Architectural Issues of JMS Compliant Group Communication

Arnas Kupšys Richard Ekwall
École Polytechnique Fédérale de Lausanne (EPFL)
CH-1015 Lausanne, Switzerland
{arnas.kupsys, nilsrichard.ekwall}@epfl.ch

Abstract

Group communication provides one-to-many *communication primitives that simplify the development of highly available services. Despite advances in research and numerous prototypes, group communication stays confined to small niches. To facilitate the acceptance of group communication by a larger community, a new specification and API, called* JMSGroups*, based on the popular Java Message Service (JMS) has previously been presented.*

As a follow-up, this paper focuses on the architectural issues of the JMSGroups implementation. We consider an implementation based on a JMS server, i.e., a JMS server that is modified internally to provide a group communication service. Usually JMS server is implemented as a single entity providing its service to numerous clients. However, single server architecture is exposed to failures and is not suitable for group communication. To address this problem, we discuss the issues related to the JMS server replication (first without providing group communication). Different replicated architecture options are presented and compared. Finally, we show how to construct a fault-tolerant JMSGroups system, by extending the replicated JMS server with a group communication service.

1. Introduction

Group communication (denoted simply by GC hereafter) provides *one-to-many* communication primitives with various semantics (e.g., reliable delivery of messages and/or delivery of messages in total order). These high-level communication abstractions among groups of processes greatly simplify the development of highly available services (through replication). Yet, despite tremendous advances in research and numerous prototypes, e.g., [3, 7, 8, 5, 4], GC stays confined to small niches and to academic prototypes. We believe that the lack of a well-defined and easily understandable standard is the reason that hinders the deployment of group communication systems.

In [1] 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. As such, it facilitates the acceptance of group communication by a larger community and provides a powerful environment for building fault-tolerant applications.

As a follow-up, this paper focuses on the architectural issues of the JMSGroups implementation. Two main architecture types have been 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 an 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. The non-centralized architecture has been well studied in the group communication context [3, 9, 7, 14, 8, 5, 4] and will not be further developed in this paper. Rather, 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 the JMS. The specification of such a modification was presented in [1]. 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. Since GC is used to provide fault-tolerance to the application, a single point of failure in its architecture is not acceptable. Therefore, the JMS server used for the implementation of JMSGroups must be fault tolerant, i.e., replicated.

For the sake of clarity, the presentation of the JMSGroups architectural issues 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).

The detailed paper structure is the following: Section 2 gives a very brief introduction on JMS and Section 3 presents the GC system models we will consider. The contribution of the paper lies mainly in Sections 4 and 5. Section 4 presents the architecture types for the fault tolerant JMS server, whereas Section 5 analyzes how the replicated JMS server should be modified to provide a group communication service. Performance comparison between the different architecture types is given in Section 6. Related work is then presented in Section 7 and finally Section 8 concludes the paper.

2. Java Message Service

2.1. The architecture

The Java Message Service (JMS) [6] is a part of Sun Microsystem's Java 2 Enterprise Edition [12]; it is a set of interfaces and associated semantics that govern the access to messaging systems. The basic architecture is shown in Figure 1. As a central part of the architecture is the JMS server, which generally acts as a hub for all communications, and has access to stable storage. The clients communicate by exchanging messages which are relayed by the server.

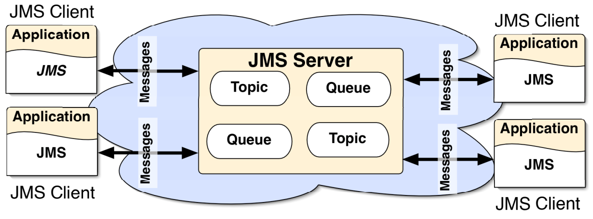


Figure 1. Basic JMS architecture.

The basic communication schema between the JMS client and server is shown in Figure 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 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.

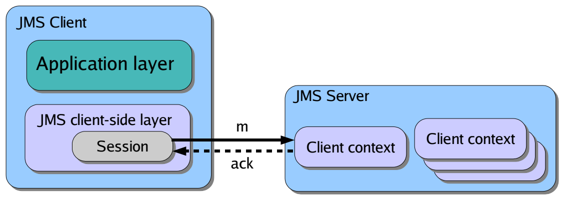


Figure 2. JMS client-server communication.

Figure 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[1]. 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 2).

2.2. JMS communication paradigms

Two communication paradigms are defined in the JMS specification: *point-to-point* and *publish-subscribe*. In point-to-point messaging, a message is sent by a JMS client to a specified *message queue*, from which it is extracted (received) by another JMS client. Hence, the message sent to a message queue is received by only one client. In contrast, publish-subscribe messaging provides one-to-many communication and is based on the concept of *topics*: a message published by a JMS client to a topic is received by all JMS clients that have subscribed to that topic. Note that the publisher does not know the set of subscribers. Since this paper focuses on group communication (one-to-many communication) using JMS, we will consider only the JMS publish-subscribe paradigm.

[1]Here we mean the application level messages.

Furthermore, JMS specifies two types of subscriptions to a topic: *non-durable* and *durable*. Consider a topic to which a client has subscribed. With a non-durable subscription the client receives messages published to the topic as long as its connection to the server is active. The connection can break (i.e., become inactive), for example because of a link failure, or because of the crash of the client. Messages published after the connection is broken are not guaranteed to be received by the client.[2] In contrast, durable subscriptions mask these failures. Indeed, the client is ensured to receive all messages that have been published to the topic it has subscribed to, even if its connection is not permanently active. During the periods when a client with durable subscription is not connected, JMS server keeps the messages for it and dispatches them as soon as the client subscribes again. In this paper, we only consider *durable* subscriptions to a topic (which are required in order to be able to provide the GC message delivery guarantees).

2.3. JMS message delivery requirements

The JMS specification [6] also defines the order of message delivery on the clients. Essentially, JMS guarantees FIFO ordering on messages that are sent between two client sessions: messages that are sent by a session must be received in the order in which they were sent. [3] However, JMS does not define the order of message receipt across several clients (the message delivery order can be m_1, m_2 on one client session and m_2, m_1 on another one if the senders of m_1 and m_2 aren't the same).

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. This ambiguity will be illustrated later in the paper.

3. Group communication system models

The previous section shortly presented JMS. In order to present the architecture of a replicated JMS server (to which the group communication service will later be added), we need to define the system models that we consider. The following system models, originally defined in the context of group communication, also apply to the replicated JMS server architecture.

[2]If the connection is broken, the client can try to re-subscribe to the topic. Let us assume that the connection breaks 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.

[3]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.

Group communication provides one-to-many communication primitives to a set of processes organized in a group. Groups can be classified into two categories: *static* and *dynamic* groups. In the case of a static group, all processes are started at system initialization. Furthermore, the group membership remains unchanged during the lifetime of the system. Processes that crash cannot be replaced by new processes.

In a dynamic group, processes can join and leave the group. The group membership can thus evolve during the lifetime of the system. Moreover, processes can start at any time (i.e. even after the system initialization) and can invoke a *join* operation to join an ongoing computation. In the dynamic group model, processes that have crashed can be replaced by new processes.

The classification into static or dynamic group category captures one dimension of the group communication system model. The second dimension relates to access to stable storage. In a system where processes do not have access to stable storage, a process crash results in the loss of the process' state (which is stored in volatile storage). In such a case, processes are said *not to recover* after a crash. We call this model the *crash-stop* failure model: a process that crashes never recovers (at least not with the same identity).

If the processes have access to stable storage, they can periodically save their state. This in turn allows a process to recover after a crash, by using the most recently saved state. This model is called the *crash-recovery* failure model: a process that crashes eventually recovers.

These two dimensions (the membership and failure models) lead to four different system models. Two of these system models will be considered later in this paper: (1) the static membership system with process recovery and (2) the dynamic membership system without process recovery. Both models can be used for replication: model (1) is used when the set of replicas does not change over time (in which case it is essential for a replica to be able to recover in case of a crash) and model (2) is used when replicas do not recover after a crash (in which case it is essential to be able to add new replicas to replace the crashed ones).

4. Fault tolerant JMS server architecture

As stated in the introduction 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 the JMS. It is possible to build such a service by internally modifying an existing JMS server and adding GC as an additional service. However, to implement JMS compliant group communication, a fault-tolerant JMS server is needed.

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 render it fault tolerant. 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.

A typical example of a replicated JMS server architecture is shown in Figure 3. 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, 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. When all server replicas have received m it can be dispatched to the subscribers of T.

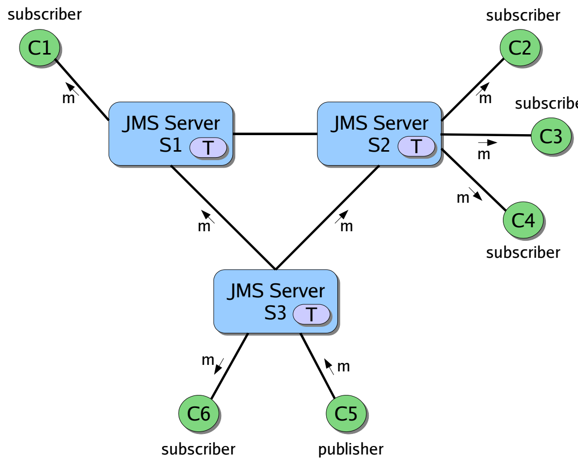


Figure 3. Replicated JMS server.

The JMS server replication should 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 requirements. 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 paper.

As already mentioned, when the JMS client connects to the server, a client context for that connection is created on the server (see Figure 2). Depending on how the client context is managed on the replicated server we distinguish two JMS server replication types: (1) server replication with *non-replicated context* and (2) server replication with *replicated context*. In case (1), which is illustrated in Figure 4(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 4(b), each client's context is replicated on all server replicas and their state is kept consistent. These two JMS server replication types are presented in detail in the following paragraphs.

4.1. Non-replicated context

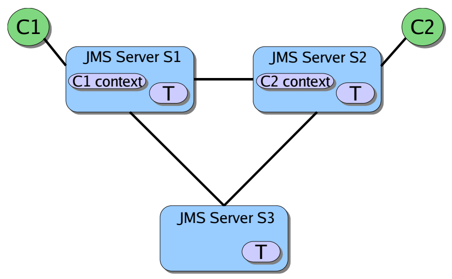
In JMS server replication with non-replicated context, each client chooses one server replica to connect to and receives the requested messaging service from it. The client context is created only on the server replica to which the client connects, and it is not shared between the other server replicas. Thus each server replica hosts only a subset of the client contexts in the system (see Figure 4(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. The solution to this problem is the recovery of the crashed server replica. After the recovery, the clients can reconnect to the same server replica and continue to receive the service. Stable storage must be used on the server replica to prevent the context loss in case of a crash. Server replica recovery and stable storage are not specific requirements for the replicated JMS server as they are defined in the JMS specification.

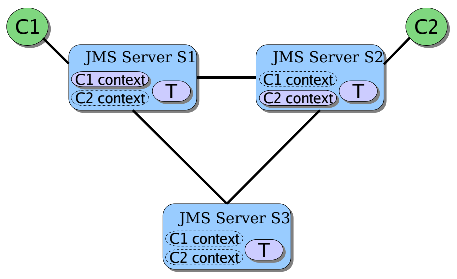
In the crash-recovery failure model, the crashed server replicas are not removed from the group (they recover with the same identity). Therefore, the static group membership model can be used for the JMS server replication with non-replicated context.

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



(a) Non-replicated client context



(b) Replicated client context

Figure 4. Different JMS server replication types.

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 replicas have to store 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 2.3, 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. 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 the 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. □

4.2. Replicated context

In the server replication with replicated context architecture, each client 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 its replica state is kept consistent during the system execution (see Figure 4(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 4(b)). The same client's contexts on the other server replicas are not active (shown with dashed line in Figure 4(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 this 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.[4]

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

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, as the client contexts (including messages) are replicated on the server and are not lost. Without such an assumption, stable storage and the crash-recovery failure model must be used for the server replicas.

For the replicated context, the client context states 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. 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 of the messages 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 order on 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. 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): 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 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.

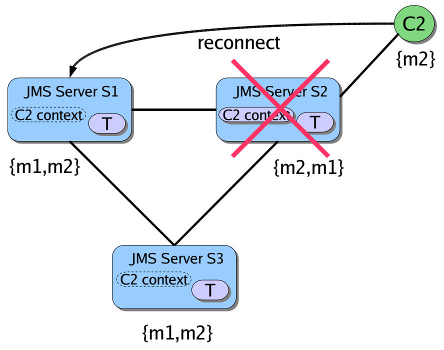


Figure 5. Client reconnection scenario.

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 2.3). $C2$ 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.

4.3. Comparison of the JMS server replication types

Table 1 presents the comparison between the non-replicated context and replicated context solutions. The replicated context solution 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 in systems with a large number of clients.

For such a system, a server with a non-replicated context can use load balancing by distributing the clients between the server replicas. Moreover, the static group membership model used by the non-replicated context solution is simpler and easier to implement than the dynamic one used by the replicated context solution. However, the non-replicated context solution requires server replica recovery which in general is more difficult to implement, but is required in the JMS specification and is 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 solution.

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

5. JMSGroups based on JMS server

Our JMSGroups specification [1] can be implemented using an existing JMS server: 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 which 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 ones 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. First, we define the two replication levels that exist in JMSGroups.

Two level replication. The replicated JMSGroups server providing GC as a service used for replication by its clients forms a system with two replication levels: (1) a *server replication level* and (2) an *application replication level* (see Figure 6). The server replication level is responsible for the server replication and the application level for the application replication respectively. Each layer uses a separate group communication 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 contains a single group (the one of the server replicas), whereas the application level supports many groups, as well as standalone clients (see Figure 6).

Both layers define separate sets of 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 for which layer 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.

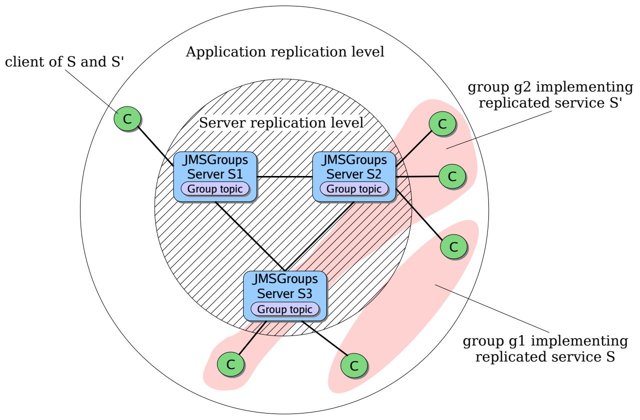


Figure 6. Two replication levels in JMSGroups.

Server replication types. In Section 4 we presented two different types of 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,

Table 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 does not crash
Number of client contexts in the system	$\lvert Clients \rvert$	$\lvert Clients \rvert \times \lvert Server\ replicas \rvert$

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, 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 composed of the primitives from both levels (see Figure 7). 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 4 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.

```
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 7. JMSGroups member's composite total order communication primitives.

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

Proof. 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. So to guarantee the total message delivery order for the group members, total order of message delivery is necessary between the server replicas. □

6. Performance comparison

We have implemented a JMSGroups prototype based on the open source JMS server JORAM v3.6 [10]. For the server replication we used *Fortika* group communication protocol stack [11], which was developed in our laboratory.

The hardware used to compare the performance was a Linux cluster consisting of nine PCs. For communication the cluster was using its private isolated LAN interconnected by a 100 Base-TX duplex Ethernet hub. Each machine was equipped with Intel Pentium III CPU running at 730 MHz and 128 MB of RAM. All machines were running RedHat Linux 7.2, kernel v2.4.18-19.7.x. The Java Virtual Machine we used was Sun's SDK v1.4.1_01.

As a performance benchmark we used a replicated JMSGroups server consisting of three replicas on three different machines. Group members (subscribers) were equally distributed on the other six machines. In each case there was one dedicated publisher which was sending the messages to the group, but was not a part of the group. We measured the actual throughput of the published messages compared with the group size.

The performance comparison between the non-replicated context and replicated context solutions is given in Figure 8. Clearly the non-replicated context scales better when the group size increases, as it does not have to maintain the additional contexts for each client. Which is the case for the replicated one.

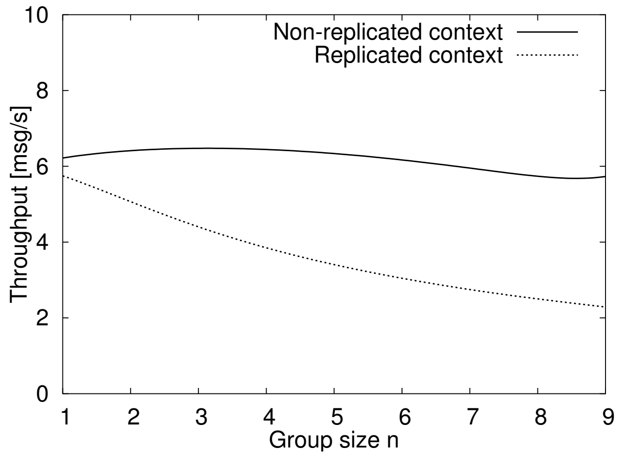


Figure 8. Performance comparison.

However, the actual throughput is rather poor for both replication solutions. One reason for that, we think, is the long and complicated message path through the internal JORAM server structure. The second reason, for the low throughput, is not full integration with Fortika toolkit (some task were processed twice: on JORAM and on Fortika). As a result, we are currently developing a lightweight JMSGroups server, which will provide only a partial JMS implementation, but will be better integrated with our group communication toolkit.

7. Related work

The open source JMS implementation called JORAM [10] provides high availability for the JMS server by replication as an option. 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 4.2. 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 4.1. 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

SonicMQ is a commercial JMS implementation from Sonic Software Corporation and also provides a replicated JMS service [13]. 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 4.2 can be used. However, unlike in the distributed 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.

8. Conclusions

JMSGroups provides a JMS compliant group communication, its specification and API were defined in [1]. As a follow-up, this paper 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 a 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. 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 this paper 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.

Acknowledgements. We would like to thank André Schiper for his useful feedback during the preparation of this paper, and Pankaj Goyal for the implementation of the replicated context architecture type using JORAM.

References

[1] A. Kupšys, S. Pleisch, A. Schiper, M. Wiesmann. Towards JMS Compliant Group Communication a semantic mapping. In *Proceedings of the Third IEEE International Symposium on Network Computing and Applications (NCA 2004)*, pages 131 – 140, Aug 2004.

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

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

[4] 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.

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

[6] 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.

[7] C. P. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phœnix: 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, 1995. IEEE. Workshop held during the 7^{th} Symp. on Parallel and Distributed Processing, (SPDP-7).

[8] 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.

[9] 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.

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

[11] Sergio Mena, Xavier Cuvellier, Christophe Grégoire, and André Schiper. Appia *vs.* Cactus: Comparing protocol composition frameworks. In *22nd Symposium on Reliable Distributed Systems. Florence, Italy*, Oct. 2003.

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

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

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