Towards a Generic Group Communication Service*

Nuno Carvalho[1], José Pereira[2], and Luís Rodrigues[1]

[1] University of Lisbon
{nunomrc,ler}@di.fc.ul.pt
[2] University of Minho
jop@di.uminho.pt

Abstract. View synchronous group communication is a mature technology that greatly eases the development of reliable distributed applications by enforcing precise message delivery semantics, especially in face of faults. It is therefore found at the core of multiple widely deployed and used middleware products. Although the implementation of a group communication system is a complex task, application developers may benefit from the fact that multiple group communication toolkits are currently available and supported.

Unfortunately, each communication toolkit has a different interface, that differs from every other interface in subtile syntactic and semantic aspects. This hinders the design, implementation and maintenance of applications using group communication and forces developers to commit beforehand to a single toolkit, thus imposing a significant hurdle to portability.

In this paper we propose jGCS, a generic group communication service for Java, that specifies an interface as well as minimum semantics that allow application portability. This interface accommodates existing group communication services, enabling implementation independence. Furthermore, it provides support for the latest state-of-art mechanisms that have been proposed to improve the performance of group-based applications. To support our claims, we present and experimentally evaluate implementations of jGCS for several major group communication systems, namely, Appia, Spread/FlushSpread and JGroups, and describe the port of a large middleware product to jGCS.

1 Introduction

View synchronous group communication is a coordination paradigm that eases the development of multi-participant applications, ranging from replicated servers, cooperative caches, multi-user cooperative application, just to name a few. The set of protocols that implement group communication services are typically bundled in a package called a *group communication toolkit*. After the pioneer work initiated two decades ago with the Isis [1], many other toolkits have been developed. Appia [2], Spread [3], and JGroups [4] are, among others, some of the group communication toolkits in use today. Therefore, group communication is, today, a mature technology that, when correctly used, greatly eases the development of reliable distributed applications. At the same time, group communication is still a hot research topic, as performance improvements and wider applicability are sought [5–9].

* This work was partially supported by the IST project GORDA (FP6-IST2-004758).

For self-containment, we provide a very brief introduction to group communication. A group communication toolkit integrates two complementary services: membership and multicast communication. Informally, the role of the membership service is to provide, to each participant in a distributed computation, information about who is active (or reachable) and who is failed (or unreachable). Such information is called a *view* of the group of participants. The multicast service allows a member to send a message to the group of participants with different reliability and ordering properties. Membership and multicast need to be integrated because reliability guarantees are usually defined in the context of the current group view. For instance, if the membership service indicates that participants A, B and C are active, the reliable multicast service will deliver all multicast messages to these three participants unless one of them fails meanwhile (in which case, a new view is installed).

From the description above it is clear that group communication is much more than just “yet another reliable multicast protocol” given that, the added value, is the precise semantics that are enforced among the communication and membership services, namely in the presence of faults [10]. By ensuring that the same messages are delivered to multiple destinations, ordered among themselves and with group membership change notifications, each message may be handled by the application in a predictable and globally consistent context.

Naturally, by enforcing strong semantics, group communication is more expensive that other weaker forms of multicast, such as best-effort multicast. Therefore, it should not be used when the application has weak consistency requirements. On the other hand, group communication excels when the application is required to maintain global invariants on distributed state. Complex distributed state maintenance problems are then greatly simplified [11]. Group communication is therefore found at the core of multiple widely deployed and used middleware products, namely, the iBus//MessageBus scalable publish/subscribe toolkit, the JBoss and Zope application servers, the Sequoia (formerly Objectweb C-JDBC) database cluster, and the CORBA FT high-availability service. Group communication is also at the cornerstone of innovative research prototypes such as Postgres-R [12] and Eternal [13].

Unfortunately, each communication toolkit has a different interface, that differs from every other interface in subtle syntactic and semantic aspects. This hinders the design, implementation and maintenance of applications using group communication and forces developers to commit beforehand to a single toolkit, thus imposing a significant hurdle to portability. Such commitment is undesirable because group communication toolkits are often optimized for specific execution environments. The ability to replace one toolkit by another has the advantage of allowing the application to use the most appropriate toolkit for each deployment scenario. If the application code is tightly coupled with a particular toolkit, changing the implementation of group communication requires a costly refactoring. This also prevents emerging loosely coupled service oriented architectures from taking full benefit of view synchronous group communication.

In this paper we tackle the problem of defining a generic interface that may be used to wrap multiple toolkits. The interface, called *Group Communication Service* for Java, or simply jCGS, has been designed for the Java programming language and leverages on several design patterns that have recently become common ground of Java-based middleware. The interface specifies not only the API but also the (minimum) semantics that allow application portability. jGCS owns a number of novel features that makes it quite distinct from previous attempts to define standard group communication interfaces, namely:

- jGCS aggregates the service in several complementary interfaces, namely a *configuration interface*, a *message passing interface*, and a set of *membership interfaces*. The configuration interface specifies several opaque configuration objects that encapsulate specifications of message delivery guarantees. These are to be constructed in an implementation dependent manner to match application requirements and then supplied using some dependency injection technique. The message passing interface exposes a straightforward interface to sending and receiving byte sequences, although concerned with high throughput, low latency and sustainable concurrency models in large scale applications. Finally, a set of membership interfaces expose different membership management concepts as different interfaces, that the application might support or need.
- jGCS provides support for recent research results that improve the performance of group communication systems, namely, *semantic annotations* [9, 8, 7] and *early delivery* [14, 15, 6, 16].
- open source implementations of jGCS for several major group communication systems have been already developed, namely, Appia [2], Spread [3] (including the FlushSpread variant), and JGroups [4]. Besides making jGCS outright useful in practice, these validate that the interface is indeed generic.
- the interface introduces negligible overhead, even when the jGCS is implemented as wrapper layer and is not supported natively by the underlying toolkit.
- the interface has been already integrated in existing middleware products. In particular, we also describe here the port to jGCS of the Sequoia database clustering middleware (formerly Objectweb C-JDBC), as an illustration of the expressiveness and performance of jGCS.

As we have already noted, there were previous attempts to define generic group communication interfaces. As it will become clear after we present our work, those attempts have approached the problem from quite different, and often orthogonal angles. We will postpone comparison with related work until Section 6.

The rest of this paper is structured as follows. In Section 2 we briefly describe the goals that we propose to achieve with the presented service and the pitfalls that need to be avoided in the design of such a service. Section 3 introduces the jGCS interface. Section 4 describes existing jGCS bindings and Section 5 offer a performance evaluation of the jGCS. Section 6 compares jGCS with related work and Section 7 concludes the paper.

2 jGCS Design Goals and Pitfalls

The design of the jGCS is shaped by a number of goals that determine also a number of tradeoffs. In this section we enumerate and describe each of these goals. Furthermore, the design of the jGCS is also shaped by the need to avoid a number of pitfalls: features that would at first sight seem desirable but which in fact are not and would force unbearable compromises. In this section we discuss not only the goals that we aimed with the design of the jGCS but also the pitfalls we have avoided in the process.

2.1 Goals to Achieve

Goal 1: *No changes to payload required.* The first goal is that when the jGCS is implemented on top of an existing toolkit, no assumptions or changes are made on message payload. This means that implementing jGCS does not require specific data formats, additional message headers or additional messages exchanged. Naturally, toolkits that adopt jGCS as their native interface are free to implement jGCS-specific optimizations. As a result, applications using a specific protocol through jGCS are interoperable with legacy versions using native interfaces. Furthermore, no Java specific constructs or data formats are forced on the application, most notably, no Java serialization is required. This makes it possible to easily translate the proposed API to languages in the same family such as C++ or C#.

Goal 2: *Support service locator and dependency injection patterns.* Complex applications of group communication can make use of multiple groups and even multiple service guarantees selected independently for each message. On the other hand, different sets of guarantees might be available on each implementation of jGCS. It is therefore required that all details regarding protocol configuration and service selection are encapsulated in objects that can be supplied to the application by a third party (i.e. the configurator) using a service locator[3] [17] or the dependency injection patterns[4]. As an example, this allows substitution by a stronger service, when the exact service required by the application is not available in the target environment.

Goal 3: *Support multiple group-based programming paradigms.* jGCS is flexible enough to support different flavors of multicast communication based on process groups. Most notably, jGCS supports both open groups (where any process can send messages to the group) and closed groups (where just group members can send messages to the group). Open groups are useful in client/server applications while closed groups are useful for coordination and synchronization among servers.

Additionally, besides the more common multicast group paradigm, in which messages are targeted to all group members, jGCS supports peer groups, in which messages are target to specific members of the group. As an example, a multicast group is useful for data replication while a peer group is useful in a load balancing application. Note that both flavors require precise knowledge of current membership to function properly.

Goal 4: *Export a flexible subsetable interface.* Supporting existing toolkits requires an extensive interface. This includes facilities for sending and receiving messages according to each group communication paradigm, as well as to receive membership notifications with various degrees of detail.

[3] http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html
[4] http://www.martinfowler.com/articles/injection.html

In some scenarios, there may be an overlap among the services provided by jGCS and other services already in-use in the target system, such as best-effort reliable multicast protocols and cluster management infrastructure. Therefore, it may be useful to deploy just parts of the jGCS to avoid redundancy. Due to this reason, jGCS has been designed to be subsetable, in the sense that parts can be independently reused, without carrying along with partially implemented interfaces and runtime exceptions.

As we will see, the subsetable property is also useful to accomodate multiple view synchrony variants (however, see Pitfall 2 in below).

Goal 5: *Non-blocking input/output and container-managed concurrency.* The jGCS supports an event-driven interface. The application registers a number of callback listener interfaces to be notified of messages arriving and changes to group composition. The rationale for this is twofold. First, avoiding the requirement to have threads blocked on input/output improves scalability and allows application containers to manage threading under a single integrated policy. Second, ordering guarantees directly translate into synchronization requirements, thus allowing jGCS implementations to cooperate with application containers to optimize the number of concurrent threads.

Goal 6: *Accomodate latest research results.* Finally, it is a goal of the jGCS to allow recent research results, such as support semantic annotations [9, 8, 7] and early delivery [14, 15, 6, 16], to be easily accomodated. In fact, the goal is to foster programming idioms that naturaly take advantage of such results as they become available. Section 3 will address this topic in detail.

2.2 Pitfalls to Avoid

Pitfall 1: *Specify a common set of service guarantees.* By assuming that protocol configuration and selection of service guarantees are hard-coded within the application, portability to multiple protocol implementations can only be achieved by standardizing on a limited set of guarantees that must exist everywhere. Such agreed set is either very small, and thus of limited use, or large and not portable to multiple implementations. jGCS avoids this pitfall by assuming a configuration step as described in Goal 2, that matches available service guarantees to application requirements.

Pitfall 2: *Exclusively reuse existing standard interfaces.* The semantics of view synchronous group communication are so different from other message passing middleware, such as JMS, that any attempt to map these semantics into other paradigms introduces substantial obstacles to all goals enumerated in the previous section. Furthermore, given the semantic mismatch, is also likely that no significant portability advantages result from the exercise. A better option is to provide a syntatically incompatible interface that embodies similar structure and the same patterns such that programmers can easily make the transition.

Pitfall 3: *Provide interfaces for protocol composition.* A lot of research effort has been invested in composing group communication protocols from fine-grained components by using uniform interfaces [18, 19, 2] or even standard ad-hoc interfaces [20]. The main problem is that the mapping of an existing implementation to a component interface is not straighforward and thus the approach is not general. Furthermore, interfaces that allow efficient assembly of fine-grained protocol components are likely to impose a specific runtime that is not acceptable as a general purpose application programming interface.

3 A generic Group Communication Service

This Section describes jGCS, a generic group communication service for Java. We provide a specification of the API and of the minimum semantics that support application portability. The service is organized in four complementary interfaces, namely: the configuration interface, the common interface, the data interface, and the control interface. Each of these interfaces is decribed below.

3.1 Configuration Interface

The configuration interface decouples the application code from specific implementations by requiring that a third party, the configurator, matches available services with application requirements. Besides the obvious portability advantages, this also fulfills Goal 2. This interface shown in Figure 1 is composed by opaque objects as follows:

ProtocolFactory The protocol factory serves as the interface entry point and triggers the initialization of runtime instances of a protocol implementation. At the semantic level, it encapsulates an implicit service guarantee specification which is enforced for all sessions.

GroupConfiguration A group configuration encapsulates the address of a group that can be used to open a session that subsequently allows messages to be sent or received, or the membership to be observed. As the ProtocolFactory, at the semantic level it also encapsulates an implicit service guarantee specification which is enforced for all messages exchanged. This object may be used as a key in hashtables.

Service A service encapsulates a specification of the guarantees to be enforced on a particular message. Upon encountering a service specification that is unknown or incompatible with group or protocol configuration, the implementation must throw an exception. A partial order is defined on guarantees provided by services by extending the Comparable interface (i.e., some services may be stronger than, and subsume, other services). Therefore, the application can use the service interface to discover if a service guarantee is subsumed by some other.

Annotation An annotation is an optional field that encapsulates semantic knowledge about a message that can be used by the protocol to optimize performance. The contents of the annotation are therefore implementation specific and protocols should silently ignore unknown annotations without erroneous or unpredictable behavior.

Configuration objects should be easily stored and retrived in configuration files and directory services. It is therefore advisable that implementations provide configuration objects with one or more of the following properties: are serializable, can be constructed from properties files, and export parameters according to JavaBean conventions. For the same reason, these objects should not be used to keep session state at runtime.

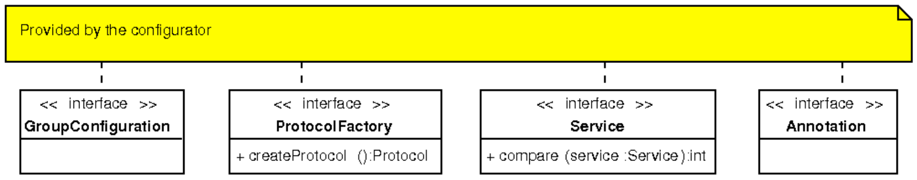


Fig. 1. Configuration interfaces.

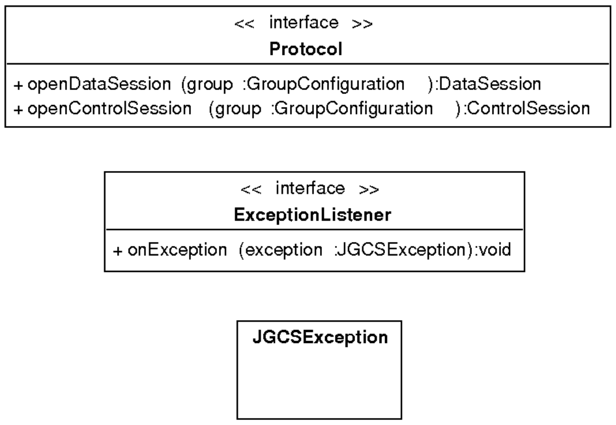

<< interface >>
Protocol
+ openDataSession (group :GroupConfiguration):DataSession
+ openControlSession (group :GroupConfiguration):ControlSession
<< interface >>
ExceptionListener
+ onException (exception :JGCSException):void
JGCSException

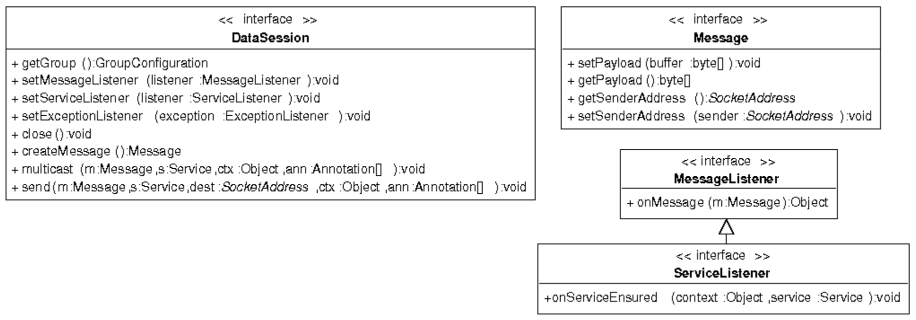
Fig. 2. Common interfaces.

3.2 Common Interface

A protocol session is represented by a Protocol instance, obtained from the configuration stored in a ProtocolFactory. Using a Protocol instance it is possible to obtain, for a specific GroupConfiguration, a *data* and a *control* session. All further operation are invoked through one of these two interfaces, as depicted in Figure 2.

Both data and control sessions identify group members using java.net.SocketAddress objects. This directly allows a large number of protocols to be supported without any form of address conversion. Protocols that use different address formats, can easily be wrapped. Examples of the use of both native and wrapped member identifiers are described in Section 4.

Finally, exceptions thrown by the jGCS extends the JGCSException class, which itself extends the java.io.IOException interface. Usually, a nested implementation dependent exception can be obtained by using the standard getCause method. Exceptions thrown asynchronously within the protocol implementation are delivered to the application using the ExceptionListener interface. This can be registered using either session object.



<< interface >>
DataSession
+ getGroup ():GroupConfiguration
+ setMessageListener (listener :MessageListener):void
+ setServiceListener (listener :ServiceListener):void
+ setExceptionListener (exception :ExceptionListener):void
+ close ():void
+ createMessage ():Message
+ multicast (m:Message ,s:Service ,ctx :Object ,ann :Annotation[]):void
+ send (m:Message ,s:Service ,dest :*SocketAddress* ,ctx :Object ,ann :Annotation[]):void
<< interface >>
Message
+ setPayload (buffer :byte[]):void
+ getPayload ():byte[]
+ getSenderAddress ():*SocketAddress*
+ setSenderAddress (sender :*SocketAddress*):void
<< interface >>
MessageListener
+ onMessage (m:Message):Object
<< interface >>
ServiceListener
+onServiceEnsured (context :Object ,service :Service):void

Fig. 3. Message passing interfaces.

3.3 Data Interface

The data interface provides the methods for messages to be sent and received. Whenever the application multicasts a message there is always a specific quality of service, i.e. a specific set of guarantees, associated with the request. The guarantees can be implicitly derived from the group or protocol configuration or explictly set using a Service parameter. The data interface shown in Figure 3 is as follows:

DataSession The data session provides methods for sending messages in both multicast and peer groups. It also allows registering listeners for the various events.

Message This interface wraps payload and sender address. According to Goal 1, the only payload supported is a byte array. As instances can only be created by the DataSession, it can be implemented as a thin layer on implementation specific objects to avoid having to perform additional buffer copy operations.

MessageListener Handles delivery of message payload. This is the main entry point for incoming data. When no separate ServiceListener is being used, implicitly does service notification.

ServiceListener Handles delivery of service notification events. As described below, applications that do not need to be optimized for concurrency can ignore this interface.

The data interface exposes one of the key features of view synchronous group communication: messages are delayed by the protocol implementation to be delivered to the application only after global guarantees have been ensured, namely, regarding order and stability. For instance, when providing uniform agreed (or safe) guarantees [10], the implementation must collect a number of acknowledgements from differente members before issuing the message delivery.

However, recent work on group communication [5, 6] has shown that it is useful to deliver the payload to the application as soon as it is received and then later notify the application that the requested service has been ensured. This allows increased concurrency and masking of latency, by allowing the application to start processing the message earlier, at least, by deserializing the message in parallel with the execution of

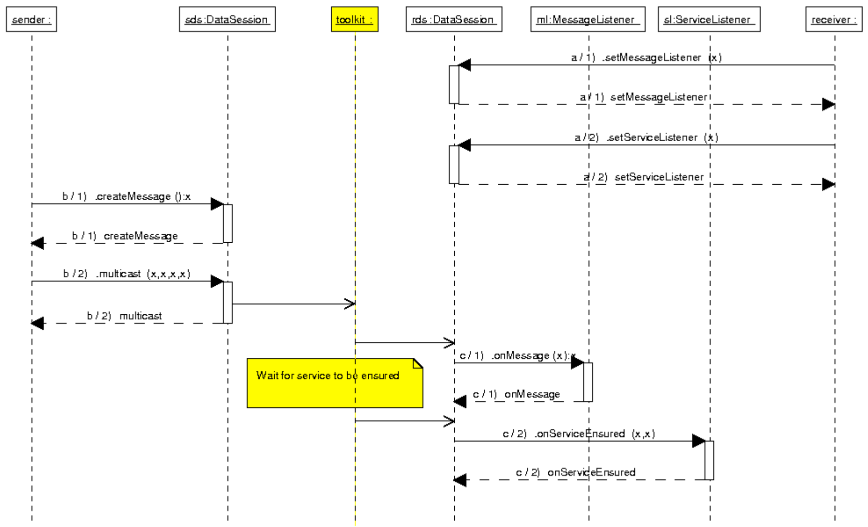

sender :
sds :DataSession
toolkit :
rds :DataSession
ml:MessageListener
sl:ServiceListener
receiver :
a / 1) .setMessageListener (x)
a / 1) setMessageListener
a / 2) .setServiceListener (x)
a / 2) setServiceListener
b / 1) .createMessage ():x
b / 1) createMessage
b / 2) .multicast (x,x,x,x)
b / 2) multicast
c / 1) .onMessage (x):x
c / 1) onMessage
Wait for service to be ensured
c / 2) .onServiceEnsured (x,x)
c / 2) onServiceEnsured

Fig. 4. Data session usage.

the remaining of the protocol. Examples of this strategy can be found in systems based on optimistic atomic multicast protocols [21].

jGCS supports this optimization as described in Figure 4. The application registers a ServiceListener with the DataSession. The protocol is allowed to deliver payload without ensuring services. Upon handling the message, the application chooses how to proceed:

– Returns a context reference (any POJO) which the protocol associates with the message. Typically, the context contains a pre-processed message. When service is ensured, the protocol calls back into the application providing references to both the context object and the service object that has been achieved. The application then resumes processing the message.
– Returns a null reference. This informs the protocol that no further notifications or service guarantees are required for this message and no further callbacks happen.

Notice that even if the protocol implementation does not natively support this interface, the binding can trivially support it by performing both callbacks only after the final delivery. Currently, the only toolkit that natively supports this interface is Appia [22].

On the sender side, the jGCS also provides mechanisms to prevent the aplication from being blocked when invoking the interface. For instance, a specific protocol implementation may not accept requests until some service is ensured. Also, an implementation may perform end-to-end flow control, thus throttling the sender in a similar fashion. The non-blocking interface works as follows. Upon sending a message, an application might also specify a context. This means that multicast does not block and the application gets notified using the service listener callback.

An additional advantage of jGCS is that it does not impose artificial limits to the application concurrency, namely in the processing of incoming messages. In fact, jGCS allows for concurrent message delivery notifications whenever the requested service does not impose ordering on messages. Therefore, only total ordering constraints prevent concurrent deliveries. Also, this applies both to payload deliveries, when no service listener has been registered, as well as to service callbacks. Notice that in the later, payload deliveries can always be performed concurrently, up to an optimal concurrency degree, that can be coordinated with application containers.

Finally, the jGCS provides support for the use of semantic knowledge to improve system performance. This is achieved by letting application annotate messages with control information that can be used by the group communication toolkit to selectively relax reliability, order and view synchrony guarantees [8, 9, 7]. For that purpose, the application should obtain one or more annotation objects in an implementation specific fashion. These are then handed to the protocol as parameters in the multicast operation. Although these objects are implementation specific, this interface does not pose a threat to portability as, by definition, a semantic annotation can be safely ignored.

3.4 Control Interface

The control interface provides a flexible subsetable interface for a wide range of membership management protocols. The most simple interface is suitable only for best-effort multicast protocols and is shown on Figure 5:

ControlSession Provides methods for entering and leaving a group, as well as for registering a listener for control events.

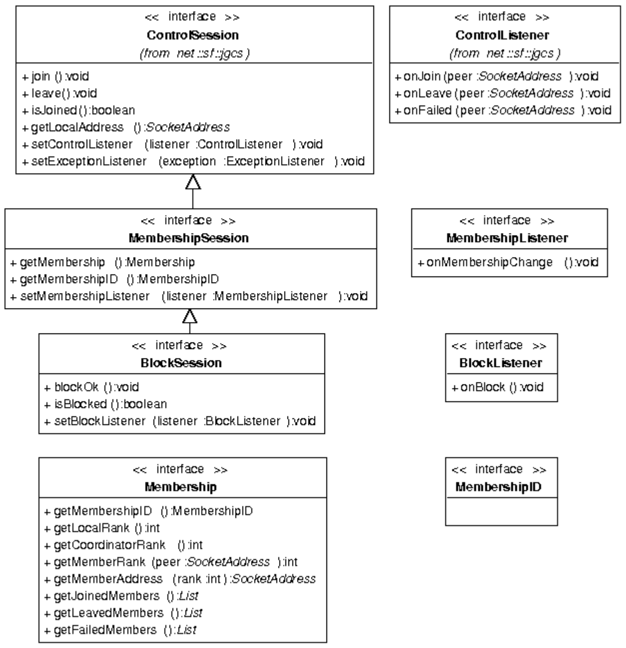
ControlListener Allows a simple notification of members entering and leaving the group. Precise semantics of these events, namely regarding concurrency with message deliveries, depends on the implementation.

This interface can be used separatly for failure detection or cluster management infrastructure, which are not directly related to group communication. Notice also that implementations can choose to distinguish members that have left the group volutarily and in a controlled fashion from members that have failed and thus been forcibly excluded. The former allows recovery from a known state and thus is more efficient.

Support for view synchronous group communication differs wether the underlying implementation provides sending view delivery [10], and thus blocks applications briefly before installing new views. This reflects in the following interfaces (Figure 5):

Membership Describes a view of the group. This can be used to obtain a ranked list of all members, whose sort order depends on the implementation but which should be the same everywhere. It can also be used to obtain information on the event leading to the view change, namely, which processes have just been included and excluded and why.

MembershipID Provides an opaque unique identifier of the view, suitable for being exchanged and stored persistently. This can be obtained from the currently installed Membership object.



<< interface >>
ControlSession
(from net::sf::jgcs)
+ join ():void
+ leave():void
+ isJoined():boolean
+ getLocalAddress ():SocketAddress
+ setControlListener (listener :ControlListener):void
+ setExceptionListener (exception :ExceptionListener):void
<< interface >>
ControlListener
(from net::sf::jgcs)
+ onJoin (peer :SocketAddress):void
+ onLeave (peer :SocketAddress):void
+ onFailed (peer :SocketAddress):void
<< interface >>
MembershipSession
+ getMembership ():Membership
+ getMembershipID ():MembershipID
+ setMembershipListener (listener :MembershipListener):void
<< interface >>
MembershipListener
+ onMembershipChange ():void
<< interface >>
BlockSession
+ blockOk ():void
+ isBlocked ():boolean
+ setBlockListener (listener :BlockListener):void
<< interface >>
BlockListener
+ onBlock ():void
<< interface >>
Membership
+ getMembershipID ():MembershipID
+ getLocalRank ():int
+ getCoordinatorRank ():int
+ getMemberRank (peer :SocketAddress):int
+ getMemberAddress (rank :int):SocketAddress
+ getJoinedMembers ():List
+ getLeavedMembers ():List
+ getFailedMembers ():List
<< interface >>
MembershipID

Fig. 5. Interface extensions for view synchrony.

MembershipSession Provides methods to obtain the current membership and register the callback for view change events.

MembershipListener Handles notifications of view change.

BlockSession Used only by implementations enforcing sending view delivery, providing methods for signaling that the application has blocked and that view change can proceed.

BlockListener Handles requests by the protocol for the application to block.

Support for view synchronous group communication requires that membership notifications are coordinated with message and service notifications performed by the corresponding data session. In detail, the implementation must ensure that the view change notification is mutually exclusive with any other view dependent event, namely, message delivery and service ensured callbacks. This means that no other notification is issued concurrently with the view change. Although protocol implementations might allow this restriction to be lifted, this should be possible only by explicitly selecting a configuration option. On the other hand, block notifications can be issued without any concurrency restrictions. This means that it is up to the application to synchronize with any other active threads.

4 jGCS Bindings

jGCS was implemented in several group communication toolkits and primitives: Appia [2], JGroups [4] and Spread [3]. To validate the generality of the service, the jGCS was also implemented using IP Multicast and NeEM [9]. All these bindings are open source and available on SourceForge.net[5]. These toolkits and the implementations are described in the following paragraphs.

Appia binding

Appia [2] is a layered communication support framework that was implemented in the University of Lisbon. It is implemented in Java and aims at high flexibility to build communication channels that fit exactly in the user needs. The QoS offered by a channel can be statically configured by an XML file or dynamically assembled by the application at run time. The application can create several channels with different QoSs and send messages to different channels, depending on the QoS required by each message. In contrast with traditional layered protocols, components of Appia channels can be shared and thus offer multiple related Qualities of Service (QoS). This makes it easy, for instance, that several channels can be bound to the same group membership.

Although Appia is protocol independent, in the sense that it can be used to compose any protocol as long as it respects the predefined interface, it includes an extensive layer library targeted at view synchronous group communication. Namely, it has protocols that implement virtual synchrony, causal order, and several implementations of total order algorithms.

The implementation of jGCS is build directly on Appia's protocol composition interfaces as an additional layer. jGCS configuration objects thus define the micro-protocols that will be used in the communication channels. Each Service identifies an Appia channel and messages are sent through the channel that fits the requested service. As Appia supports early delivery in totally ordered multicast, this is exposed in the jGCS binding using the ServiceListener interface. Appia implements all extensions of the ControlSession, depending on the channel configuration.

JGroups binding

JGroups [4] is a group communication toolkit modelled on Ensemble [18] and implemented in Java. It provides a stack architecture that allows users to put together custom stacks for different view synchronous multicast guarantees as well as supporting peer groups. It provides an extensive library of ordering and reliability protocols, as well as support for encryption and multiple transport options. It is currently used by several large middleware platforms such as JBoss and JOnAS.

The JGroups implementation of jGCS also uses the configuration interface to define the micro-protocols that will be used in the communication channel. JGroups can provide only one service by the applications, since configurations only support one JGroups channel per group communication instance. JGroups implements all extensions of the ControlSession.

[5] jGCS and its bindings are available in `http://jgcs.sf.net`

Spread binding

Spread/FlushSpread [3] is a toolkit implemented by researchers of the Johns Hopkins University. It is based on an overlay network that provides a messaging service resilient to faults across local and wide-area networks. It provides services ranged from reliable message passing to fully ordered messages with delivery guarantees. The Spread system is based on a daemon-client model where generally long-running daemons establish the basic message dissemination network and provide basic membership and ordering services, while user applications linked with a small client library can reside anywhere on the network and will connect to the closest daemon to gain access to the group communication services. Although there are interfaces for Spread in multiple languages, these do not support the FlushSpread extension, which provides additional guarantees with a different interface.

The Spread and FlushSpread implementations of jGCS use the configuration interface to define the location of the daemon and the group name. The implementation to use (FlushSpread or just Spread) is also defined at configuration time. In Spread, the quality of service is explicitly requested for each message, being thus encapsulated in Service configuration objects.

Other bindings

To prove the generality of jGCS, we provide also two implementations, based on the well known IP Multicast and on the Network-friendly Epidemic Protocol (NeEM) [9]. The NeEM protocol is an epidemic multicast protocol (also called probabilistic or gossip-based) in wide-area networks that uses multiple TCP/IP connections in a non-blocking fashion. The resulting overlay network is automatically managed by the protocol. The implementations of jGCS that use IP Multicast and NeEM allow peers to join and leave the multicast group, and send and receive messages to/from other peers. One application that uses only these functionalities can easily be ported to other implementations.

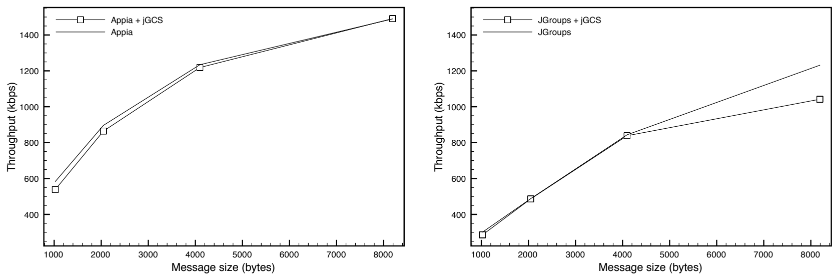
5 Performance Evaluation

We have done a number of experiments to assess the overhead imposed by the use of jGCS to wrap different group communication toolkits. Namely, we want to assess the impact of the extra level of indirection between the application and the toolkit introduced by jGCS.

For this purpose we have made two different sets of tests. In the first set we have made standalone throughput measurements for two different toolkits, both with and without the jGCS. In the second set of tests we have integrated jGCS in a production environment, namely in the Sequoia database clustering middleware.

5.1 jGCS for Appia and JGroups

To measure the impact of the jGCS on the maximum throughput of existing group communication toolkits we have selected Appia and JGroups. To run the experiments, we



(a) Appia throughput. (b) JGroups throughput.

Fig. 6. Throughput of the toolkits with and without the jGCS.

have implemented three different versions of a test application that transmits a number of messages of a configurable payload size to the group. One version uses the Appia native interface, other uses the JGroups native interface, and the last version uses jGCS. This allowed us to run four different configurations: *(i)* the test application with Appia; *(ii)* the test application with JGroups, *(iii)* the test application with jGCS, configured to use Appia and *(iv)* the test application with jGCS, configured to use JGroups.

Measurements were obtained with the following environment. The JGroups and the Appia protocol stack were created using similar configurations. All tests used a virtual synchrony protocol stack and a token based total order protocol. All tests were made with a group of three members, each member sending 10 000 totally ordered messages to the group. Each member of the group runs in a Pentium IV/2.8GHz server with 1Gb of memory. The three machines are connected through 100Mbps ethernet switch. Each test was made with different message sizes.

The Figure 6 shows the throughput of the two group communication toolkits, using directly the interface provided by the toolkit and using jGCS. As we can see in 6(a), the Appia implementation of jGCS does not cause a significant overhead and this overhead is increasingly less noticeable as the message size grows. In the case of JGroups, in 6(b), the overhead caused by the jGCS is also very small but it grows as the message size increases. This is explained as follows: For improved performance, JGroups delivers messages in a buffer that can be reused later by the protocol, forcing the application to locally copy data during delivery. The native JGroups test application does not do this copying, and thus has better performance. On the other hand, the current jGCS binding does this copying in order to provide the same service as other bindings and thus incurs in additional overhead. In the future, this decision should probably be left to the configurator, thus making it possible to achieve the same performance as with the native interface.

5.2 jGCS in Sequoia

The second set of tests measure the overhead of having jGCS in a real application. To do these tests we used Sequoia, a middleware database replication system that exports a

Implementation	Mean	Std. Dev.	Samples
Native	39.96	41.10	3846
With jGCS	40.26	52.97	3832

Table 1. Latency of client requests of TPC-W (ms).

JDBC interface to applications and routes client requests to a set of databases. Sequoia is composed by a JDBC driver, that is used by applications that want to access the databases and a controller that receives the client requests and forward them to a set of databases. For availability and fault tolerance, the Sequoia controller can (and should) be replicated. Each controller manages a set of databases. In a system with more than one controller, the application can use any controller to make the requests. The controllers exchange their requests using view synchronous total order, in order to execute the same set of requests in the same order in all databases.

The implementation of primitives that make use of group communication is distributed as a separate package, Hedera (formerly ObjectWeb Tribe). In detail, it provides access to an application specific subset of group communication and additional functionality for explicitly acknowledged messages, multiplexing and dispatching. Hedera has been previously implemented twice, using JGroups and Appia. We thus ported Hedera to jGCS which allowed us to use Sequoia with any jGCS implementation that supports the required service guarantees.

Performance figures were obtained in a system configured as follows. The clients are a Java implementation of the TPC-W^{TM}[6] that use the Jakarta Tomcat to make requests to a database. The requests are made to a sequoia controller that replicates the requests. Sequoia is configured to use three controllers, each one controls one MySQL database. The emulated browsers of the benchmark used and Tomcat run in one machine. The other three machines have one instance of the Sequoia controller and one instance of the MySQL database each. All four machines are connected by a 100Mbps ethernet switch and have the same memory and processing power of the machines used in the previous tests. In these tests, the benchmark was configured to have allways 20 clients (emulated browsers) making requests to the database, in the Ordering Mix (50% of write operations).

Table 1 shows that latency results of test system using the Appia toolkit, either through the native interface or through jGCS, are in practice the same. In fact, the difference is not statistically relevant, even with a very low confidence level, as confidence intervals overlap significantly. This shows that the use of jGCS is negligible in the overall performance of a complex system.

6 Related Work

Although there have been multiple attempts to ease the development of applications based on group communication by standardising their interfaces and their semantics,

[6] `http://www.ece.wisc.edu/~pharm/tpcw.shtml`

most of these efforts have quite different goals and, therefore, can be seen as complementary (instead of competing) efforts. We discuss four non mutually exclusive approaches.

The first approach attempts to hide the complexity of group communication by wrapping it in higher level abstractions. The rationale for this line of work is that there is a category of users that would like to benefit from the advantages provided by group communication (namely, easy maintenance of consistent global states) but that do not want to invest in understanding the semantics of view synchronous communication. A particularly popular approach is to wrap view synchronous communication in RPC-like interface, such as RMI [23]. While the above may be true, using such interfaces often introduce a significant performance overhead that is unacceptable for programmers attempting to build high-performance applications such as replicated database middleware. In our view, the widespread use of the Java programming language and the broad adoption of several design patterns used in jGCS allow to conciliate the programmer familiarity with the satisfaction of performance constraints.

A second line of work attempts to fit the view synchronous interface into widely adopted interfaces such as, for instance, SNMP messages [20] or JMS [24, 25]. The rationale for this line of work is that a view synchronous services can be easily adopted if exported using an interface that is familiar to most programmers. We see this category as a complementary line of work, given that the simplicity is achieved at the cost of loosing some of the benefits of both paradigms. Namely, many JMS applications rely of persistence or transactional services that are not provided directly by group communication toolkits. On the other hand, the notion of explicit membership, a keystone of view synchronous communication, conflicts with the goal of decoupling publishers from subscribers. Therefore, group communication can be a valuable tool to increase the reliability of messaging services but, certainly, one service cannot simply replace the other.

A third alternative aims at standard semantics for view synchronous communication [26]. The rationale for this type of work is that it would be easier to see a wider adoption of view synchronous communication toolkits if all provide the same guarantees. Although this is a valid goal, practice has shown that it is hard to make the community converge on such an common semantics, given that each toolkit exploits a different angle to provide better performance for some distinct target application areas. Instead of trying to define unique semantics, jGCS only defines minimum common semantics and provides the necessary hooks to support specific features using a flexible interface.

A final approach is to make the service provided by the group communication toolkit highly reconfigurable, usually through the composition of micro-protocols [18, 2]. The rationale for this type of work is that applications are better served if the toolkit can be tailored exactly to their specific needs. Toolkit reconfiguration is an important topic. Our own work with the Appia toolkit has addressed this facet extensively. However, existing protocol composition mechanisms are typically tied to concrete language or run-time constructs from which the application should be shielded. The configuration interface of the jGCS decouples the way the application expresses its requirements from the mechanisms used to configure the underlying toolkit.

The work with jGCS is unique in the sense that it provides a low-level interface to view-synchronous communication, that allows the implementation of high-performance higher level abstractions while, at the same time, promotes a level of decoupling between the application and the underlying toolkit, that allows the application to be portable. Higher level primitives, such as state transfer, multiplexing and dispatching, or explicit aknowldgement, commonly found in group communication wrappers as Maestro [27] or Hedera should be built on jGCS services.

7 Conclusions

This paper describes a generic interface for group communication to be used as a service to the applications – the Group Communication Service, or simply jGCS. The paper presents the goals to achieve and features to avoid and presents the interfaces and how should it work in order to achieve the desired goals.

Results show that the jGCS interfaces can be implemented using most of the state of the art group communication toolkits. It is also shown that the overhead caused by the jGCS service is negligible and do not affect real applications, improving modularity and configurability. This service was implemented in Java and is hosted at SourceForge.net (`http://jgcs.sf.net`).

References

1. Birman, K., van Renesse, R., eds.: Reliable Distributed Computing with the Isis Toolkit. IEEE Computer Society Press (1993)
2. Miranda, H., Pinto, A., Rodrigues, L.: Appia, a flexible protocol kernel supporting multiple coordinated channels. In: Proceedings of the 21st International Conference on Distributed Computing Systems, Phoenix, Arizona, IEEE (2001) 707–710
3. Amir, Y., Danilov, C., Stanton, J.: A low latency, loss tolerant architecture and protocol for wide area group communication. In: IEEE International Conference on Dependable Systems and Networks. (2000)
4. Ban, B.: Design and implementation of a reliable group communication toolkit for java (1998)
5. Vicente, P., Rodrigues, L.: An indulgent uniform total order algorithm with optimistic delivery. In: Proceedings of the 21th IEEE Symposium on Reliable Distributed Systems (SRDS'02), Osaka, Japan (2002) 92–101
6. Sousa, A., Pereira, J., Moura, F., Oliveira, R.: Optimistic total order in wide area networks. In: Proc. 21st IEEE Symposium on Reliable Distributed Systems, IEEE CS (2002) 190–199
7. Pedone, F., Schiper, A.: Handling message semantics with generic broadcast protocols. Distributed Computing Journal (2002) 97–107
8. Pereira, J., Rodrigues, L., Oliveira, R.: Semantically reliable multicast: Definition, implementation and performance evaluation. IEEE Transactions on Computers, Special Issue on Reliable Distributed Systems **52** (2003) 150–165
9. Pereira, J., Rodrigues, L., Monteiro, M.J., Oliveira, R., Kermarrec, A.M.: Neem: Network-friendly epidemic multicast. In: Proceedings of the 22th IEEE Symposium on Reliable Distributed Systems (SRDS'03), Florence,Italy (2003) 15–24
10. Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Comput. Surv. **33** (2001) 427–469

11. Guerraoui, R., Schiper, A.: Software-based replication for fault-tolerance. IEEE Computer **30** (1997) 68–74
12. Kemme, B., Alonso, G.: Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In: The VLDB Journal. (2000) 134–143
13. Narasimhan, P., Moser, L.E., Melliar-Smith, P.M.: Eternal - a component-based framework for transparent fault-tolerant corba. Software Practice and Experience (2002)
14. Pedone, F., Schiper, A.: Optimistic atomic broadcast. In: Proceedings of the 12th International Symposium on Distributed Computing (DISC'98). (1998)
15. Sussman, J.B., Keidar, I., Marzullo, K.: Optimistic virtual synchrony. In: Symposium on Reliability in Distributed Software. (2000) 42–51
16. Rodrigues, L., Mocito, J., Carvalho, N.: From spontaneous total order to uniform total order: different degrees of optimistic delivery. In: Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06), Dijon, France, ACM (2006)
17. Alur, D., Crupi, J., Malk, D.: Core J2EE Patterns: Best Practices and Design Strategie. Prentice Hall / Sun Microsystems Pres (2001)
18. Hayden, M.: The Ensemble System. PhD thesis, Cornell University, Computer Science Department (1998)
19. Hiltunen, M., Schlichting, R., Wong, G.: Implementing integrated fine-grain customizable qos using cactus. In: Fast Abstracts, The 29th International Symposium on Fault-Tolerant Computing Systems, Madison, Wisconsin, USA (1999) 59–60
20. Wiesmann, M., Défago, X., Schiper, A.: Group communication based on standard interfaces. In: Proc. 2nd Int'l Symp. on Network Computing and Applications (NCA-03), Cambridge, MA, USA, IEEE (2003) 140–147
21. Kemme, B., Pedone, F., Alonso, G., Schiper, A.: Processing transactions over optimistic atomic broadcast protocols. In: Proceedings of 19th International Conference on Distributed Computing Systems (ICDCS'99). (1999)
22. Mocito, J., Respício, A., Rodrigues, L.: On statistically estimated optimistic delivery in wide-area total order protocols. Technical Report DI/TR/2006, University of Lisbon (2006)
23. Montresor, A., Davoli, R., Babaoglu, O.: Group-enhanced remote method invocations (1999)
24. Amir, Y., Munjal, A.: (Jms4spread: http://www.spread.org/jms4spread/)
25. SoftWired: (ibus/messagebus: http://www.softwired.ch/)
26. Kupsys, A., Pleisch, S., Schiper, S., Wiesmann, M.: Towards jms compliant group communication - a semantic mapping. In: NCA '04: Proceedings of the Network Computing and Applications, Third IEEE International Symposium on (NCA'04), Washington, DC, USA, IEEE Computer Society (2004) 131–140
27. Birman, K., Friedman, R., Hayden, M.: The maestro group manager: A structuring tool for applications with multiple quality of service requirements. Technical report, Ithaca, NY, USA (1997)