An Implementation of Reliable Group Communication based on the Peer-to-Peer Network JXTA

Karsten Loesing and Guido Wirtz
Distributed and Mobile Systems Group
Otto-Friedrich-Universität Bamberg
Feldkirchenstraße 21, 96052 Bamberg, GERMANY
{karsten.loesing|guido.wirtz}@wiai.uni-bamberg.de
3rd June 2004

Abstract

We propose a completely distributed peer-to-peer protocol for atomic reliable multicast based on view consistency. We implemented our protocol using the peer-to-peer middleware JXTA for use in TCP/IP-connected networks. This implementation offers its service to other applications via a Java API. Typical applications are those requiring group communication semantics without relying on dedicated, fixed servers.

1 Introduction

Group communication [TvS02] is a communication paradigm in which groups of processes can reliably communicate with each other. It stands in contrast to point-to-point communication which is inherently restricted to two processes. Reliable group communication guarantees that messages are delivered either to all members in a process group, or to none of them. There are applications which require reliable group communication, e.g. drawing on a shared whiteboard or application sharing [BAM+01]. Group communication middleware often is heavy-weighted software relying on dedicated servers.

Peer-to-peer networks have evolved because of the need for decentralized services [MKL+02]. Dedicated servers are a single point of failure which certain services have
to avoid. In peer-to-peer networks availability of service is enhanced by distributing it
to a great number of nodes which are not assumed to fail all at the same time. Though
single nodes may fail or leave the network the service remains available.

In this paper we specify a group communication protocol for peer-to-peer networks.
It was designed to face arbitrary node and communication failures without relying on
certain server nodes. Further we discuss an implementation of this protocol based on the
peer-to-peer middleware JXTA [JXT03a]. This prototype was designed as a Java API
and may be used by other applications. Both our protocol and JXTA are based on XML
so that future implementations might cover other platforms as well.

The rest of the paper is organized as follows. Section 2 describes our protocol and
the messages used by it. Therefore we discuss possible situations which may occur in a
distributed, decentralized system and present solutions to keep up view consistency. The
third section describes the implementation of our protocol and points out some JXTA-
specific implementation details. Section 4 relates our approach to other work, and section
5 points out some possible directions for future work.

2 Protocol specification

Before going into the details of our group communication protocol we define what re-
liable group communication means. Then, we point out the requirements our protocol
is assumed to meet. Next, we present the parts of the protocol needed to achieve view
consistency. Finally, we describe how atomic multicast messages are sent and delivered
within a group.

2.1 Definition of reliable group communication

Group communication systems introduce an additional layer between application and net-
work. Reliable multicast messages are sent to and delivered by the group communication
system which in turn sends and receives (non-reliable) messages to/from the network. Reliable group communication guarantees that a reliable multicast message is either delivered to all members of a group or—in case of node or network failures—to none of them. Figure 1 shows a possible situation in a group of three processes: Message $M_1$ may be delivered, because all processes in the group have received it. In contrast, $M_2$ was corrupted and not correctly received by $P_2$, and may not yet be delivered by $P_1$ and $P_3$.

![Figure 1: Messages are not delivered to the application until all processes in the group have received them: While $M_1$ may be delivered, $M_2$ has to be queued until $P_2$ has correctly received it.](image)

The so-called view is essential for this kind of semantics. It contains a set of current members in a group. Before any message can be sent, all members of a group have to agree on a common view. Whenever membership changes occur a new view has to be installed. A message is called safe when all members of a view have received it. It is not allowed to be delivered to the application before it is safe. Atomic multicast further requires all messages to be delivered in total order to all processes in the group. For a comprehensive overview of group communication middleware see [CKV01].

### 2.2 Requirements

In order to provide reliable group communication in a peer-to-peer network we propose a new protocol which fits the peer-to-peer model. It explicitly avoids servers or central nodes and distributes responsibility of service among all peers in a group. Due to the transient nature of peer-to-peer networks the protocol has to face sudden node and communication failures.
The underlying platform is required to support best-effort, bidirectional unicast communication between any two processes and best-effort multicast communication between any processes in the network. Though multicast communication also can be replaced by sending unicast messages to all members in a group, the multicast ability is at least needed in order to bootstrap the system when there is no known group member. The network is assumed to be asynchronous, i.e. there is no upper bound for message delivery.

2.3 View consistency

A consistent view is required for reliable multicast messaging. In this subsection we consider possible situations which may occur in a group of processes with regard to joining, leaving, and crashing processes, and describe how our protocol keeps up a consistent view between the processes in a group.

2.3.1 Announcing presence

On startup a process has to become aware of other processes in the group. Therefore new processes send out a HELLO message using (unreliable) multicast to announce their presence in the group. Apart from this, HELLO messages are used to reunite subgroups which have evolved due to network partitions.

2.3.2 View change

If the group membership changes due to joining, leaving or crashing processes, the members of the group have to agree on the next view. Therefore a couple of messages have to be exchanged to make sure that the consensus concerning the new view is accepted by all peers. The view change procedure is initiated by a VIEW_CHNG message sent by the process which notices the necessity of changing the current view. This message contains a so-called proposal which in turn consists of a unique ID and the proposed view. On receiving a VIEW_CHNG message every member of the proposed view interrupts com-
munication in the old view, stores the attached proposal and acknowledges the arrival of
the VIEW\_CHNG message by sending a VIEW\_CHNG\_ACK message to the proposing
peer. It cannot immediately install the view as other members of the view might already
have crashed or disconnected.

The proposing process collects all acknowledgements to assure that all other processes
have agreed to the proposal. Now the proposed view can be installed. Therefore the
proposing process sends out a VIEW\_INST message. The VIEW\_INST message again is
acknowledged by VIEW\_INST\_ACK messages to assure that all peers have installed this
view.

2.3.3 Concurrent view changes

The simplified procedure as described above only works, if there are no conflicting propos-
als. These might lead to inconsistent views, e.g. if two processes acknowledge two distinct
proposals before installing one of them, they might install different but overlapping views
in a single group which is not allowed. Therefore concurrent proposals should first of all
be prevented or at least have to be resolved if they occur.

As an example for an unresolved conflict see figure 2. Two processes $D$ and $E$ attempt
to join the same group with view $ABC$ at the same time. The first HELLO message is
broadcasted by $A$. $D$ and $E$ in turn start two proposals $ABCD$ and $ABCE$ respectively
and send them to $A$, $B$, and $C$. By accident $D$’s message to $C$ is delayed or lost and
resent later, such that $D$ only receives two VIEW\_CHNG\_ACK messages by $A$ and $B$. $E$
receives all three acknowledgement messages and forces the installation of $ABCE$. Soon
after $D$ receives the acknowledge message by $C$ it forces the installation of $ABCD$. At
the end $A$, $B$, $C$, and $D$ reside in view $ABCD$ while $E$ still remains in $ABCE$, which is
not allowed.

Preventing concurrent proposals is done by allowing as few processes as necessary in
Figure 2: Sequence of exchanged messages in case of an unresolved view change conflict. White arrows denote messages which are disposed directly after receiving them.

a group to make a new proposal. Any process in a group as well as processes attempting to join a group might notice the necessity of a view change, possibly over a short period of time before any message can be exchanged. If all of them would send VIEW_CHNG messages simultaneously, a lot of conflicts would arise. This is why only selected processes may start changing a view, e.g. only the process with highest ID in a group may answer a HELLO message and add new processes to the group. Such rules can only succeed, if processes know about each other, i.e. reside in the same group, which is not the case, if new processes join or network partitions heal. Hence, prevention may reduce the problem of conflicts, but does not solve it completely.

If two or more view changes are initiated concurrently, the conflict has to be resolved. This is done by attaching information about previously received proposals on VIEW_CHNG_ACK messages in form of so-called proposal sets. These sets also include proposals made by a process itself. After collecting all responses to a proposal a process can decide locally whether there is a conflict and how to handle it. Three situations are possible: A proposing process might receive only empty proposal sets (1), only non-empty sets (2), or a mixture of empty and non-empty sets (3):
1. If all sets are empty, there is no conflict and the view can be installed.

2. If none of the sets is empty, one or more other processes must have made proposals before. Another process might already have collected only empty proposal sets and have started installing the view due to (1). Thus a process receiving only non-empty proposal sets has to abort its proposal, return to an initial view and wait for the other process(es) to resolve this conflict.

A special case arises, if two or more processes propose the same view at the same time. Then it also can happen that both/all of them only receive non-empty proposal sets. As a result both/all of them would stop proposing which might possibly end up in a life lock. As a solution the peer with higher ID then is allowed to continue its proposal while the others abort their proposals.

3. If a process receives both empty and non-empty sets, at least one other process is proposing at the same time. This process must have received at least one empty and possibly a couple of non-empty sets. If all processes would stop their proposals, a life lock was possible. So, there must be found a solution that allows only one process to continue its proposal and forces the other processes to abort proposing. Therefore only the process with highest ID is permitted to continue its proposal. Since this process has received at least one empty set from a process $X$, other processes must have received non-empty sets from $X$ and aborted their proposals due to (2) or (3).

If a proposing process notices that other proposals are running and is allowed to continue its proposal, it combines all proposals it is aware of and proposes the combined view. Therefore a proposal does not only contain a list of processes which are a member of the next view, but also a list of processes which have left since the last view in order to avoid ambiguity. E.g. two proposals $ABC$ and $BCD$ can be combined to $ABCD$, if $A$ and $D$ are joining $BC$, or result in $BC$, if $A$ and $D$ are leaving the group.
Whenever a process notices that it cannot succeed its proposal due to (2) or (3), it sends a VIEW\_CHNG\_FAIL message. Other processes can then clean up their local proposal sets which accelerates consensus for following proposals.

Figure 3 demonstrates how a conflict is resolved using the rules stated above. The situation is similar to the situation in figure 2 up to the point when E decides to force the installation of ABCE. Since E has received both empty (C) and non-empty sets (A and B) rule (3) has to be applied. In the example E is assumed to have a lower ID than D, hence it stops its proposal. D learns about the proposal of D from the VIEW\_CHNG\_ACK message by C which was delayed. It then combines both proposals to ABCDE and proposes it.

Figure 3: Sequence of exchanged messages evoked by a view change conflict which is resolved by combining proposals. White arrows denote messages which are disposed directly after receiving them.

2.3.4 Failure detection

The multicast service requires all peers of a view to be alive and connected. In order to assure that process and network failures are discovered a periodic ping mechanism is introduced. Therefore all processes in a view form a ring in the order of their IDs. Every process periodically notifies its neighbor that it is still alive by sending an ALIVE
message. With the use of timeouts failures are detected and new views may be proposed. In order to compensate single message losses PING messages are introduced, by which processes can explicitly request an ALIVE message. Further all messages are resent based on a heuristic, e.g. for three times, until a process is suspected to have become unavailable and is removed from the view.

2.3.5 Leaving a group

If a process leaves the group, it sends a LEAVE message to its neighbor before stopping communication. The neighbor then proposes a new view without the leaving process.

2.4 Atomic multicast messages

Atomic multicast messages have to be reliably delivered to all processes in a group in the same order. As a special case this has to be assured considering arbitrary view changes, too. Reliable message delivery in these cases as well as total ordering of messages is described in the following.

2.4.1 Multicast messaging in a stable view

Up to this point the protocol allows processes to agree on a common view which is the prerequisite for sending atomic multicast messages. Once a view is installed the application may send a multicast message to the group. In order to guarantee safe delivery of this message to all processes in the group, multiple message rounds are necessary. First, a MSG message is sent via multicast to the group and answered by a MSG-ACK message. After all acknowledgement messages have been received the sending peer may deliver the message to the application and send a SAFE message to the group which is in turn acknowledged by a SAFE-ACK message. The other processes may not deliver the message to the application layer until they receive a SAFE message.
2.4.2 View change while multicast operation proceeds

In case that the view changes while a multicast operation is performed, it has to be made sure that the message is delivered either to none of the processes, or to all of them. I.e. if only one process which is about to install the next view has delivered a message, all processes in the old view have to do this. Therefore every process stores the information about those messages it has delivered to the application. When a new view is proposed every process attaches this information to the VIEW.CHNG.ACK message. The proposing process collects this information, combines it and attaches it to the VIEW.INST message.

2.4.3 Total order

Atomic multicast prescribes that multicast messages have to be totally ordered. This is performed by the algorithm used in the ISIS toolkit as described in [CDK01] which is only sketched in the following. Every peer maintains a message queue in which it stores received messages until they are declared safe and may be delivered due to their message sequence number. Any process acknowledging the receipt of a MSG message via a MSG.ACK message proposes its own sequence number for this message. The sending process then collects all proposed sequence numbers and determines an agreed sequence number. This is transmitted with the SAFE message so that all processes receive the same agreed sequence number for a message. Messages in the message queue are ordered by their sequence numbers. They may only be delivered, if they have been declared safe and have the lowest sequence number in the queue.

Table 1 contains an overview of all message types and their purpose in our protocol.
### Message types and their purposes

<table>
<thead>
<tr>
<th>Message type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>HELLO</td>
<td>Announce presence in group</td>
</tr>
<tr>
<td>VIEW_CHNG</td>
<td>Initiate view change by proposing a new view</td>
</tr>
<tr>
<td>VIEW_CHNG_ACK</td>
<td>Acknowledge proposal</td>
</tr>
<tr>
<td>VIEW_INST</td>
<td>Force installation of previously proposed view</td>
</tr>
<tr>
<td>VIEW_INST_ACK</td>
<td>Acknowledge view installation</td>
</tr>
<tr>
<td>VIEW_CHNG_FAIL</td>
<td>Notify that proposal cannot succeed</td>
</tr>
<tr>
<td>ALIVE</td>
<td>Inform that process is still running</td>
</tr>
<tr>
<td>PING</td>
<td>Explicitly request proof of live</td>
</tr>
<tr>
<td>LEAVE</td>
<td>Declare intention to leave group</td>
</tr>
<tr>
<td>MSG</td>
<td>Send application message to group</td>
</tr>
<tr>
<td>MSG_ACK</td>
<td>Acknowledge application message</td>
</tr>
<tr>
<td>SAFE</td>
<td>Force delivery of previously sent application message</td>
</tr>
<tr>
<td>SAFE_ACK</td>
<td>Acknowledge delivery of a message</td>
</tr>
</tbody>
</table>

Table 1: Message types and their purposes.

### 3 Protocol implementation

In the following we sketch the architecture of our protocol implementation. Then, we describe the connection to the JXTA API including details about JXTA-specific implementation issues. Next, we state how JXTA could be exchanged by another communication platform. Finally, we show how to use our group communication API in an application.

#### 3.1 Implementation overview

Our implementation is divided into communication and service layer. The communication layer is responsible for the interaction with the network, hiding JXTA details from the service layer which actually implements our protocol. The communication layer performs three tasks: It manages peer group membership, provides means for communication between peers, and encapsulates service-specific data structures for exchange via the JXTA Network. Figure 4 contains an overview of the packages.
Figure 4: The two-layered architecture splits the implementation into service layer which is used by the application and communication layer which accesses the communication platform.

3.2 Binding to JXTA

The open-source project JXTA [JXT03b, TAA+03] is a common peer-to-peer middleware for application developers. The JXTA virtual network creates an overlay network of peers on top of an IP-based network using TCP/IP or HTTP over TCP/IP. This network abstracts from obstacles like firewalls or NAT routers and provides direct connectivity for unreliable unicast and multicast communication. Further it has a built-in mechanism for managing peer groups. We decided to use JXTA as the basis for our group communication implementation as it solves a couple of problems, e.g. direct peer connectivity. But we left open the architecture so that other platforms might be added in the future.

3.2.1 Group management

JXTA introduces the concept of peer groups to form subsets of peers which share a common interest. The main motivation for this is to establish virtual domains for applications.
In addition to this we use peer groups to separate distinct groups in our group communication system. At a later stage secure peer groups might be applied to provide a secure group communication service.

Peer groups can be created, discovered, and joined by the means of Peer Group Advertisements. Like other resources Advertisements are locally created by a peer and published using JXTA’s Discovery Service. Other peers then can discover Peer Group Advertisements and join the belonging peer groups. Removing a peer group from the network is not possible, but a lifetime can be set after which the corresponding Advertisement expires.

It is important to notice that joining a JXTA peer group is only a precondition for being a member of the group, but it is not sufficient. Group membership—in the sense of view membership—is managed exclusively by the group communication protocol which is implemented in the service layer.

3.2.2 Peer communication

Our protocol requires processes to communicate via unicast and multicast messages. In JXTA communication between peers can be performed by using so-called Pipes. There exist Unicast Pipes for point-to-point communication as well as Propagate Pipes for multicast communication.

However, we do not make use of JXTA Pipes. In so doing we avoid to publish and discover Pipe Advertisements for every peer in a group. Every peer would have to manage 1 Propagate Pipe and \( n - 1 \) Unicast Pipes in a group of \( n \) processes which does not seem to be practicable. Instead of Pipes we directly use the two JXTA services which perform message exchange, namely the Endpoint and Rendezvous Services. The Endpoint Service allows any peer to discover a route (sequence of hops) to another peer. The Rendezvous Service propagates messages in a group. This is done by establishing a network of Rendezvous Peers which transmit messages among themselves and to attached
edge peers. Both Propagate and Unicast Pipes base on the Endpoint and Rendezvous Service as well.

3.2.3 Data structures

The data structures used in our protocol, e.g. view or proposal, have to be transmitted using JXTA messages. Therefore all data structures used in the service layer contain two methods for the conversion from/to XML representation. The XML strings then can be attached to JXTA messages.

3.2.4 Comments on working with JXTA

Though JXTA promises to simplify the development of peer-to-peer applications and services, we faced a couple of problems when implementing our service. As an example the Rendezvous Service used to propagate messages in a group comprises a serious inconvenience: A peer may either act as Rendezvous, or as Edge Peer whereas only Edge Peers try to discover Rendezvous Peers. After becoming a Rendezvous Peer a peer only waits for requests by other Edge peers. Concerning peers which newly join a group it is not trivial to determine how long they should search for Rendezvous Peers until they become one themselves.

Further the JXTA Reference Implementation has a number of bugs and unimplemented parts. As an example, the method to flush all Advertisements in the local cache returns without throwing an exception indicating to the user that the processing was successful. But a look into the source code reveals that the method immediately returns without doing anything.

3.3 Replacing the communication platform

The two-layered architecture of our implementation facilitates the support of other communication platforms in the future. This can be accomplished by replacing the JXTA-
specific classes of the communication layer. A communication platform which can be used for the implementation of our protocol has to fulfill the following requirements:

- Peers are able to send directly point-to-point messages. Reliable delivery is not required.
- Peers can form groups for multicast operations. Again, delivery is not required to be reliable.
- Messages exchanged via unicast and multicast can contain arbitrary XML documents.

Figure 5 gives an overview of the classes in the communication layer. The class `GroupManagementCommunicationFactory` is used to instantiate a group manager according to a Java property stating which platform to use. The two interfaces `GroupManagementCommunicator` and `GroupCommunicator` provide methods which are invoked by the service layer. Asynchronous replies are made by invocation of methods on the two listeners `GroupManagementCommunicatorListener` and `GroupCommunicatorListener`. The class `CommunicationException` is used to notify the service about communication failures. The JXTA-specific implementation is capsulated in the two classes `GroupManagementCommunicatorImpl` and `GroupCommunicatorImpl` of the subpackage `jxta`. In order to replace the communication platform only these two new implementation classes as well as a few lines of code in the factory class have to be implemented.

Another communication platform which could be used by our protocol implementation is Proem [KSP+01]. Proem is a peer-to-peer middleware focussing on mobile ad-hoc networks rather than on Internet-scale applications. It allows the creation of so-called communities which can be compared to JXTA’s peer groups. Communication between peers is performed in a connectionless, asynchronous way by passing events. Reliability
of message delivery depends on the underlying transport protocol, e.g. TCP/IP, UDP or HTTP. Hence, Proem meets the requirements we stated above.

Further, communication also could be performed by relying on TCP or UDP directly. Though this might increase performance of our protocol, the problems of finding other peers or direct communication between nodes would have to be solved manually.

Apart from supporting the JXTA platform we implemented a separate communication platform based on Java threads. This is not supposed to be used in a production environment but rather to test and benchmark the protocol with a great number of processes in a group locally.
3.4 Using the group communication API

Our implementation simply may be plugged between the JXTA platform and a custom application. Therefore we developed a lean API which allows application developers to manage groups, send atomic multicast messages, or listen for messages from a group. No internal knowledge of the protocol is required to use this API.

Figure 6: An application using the API of our implementation interacts with these five types.

An application which uses the group communication API first creates an instance of GroupManager, respectively a class implementing this interface, using the GroupManagerFactory (see figure 6). The manager is then used to create or discover groups and to obtain a reference on a certain group. A Group may be joined or left and atomic multicast messages may be sent to it. The two listeners GroupManagerListener and GroupListener inform the application about new groups and view changes and deliver safe messages to it. Listing 7 shows an example code of the invocations which are necessary to apply our group communication service.

4 Related work

Traditional group communication systems such as ISIS [Bir86, BR94] or Totem [MMSA+96] are designed for use in a LAN environment, e.g. for replicating local
public void testGroupCommunicationAPI() {

    // start service
    GroupManager myGroupManager = GroupManagerFactory.createGroupManager(
        myGroupManagerListener, aPeerID, aPeerName);

    // determine peer id
    String peerID = myGroupManager.getPeerID();

    // lookup if group "lspi" exists
    boolean lspiExists = myGroupManager.lookupGroups().containsKey("lspi");

    // create group
    myGroupManager.createGroup("lspi", "A new group");

    // join group
    myGroup = myGroupManager.getGroup("lspi", myGroupListener);
    myGroup.joinGroup();

    // send message
    myGroup.sendMessage("This is a test.");

    // leave group
    myGroup.leaveGroup();
}

// deliver message - class implements GroupListener
public void deliverMessage(
    String message, String sendingPeerName) {

    System.out.println("Delivering message: " + message + " by " + sendingPeerName);
}

Figure 7: Example code of the invocations on the group communication API.

servers. Thus high bandwidths and low latencies are assumed, and process or network failures as well as membership changes are not supposed to occur very often. These assumptions do not meet the peer-to-peer model.

The InterGroup protocols [BAC02] make an approach to scale group communication to WAN for collaborative applications, e.g. video-conferencing. This is done by dividing a group into senders which are explicitly known and receivers of which membership is not maintained. This compromise was made to achieve better scalability and efficiency. Our approach makes similar assumptions, but maintains membership for all group members.
The P2P Messaging System [Jun03, JL02] realizes a group communication system for peer-to-peer networks based on JXTA. In contrast to our approach the system uses JXTA only to look up peers. Communication is then performed directly by using TCP or UDP. Though this approach should have better performance, direct connectivity between peers is not guaranteed.

5 Conclusion and future work

In this paper we proposed a protocol for reliable group communication for peer-to-peer networks. It focuses on the replication of services across all peers in a network and therefore avoids central servers as single points of failures.

Further we implemented a group communication API of our protocol for the development of Java applications. At the moment this prototype bases on the peer-to-peer middleware JXTA. Due to the two-layered architecture of our implementation other platforms may be applied as well.

Future work aims to measure performance of our protocol and optimize its efficiency, e.g. reduce idle times while view proposals are running. Another field is introducing security issues to the protocol, e.g. to prevent denial of service attacks which could easily be performed in the current version.

References


