service is available to the clients despite of a failure. We say that a client invokes a replicated server or rather a service on it. If the client itself is replicated, we speak of a *replicated invocation*.

Definition 1 (Replicated Invocation)**.** *A replicated invocation occurs if a replicated client invokes a server (not necessarily replicated).*

In Fig. 7.2, the invocation from replicated server R to server S is a replicated invocation. If S is replicated, we do not make any assumptions about the replication strategy that can be used by S (e.g., passive [28], active [70], semi-passive [14], or semi-active [61]), as the replication strategy of S is not relevant for our contribution. Indeed, although S may be replicated, its replication strategy makes it behave like a non-replicated server from the point of view of R. For simplicity we represent server S as a single, non-replicated server, which is sufficient to illustrate the problem addressed in the chapter and our solution. However, in real systems S would be replicated in order to prevent the existence of a single point of failure.

The replicated invocation problem can be addressed in the context of a deterministic or a non-deterministic server, R. Non-determinism can occur with respect to communication and computation. The former is caused by a different order of message arrivals at the replicas. The latter, i.e., non-determinism related to computation, occurs if the replica, for instance, is multithreaded or uses asyn-

[2] Actually, crashed processes can recover with a different process ID. From the application's perspective this corresponds to a new process.

chronous system calls (e.g., interrupts). A *deterministic server* (or *non-deterministic server*) is deterministic (non-deterministic) with respect to its computation, but not necessarily with respect to communication. In other words, the order in which client requests arrive at a deterministic server is arbitrary.

7.1.1 Deterministic servers

Server replicas are said to be deterministic if, given the same initial state and the same request, all transit to the same state and return the same reply. We show that orphan requests do not occur with replicated deterministic servers.

In the case of deterministic servers *active replication* [70] can be used. In active replication clients multicast (using total order multicast) the request to all server replicas, which process the requests in parallel (see Fig. 7.2, ①). If this processing requires the invocation of another server, each replica issues exactly the same invocation ②. Because these invocations are identical, duplicate invocations can easily be detected and filtered, in order not to process them multiple times ③. This is done by having the replicas R_i assign IDs to their invocation.[3] Duplicate invocation filtering is addressed for instance in [45, 54]. The result of the processing on S is valid for every replica R_i and is multicast to them ④. Also, each replica R_i sends the reply back to the client C ⑤. Generally, the client accepts the first one and discards the others.

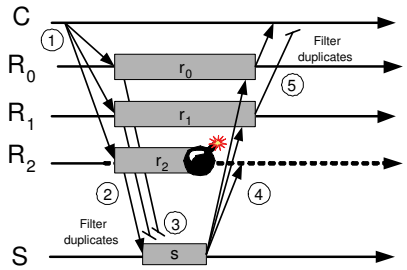


Figure 7.2: Deterministic server R is replicated using active replication.

As long as there is at least one *correct* (not failed) replica R_i, the orphan request problem does not occur with replicated deterministic server R.[4] The reason is that all replicas share the same request s (see Fig. 7.2) and thus the failure of one or multiple R_i does not leave s as an orphan.

7.1.2 Non-Deterministic servers

Non-deterministic execution of R_i prevents the use of active replication; rather, it requires another replication strategy, such as passive (also called *primary-backup* [10]), semi-passive, or semi-active replica-

[3] One could argue that the client can assign a unique ID to its invocation, which can be reused for the nested invocations as well. Unfortunately, this does not always work. Indeed, assume that processing on R leads to a number of invocations to S that is not known a priori. Hence, the request ID must be assigned by R.

[4] Usually, replication techniques in the asynchronous system model assume that a majority of replicas do not fail [28].

tion. Without loss of generality, we discuss the use of passive replication for the server R. However, our approach is also valid in case the server R is semi-passively or semi-actively replicated. In passive replication only one replica, the *primary*, executes C's request. The update is then sent to the backup replicas. The backup replicas do not directly communicate with C; rather, they only communicate with the primary. As only the primary executes the request, passive replication supports non-deterministic execution. However, a passively replicated server needs to handle failures of the primary. If the primary fails or is erroneously suspected, one of the backups takes over the role of the primary (Fig. 7.3). The client C eventually times out, has to learn the identity of the new primary (e.g., R_1), and reissues the request.

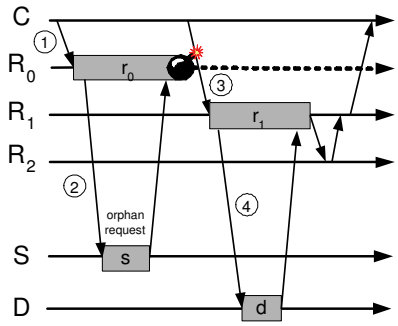


Figure 7.3: Non-deterministic server R is replicated using passive replication (with a failure of the primary).

Consider nested invocation in the context of the passively replicated non-deterministic server R (see Fig. 7.3). To service client C's request ①, the primary replica R_0 invokes server S ②, but fails before updating the backups. A new primary, say R_1, is elected and the client C resends its request ③. As the replicas of R are non-deterministic, R_1 might issue a different invocation to server S to serve the same request from C. It might even choose a different server D ④, or it might not issue the invocation at all. The result computed for r_0 thus cannot be reused for r_1, and d must be processed separately. This leaves s as an orphan request, which has a pending effect on the state of S. A new invocation of S at this point, would likely lead to an inconsistent reply. So the problem of the orphan request s needs to be addressed. In the rest of the chapter we focus on replicated invocation in the context of non-deterministic replicated servers. In the next section, we introduce the specification and notation we use to model this problem.

7.2 Specification with Non-Deterministic Servers

In this section, we give a specification of replicated invocation in terms of transactions. The problem of orphan requests is caused by a partial execution of C's request. As the transaction model addresses the issue of atomicity of a set of operations it is useful to model replicated invocation. Informally, a transaction always terminates by either committing its modifications, or aborting them.

Transactions (recursively) decomposed into subtransactions are called *nested transactions* [53]. Every subtransaction forms a logically related subtask. A successful subtransaction becomes permanent, i.e., commits, if all its parent transactions (the transaction that encompasses this subtransaction) commit as well. In contrast, a parent transaction can commit (provided its parent transaction commits) although some of its subtransactions may have aborted. A subtransaction is ready to commit, if it has successfully executed and is waiting for the commit or abort decision of its parent transaction. A ready-to-commit transaction t, denoted $ReadyToCommit_t$, can no longer spontaneously abort (i.e., itself decide abort), but only aborts if its parent transaction aborts. Finally, $\rightarrow$ denotes the precedence operator as specified in [37]. More specifically, if $t_1 \rightarrow t_2$, t_1 is executed before t_2. In other words, any operation of t_1 that conflicts with an operation of t_2 is executed before that operation of t_2.

We first specify the invocation between the client C and the server R (i.e., the traditional passive replication approach) in terms of transactions (Section 7.2.1), and then extend this specification to also encompass the invocation between server R and server S (Section 7.2.2).

7.2.1 Invocation $C \longleftrightarrow R$

We model the execution of C's request on server R as follows. Upon reception of C's request, the primary replica R_0 starts transaction t_0 (see Fig. 7.4). This transaction contains subtransactions pr_0 (pr stands for *processing*) and up_0 (up stands for *update*). For the moment, we ignore the invocation to server S in Fig. 7.4. Subtransaction pr_0 executes the client request on the primary, subtransaction up_0's task is to update the backup replicas of R, i.e., R_1 and R_2.

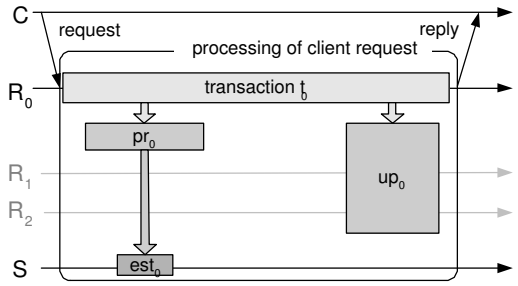


Figure 7.4: Representation in terms of (sub)transactions (invocations between C, R, and S).

The specification is stated in terms of properties of transactions. We mention only those that are related to the replicated invocation, and omit basic transaction properties. The full set of properties for nested transactions can be found in [13]. The invocation $C \longleftrightarrow R$ can be specified as follows (the subscript *prim* refers to the primary replica, e.g., R_0 in Fig. 7.4):

1. (*Abort Atomicity*) If transaction t_{prim} aborts, all of its subtransactions (i.e., pr_{prim} and up_{prim}) must abort.
   $Abort_{t_{prim}} \Rightarrow (Abort_{pr_{prim}} \wedge Abort_{up_{prim}})$

2. (*Cruciality*) If one of subtransactions pr_{prim} or up_{prim} aborts, t_{prim} also aborts: pr_{prim} and up_{prim} are crucial for t_{prim}. Transaction t_{prim} doesn't make sense without either of these transactions.
   $Abort_{pr_{prim}} \vee Abort_{up_{prim}} \Rightarrow Abort_{t_{prim}}$

3. *(Sequence)* Transaction pr_{prim} always executes before up_{prim}. This is because up_{prim} updates the backups with the results of the execution of pr_{prim}.
   $pr_{prim} \rightarrow up_{prim}$

4. *(Termination)* If R_{prim} executes t_{prim}, then all correct replicas of R eventually know the outcome (i.e. commit or abort) of t_{prim}. This is modeled by either an abort or a commit of transaction up_{prim}.
   $ReadyToCommit_{t_{prim}} \Rightarrow \Diamond(Abort_{up_{prim}} \vee Commit_{up_{prim}})$

5. *(Non-triviality)* If both subtransactions pr_{prim} and up_{prim} have successfully executed (i.e., are ready to be committed), eventually transaction t_{prim} is committed.
   $ReadyToCommit_{pr_{prim}} \wedge ReadyToCommit_{up_{prim}} \Rightarrow Commit_{t_{prim}}$

Property 1 is a standard nested transaction property. The success of subtransactions pr_{prim} and up_{prim} is crucial for the success of t_{prim}: in other words, t_{prim} can only commit if the processing transaction pr_{prim} and the update transaction up_{prim} of the backup replicas have succeeded (Property 2).

The sequence property (Property 3) is inherited from passive replication: first the client request is processed on the primary, then the backups are updated with the result obtained from the processing. Note that this property, together with Property 2, specifies a particular case of nested transactions, namely a distributed flat transaction [27]. We use the nested transaction model because it is needed when we extend our specification to the invocation between R and S in Section 7.2.2.

The termination property (Property 4) ensures that once transaction t_{prim} is ready to commit, its subtransaction up_{prim} eventually terminates by either commit or abort. Although the primary may fail, transaction up_{prim} eventually must be terminated. As a consequence, the other replicas need to somehow learn of the failure of up_{prim}. Property 4 is thus also a liveness property, which ensures that the outcome of transaction t_{prim} is eventually decided and that all subtransactions executing on correct processes eventually terminate. This property is essential in preventing orphan requests or subtransactions.

Finally, the non-triviality property (Property 5) specifies, that if both subtransactions pr_{prim} and up_{prim} are ready to be committed, then the outcome of t_{prim} is commit. Note that we do not require that t_{prim} be committed by R_{prim} (where t_{prim} executes), as R_{prim} may have failed. Moreover, the specification still allows R_{prim} to always immediately abort pr_{prim} despite this property.

In our system model, we assume that crashed processes do not recover[5] (see Section 7.1). Consequently, the failure of a replica R_{prim} erases all traces of the transaction on R, unless the other replicas have been updated.

[5]This is the standard assumption that forces a protocol to be non-blocking. In other words the protocol presented later is non-blocking.

7.2.2 Invocation $R \longleftrightarrow S$

In the previous section, we have specified the invocation between C and R. In this section, we extend this specification to the cases where the primary R_0 invokes transaction est_0 (*external server transaction*) on another server S (see Fig. 7.4). The invocation between R and S corresponds to the replicated invocation presented in Section 7.1.2. Transaction est_0 is a subtransaction of transaction pr_0. Recall that server S is represented as a single, non-replicated server. For our discussion, it is not relevant whether S is replicated or not.

Compared to the model of the invocation $C \longleftrightarrow R$, the invocation of S adds another level of nesting. Indeed, the subtransaction pr now contains subtransaction est. While subtransactions pr and up are crucial for the successful outcome of t, subtransaction est may not be. In other words, in some applications pr can commit although est aborts.

Replicated invocation between R and S can thus be specified by the properties mentioned in Section 7.2.1 and the following two additional properties:

6. (*Remote Abort Atomicity*) If subtransaction pr_i is aborted, est_i is aborted as well.
   $Abort_{pr_i} \Rightarrow Abort_{est_i}$

7. (*Remote Commit Atomicity*) If subtransaction est_i has successfully executed and is ready to be committed, and its parent transaction pr_i commits, then est_i is also committed.
   $ReadyToCommit_{est_i} \wedge Commit_{pr_i} \Rightarrow Commit_{est_i}$

Properties 6 and 7 ensure that subtransaction est_i eventually is terminated, i.e., either commits or aborts. Clearly, we assume here that server S is available, which is the case if S is fault-tolerant (i.e., if S is itself replicated). Note that properties 6, 1 and 2 specify that if up aborts then est must also be aborted.

7.3 The Problem of Orphan Subtransactions with Replicated Invocation

According to Properties 1 to 7 the outcome of the entire execution (i.e., commit or abort) is decided by the top-level transaction, and then this decision is propagated to the subtransactions, which in turn propagates to their subtransactions. However, the failure of a replica R_i may interrupt the mechanism that notifies the subtransactions of the commit or abort decision. In this case, est_0 is unaware of the outcome of t_0 and thus cannot terminate (see Fig. 7.4): est_0 is called an *orphan subtransaction*. Clearly, orphan subtransactions are undesirable, because they maintain locks on data items and prevent other transactions from accessing these items. Note that subtransaction est_0 cannot spontaneously abort, because its parent transaction decides the final outcome.

Depending on the processing of server S (optimistic or pessimistic) orphan subtransactions cause different problems.

7.3.1 Pessimistic *vs.* optimistic server S

To ensure transaction atomicity, data items are locked. When a transaction starts, the needed locks are acquired; when the transaction finishes, the locks are released, and the result of the processing becomes visible in the system. If the locks are not available for some transaction t, the processing blocks until the locks are released by the transaction holding the locks. A *subtransaction* holding the locks has two options upon finishing its processing: (1) it can release the locks immediately (i.e., temporary commit), or (2) it can wait for the commit/abort decision from a higher entity (i.e., parent transaction), keeping the locks on the data. The latter option, i.e., option (2), is called *pessimistic processing*, and the server is called a *pessimistic server*. Option (1) is called *optimistic processing*, and the server is called an *optimistic server*.

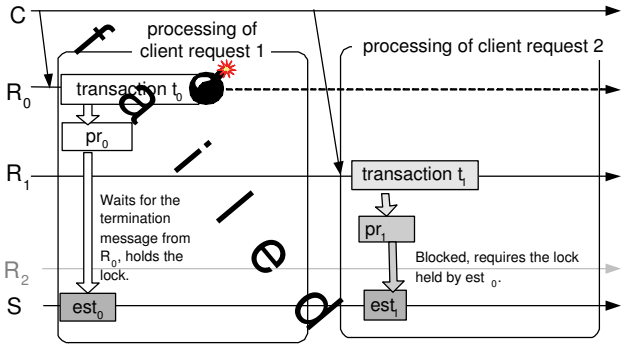


Figure 7.5: An orphan subtransaction est_0 on pessimistic server S.

Blocking with Pessimistic Server S. Consider the case of an orphan subtransaction (Fig. 7.5) with a pessimistic server S. Assume that after the crash of the primary R_0, the new primary R_1 calls the same server S, and executes subtransaction est_1, which accesses some of the same data items accessed by est_0. In this case est_1 has to wait until est_0 releases the locks. Hence, the entire client R is blocked. Blocking of R is undesirable, as it acts itself as a server for other applications. Moreover, other clients may also block when accessing server S.

Inconsistency with Optimistic Server S. The problem with optimistic servers is different: the temporary commit might have to be undone. This can be handled by *compensating actions*: to abort a committed transaction a *compensating transaction* [24, 26] is executed on the server. A compensating transaction t^{comp} semantically undoes the modifications caused by the original transaction. Assume, for instance, that transaction t reserves a ticket on a flight, then t^{comp} simply cancels this reservation.

Using an optimistic approach, blocking is prevented. Indeed, the locks held by subtransaction est_0 (Fig. 7.6) are immediately released and the data items are again accessible by est_1 (unless another trans-

action has acquired them in the meantime). However, in this case, subtransaction est_0 needs to be compensated, since the state of server S reflects est_0, but after the crash of R_0, est_0 is not valid any more.

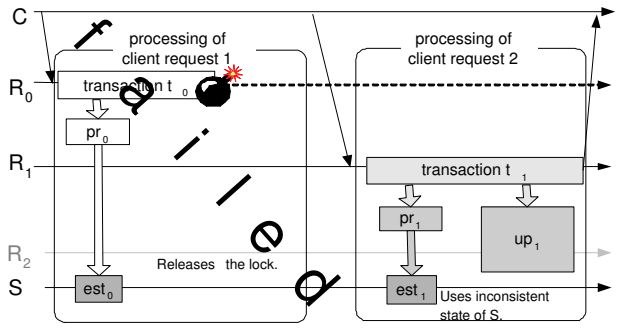


Figure 7.6: An orphan subtransaction est_0 on optimistic server S.

In the next section we present solutions for pessimistic and optimistic servers.

7.4 Replicated Invocation Protocol

In the previous section we have used transactions to model the problem of replicated invocation in the context of a passively replicated client R. In particular, we have used orphan subtransactions as a model of orphan requests. In this section, we show an approach that prevents orphan requests. For this purpose, we first present the basic idea to solve the problem of orphan requests/subtransactions in the context of replicated invocation, and then the protocol that implements this idea.

7.4.1 Basic idea: sharing undo information among replicas of R

The Problem of Finding Out About S. To prevent orphan requests in replicated invocations, it is crucial that a new primary replica R_j is able to find out the identities of the servers that have been accessed by the previous primary R_i. This allows R_j to send abort message(s) or compensating transaction(s) to S. How can S be known to R_j? The identity of S is trivially known if (1) it can be deterministically computed by the replicas of R, or (2) the set of servers is sufficiently small. Note that in case (1), the replicas still may generate different requests to S, or not send any request to S at all. In case (2), a message is sent to all servers to find out which one has been invoked by R_j. In the following, we address the more complex cases in which the identity of S cannot be deduced a posteriori. This is especially the case if:

- the identity of S is dynamically computed during the processing of R_j. In other words, the identity of S is not known to the replica prior to the processing of C's request, and it is impossible for R_j to find out the identity of S computed by another replica R_i, and
- the set of potential servers is large.

Sending Undo Information. We call *undo information*, the information that allows a particular request to be undone; it includes the name of the server S to which the request is sent, and the description of an action to perform. The solution to orphan requests consists of making the undo information available to other replicas of R before the primary invokes S. In the context of the undo information, we distinguish between *termination requests* and *compensation requests*:

(1) non terminated orphan requests (or subtransactions) on pessimistic servers need to be terminated, and

(2) terminated orphan requests (or subtransactions) on optimistic servers need to be compensated.

In case (1), *termination requests* are *COMMIT* and *ABORT* messages.[6] In case (2) compensating actions are included in the undo information in order to restore the consistent state of the system. However, note that compensating the request of a replica is not easy. For example the sequence of requests $(rq_x; rq_y; rq_x^{comp})$ must be a valid sequence and must be *semantically* equivalent to the sequence that consists only of rq_y. Note that this is not a consequence of our solution; rather, this assumption is required also in the case S is accessed by multiple different clients.

If a new primary R_j is elected as a result of an erroneous suspicion of the old primary R_i, R_i can itself send the termination or compensation request to S, if needed. However, from the perspective of the replicas R_k $(k \neq i)$ it is impossible to distinguish between an erroneous and a correct suspicion of R_i (see Section 7.1).

7.4.2 The protocol

The *Replicated Invocation Protocol* for non-deterministic execution is presented in Figures 7.7 and 7.8. The protocol consists of six procedures executed on the primary of R. When the primary gets a request from client C, Procedure 1 (Fig. 7.8) is executed. If the request was not processed previously, the primary starts processing it. This corresponds to transaction t_0 in our model (see Fig. 7.4). After executing procedures *Process_Request* (Procedure 2) and *Update_Backups* (Procedure 3) the processing on remote pessimistic servers must be committed and undo information sent to backups during the processing must be garbage collected. Procedure 1 terminates after sending the reply to the client.

Procedure 2 corresponds to transaction pr_0 in our model (see Fig. 7.4). Assume that during processing, the primary needs to send a nested request to some other server S. Before doing so, a message of type *UndoInfo* is prepared for that request and multicast to the backups (this multicast is denoted by *Uniform-VScast*).[7] The content of the undo message depends on the type of server the original request

[6] We assume that the execution of termination requests is idempotent.

[7] In the context of group communication, this multicast corresponds to what is called *uniform view synchronous broadcast* [12, 68]. Roughly speaking, uniform view synchronous broadcast ensures that if some process VSdelivers the message, then all correct processes eventually VSdeliver the message. For simplicity, we assume Sending View Delivery [12]. However, our approach can be easily extended to encompass also Same View Delivery. More information about using group communication for passive replication can be found in [28]. We do not discuss these issues here, since they are not needed to understand the contribution of the chapter.

```
New Message TYPE StandardRequest = {req, ID};
   req - (the request to be sent);
   ID - (ID, which uniquely identifies the request);

New Message TYPE UndoInfo = {comp, reqID, parentID, target};
   comp - (compensating request, used only with optimistic servers);
   reqID - (ID of the request this undo information corresponds to);
   parentID - (ID of request from client C, whose processing
              triggered the undo message);
   target - (the server, to send this undo message to, if needed);

Pessimistic(S) - (predicate that evaluates to true if S is pessimistic);
Optimistic(S) - (predicate that evaluates to true if S is optimistic);
```

Figure 7.7: Message type declaration and predicate definition.

is sent to. Upon delivery, the undo information messages are stored locally on backups R_j in the set U_j. Then, server S is invoked.

Procedure 3 multicasts the result of the request processing to the backups. It corresponds to the transaction up_0 in our model (see Fig. 7.4) and to uniform VScast traditionally used in passive replication.[8]

Procedure 4 is called when a replica becomes a primary, which occurs if the previous primary fails or is wrongly suspected to have failed. Before starting to serve the clients' requests, the new primary takes care of orphan requests.

Managing orphan requests in Procedure 5 depends on the type of server: pessimistic or optimistic. In the following two paragraphs, we describe each case separately. We assume the same system as in the Fig. 7.3, i.e. R_0 is the initial primary. If R_0 crashes, R_1 takes the role of the primary.

Procedure 6 enables garbage collection (GC) of the undo information on the backups. It multicasts (using uniform VScast, mentioned above) the set of request IDs whose undo information has become obsolete. In a practical setting, the messages related to garbage collection can be piggy backed on the messages of the next uniform VScast in Procedure 3. Upon reception of the messages related to garbage collection (not shown in Fig. 7.8) the backups discard all the corresponding undo information. If they have not yet received the undo information that corresponds to a particular request ID, they store this GC information for later use.

Pessimistic Server S. If server S executes pessimistically, a termination message is always required. Indeed, assume that primary R_0 fails after updating the backups, but before sending the result to the client. In this case, as the new primary R_1 has received the update, a *COMMIT* message is sent to S together with the ID of the request to be committed. In contrast, an *ABORT* message is sent to S by R_1, if

[8] If processes can communicate using *reliable* communication channels, then uniform VScast is not needed to send the undo information to the backups. Rather, simple point-to-point communication is sufficient.

```
1  r: StandardRequest;
2  u: UndoInfo;
3  U_prim: set of UndoInfo messages;

4  Procedure 1. Upon reception of r from C on primary R_prim
5  begin
       // a set of IDs for Garbage Collection
6      UReqIDs ← ∅;
7      if update for request ID=r.ID is available on R_prim then
8          send(reply for r) to C;
9      else
10         Process_Request(r);
11         Update_Backups(update for r);
12         foreach u : u ∈ U_prim and u.reqID = r.ID do
13             if Pessimistic(u.target) then
14                 send(COMMIT, u.reqID) to u.target;
15                 U_prim ← U_prim \ {u};
16                 UReqIDs ← UReqIDs ∪ {u.reqID};
17                 Garbage_Collection(UReqIDs);
18             end
19         end
20         send(reply for r) to C;
21     end
22 end

23 Procedure 2. Process_Request(r) on the primary R_prim
24 begin
25     ...
26     if primary needs to send nested request to S then
27         new s: StandardRequest;
28         s.req ← request to S;
29         s.ID ← assign unique ID;
30         new u: UndoInfo;
31         u.parentID ← r.ID;
32         u.target ← S;
33         if Pessimistic(S) then
34             u.comp ← NULL;
35             u.reqID ← s.ID;
36         else if Optimistic(S) then
37             u.comp ← compensating request for s;
38             u.reqID ← NULL;
39         end
40         U_prim ← U_prim ∪ {u};
41         Uniform-VScast(U_prim);
42         wait to deliver(u);
43         send(s) to S;
44         wait for reply;
45     end
46     ...
47 end

48 Procedure 3. Update_Backups(update for r)
49 begin
50     Uniform-VScast(update for r);
51     wait to deliver(update for r);
52 end

53 Procedure 4. When R_i becomes a primary
54 begin
       // set of IDs for Garbage Collection
55     UReqIDs ← ∅;
56     foreach u : u ∈ U_prim do
57         Manage_Orphan(u);
58         U_prim ← U_prim \ {u};
59         UReqIDs ← UReqIDs ∪ {u.reqID};
60     end
61     Garbage_Collection(UReqIDs);
62 end

63 Procedure 5. Manage_Orphan(u)
64 begin
65     if update available for request ID=u.parentID then
66         if Pessimistic(u.target) then
67             send(COMMIT, u.reqID) to u.target;
68         end
69     else
           // if no update is available
70         if Pessimistic(u.target) then
71             send(ABORT, u.reqID) to u.target;
72         else if Optimistic(u.target) then
73             send(u.comp) to u.target;
74         end
75     end
76 end

77 Procedure 6. Garbage_Collection(UReqIDs: set of IDs)
78 begin
79     Uniform-VScast(UReqIDs);
80     wait to deliver(UReqIDs);
81 end
```

Figure 7.8: Replicated invocation protocol.

R_0 fails before it updates the backups (see Fig. 7.9). When C resends its request, this request is executed by R_1.

Consider the particular case of R_0's failure after sending the undo information, but before sending the request to S. In the case of pessimistic server S, no special mechanisms are needed: termination messages not related to an actual request are simply ignored by S.

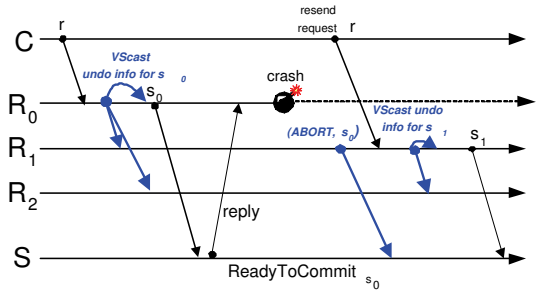


Figure 7.9: Primary R_0's failure after invoking pessimistic server S (the primary fails before backups are updated).

Optimistic Server S. To undo request s that has been sent to the optimistic server S, a compensating request is used. Consider first the case where no compensating request is required. In this case, the primary (i.e., R_0) executes C's request (which requires the sending of a request to S), updates the backups, and crashes. As the state of the backups has been updated, the new primary R_1 simply returns the result previously computed by R_0 when C resends its request.

However, if R_0 fails before updating the backups, the processing on S needs to be undone. Hence, a compensating request is sent to server S. Eventually, C resends its request to the new primary R_1, which recomputes the result. Note that the order of compensating an original request is not significant. This is a consequence of the properties of the compensating request (see Section 7.4.1).

A particular case arises if R_0 fails after having sent the undo information to the backup replicas, but before sending the request to S. As the backup replicas have received the undo information u (see Procedure 2, lines 41-42), the new primary will use this undo information to send a compensating request to S (see Procedure 5, line 73). Similarly, the undo information may arrive at S before the original request. Server S must handle this case: if the original request has not been received, then the undo information is not executed, but stored to be reused in case it eventually arrives (if it does at all). Such early undo messages are possible even if R_0 fails after sending the original request.

7.4.3 Correctness Issues

In this section, we argue about the correctness of the replicated invocation approach. Basically, we have to show that our algorithm satisfies Properties 1 - 7. As Properties 1 to 5 are already satisfied by passive

replication based on uniform VScast, the reader is referred to [69] for the proofs.

We only give an informal proof of properties 6 and 7. For this purpose, we first prove the following lemma:

Lemma 4. *If primary p VSdelivers undo information u and a new primary q takes over (because of a failure of p or an erroneous suspicion), q has also VSdelivered u.*

Proof sketch. The proof of this lemma is based on the specification of uniform VScast. Indeed, uniformally vscasting u (see Procedure 3 in Fig. 7.8) ensures exactly this property. Note that the uniformity is needed here. Indeed, assume that the primary VSdelivers the undo information message u, invokes S and then fails. Although the primary has failed, the other group members must also VSdeliver u to prevent orphan requests on S. □

From this lemma, the proof of Properties 6 and 7 is immediate. When q becomes primary, it first processes all undo information by sending it to the corresponding servers S. This and the fact that undo information is only garbage collected when an ACK has been received ensure that Properties 6 and 7 are satisfied.

7.4.4 Evaluation

Message Costs. Compared to the costs of passive replication (more specifically VScast) our mechanism adds an additional overhead. We are interested in the costs of the execution in which no processes fail (which generally is the case most of the time) and compute (1) the total number of messages and (2) the messages on the critical path of the execution. A message is on the critical path if the algorithm cannot proceed until this message is received.

If we assume that R consists of n replicas, then the total number of additional messages is $(n-1)^2 + 3(n-1)$, the cost of *Uniform VScast*. Indeed, sharing the undo information among the replicas requires to send additional $(n-1)^2 + (n-1)$ messages, $(n-1)$ ACKs and $(n-1)$ notifications to deliver. Among these messages, $\lceil \frac{n+1}{2} \rceil n + (n-1)$ are on the critical path. Clearly, if a broadcast medium is available among the replicas, then the costs of sending undo information is greatly reduced. However, the ACKs are still sent using point-to-point communication.

The costs of the undo messages are added to every single remote server invocation. Hence, response time has increased. On the other hand, these messages are usually very small and thus not very expensive.

Limitations. The approach presented in Section 7.4.2 has two limitations. However, we believe that these limitations are inherent to the replicated invocation itself, and not at all related to our solution.

The first drawback is that server(s) S are not allowed to spontaneously abort unterminated invocations. In our solution, the client replicas R are responsible for terminating pending invocations, and the server(s) S relies entirely on the replicas R. In other words, the server(s) S must trust the clients R to do their job.

A pessimistic server S needs to support the abort/commit of a transaction (i.e., invocation) by another process than the one that has issued the invocation (see Section 7.4.2). To our knowledge, although a

mechanism to pass on the responsibility for a transaction to another process is foreseen in the XA Specification for distributed transaction processing [35], this mechanism seems not to encompass the situation where processes fail. Rather, in this case, the unterminated transaction is simply aborted.

7.5 Related Work

Most of the work performed in the context of replicated invocation assumes deterministic execution. For example, Mazouni's work [45] addresses transparency of the replication technique in the context of replicated invocation. More specifically, the replication mechanism of the client needs to be hidden from the server, and vice-versa. Mazouni advocates the use of proxies to achieve transparency, for both the invocation and the reply to the invocation. Hence, a proxy is located with each client and server replica. To achieve transparency, these proxies also filter duplicate invocations and results, assuming that the clients and the actively replicated servers are deterministic.

Zhao, Moser, and Melliar-Smith [76] unify fault-tolerant CORBA (FT-CORBA) and the CORBA Object Transaction Service (OTS) in the context of a three-tier architecture. Their work also assumes deterministic execution. The proposed infrastructure replicates transactional application servers (business-logic tier) to protect them from failures. Moreover, they are augmented with an automatic transaction retry mechanism, which in the case of failure prevents the client from reissuing the request (this prevents duplicate invocations from the client tier). Replicated gateways are introduced between the business-logic tier and the data tier: they are responsible for filtering duplicate invocations and manage transaction retry. If a failure occurs and an ongoing transaction is not *ReadyToCommit*, the infrastructure, transparently to the client, aborts and retries the transaction. For this purpose, the state of all objects involved in the transaction is checkpointed [27].

Similarly in [19] Felber and Narasimhan use fault-tolerant CORBA and the CORBA Object Transaction Service in the context of a three-tier architecture. Additionally they allow middle tier (server) non-determinism. FT-CORBA mechanisms are used for the middle tier replication and CORBA OTS is used for the transactional invocation of the third (data) tier. The protocol in [19] is an implementation example of the more general protocol presented in this chapter. In such implementation the complexity of undoing orphan requests is hidden inside the CORBA OTS service. Also the protocol in [19] considers only the pessimistic data tier servers.

In contrast, Narasimhan enforces determinism (in the context of multithreaded applications) instead of assuming determinism. The work was performed in the context of Eternal [54], a replication infrastructure for CORBA objects. It introduces the notion of *MT-domain* (MT stands for multithreaded), to refer to any CORBA client or server that supports multiple (application level or ORB level) threads, which may access shared data. The Eternal system enforces deterministic behavior within the MT-domain by allowing only *a single logical thread of control* within each replica of the MT-domain at any point in time. Although multiple threads may exist in a MT-domain, all of them relate to the same logical thread of control, and only one is allowed to execute at a time. Consistent dispatching of threads within replicated MT-domain is achieved using *deterministic operation scheduler*.

Jimenez et al. [36] enforce determinism of transactional multithreaded replicas in the context of active replication. More specifically, they identify two levels of non-determinism: external and internal. External non-determinism corresponds to non-determinism related to communication, while internal non-determinism relates to computation, in particular thread scheduling. External non-determinism is handled using totally ordered multicast. Internal non-determinism is addressed with deterministic thread scheduling and selective message reception from two-level queues. In [36], no replicated invocation is considered.

Frølund and Guerraoui present a correctness criterion for exactly-once in the context of replication [22], that also addresses non-determinism in the execution and external side-effects. They also propose a replication protocol, called *asynchronous replication* [21]. The protocol is targeted towards the classical three-tier architecture, with slim client, stateless application servers, and databases. In contrast, our approach is more general in that it also addresses stateful components (our approach does not make the distinction between clients and servers). Rather, any client can at the same time act as a server for another client. Assuming stateful components clearly leads to stronger requirements, e.g., the update of all replicas.

A large body of work in the context of checkpointing and rollback recovery exists [16]. However, even if undo messages appear in this chapter and in checkpointing/rollback recovery, the issues are only loosely related. Checkpointing techniques do not address availability: progress is only possible upon recovery. In contrast, the chapter addresses issues in the context of replication, a technique masking failures, i.e., allowing progress even while processes are down.

7.6 Summary

In the chapter we have presented the problem of orphan requests. In the context of replicated invocations, orphan requests occur when a server replica R_i invokes another server S, but fails before updating the other replicas R_j $(j \neq i)$. Hence, the results of the execution on R_i are lost. As the state of server S reflects the invocation by R_i, the state of S may become inconsistent with respect to the other replicas R_j. This problem, which is easily addressed with deterministic replicated servers R [45], has not been solved in the context of non-deterministic replicated servers. We proposed a protocol for preventing orphan invocations based on undo information shared by R_i with other replicas of R. More specifically, R_i sends undo information to its replicas before issuing the nested invocation to S. Based on this undo information, another replica R_k $(k \neq i)$ can undo R_i's invocation on S in case R_i fails or is erroneously suspected. Our protocol handles both pessimistic and optimistic execution of the invocation on S.

Chapter 8

Conclusion

8.1 Research Assessment

In the thesis we propose a standard interface for group communication (GC) based on JMS. We believe that its acceptance will enable the wider use of GC not only in the academic community, but also in the enterprise world. The adoption of the JMS interface for GC led to three major assessments:

The mapping and specification. When defining a standard for GC we provided an API mapping between GC primitives and JMS. But we did not constrain the standard only to the API. We wanted to profit from the properties provided in the JMS specification. This lead to the semantic mapping between the JMS and GC. Based on this mapping a new GC specification that incorporates JMS properties was proposed. As the API and the specification reflect the spirit of JMS, we hope that our proposal will contribute to a wider use of the GC abstractions, and that GC will become an integral part of future applications.

The architecture and implementation. To implement our specification we have chosen a centralized architecture. It provides GC as a service. Two implementations were proposed: a JORAM based and a Component Chain based. Both rely on the component architecture. The two implementations differ in the way the service is composed from the smaller components. In the JORAM based implementation the components (agents), in order to send the message to the right component, need the information about the other components and the services they provide. In the Component Chain based implementation the components are concerned only with their own task (with just a few exceptions for efficiency): they do not have the information who will use the results they produce. This information is provided by the "component chain" which is constructed at the system initialization time.

The replicated invocation. In the centralized server architecture when the server providing GC itself is replicated, we have two level of replication. This raises the problem of replicated invocation, when a replicated server invokes another replicated server and the invoking server is non-deterministic. We

specified the replicated invocation problem in the transactional environment and presented the solution to it.

In our implementations the server is deterministic and uses active replication. So our solution is not needed here. But in a case of non-deterministic server, the problem must be taken into account.

8.2 Open Questions and Future Research Directions

Performance. Middleware systems in general does not show good performance. The performance is the price for the high abstraction level that middleware provides. Our JMSGroups implementation (CCB) also suffers from the performance problem. But we think that its performance can be improved by refactoring and optimizing the code. Performance improvement is one of the top priorities for the future development of the JMSGroups implementation.

Transactions. The JMS specification specifies a basic transaction mechanism. Transactions were not included in our mapping and specification. The open questions is "How can GC be mapped to JMS transactions and what are the benefits of such mapping for GC?".

Further J2EE integration. In the thesis we focused only on the JMS specification, but JMS is a part of the Java Enterprise Edition specification (J2EE). Further GC integration into J2EE would make GC even more popular among the industry users, which could benefit from the GC services.

Bibliography

[1] M. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13:99–125, 2000.

[2] B. Ban. Design and Implementation of a Reliable Group Communication Toolkit for Java, September 1998.

[3] B. Ban. *JavaGroups User's Guide*, Nov 2002.

[4] G. Banavar, T. Chandra, R. Strom, and D. Sturman. A Case for Message Oriented Middleware. LNCS 1693, pages 1–17. Springer Verlag, 1999.

[5] BEA. BEA Tuxedo: The programming model. white paper, BEA Systems, USA, November 1996.

[6] L. Bellisard, N. De Palma, A. Freyssiinet, M. Herrmann, and S. Lacourte. An Agent Platform for Reliable Asynchronous Distributed Programming. In *Proceedings of the 18th Symposium on Reliable Distributed Systems (SRDS-18)*, pages 292–293, Lausanne, Switzerland, October 1999.

[7] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138. ACM Press, 1987.

[8] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993.

[9] K. P. Birman, R. Constable, M. Hayden, C. Kreitz, O. Rodeh, R. van Renesse, and W. Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *Proceedings of the DARPA Information Survivability Conference & Exposition (DISCEX '00)*, Hilton Head, South Carolina USA, 2000.

[10] N. Budhirja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In *Distributed Systems*, pages 199–216.

[11] D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems (TOCS)*, 3(2):77–107, May 1985.

[12] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 4(33):1–43, December 2001.

[13] P.K. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems (TODS)*, 19(3):450–491, 1994.

[14] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proc. of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)*, pages 43–50, West Lafayette, Indiana, October 1998.

[15] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, 1996.

[16] E.N. Elnozahy, L. Alvisi, Y.-M. Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[17] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1867.

[18] P. Felber, R. Guerraoui, and A. Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[19] P. Felber and P. Narasimhan. Reconciling Replication and Transactions for the End-to-End Reliability of CORBA Applications. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'02)*, Irvine, California, USA, October 2002. IEEE Computer Society Press.

[20] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. In *Proc. of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–7, Atlanta, Georgia, March 1983.

[21] S. Frølund and R. Guerraoui. Implementing e-transactions with asynchronous replication. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):133–146, February 2001.

[22] S. Frølund and R. Guerraoui. X-ability: a theory of replication. *Distributed Computing*, 14(4):231–249, December 2001.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[24] H. Garcia-Molina and K. Salem. Sagas. In *Proc. of ACM SIGMOD Int. Conference on Management of Data and Symposium on Principles of Database Systems*, pages 249–259, 1987.

[25] A. S. Gokhale and D. C. Schmidt. Measuring and Optimizing CORBA Latency and Scalability Over High-Speed Networks. *IEEE Transactions on Computers*, 47(4):391–413, 1998.

[26] J. Gray. The transaction concept: virtues and limitations. In *Proc. of Int. Conference on Very Large Databases*, pages 144–154, Cannes, France, 1981.

[27] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, USA, 1993.

[28] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

[29] V. Hadzilacos and S. Toueg. A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. Technical Report TR94-1425, CS, University of Toronto; CS, Cornell University, May 1994.

[30] M. Hapner, R. Sharma, J. Fialli, and K. Stout. *JMS specification*. Sun Microsystems Inc., USA, 1.1 edition, April 2002. http://java.sun.com/products/jms/docs.html.

[31] M. Hayden. The Ensemble system. Technical Report TR98-1662, Department of Computer Science, Cornell University, January 1998.

[32] M. A. Hiltunen and R. D. Schlichting. The Cactus approach to building configurable middleware services. In *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC'00)*, Nürnberg, Germany, 2000.

[33] IBM. *MQSeries Application Programming Guide*. USA, 11 edition, 2000. SC33-0807-10.

[34] Isis Distributed Systems, Inc. *The Isis Distributed Toolkit, User Reference Manual, Version 3.0*. Ithaca, NY, 1992.

[35] ISO/IEC. *Information Technology - Distributed Transaction Processing - The XA Specification*, 1st edition, 1996. ISO/IEC 14834.

[36] R. Jiménez-Peris, M. Patiño-Martínez, and S. Arevalo. Deterministic Scheduling for Transactional Multithreaded Replicas. In *Proceedings of the 19th IEEE Symposium on Distributed Reliable Systems, SRDS'00*, pages 164–173, Nuremberg, Germany, October 2000.

[37] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[38] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[39] C. Loosley and F. Douglas. *High-Performance Client/Server*. Wiley Computer Publishing, New York, USA, 1998.

[40] D. Malki. *The Transis User Tutorial*. Institute of Computer Science, The Hebrew University of Jerusalem, Israel, first edition, March 1994.

[41] D. Malki, Y. Amir, D. Dolev, and S. Kramer. The Transis Approach to High Availability Cluster Communication. Technical Report CS94-14, 1994.

[42] C. P. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, September 1996.

[43] C. P. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A toolkit for building fault-tolerant distributed applications in large scale. In *Workshop on Parallel and Distributed Platforms in Industrial Products*, San Antonio, Texas, USA, October 1995. Workshop held during the 7th IEEE Symp. on Parallel and Distributed Processing, (SPDP-7).

[44] C. Marchetti. Interoperable Replication Logic, `http://www.dis.uniroma1.it/~irl`.

[45] K.R. Mazouni, B. Garbinato, and R. Guerraoui. Filtering duplicated invocations using symmetric proxies. In *Proc. of the 4th IEEE International Workshop on Object Orientation in Operating Systems (IWOOOS'95)*, Lund, Sweden, August 1995.

[46] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast Protocols for Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, Jan 1990.

[47] S. Mena, X. Cuvellier, C. Grégoire, and A. Schiper. Appia *vs.* Cactus: Comparing protocol composition frameworks. In *22nd Symposium on Reliable Distributed Systems. Florence, Italy*, October 2003.

[48] S. Mena, A. Schiper, and P. Wojciechowski. A Step Towards a New Generation of Group Communication Systems. In *Proceedings of the Int. ACM/IFIP/USENIX Middleware Conference*, LNCS 2672, Rio de Janeiro, Brazil, June 2003. Springer-Verlag.

[49] Sun Microsystems. `http://www.sun.com`.

[50] H. Miranda, A. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the* 21^{st} *International Conference on Distributed Computing Systems (ICDCS'01)*, pages 707–710, Phoenix, Arizona, USA, 2001. IEEE.

[51] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, 1996.

[52] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. A fault tolerance framework for CORBA. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS)*, Washington, DC, 1999. IEEE Computer Society.

[53] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, MA, USA, 1985.

[54] P. Narasimhan. *Transparent Fault Tolerance for CORBA*. PhD thesis, University of California, Santa Barbara, USA, September 1999.

[55] P. Narasimhan, K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Providing support for survivable CORBA applications with the Immune system. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 507–516. IEEE Computer Society, 1999.

[56] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, 3.0.3 edition, March 2004.

[57] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. The design and implementation of Arjuna. Technical Report TR94-65, ESPRIT Basic Research Project BROADCAST, 1994.

[58] F. Pedone and A. Schiper. Handling message semantics with Generic Broadcast protocols. *Distributed Computing*, 15(2):97–107, 2002.

[59] S. Pleisch. *Fault Tolerant and Transactional Mobile Agent Execution*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, October 2002. Thesis Number 2654.

[60] S. Poledna. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems*, 6(3):289–316, May 1994.

[61] D. Powell. Delta4: A generic architecture for dependable distributed computing. volume 1 of *ESPRIT Research Reports*. Springer Verlag, 1991.

[62] R. Baldoni, C. Marchetti, A. Termini. Active Software Replication through a Three-tier Approach. In *Proceedings of the 22th IEEE International Symposium on Reliable Distributed Systems (SRDS02)*, pages pp. 109–118, Osaka, Japan., October 2002.

[63] L. Rodrigues and M. Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems. In *Proceedings of the 20th IEEE International Conference on Distributed Com puting Systems (ICDCS'20)*, pages 288–295, Taipe, Taiwan, April 2000.

[64] ScalAgent. *JORAM*. http://joram.objectweb.org.

[65] ScalAgent. *ScalAgent*. http://www.scalagent.com.

[66] A. Schiper. Dynamic Group Communication. Technical Report Nbr:200327, École Polytechnique Fédérale de Lausanne (EPFL), 2003.

[67] A. Schiper, K. Birman, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.

[68] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *IEEE 13th Intl. Conf. Distributed Computing Systems*, pages 561–568, May 1993.

[69] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *Proceedings of the 13th International Conference on Distributed Computing Systems (ICDCS-13)*, pages 561–568, Pittsburgh, Pennsylvania, USA, May 1993. IEEE Computer Society Press.

[70] F. B. Schneider. Replication management using the state-machine approach. pages 169–198.

[71] B. Shannon. *Java 2 Enterprise Edition specification*. Sun Microsystems Inc., USA, 1.4 edition, April 2003.

[72] Sonic Software Corporation. Clustering and Dynamic Routing in SonicMQ. White paper, USA, January 2004.

[73] TIBCO. Rendezvous, TIBCO Messaging Solutions.

[74] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. A. Karr. A framework for protocol composition in horus. In *Symposium on Principles of Distributed Computing*, pages 80–89, 1995.

[75] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[76] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. Unification of replication and transaction processing in three-tier architectures. In *Proc. of Int. Conference on Distributed Computing Systems (ICDCS'02)*, pages 290–297, Vienna, Austria, July 2002.

Appendix A

Group Communication Toolkits API

A.1 Isis

Isis interface description (the full description can be found in [34]).

`isis_remote_init()` Full signature: `int isis_remote_init(host_list, lport_no, rport_no, flags)`. Connects to the Isis backbone, the set of machines running Isis system, in order to use the group communication services provided. Argument `host_list` specifies the list of the machines in the backbone; the `lport_no` and `rport_no` give the port numbers to be used for connecting, and are respectively the Isis "TCP" port number and the "bcast" port number; the `flags` argument specifies a number of flags which control the actions Isis will take on the connection.

`site_getview()` Full signature: `sview* site_getview()`. Returns a pointer to the member view structure. The Isis system keeps track of which members are currently active and stores this information in the site view structure. The structure is automatically updated when members join or leave.

`bcast()` Full signature: `int bcast(addr_p, entry, fmt1, arg1, arg2, ..., nwanted, fmt2, rep1, rep2,...)`. Broadcasts message and collects the replies. The argument `addr_p` is the pointer to the address of the group to which to deliver the messages; in Isis each of the recipients must have an associated entry number with a task (callback). The argument `entry` is the entry number in recipient process; `fmt1` is the format string for the data to be put into the outgoing message (format strings are used to denote the number and the type of the arguments following the string); argument `nwanted` specifies number of reply messages wanted (ALL or MAJORITY); argument `fmt2` is a format string for data to be read from the reply messages (this and the following arguments can be omitted if `nwanted=0`). There are four different broadcast primitives in Isis: `fbcast`, `cbcast`, `abcast` and `gbcast`. They use the same interface

as `bcast`, but differ in the type of multicast ordering they provide. Also there is long form of the broadcast method `bcast_l`, which provides more options upon the call, but we do not include the full description of it here.

`pg_join()` Full signature: `address* pg_join(gname, ... PG_KEYWORDi, argi1, argi2, ..., 0)`. Adds a process to the group. Argument `gname` is the name of the group to join; options to the method are placed between the first argument and the last, which is always `0`. Each option consists of a keyword followed by the arguments required by that option and specifies additional tasks such as state transfer, setting membership monitoring callback, etc. The method returns the address of the group.

`pg_subgroup()` Full signature: `address* pg_subgroup(pgaddr_p, sgname, incarn, members, NULL)`. Creates a process group with the prespecified initial membership. Argument `pg_subgroup` is the address of the parent group, the caller must be a member; `sgname` the name of the new subgroup, as in `pg_join`; `incarn` is the incarnation number; `members` null-terminated list of initial members, subset of the parent group membership. Return value is the address of the subgroup.

`pg_leave()` Full signature: `pg_leave(gaddr_p)`. Makes a calling process leave a group. `gaddr_p` pointer to the address of the group to leave.

`pg_client()` Full signature: `pg_client(gaddr_p, credentials)`. Registers the calling process as a client of the group. `gaddr_p` the pointer to the group address; `credentials` null-terminated string used as credentials.

`pg_getview()` Full signature: `groupview* pg_getview(gaddr_p)`. Returns a pointer to a group view structure. Argument `gaddr_p` is the pointer to the address of the group.

`sv_monitor()`[†] Full signature: `int sv_monitor(routine, arg)`. Sets the `routine` (function) to monitor the site view changes; `arg` is the argument to be passed for the routine. Returns monitor ID as an integer.

`sv_watch()`[†] Full signature: `int sv_watch(sid, event, routine, arg)`. Registers a `routine` to monitor the given site to fail or recover. Argument `sid` is the ID of the site to monitor; `event` is the event on which to invoke the routine (`W_FAIL or W_RECOVER`); `arg` is the argument to be passed to the routine.

`pg_monitor()`[†] Full signature: `int pg_monitor(gaddr_p, routine, arg)`. Registers a `routine` (function) to monitor changes of the membership of a process group. `gaddr_p` is the pointer to the group address; `arg` is the argument to be passed to the routine. Returns an integer monitor ID.

[†]The method has the corresponding `cancel` method.

A.2 Transis

Transis interface description (the full description can be found in [40]).

zzz_Connect() Full signature: `zzz_mbox_cap zzz_Connect(char *mbox_name, char *select-layers, int flags)`. Method to connect to Transis. `mbox_name` string specifies a unique name for the program, `select-layers` string specifies the list of layers to be activated, `flags` bit mask specifies option flags. The return value of type `zzz_mbox_cap` is used for the all further calls on the Transis Groups Facility.

zzz_Join() Full signature: `void zzz_Join( zzz_mbox_cap mbox_c, char *set_name)`. Join the group named `set_name`. `mbox_c` the reference to the group control object assigned by Transis upon connecting, `set_name` the name of the group to join.

zzz_Leave() Full signature: `void zzz_Leave(zzz_mbox_cap mbox_c, char *set_name)`. Leave the group named `set_name`. `mbox_c` the reference to the group control object assigned by Transis upon connecting, `set_name` the name of the group to leave.

zzz_Send() Full signature: `void zzz_Send(zzz_mbox_cap mbox_c, int send_type, int flag, int len, char *buf, char *targets[])`. Send an untyped message `buf`, of length `len` bytes, to the specified target groups. `send_type` may be one of `ATOMIC`, `CAUSAL`, `AGREED`, `SAFE`. Return value indicates the number of bytes sent.

zzz_Receive() Full signature: `int zzz_Receive(zzz_mbox_cap mbox_c, char buf[], int max_len, int *receive_type, view **view)`. Attempts to receive the next message, of at most `max_len` bytes into `buf`. It returns the number of bytes received.

zzz_Add_Upcall() Full signature: `void zzz_Add_Upcall(zzz_mbox_cap mbox_c, void (* func)(int, void *), int priority, void *param)`. Add an upcall event handler, that will be invoked automatically whenever there are pending messages. This call together with the corresponding `zzz_Remove_Upcall()` are used in the Transis Event mechanism ([40], Chapter 4).

A.3 Phoenix

Phoenix interface description (the full description can be found in [42]).

SinkSubscribe() Full signature: `void SinkSubscribe(char *groupname)`. Allows a sink to become sink of a group `groupname`.

SinkUnsubscribe() Full signature: `void SinkSubscribe(char *groupname)`. Allows a sink to leave the group `groupname` to which it is a sink.

ClientSubscribe() Full signature: `void ClientSubscribe(char *groupname)`. Makes an object a client of the group `groupname`.

ClientUnsubscribe() Full signature: `void ClientUnsubscribe(char *groupname)`. Makes an object to leave the group `groupname` of which it is a client.

ViewChangeDeliveryCB() Full signature: `void ViewChangeDeliveryCB(View view)`. This method is invoked on the client of the group each time the view of the group changes. Argument view contains the new membership of the group.

Join() Full signature: `void Join(char *groupname)`. Adds the calling object to the group `groupname`.

Leave() Full signature: `Leave()`. Makes the object leave the group of which it is currently a member.

Multicast() Full signature: `void Multicast(Message msg, Order ord)`. Multicast the message `msg` to the group. Optional parameter `ord` specifies the order type for the multicast: FIFO, uniform, weak total, strong total or global order.

IntermediateViewDelivery() Full signature: `void IntermediateViewDelivery(View view)`. This method is invoked each time a different intermediate view is delivered by the Phoenix system at the beginnind and during the execution of view change protocol.

A.4 JGroups

JGroups interface description (the full description can be found in [3]).

connect() Full signature: `void connect(String groupname)`. Adds a calling client to the group. The argument `groupname` specifies the name of the group to join. If the group with the given name does not exist, this method will create it. The method changes group membership.

getView() Full signature: `View getView()`. Returns the current view of the group, to which channel is connected to.

send() Full signature: `void send(Message msg)`. Sends a message `msg` to the group members (including itself). If the message contains a destination address, it can also be sent to a single group member.

receive() Full signature: `Object receive(long timeout)`. Receives the next available message from the channel. The method is blocking, i.e., if there are no available messages it will block until a message arrives. If the argument `timeout` is greater than 0, and the message does not arrive until the timeout expires, the exception will be thrown. The

received object can be an ordinary message, or can contain some special object such as view, suspicion, state transfer, etc.

disconnect() Full signature: `void disconnect()`. Disconnects the calling client from the channel, this is equivalent to leaving the group. The method changes the group's membership.

receive() Full signature: `void receive(Message msg)`. This `MessageListener` interface callback is automatically called when the message is received by the `Channel`. The argument `msg` contains the message received. The `MessageListener` interface must be provided by the application, which wants to receive the messages using this callback.

viewAccepted() Full signature: `void viewAccepted(View new_view)`. This `MembershipListener` interface callback is automatically called when the new member joins the group, or the existing member leaves or crashes. The `new_view` argument contains the new view of the group. `MembershipListener` interface must be provided by the application, which wants to be notified about the group membership events.

suspect() Full signature: `void suspect(Object suspected_mbr)`. This `MembershipListener` interface callback is automatically called when a member is suspected of having crashed, but not yet excluded from the view. The argument `suspected_mbr` identifies the suspected member.

A.5 Object Group Service

OGS interface description (the full description can be found in [17]).

join_group() Full signature: `void join_group(Groupable member, InterfaceSemantics semantics)`. This `GroupAdministrator` interface method adds the specified `member` to the group; the member must implement the `Groupable` interface. The argument `semantics` specifies the semantics associated with each of its operations.

leave_group() Full signature: `void leave_group(Groupable member)`. This `GroupAdministrator` interface method removes the specified member from the group. The argument `member` is the member to be removed.

view_change() Full signature: `void view_change(GrouView view)`. This `Groupable` interface method is called on the member of the group upon membership change. The argument `view` contains the new view of the group.

deliver() Full signature: `any deliver(any msg)`. This `Invocable` interface method is called to deliver a message to the individual group member. The argument `msg` contains the message to deliver.

multicast() Full signature: `AnySeq multicast(any msg, NumReplies replies, Semantics semantics)`. This `GroupAccesor` interface method issues a multicast to the group. The argument `msg` is a message to multicast, the argument `replies` specifies the number of replies to wait for (ONEWAY, ZERO, ONE, MAJORITY, ALL), and the argument `semantics` specifies the semantics of the multicast (UNRELIABLE, RELIABLE, FIFO, TOTAL_ORDER). The method returns the replies as a sequence of objects of type `Any`.

get_view() Full signature: `GroupView get_view()`. This `GroupAccessor` interface method returns the view of the group to the client in the `GroupView` structure. There is no guarantee that the view is up to date as the client is usually not a member of the group.

A.6 Eternal System

Eternal system interface description (the full description can be found in [54]).

create_object() Full signature: `Object create_object(TypeId type_id, Criteria the_criteria, FactoryCreationId factory_creation_id)`. Creates an object group. The argument `type_id` is the repository identifier for the object type, the argument `the_criteria` contains the parameters to be passed for the object factory, the argument `factory_creation_id` is the factory local identifier used to delete the object. The method returns the created object group reference.

delete_object() Full signature: `void delete_object(FactoryCreationId factory_creation_id)`. Deletes an object group. The argument `factory_creation_id` is the local factory identifier specified at the object group creation time.

create_member() Full signature: `ObjectGroup create_member(ObjectGroup object_group, Location the_location, TypeId type_id, Criteria the_criteria)`. Creates a member of the specified object group `object_group` at the location `the_location`. The argument `type_id` is the repository identifier for the object type, the argument `the_criteria` contains the parameters to be used for the member creation. The method returns the object group reference with the member added.

add_member() Full signature: `ObjectGroup add_member(ObjectGroup object_group, Location the_location, Criteria the_criteria)`. Adds the existing member to an object group at a particular location. The argument `object_group` is the group to which the member is added, the argument `the_location` is the location at which the member resides, the argument `member` is the reference to the object to be added. The method returns the object group reference with the member added.

remove_member() Full signature: `ObjectGroup remove_member(ObjectGroup object_group, Location the_location)`. Removes the existing member from an object group at a particular location. The argument `object_group` is the group from which the member is removed, the argument `the_location` is the location at which the member resides. The method returns the object group reference with the member removed.

A.7 Interoperable Replication Logic

IRL interface description (the full description can be found in [44]).

create_object() Full signature: `org.omg.CORBA.Object create_object(String type_id, org.omg.CORBA.FT.Property[] the_criteria, org.omg.CORBA.AnyHolder factory_creation_id)`. Creates an Object Group. The argument `type_id` is the interface repository ID type for the created group. The second argument `the_criteria` is the array of properties for the created group. The `factory_creation_id` is the ID of a factory, which is responsible to create the group object. The method returns the created object group.

delete_object() Full signature: `void delete_object(org.omg.CORBA.Any factory_creation_id)`. Deletes an Object Group. The argument `factory_creation_id` is the ID of a factory, which is responsible to delete the group object.

add_member() Full signature: `org.omg.CORBA.Object add_member(org.omg.CORBA.Object og, org.omg.CosNaming.NameComponent[] the_location, org.omg.CORBA.Object member)`. Adds a member into an Object Group. The argument `og` is the object group to use. The second argument `the_location` is the CORBA naming service reference to the new member. The last argument `member` is the object to be added to the object group. The method returns the object group with the new member.

remove_member() Full signature: `org.omg.CORBA.Object remove_member(org.omg.CORBA.Object og, org.omg.CosNaming.NameComponent[] the_location)`. Removes a member from an Object Group. The argument `og` is the object group to use. The second argument `the_location` is the CORBA naming service reference to the member to be removed. The method returns the object group without the removed member.

Appendix B

Replicated Table Source Code

B.1 ReplTable.java

```
1  /*
2   * JORAM: Java(TM) Open Reliable Asynchronous Messaging
3   * Copyright (C) 2001 - ScalAgent Distributed Technologies
4   * Copyright (C) 1996 - Dyade
5   *
6   * This library is free software; you can redistribute it and/or
7   * modify it under the terms of the GNU Lesser General Public
8   * License as published by the Free Software Foundation; either
9   * version 2.1 of the License, or any later version.
10  *
11  * This library is distributed in the hope that it will be useful,
12  * but WITHOUT ANY WARRANTY; without even the implied warranty of
13  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
14  * Lesser General Public License for more details.
15  *
16  * You should have received a copy of the GNU Lesser General Public
17  * License along with this library; if not, write to the Free Software
18  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
19  * USA.
20  *
21  * Initial developer(s): Frederic Maistre (INRIA)
22  * Contributor(s):
23  */
24 package lsr.epfl.hash.active;
25
26 import java.io.*;
27 import java.util.*;
28 import javax.jms.*;
29 import javax.naming.*;
30 import fr.dyade.aaa.joram.TemporaryQueue;
31 import java.util.logging.Logger;
32
33 /**
34  * Hastable replica class. Subscribes and sets a listener to the GroupTopic.
35  */
36 public class ReplTable {
37         // variable for the logging messages
38         private static Logger log = Logger
39                         .getLogger("lsr.epfl.hash.active.ReplTable");
40
41         static Context ictx = null; // initial context for NamingService
42
43         // variables to subscribe to the topic
44         TopicSession groupSession = null;
45         TopicSubscriber groupSubscriber = null;
46         TopicPublisher groupPublisher = null;
47
48         /** Data structure where the state of a Hastable is kept. */
49         Hashtable table = null;
```

```
50
51          // constants for the hashtable invocation message property names
52          public static final String KEY = "#key";
53          public static final String VALUE = "#value";
54          public static final String PUT_PROPERTY = "#put_property";
55          public static final String REMOVE_PROPERTY = "#remove_property";
56          public static final String GET_PROPERTY = "#get_property";
57          public static final String REPLY_PROPERTY = "#reply_property";
58          public static final String STATE = "#state";
59
60          /*-------------------------------------------------------*/
61          /** Constructor for Hashtable replica. */
62          public ReplTable(String _ID) {
63                  table = new Hashtable();
64          }
65
66          /*-------------------------------------------------------*/
67          /**
68           * Establishes the connection with the server, initializes publisher and
69           * subscriber to the GroupTopic.
70           */
71          public void join(String[] args) {
72                  try {
73                          ictx = new InitialContext(); // obtain NamingService initial context
74                          // get the GroupTopic reference
75                          Topic topic = (Topic) ictx.lookup("group_topic");
76                          // get the TopicConnectionFactory reference
77                          TopicConnectionFactory tcf = (TopicConnectionFactory) ictx.lookup("tcf");
78                          ictx.close(); // close the NamingService context
79
80                          // select which subscriber to create: durable or not
81                          // this is specified from the command line argument
82                          boolean durable = false;
83                          String id = null;
84                          if ((args.length == 2) && (args[0].equalsIgnoreCase("durable"))) {
85                                  id = args[1];
86                                  if (id != null) {
87                                          durable = true;
88                                          log.fine("Creating durable subscriber with ID=" + id);
89                                  } else {
90                                          System.err.println("The ID for the durable subcription is not "
91                                                  + "specified, creating NON-DURABLE subscriber");
92                                  }
93                          } else {
94                                  log.fine("Creating non-durable subscriber.");
95                          }
96
97                          // create a TopicConnection
98                          TopicConnection groupConnection;
99                          if (durable)
100                                 groupConnection = tcf.createTopicConnection("durable_user",
101                                                 "durable_user");
102                         else
103                                 groupConnection = tcf.createTopicConnection("group_user",
104                                                 "group_user");
105
106                         // create a TopicSession
107                         groupSession = groupConnection.createTopicSession(false,
108                                         javax.jms.Session.AUTO_ACKNOWLEDGE);
109
110                         // create a TopicSubscriber: durable or not
111                         if (durable)
112                                 groupSubscriber = groupSession.createDurableSubscriber(topic, id);
113                         else
114                                 groupSubscriber = groupSession.createSubscriber(topic);
115
116                         // set the message listener to handle the messages
117                         groupSubscriber.setMessageListener(new ReplTableMsgListener(this));
118                         // start the connection
119                         groupConnection.start();
120
121                         // create a TopicPubliesher: join the group
122                         groupPublisher = groupSession.createPublisher(topic);
123
124                         // wait untill the key is pressed to terminate
125                         try {
```

```
126                             System.in.read();
127                     } catch (IOException e) {
128                             e.printStackTrace();
129                     } // catch
130                     // close the connection
131                     groupConnection.close();
132
133                     System.out.println("Table Replica terminated.");
134             } catch (NamingException e) {
135                     e.printStackTrace();
136             } catch (JMSException e) {
137                     e.printStackTrace();
138             }
139     } // Join
140
141     /*-------------------------------------------------------------*/
142     /** Implements Hashtable "get" method. */
143     public void get(String key, TemporaryQueue answer_q) {
144             try {
145                     String value = (String) table.get(key);
146                     if (value != null) {
147                             // construct and send the reply to the client
148                             TextMessage msg = groupSession.createTextMessage();
149                             msg.setStringProperty(ReplTable.REPLY_PROPERTY, "");
150                             msg.setStringProperty(ReplTable.KEY, key);
151                             msg.setStringProperty(ReplTable.VALUE, value);
152
153                             javax.jms.MessageProducer answer_prod = groupSession
154                                             .createProducer(answer_q);
155                             answer_prod.send(msg);
156                     } // if
157
158             } catch (JMSException e) {
159                     e.printStackTrace();
160             }
161     }
162
163     /*-------------------------------------------------------------*/
164     /** Implements Hashtable "put" method. */
165     public void put(String key, String value) {
166             table.put(key, value);
167     }
168
169     /*-------------------------------------------------------------*/
170     /** Implements Hashtable "remove" method. */
171     public void remove(String key) {
172             table.remove(key);
173     }
174
175     /*-------------------------------------------------------------*/
176     /** Transfer the state to the new member. */
177     public void stateTransfer() {
178             try {
179                     ObjectMessage m = groupSession.createObjectMessage();
180                     m.setObject(table);
181                     m.setStringProperty(ReplTable.STATE, "");
182                     groupPublisher.publish(m);
183             } catch (JMSException e) {
184                     e.printStackTrace();
185             }
186     }
187
188     /*-------------------------------------------------------------*/
189     /** Remove the specified members from the group. */
190     public void removeMembers(String members) {
191             try {
192                     TextMessage msg = groupSession.createTextMessage();
193                     msg.setStringProperty(``JMS_remove'', members);
194                     groupPublisher.publish(msg);
195             } catch (JMSException e) {
196                     e.printStackTrace();
197             }
198     }
199
200     /*-------------------------------------------------------------*/
201     /** Print the table to the console. */
```

```
202         public void print() {
203                 log.info("TABLE IS: " + table.toString());
204         }
205
206         /*-------------------------------------------------------------*/
207         /** Set the new Hastable state. */
208         public void setTable(Hashtable new_table) {
209                 table = new_table;
210         }
211
212         /*-------------------------------------------------------------*/
213         /** The execution entry method. */
214         public static void main(String[] args) throws Exception {
215                 ReplTable replTable = new ReplTable(args[0]);
216                 replTable.join(args);
217         }
218         /*-------------------------------------------------------------*/
219 } // ReplTable
```

B.2 ReplTableMsgListener.java

```
1  package lsr.epfl.hash.active;
2
3  import javax.jms.*;
4  import lsr.epfl.util.*;
5  import java.util.*;
6  import fr.dyade.aaa.mom.messages.GroupMessage;
7  import java.util.logging.Logger;
8
9  /**
10  * Implements the <code>javax.jms.MessageListener</code> interface,
11  * which  handles the incomming messages according to their type
12  * and properties.
13  */
14 public class ReplTableMsgListener implements MessageListener {
15                  // variable for the logging messages
16         private static Logger log = Logger
17                         .getLogger("lsr.epfl.hash.active.ReplTableMsgListener");
18
19         int prev_count = 1; // previous view member count
20
21         ReplTable master = null; // reference to the Hashtable replica
22
23         View curr_view = null; // currenrt view of the group
24
25         // variable to know for the member that it is joined a goup
26         boolean me_joined = false;
27
28         /*-------------------------------------------------------------*/
29         /** The constructor for the message Hashtables MessageLisener.*/
30         public ReplTableMsgListener(ReplTable master) {
31                 this.master = master; // reference to the Hashtable
32         }
33
34         /*-------------------------------------------------------------*/
35         /** Process th ereceived messages.*/
36         public void onMessage(Message msg) {
37                 try {
38                         // if the received message is TEXT MESSAGE
39                         if (msg instanceof TextMessage) {
40                                 log.fine("Text message received");
41                                 if (msg.propertyExists(ReplTable.PUT_PROPERTY)) {
42                                         processPutMessage(msg);
43                                 } else if (msg.propertyExists(ReplTable.GET_PROPERTY)) {
44                                         processGetMessage(msg);
45                                 } else if (msg.propertyExists(ReplTable.REMOVE_PROPERTY)) {
46                                         processRemoveMessage(msg);
47                                 } else if (msg.propertyExists(``JMS_suspect'')) {
48                                         processSuspitionMessage(msg);
49                                 } else
50                                         log.fine(((TextMessage) msg).getText());
51
52                                 // if the received message is OBJECT MESSAGE
53                         } else if (msg instanceof javax.jms.ObjectMessage) {
54                                 log.fine("Object message received");
55                                 ObjectMessage ob_msg = (ObjectMessage) msg;
56                                 if (msg.propertyExists(``JMS_view'')) {
57                                         processViewMessage(ob_msg);
58                                 } else if (msg.propertyExists(ReplTable.STATE)) {
59                                         processStateMessage(ob_msg);
60                                 }
61                         }
62                         // Other message types: do not process
63                         else {
64                                 log.fine("Other message received: " + msg.getClass().getName());
65                         } // else
66
67                 } catch (JMSException jE) {
68                         System.err.println("Exception in listener: " + jE);
69                 }
70         }
71
72         /*-------------------------------------------------------------*/
73         /** Process the state update message.
```

```
74          * Updates the state of the Hashtable replica.*/
75         public void processStateMessage(ObjectMessage msg) {
76                 try {
77                         if (me_joined) {
78                                 log.fine("Seting the state!");
79                                 ObjectMessage m = (ObjectMessage) msg;
80                                 Hashtable received_table = (Hashtable) m.getObject();
81                                 log.fine("To received table: " + received_table.toString());
82                                 master.setTable(received_table);
83                                 master.print();
84                         } // me_joined
85                 } catch (JMSException e) {
86                         e.printStackTrace();
87                 }
88         }
89
90         /*-------------------------------------------------------*/
91         /** Process View change message. */
92         public void processViewMessage(ObjectMessage msg) {
93                 try {
94                         // new view received, extract it
95                         View v = (View) msg.getObject();
96                         v.print(); // print the view on console
97                         // if it is the first view I receive, meens I just joined
98                         if (curr_view == null)
99                                 me_joined = true;
100                        else {
101                                // otherwise remember how many members were in the previous view
102                                // in order to detect if somebody joined
103                                me_joined = false;
104                                prev_count = curr_view.size();
105                        } // else
106                        curr_view = v; // make new view the current my view
107                        // if somebody joined and its not me: transfer the state
108                        if (msg.propertyExists(``JMS_join'') && !me_joined) {
109                                master.stateTransfer();
110                        }
111                } catch (JMSException e) {
112                        e.printStackTrace();
113                }
114                master.print(); // print the state of the table
115        }
116
117        /*-------------------------------------------------------*/
118        /** Process "put" request.*/
119        public void processPutMessage(Message msg) {
120                try {
121                        master.put(msg.getStringProperty(ReplTable.KEY), msg
122                                        .getStringProperty(ReplTable.VALUE));
123                        master.print();
124                } catch (JMSException e) {
125                        e.printStackTrace();
126                }
127        }
128
129        /*-------------------------------------------------------*/
130        /** Process "get" request.*/
131        public void processGetMessage(Message msg) {
132                try {
133                        master.get(msg.getStringProperty(ReplTable.KEY),
134                                        (fr.dyade.aaa.joram.TemporaryQueue) msg.getJMSReplyTo());
135                } catch (JMSException e) {
136                        e.printStackTrace();
137                }
138        }
139
140        /*-------------------------------------------------------*/
141        /** Process "remove" request.*/
142        public void processRemoveMessage(Message msg) {
143                try {
144                        master.remove(msg.getStringProperty(ReplTable.KEY));
145                        master.print();
146                } catch (JMSException e) {
147                        e.printStackTrace();
148                }
149        }
```

```
150
151         /*-------------------------------------------------------------*/
152         /** Process a suspition. */
153         public void processSuspitionMessage(Message msg) {
154                 try {
155                         String suspected = msg.getStringProperty(``JMS_suspect'');
156                         for (Enumeration e = GroupMessage.getSubscriptions(suspected).elements();
157                                                                 e.hasMoreElements();) {
158                                 log.info("GOT_SUSPITION_FOR_MEMBER:_" + e.nextElement());
159                         } // for
160                         // Remove the suspected member from the group
161                         log.info("REMOVING_MEMBER(S):_" + suspected);
162                         master.removeMembers(suspected);
163                 } catch (JMSException e) {
164                         // TODO Auto-generated catch block
165                         e.printStackTrace();
166                 }
167         }
168         /*-------------------------------------------------------------*/
169 } // ReplTableMsgListener
```

B.3 Client.java

```
1  package lsr.epfl.hash.active;
2
3  import javax.jms.*;
4  import javax.naming.*;
5  import java.util.logging.Logger;
6
7  /**
8   * Replicated Hashtable client. The requests to invoke on the Hashtable are
9   * supplied through the command line.
10  */
11 public class Client {
12         // variable for the logging messages
13         private static Logger log = Logger.getLogger("lsr.epfl.hash.active.Client");
14
15         // constants to mark the request messages
16         public static final String PUT = "put";
17         public static final String GET = "get";
18         public static final String REMOVE = "remove";
19
20         // variables to subscribe to the topic
21         InitialContext ictx = null;
22         TopicConnection topicConnection = null;
23         TopicSession topicSession = null;
24         Session session = null;
25         TopicPublisher topicPublisher = null;
26
27         /*-------------------------------------------------------*/
28         /**
29          * Subscribes to the GroupTopic, calls the method to invoke the specified
30          * operations.
31          */
32         public void join(String[] operations) {
33                 try {
34                         // get initial JNDI context
35                         ictx = new InitialContext();
36
37                         // the topic from JNDI
38                         Topic topic = (Topic) ictx.lookup("group_topic");
39
40                         // get the topic connection factory
41                         TopicConnectionFactory tcf = (TopicConnectionFactory) ictx
42                                         .lookup("tcf");
43                         ictx.close();
44
45                         // Create the user connection
46                         topicConnection = tcf.createTopicConnection("group_user",
47                                         "group_user");
48
49                         // Create topic session
50                         topicSession = topicConnection.createTopicSession(false,
51                                         Session.AUTO_ACKNOWLEDGE);
52
53                         // Start the connection
54                         topicConnection.start();
55
56                         // create a publischer
57                         topicPublisher = topicSession.createPublisher(topic);
58
59                         // process the specified actions
60                         if (operations.length > 0)
61                                 process(operations);
62
63                         // close the connection
64                         topicConnection.close();
65                         System.out.println("Subscription closed.");
66                 } catch (NamingException e) {
67                         e.printStackTrace();
68                 } catch (JMSException e) {
69                         e.printStackTrace();
70                 }
71         }
72
73         /*-------------------------------------------------------*/
```

```
74          /** Invoke the specified operations on the Hashtable. */
75          public void process(String[] operations) {
76
77                  if (operations[0] == null) {
78                          System.err.println("ERROR: No operation specified, exiting!!!");
79                  } // if action[0] null
80
81                  else if (operations[0].equalsIgnoreCase(Client.PUT)) {
82                          log.fine("Putting to the table: key=" + operations[1] + " value="
83                                          + operations[2]);
84                          put(operations[1], operations[2]);
85                  } // if put
86
87                  else if (operations[0].equalsIgnoreCase(Client.GET)) {
88                          log.fine("Getting the value for: key=" + operations[1]);
89                          get(operations[1]);
90                  } // if put
91
92                  else if (operations[0].equalsIgnoreCase(Client.REMOVE)) {
93                          log.fine("Removing the key=" + operations[1]);
94                          remove(operations[1]);
95                  } // if remove
96
97          } // process
98
99          /*-------------------------------------------------------*/
100         /** Invokes a "put" operation on the Hashtable. */
101         public void put(String key, String value) {
102                 log.info("Putting key=" + key + " value=" + value);
103                 try {
104                         javax.jms.TextMessage msg = topicSession.createTextMessage();
105                         msg.setStringProperty(ReplTable.PUT_PROPERTY, "");
106                         msg.setStringProperty(ReplTable.KEY, key);
107                         msg.setStringProperty(ReplTable.VALUE, value);
108                         topicPublisher.publish(msg);
109                         msg.clearProperties();
110                 } catch (JMSException e) {
111                         e.printStackTrace();
112                 } // catch
113
114         }
115
116         /*-------------------------------------------------------*/
117         /** Invokes a "get" operation on the Hashtable. */
118         public void get(String key) {
119
120                 log.info("Getting the value for key=" + key);
121                 try {
122                         javax.jms.TextMessage msg = topicSession.createTextMessage();
123                         msg.setStringProperty(ReplTable.GET_PROPERTY, "");
124                         msg.setStringProperty(ReplTable.KEY, key);
125
126                         Session ses = topicConnection.createSession(false,
127                                         Session.AUTO_ACKNOWLEDGE);
128                         TemporaryQueue temp_queue = ses.createTemporaryQueue();
129                         MessageConsumer answer_consumer = ses.createConsumer(temp_queue);
130
131                         msg.setJMSReplyTo(temp_queue);
132                         topicPublisher.publish(msg);
133
134                         log.fine("Waiting for the answer...");
135                         Message answer = answer_consumer.receive();
136                         if (answer.propertyExists(ReplTable.REPLY_PROPERTY)) {
137                                 String value = answer.getStringProperty(ReplTable.VALUE);
138                                 log.info("Answer received -- key=" + key + " value=" + value);
139                         } // if
140
141                         answer_consumer.close();
142                         temp_queue.delete();
143
144                 } catch (JMSException e) {
145                         e.printStackTrace();
146                 } // catch
147
148         }
149
```

```
150         /*-------------------------------------------------------*/
151         /** Invokes a "remove" operation on the Hashtable. */
152         public void remove(String key) {
153
154                 log.info("Removing key=" + key);
155                 try {
156                         javax.jms.TextMessage msg = topicSession.createTextMessage();
157                         msg.setStringProperty(ReplTable.REMOVE_PROPERTY, "");
158                         msg.setStringProperty(ReplTable.KEY, key);
159                         topicPublisher.publish(msg);
160                 } catch (JMSException e) {
161                         e.printStackTrace();
162                 } // catch
163
164         } // remove
165
166         /*-------------------------------------------------------*/
167         /** The execution entry method. */
168         public static void main(String[] args) throws Exception {
169                 Client p = new Client();
170                 p.join(args);
171
172         }
173         /*-------------------------------------------------------*/
174 } // Client
```

List of Publications

[1] A. Kupšys, R. Ekwall. Architectural Issues of JMS Compliant Group Communication. In *Proceedings of the 4th IEEE International Symposium on Network Computing and Applications, 2005. (NCA 2005)*, Cambridge, MA, USA, Jul. 2005

[2] A. Kupšys, S. Pleisch, A. Schiper, M. Wiesmann. JMSGroups: Towards JMS-Compliant Group Communication. In *Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications, 2004. (NCA 2004)*, Cambridge, MA, USA, Aug. 2004

[3] S. Pleisch, A. Kupšys, A. Schiper. Preventing orphan requests in the context of replicated invocation. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, (SRDS 2003), Florence, Italy, Oct. 2003

Curriculum Vitae

Arnas Kupšys was born in 1976 in Lithuania. He graduated from high school in 1994, at the same time finishing a correspondence school in physics. From 1994 to 1998 he studied computer science and obtained B.Sc. degree at Kaunas University of Technology, Lithuania. After one year of master studies he enrolled for a Pre-Doctoral school in Computer Science at École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, which he finished in 2000. Since 2000 he has been working as a research assistant, and since 2002, was a Ph.D. student in the Distributed Systems Laboratory (LSR-EPFL), under the supervision of Prof. André Schiper. From 2003 he was also a system administrator at LSR.