Project no. 004758

GORDA

Open Replication Of Databases

Specific Targeted Research Project

Software and Services

Draft Standard

(GORDA Group Communication Service Specification)

GORDA Deliverable 6.4b

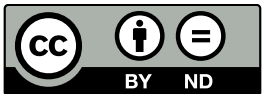
Due date of deliverable:
Actual submission date: 2007/04/04

Start date of project: 1 October 2004 Duration: 42 Months

FFCUL

Revision 0.1

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	



Contents

Preface

This document, *GORDA Group Communication Service Specification*, specifies the programming interfaces for generic group communication systems.

Revision History

Date	Version	Description
2007-04-04	0.1	Initial Draft

Who Should Use This Specification

The audience for this document are:

- implementors of database replication protocols;
- implementors of distributed systems that require group communication.

How This Specification Is Organized

Section 1 introduces the interface in the context of the GORDA project as well as document conventions used. Section 2 describes the goals, scope, and requirements of the proposed interface. Section 3 presents the abstract model of transaction processing underlying the interface as well as key design patterns. Section 4 discusses the interface in detail. Finally, Section 5 is a guide to sample code distributed with the interface.

1 Introduction and Background

1.1 Introduction

This document specifies a programming interface for Group Communication as well as minimum semantics that allow application portability. This interface accommodates existing group communication services, enabling implementation independence. The interface is called Group Communication Service, or simply GCS.

Group Communication is understood as a coordination paradigm that eases the development of multi-participant applications. Some examples are replicated servers, cooperative caches and multi-user cooperative applications.

1.2 The GORDA Project

The goal of the GORDA project is to foster database replication as a means to address the challenges of trust, integration, performance, and cost in current database systems underlying the information society. This is to be achieved by standardizing architecture and interfaces, and by sparking their usage with a comprehensive set of components ready to be deployed.

GORDA is supported by the European Community under the Sixth European Union Framework Programme for Research and Technological Development, thematic priority Information Society Technologies, contract number 004758. The consortitum is composed by U. Minho, U. della Svizzera Italiana, U. Lisboa, INRIA Rhône-Alpes, Continuent, and MySQL.

More information is available at:

- http://gorda.di.uminho.pt

1.3 Relation with GCS

The specification is based on the GORDA Architecture and Programming Interfaces as described in GORDA deliverables D2.2 and D2.3. The main difference is that a new interface to handle the exclusion of a member from a group was created.

In the scope of the project, the presented interfaces were implemented using several group communication toolkits. A Java version of the interfaces and all its implementations are available as open source code in the URL: http://jgcs.sf.net.

1.4 Document Conventions

1.4.1 Definitions

This document uses definitions based upon those specified in RFC-2119 (See http://www.ietf.org/). For a better reading experience these terms are written in small letters.

Table 1: Specification terms.

Term	Definition
MUST	The associated definition is an absolute requirement of this specification.
MUST NOT	The definition is an absolute prohibition of this specification.
SHOULD	Indicates a recommended practice. There may exist valid reasons in particular circumstances to ignore this recommendation, but the full implications must be understood and carefully weighed before choosing a different course.
SHOULD NOT	Indicates a non-recommended practice. There may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.
MAY	Indicates that an item is truly optional.

1.4.2 Formatting Conventions

This specification uses the following formatting conventions.

Table 2: Formatting conventions.

Convention	Description
`fixed`	Used in all Java code including keywords, data types, constants, method names, variables, class names, and interface names.
italic	Used for emphasis and to signify the first use of a term.

1.5 Contributors

- Alfranio Correia Jr., U. Minho
- Nuno Carvalho, U. Lisboa
- Nuno A. Carvalho, U. Minho

- Rui Oliveira, U. Minho
- José Pereira, U. Minho
- Luís Rodrigues, U. Lisboa
- Ricardo Vilaça, U. Minho

1.6 Feedback

Please send any comments and questions concerning this specification to:

community@gorda.di.uminho.pt

or

jgcs@lasige.di.fc.ul.pt

2 Scope and Requirements

2.1 Goals

No changes to payload required. No assumptions or changes should be made on message payload. This means that implementing GCS does not require specific data formats, additional message headers or additional messages exchanged. The toolkits that adopt GCS as their native interface can implement GCS-specific optimizations. As a result, applications that use a specific protocol through GCS should be interoperable with legacy versions using native interfaces. Furthermore, no specific constructors or data formats are forced on the application. It must be possible to translate the interface to languages in the same family such as C++ or C#.

Support service locator and dependency injection patterns. All details regarding protocol configuration and service selection must be encapsulated in objects that may be supplied to the application by a third party (i.e. the configurator) using a service locator or the dependency injection patterns. 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.

Support multiple group-based programming paradigms. The GCS interface should be flexible enough to support different flavors of multicast communication based on process groups. The GCS should support both open groups (where any process can send messages to the group) and closed groups (where only group members can send messages to the group). It should also support 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.

Export a flexible subsetable interface. The GCS should support the deployment of just parts of the interface to avoid redundancy. The GCS 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.

Non-blocking input/output and container-managed concurrency. GCS 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. This avoids the requirement to have threads blocked on input/output. It also allow the GCS implementations to cooperate with application containers to optimize the number of concurrent threads, when concurrency requirements arise.

Accommodate latest research results. The interface should allow recent research results, such as support semantic annotations and early delivery, to be easily accommodated. In fact, the goal is to foster programming idioms that naturally take advantage of such results as they become available.

2.2 Non-Goals

Specify a common set of service guarantees. The GCS avoids this pitfall by assuming a configuration step that matches available service guarantees to application requirements.

Exclusively reuse existing standard interfaces. It is a better option to provide a syntactically incompatible interface that embodies similar structure and the same patterns such that programmers can easily make the transition.

Provide interfaces for protocol composition. The main problem is that the mapping of an existing implementation to a component interface is not straightforward 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 Design

3.1 Approach and Terminology

This specification is based on the basis needed to implement group communication in general. We use Java to illustrate the interfaces, but any object oriented language may be used to implement the GCS.

3.2 Overview

The GCS interface 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.

3.3 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. It is composed by opaque objects as follows:

ProtocolFactory The protocol factory must serve 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 configuration 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 return some error. A partial order must be defined on guarantees provided by services (i.e., some services may be stronger than, and subsume, other services). Therefore, the application may use the service interface to discover if a service guarantee is subsumed by some other.

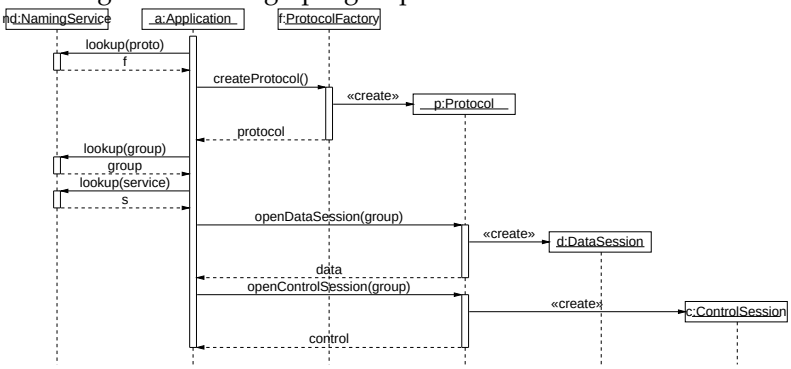
Annotation An annotation is an optional field that encapsulates semantic knowledge about a message that may be used by the protocol. The contents of the annotation are therefore implementation specific and protocols should silently ignore unknown annotations without erroneous or unpredictable behavior.

Configuration objects should be easily stored and retrived in configuration files and directory services. The implementations should provide configuration objects with one or more of the following properties: are serializable and can be constructed from properties files. For the same reason, these objects should not be used to keep session state at runtime.

3.4 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 operations must be invoked through one of these two interfaces. This sequence is shown in Figure 1. Both data and control sessions identify group members. Protocols may use different address formats and should wrap the addresses.

Figure 1: Starting up a group communication instance.



Finally, exceptions thrown asynchronously within the protocol implementation are delivered to the application using the ExceptionListener interface. This can be registered using either session object.

3.5 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 may be implicitly derived from the group or protocol configuration or explicitly set using a Service parameter. The data interface 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. The only payload supported is a byte array. The instances must be created by the DataSession. Implementors may provide this interface 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.

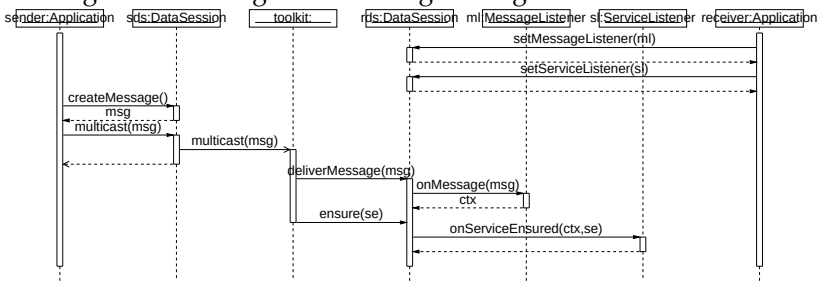
ServiceListener Handles delivery of service notification events. Applications that do not need to be optimized for concurrency may ignore this interface.

The data interface may expose the early delivery feature to the application, using the Services interface. This should be done by delivering 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 the remaining of the protocol. GCS should support this optimization as described in Figure 2. The application registers a ServiceListener with the DataSession. The protocol may 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. When the 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 should happen.

Protocols that do not natively support this interface may perform both callbacks only after the final delivery.

Figure 2: Sending and receiving messages with the services.



On the sender side, the GCS also provides mechanisms to prevent the application 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 may also specify a context. This means that multicast does not block and the application gets notified using the service listener callback.

GCS does not impose artificial limits to the application concurrency, namely in the processing of incoming messages. This interface allows for concurrent

message delivery notifications whenever the requested service does not impose ordering on messages. 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 may be coordinated with application containers.

Finally, the GCS provides support for the use of semantic knowledge. 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. 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. Unknown semantic annotations should be ignored by the protocols.

3.6 Control Interface

The control interface is subsetable and the most simple interface should be implemented only by best-effort multicast protocols. The basic interface is composed by the following:

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 may be used separatly for failure detection or cluster management infrastructure, which are not directly related to group communication. The implementations may choose to distinguish members that have left the group voluntarily and in a controlled fashion from members that have failed and thus been forcibly excluded.

If the implementation supports view synchronous, the extensions of the control session must be used. The extensions are reflected in the following interfaces:

Membership Describes a view of the group. This may 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 may 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 may be obtained from the currently installed Membership object.

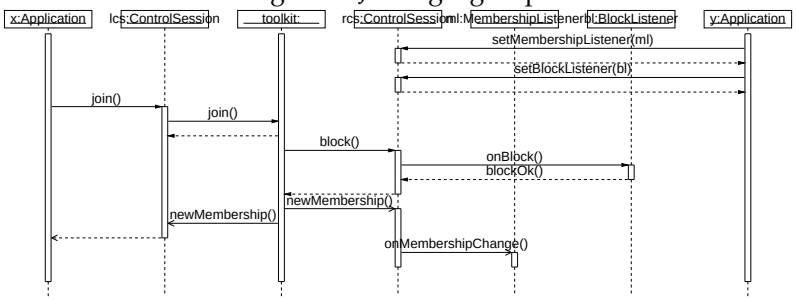
MembershipSession Provides methods to obtain the current membership and register the callback for view change events.

MembershipListener Handles notifications of view change.

BlockSession Should be 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.

Figure 3: Joining a group.



The Figure 3 shows how the system should work when a member joins a group. Support for view synchronous group communication requires that membership notifications are coordinated with message and service notifications performed by the corresponding data session. 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 notification must not be issued concurrently with the view change. Protocol implementations may allow this restriction to be lifted, but this should be possible only by explicitly selecting a configuration option. Block notifications may be issued without any concurrency restrictions. This means that it is up to the application to synchronize with any other active threads.

4 API Description

The specification is contained in package net.sf.jgcs and net.sf.jgcs.membership. A diagram outlining the relations between individual interfaces is shown in the Figures 4 and 5.

Detailed descriptions of the specification are provided in the following sections.

Figure 4: Group communication interfaces.

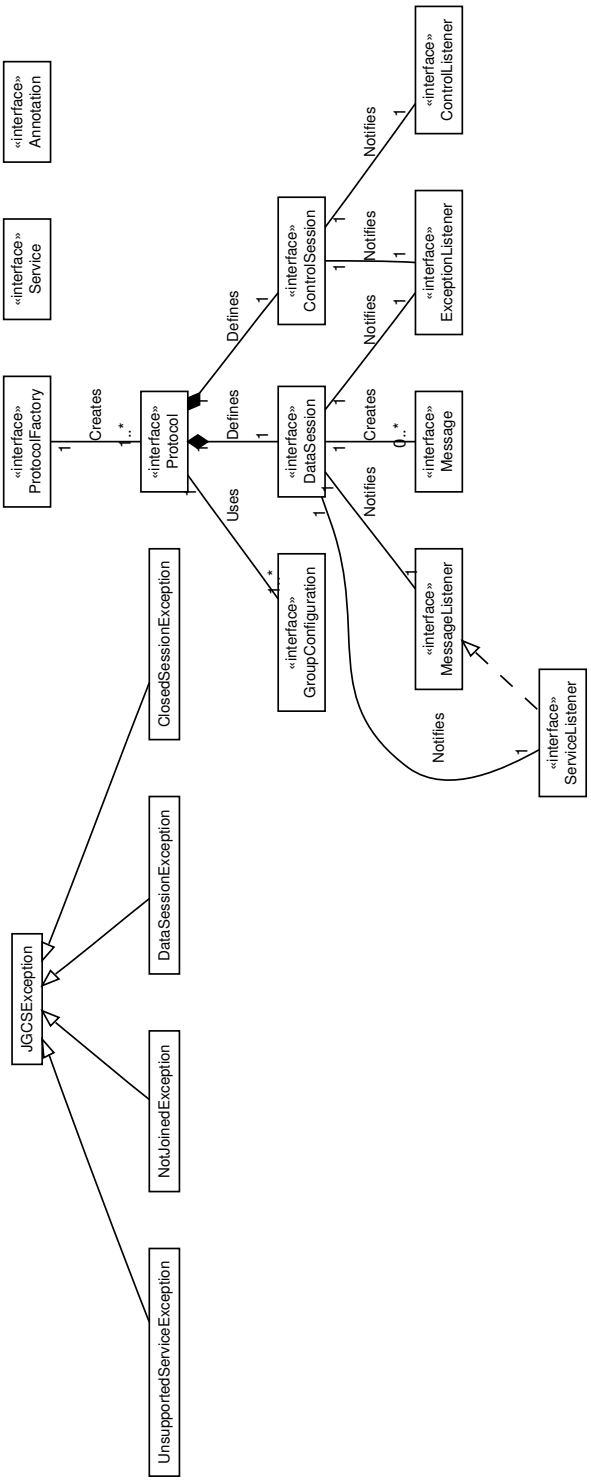
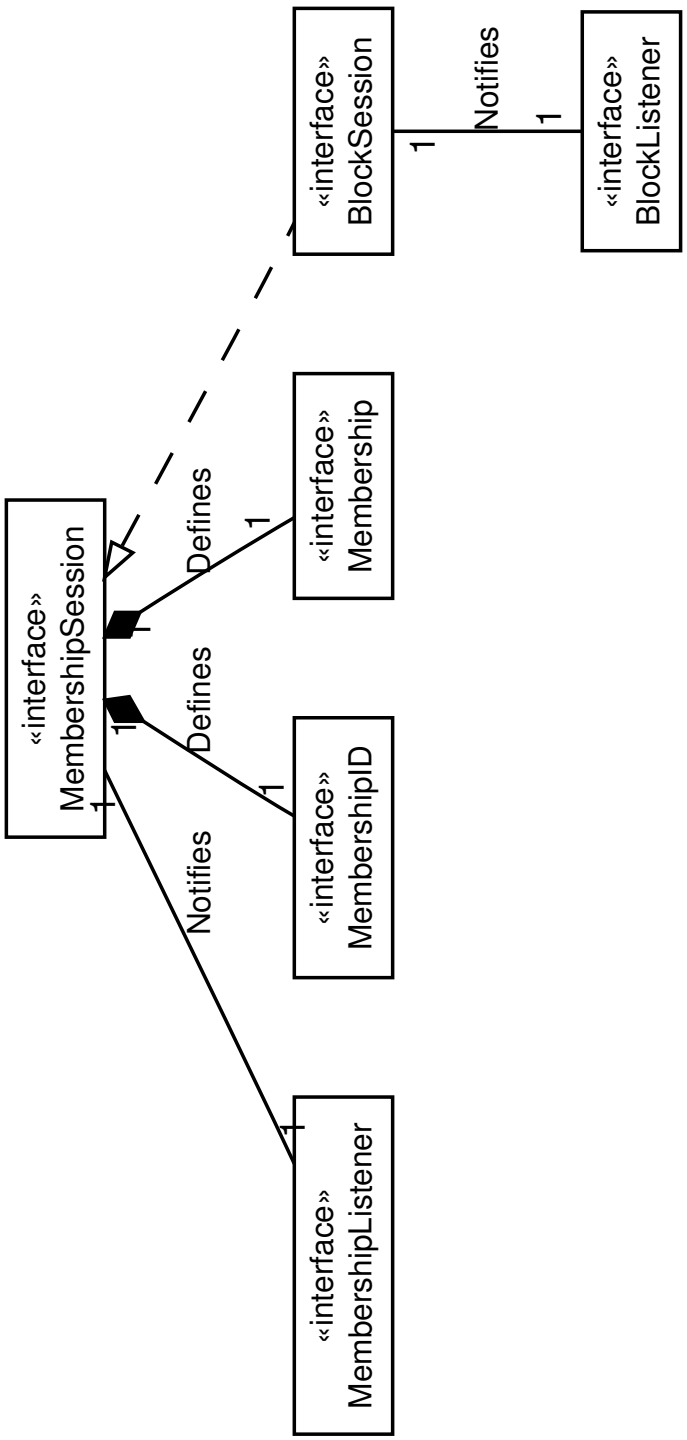


Figure 5: Extension interfaces for virtual synchrony.



4.1 Package net.sf.jgcs

4.1.1 Interface Annotation

This class defines a Annotation. An Annotation should be used by the application to give semantic information about the message to the Channel protocols implementation.

Declaration public interface Annotation

4.1.2 Interface ControlListener

This class defines a ControlListener. This listener must be used by clients that wish to be notified of changes in the members that join, leave or fail in a simple group.

Declaration public interface ControlListener

Methods

- **onFailed**
  `void` **onFailed**`( java.net.SocketAddress` **peer** `)`
  - **Description**
    Notification of a member that was detected as failed. This notification means also that the member does not belong to the group any more.
  - **Parameters**
    * `peer` – the address of the member that failed.
- **onJoin**
  `void` **onJoin**`( java.net.SocketAddress` **peer** `)`
  - **Description**
    Notification of a new member in the group.
  - **Parameters**
    * `peer` – the address of the new member.
- **onLeave**
  `void` **onLeave**`( java.net.SocketAddress` **peer** `)`
  - **Description**
    Notification of a member that leaved the group.
  - **Parameters**
    * `peer` – the address of the leaved member.

4.1.3 Interface ControlSession

This class defines a ControlSession. This Session is used to join and leave a simple group. It is also used to register a ControlListener. An instance of this session must be created by the Protocol interface.

4.1.4 See also

– 4.1.13, page 25

Declaration public interface ControlSession

All known subinterfaces MembershipSession (in 4.2.7, page 37), BlockSession (in 4.2.2, page 33)

Methods

- **getLocalAddress**
  `java.net.SocketAddress` **getLocalAddress**`( )`
  - **Description**
    Gets the local address. It should return null if the member is not joined to any group.
  - **Returns** – the local address.
- **isJoined**
  `boolean` **isJoined**`( )`
  - **Description**
    Verifies if the member belongs to a group.
  - **Returns** – true if the member is correctly joined, false otherwise.
- **join**
  `void` **join**`( )`
  `throws net.sf.jgcs.ClosedSessionException, net.sf.jgcs.JGCSException`
  - **Description**
    Joins the group. It must block until the join process is finished.
- **leave**
  `void` **leave**`( )`
  `throws net.sf.jgcs.ClosedSessionException, net.sf.jgcs.JGCSException`
  - **Description**
    Leaves the group. It must block until the leave process is finished.

- **setControlListener**
  `void` **setControlListener**`( ControlListener` **listener** `)`
  - **Description**
    Adds a listener to deliver group membership notifications.
  - **Parameters**
    * `listener` – The listener to be bound to the membership service.
- **setExceptionListener**
  `void` **setExceptionListener**`( ExceptionListener` **exception** `)`
  `throws net.sf.jgcs.ClosedSessionException`
  - **Description**
    Adds a listener to deliver exceptions related to message reception and membership notifications.
  - **Parameters**
    * `exception` – the exception thrown by the implementation of the interface.

4.1.5 Interface DataSession

This class defines a DataSession. This Session must be used to send and receive messages to/from the group. An instance of a DataSession must be created on the Protocol interface.

4.1.6 See also

– 4.1.13, page 25

Declaration public interface DataSession

Methods

- **close**
  `void` **close**`( )`
  - **Description**
    Closes the session. All resources that the session holds should be freed and therefore no subsequent communication can be done.
- **createMessage**
  `Message` **createMessage**`( )`
  `throws net.sf.jgcs.ClosedSessionException`
  - **Description**
    Creates an empty message that can be used (transmitted) through the session.

- **Returns** – The message created.

- **getGroup**
  `GroupConfiguration` **getGroup**`( )`
  - **Description**
    Gets the group associated with this session.
  - **Returns** – the group associated with this session.

- **multicast**
  `void` **multicast**`( Message` **msg**`, Service` **service**`,`
  `java.lang.Object` **cookie**`, Annotation[]` **annotation** `)`
  `throws java.io.IOException, net.sf.jgcs.UnsupportedServiceException`
  - **Description**
    Sends a message to the group.
  - **Parameters**
    * `msg` – The message to be sent.
    * `service` – the service needed by the application for message delivery (e.g. total order) or null to use the default channel service.
    * `cookie` – a cookie used to identify the message in the future (e.g. service notifications).
    * `annotation` – semantic information provided by the application to be used by communication protocols (e.g. semantic reliability).
  - **Throws**
    * `java.io.IOException` –

- **send**
  `void` **send**`( Message` **msg**`, Service` **service**`,`
  `java.lang.Object` **cookie**`, java.net.SocketAddress` **destination**`, Annotation[]` **annotation** `)`
  `throws java.io.IOException, net.sf.jgcs.UnsupportedServiceException`
  - **Description**
    Sends a message to one particular member of the group.
  - **Parameters**
    * `msg` – The message to be sent.
    * `service` – the service needed by the application for message delivery (e.g. total order) or null to use the default channel service.
    * `cookie` – a cookie used to identify the message in the future (e.g. service notifications).
    * `destination` – the destination of the message.

    * `annotation` – semantic information provided by the application to be used by communication protocols (e.g. semantic reliability).
  - **Throws**
    * `java.io.IOException` –

- **setExceptionListener**
  `void` **setExceptionListener**`( ExceptionListener` **exception** `)`
  `throws net.sf.jgcs.ClosedSessionException`
  - **Description**
    Adds a listener to deliver exceptions related to message reception.
  - **Parameters**
    * `exception` – the exception thrown by the implementation of the interface.

- **setMessageListener**
  `void` **setMessageListener**`( MessageListener` **listener** `)`
  `throws net.sf.jgcs.ClosedSessionException`
  - **Description**
    Adds a listener to deliver messages from this channel.
  - **Parameters**
    * `listener` – The listener to be bound to the channel.

- **setServiceListener**
  `void` **setServiceListener**`( ServiceListener` **listener** `)`
  `throws net.sf.jgcs.ClosedSessionException`
  - **Description**
    Adds a listener to deliver notifications from this channel.
  - **Parameters**
    * `listener` – the listener to be bound to the channel.

4.1.7 Interface ExceptionListener

This class defines a ExceptionListener. This listener must be used to receive exceptions that could occour on message reception.

Declaration public interface ExceptionListener

Methods

- **onException**
  `void` **onException**`( JGCSException` **exception** `)`
  - **Description**
    Notification of an exception that occurred when the underlying implementation was receiving a message.
  - **Parameters**
    * `exception` – the exception.

4.1.8 Interface GroupConfiguration

This class defines a GroupConfiguration. Interface that provides a Group configuration to open Sessions (in 4.1.5, page 21). This Interface must be used together with the Protocol (in 4.1.13, page 25) to create a DataSession (in 4.1.5, page 21) and a ControlSession (in 4.1.3, page 20).

4.1.9 See also

- 4.1.5, page 21
- 4.1.3, page 20
- 4.1.13, page 25

Declaration public interface GroupConfiguration

4.1.10 Interface Message

This class defines a Message. Messages exchanged using the underlying toolkit must implement this interface. Instances of this interface must be retrieved from the DataSession (in 4.1.5, page 21).

Declaration public interface Message

Methods

- **getPayload**
  `byte[]` **getPayload**`( )`
  - **Description**
    Gets the payload from the message.
  - **Returns** – the payload from the message.
- **getSenderAddress**
  `java.net.SocketAddress` **getSenderAddress**`( )`
  - **Description**
    Gets the sender address.
  - **Returns** – the sender address
- **setPayload**
  `void` **setPayload**`( byte[]` **buffer** `)`
  - **Description**
    Sets the payload for the message.
  - **Parameters**
    * `buffer` – The payload to be stored in the message.

- **setSenderAddress**
  `void` **setSenderAddress**`( java.net.SocketAddress` **sender** `)`
  - **Description**
    Sets the sender address.
  - **Parameters**
    * `sender` – the sender address.

4.1.11 Interface MessageListener

This class defines a MessageListener. This listener must be used to receive messages.

4.1.12 See also

- 4.1.5, page 21
- 4.1.16, page 27
- 4.1.17, page 27

Declaration public interface MessageListener

All known subinterfaces ServiceListener (in 4.1.17, page 27)

Methods

- **onMessage**
  `java.lang.Object` **onMessage**`( Message` **msg** `)`
  - **Description**
    Delivers a message from the channel to the application. To use this listener together with the Services, a cookie must be returned by the application.
  - **Parameters**
    * `msg` – The message received from the channel.
  - **Returns** – the cookie of the message.

4.1.13 Interface Protocol

This interface defines a Protocol represents an instance of the toolkit used to implement the Group Communication Service (GCS). This interface must be used to create instances of DataSession and Control Session.

4.1.14 See also

- 4.1.5, page 21
- 4.1.3, page 20
- 4.1.8, page 24

Declaration public interface Protocol

Methods

- **openControlSession**
  `ControlSession` **openControlSession**`( GroupConfiguration` **group** `)`
  `throws net.sf.jgcs.JGCSException`
  - **Description**
    Creates a new Control Session. This session must be used to join a group and register a listener to receive asynchronous notifications about the other members of the group (join, leave, fail).
  - **Parameters**
    * `group` – the group configuration.
  - **Returns** – a new control session.
  - **Throws**
    * `net.sf.jgcs.JGCSException` –
- **openDataSession**
  `DataSession` **openDataSession**`( GroupConfiguration` **group** `)`
  `throws net.sf.jgcs.JGCSException`
  - **Description**
    Creates e new Data Session. This session must be used to send messages and to register a listener to receive messages from the other members of the group.
  - **Parameters**
    * `group` – the configuration.
  - **Returns** – a new data session.
  - **Throws**
    * `net.sf.jgcs.JGCSException` –

4.1.15 Interface ProtocolFactory

This class defines a ProtocolFactory This factory must be used to create instances of Protocols. It should be stateless and represents one toolkit.

Declaration public interface ProtocolFactory

Methods

- **createProtocol**
  `Protocol` **createProtocol**`( )`
  `throws net.sf.jgcs.JGCSException`

- **Description**
  Creates a new Protocol that represents a toolkit.
- **Returns** – a new protocol.
- **Throws**
  * `net.sf.jgcs.JGCSException` –

4.1.16 Interface Service

This class defines a Service. A Service is some functionality that the channel needs to provide to the application. One example is the optimistic total order. If an application creates a channel that provides optimistic total order, the application will receive the message payload with out guarantees and will be notified later about optimistic delivery, regular delivery, uniform delivery, etc. These notifications must implement this interface. All related services must be comparable with each other (e.g. uniform delivery is a stronger service than regular delivery, so if the message is uniform, it's also regular and optimistic – optimistic lower than regular lower than uniform).

Declaration public interface Service

Methods

- **compare**
  `int` **compare**`( Service` **service** `)`
  `throws net.sf.jgcs.UnsupportedServiceException`
  - **Description**
    Compares two Services of the same protocol. return 0 if the services are the same, -1 if the service has lower properties than the given service, 1 if the service has greater properties than the given service.
  - **Parameters**
    * `service` – the service to compare.
  - **Returns** – 0 - same service, 1 greater service, -1 otherwise
  - **Throws**
    * `net.sf.jgcs.UnsupportedServiceException` – if the service is not comparable.

4.1.17 Interface ServiceListener

This class defines a ServiceListener. Listeners interested in receiving notifications about guarantees of requested services on messages must implement this interface.

4.1.18 See also

- 4.1.5, page 21
- 4.1.16, page 27

Declaration public interface ServiceListener
extends MessageListener

Methods

- **onServiceEnsured**
  `void` **onServiceEnsured**`( java.lang.Object` **context**`, Service` **service** `)`
  - **Description**
    Notifies the application that one certain service to a message delivery is already ensured. The message is identified by the context. This context must be previously provided by the application.
  - **Parameters**
    * `context` – context previously provided by the application that identifies a message.
    * `service` – service ensured.

4.1.19 Exception ClosedSessionException

This class defines a ClosedSessionException.

Declaration public class ClosedSessionException
extends net.sf.jgcs.JGCSException (in 4.1.21, page 29)

Constructors

- **ClosedSessionException**
  `public` **ClosedSessionException**`( )`
- **ClosedSessionException**
  `public` **ClosedSessionException**`( java.lang.String` **s** `)`
- **ClosedSessionException**
  `public` **ClosedSessionException**`( java.lang.String` **s**`,`
  `java.lang.Throwable` **t** `)`

4.1.20 Exception DataSessionException

This class defines a DataSessionException.

Declaration public class DataSessionException
extends net.sf.jgcs.JGCSException (in 4.1.21, page 29)

Constructors

- **DataSessionException**
  `public` **DataSessionException** `( )`
  - **Description**
    Creates a new DataSessionException.
- **DataSessionException**
  `public` **DataSessionException** `( java.lang.String` **message** `)`
  - **Description**
    Creates a new DataSessionException.
  - **Parameters**
    * `message` – the error message.
- **DataSessionException**
  `public` **DataSessionException** `( java.lang.String` **message,** `java.lang.Throwable` **cause** `)`
  - **Description**
    Creates a new DataSessionException.
  - **Parameters**
    * `message` – the error message
    * `cause` – the thowable that caused this exception.

4.1.21 Exception JGCSException

This class defines a JGCSException.

Declaration public class JGCSException
extends java.io.IOException

All known subclasses UnsupportedServiceException (in 4.1.23, page 32), NotJoinedException (in 4.1.22, page 31), DataSessionException (in 4.1.20, page 28), ClosedSessionException (in 4.1.19, page 28)

Constructors

- **JGCSException**
  `public` **JGCSException** `( )`
  - **Description**
    Creates a new JGCSException.

- **JGCSException**
  `public` **JGCSException**`( java.lang.String` **s** `)`
  - **Description**
    Creates a new JGCSException.
  - **Parameters**
    - `s` – the error message.
- **JGCSException**
  `public` **JGCSException**`( java.lang.String` **s,** `int` **code** `)`
  - **Description**
    Creates a new JGCSException.
  - **Parameters**
    - `s` – the error message.
    - `code` – the error code.
- **JGCSException**
  `public` **JGCSException**`( java.lang.String` **s,**
  `java.lang.Throwable` **cause** `)`
  - **Description**
    Creates a new JGCSException.
  - **Parameters**
    - `s` – the error message.
    - `cause` – the throwable that caused this exception.
- **JGCSException**
  `public` **JGCSException**`( java.lang.String` **s,**
  `java.lang.Throwable` **cause,** `int` **code** `)`
  - **Description**
    Creates a new JGCSException.
  - **Parameters**
    - `s` – the error message
    - `cause` – the throwable that caused this exception.
    - `code` – the error code

Methods

- **getCause**
  `public java.lang.Throwable` **getCause**`( )`
  - **Description**
    Gets the throwable that caused this exception.
- **getErrorCode**
  `public int` **getErrorCode**`( )`
  - **Description**
    Gets the error code that identifies the error ocurred.
  - **Returns** – the error code.

4.1.22 Exception NotJoinedException

This class defines a NotJoinedException.

Declaration public class NotJoinedException
extends net.sf.jgcs.JGCSException (in 4.1.21, page 29)

Constructors

- **NotJoinedException**
  `public` **NotJoinedException**`( )`
  - **Description**
    Creates a new NotJoinedException.
- **NotJoinedException**
  `public` **NotJoinedException**`( java.lang.String` **s** `)`
  - **Description**
    Creates a new NotJoinedException.
  - **Parameters**
    * `s` – the error message
- **NotJoinedException**
  `public` **NotJoinedException**`( java.lang.String` **s**`, int` **code** `)`
  - **Description**
    Creates a new NotJoinedException.
  - **Parameters**
    * `s` – the error message.
    * `code` – the error code.
- **NotJoinedException**
  `public` **NotJoinedException**`( java.lang.String` **s**`, java.lang.Throwable` **cause** `)`
  - **Description**
    Creates a new NotJoinedException.
  - **Parameters**
    * `s` – the error message.
    * `cause` – the throwable that caused this exception.
- **NotJoinedException**
  `public` **NotJoinedException**`( java.lang.String` **s**`, java.lang.Throwable` **cause**`, int` **code** `)`
  - **Description**
    Creates a new NotJoinedException.
  - **Parameters**
    * `s` – the error message.
    * `cause` – the throwable that caused this exception.
    * `code` – the error code.

4.1.23 Exception UnsupportedServiceException

This class defines a UnsupportedServiceException.

Declaration public class UnsupportedServiceException
extends net.sf.jgcs.JGCSException (in 4.1.21, page 29)

Constructors

- **UnsupportedServiceException**
  `public` **UnsupportedServiceException** `( )`
  - **Description**
    Creates a new UnsupportedServiceException.
- **UnsupportedServiceException**
  `public` **UnsupportedServiceException** `( java.lang.String` **message** `)`
  - **Description**
    Creates a new UnsupportedServiceException.
  - **Parameters**
    * `message` – the error message.
- **UnsupportedServiceException**
  `public` **UnsupportedServiceException** `( java.lang.String` **message,** `java.lang.Throwable` **cause** `)`
  - **Description**
    Creates a new UnsupportedServiceException.
  - **Parameters**
    * `message` – the error message.
    * `cause` – the throwable that caused this exception.

4.2 Package net.sf.jgcs.membership

4.2.1 Interface BlockListener

This class defines a BlockListener. This listener must be used to receive notifications that a group membership will block.

Declaration public interface BlockListener

Methods

- **onBlock**
  `void` **onBlock**`( )`
  - **Description**
    Block notification. Upon this notification, the application must flush all pending messages and notify the session with the (in 4.2.2, page 33) method. The view change will not continue if this does not happen. After the group is blocked, the members cannot send more messages until a new Membership view is received.

4.2.2 Interface BlockSession

This class defines a BlockSession. This session should be used by toolkits that implement Group Communication with flush of messages before a view change.

Declaration public interface BlockSession
extends MembershipSession

Methods

- **blockOk**
  `void` **blockOk**`( )`
  `throws net.sf.jgcs.NotJoinedException,`
  `net.sf.jgcs.JGCSException`
  - **Description**
    This method must be used by the application after it received a block notification and flushed all pending messages. After calling this method, the application cannot send any more messages until it receives a notification of a membership change.
  - **Throws**
    * `net.sf.jgcs.NotJoinedException` – if the member is not in a group.
    * `net.sf.jgcs.JGCSException` – if an error ocurs.

- **isBlocked**
  `boolean` **isBlocked**`( )`
  `throws net.sf.jgcs.NotJoinedException`
  - **Description**
    Verifies if the group is blocked or not.
  - **Returns** – true if the group is blocked, false otherwise.
  - **Throws**
    * `net.sf.jgcs.NotJoinedException` – if the member is not in a group.
- **setBlockListener**
  `void` **setBlockListener**`( BlockListener` **listener** `)`
  `throws net.sf.jgcs.JGCSException`
  - **Description**
    Registers a listener for the block notification.
  - **Parameters**
    * `listener` – the listener to register.
  - **Throws**
    * `net.sf.jgcs.JGCSException` – if an error ocurs.

4.2.3 Interface Membership

This class defines a Membership.

Declaration public interface Membership

Methods

- **getCoordinatorRank**
  `int` **getCoordinatorRank**`( )`
  - **Description**
    Gets the rank of the coordinator of this group.
  - **Returns** – the rank of the coordinator of the group.
- **getFailedMembers**
  `java.util.List` **getFailedMembers**`( )`
  - **Description**
    Gets a list of members that failed since the previous membership.
  - **Returns** – a list of failed members or null if there are none.
- **getJoinedMembers**
  `java.util.List` **getJoinedMembers**`( )`

  - **Description**
    Gets a list of members that joined the group since the previous membership.
  - **Returns** – a list of new members or null if there are none.

- **getLeavedMembers**
  `java.util.List` **getLeavedMembers**`( )`
  - **Description**
    Gets a list of members that leaved the group since the previous membership.
  - **Returns** – a list of old members or null if there are none.

- **getLocalRank**
  `int` **getLocalRank**`( )`
  `throws net.sf.jgcs.NotJoinedException`
  - **Description**
    Gets the local rank of the member in this membership.
  - **Returns** – the local rank of this member.
  - **Throws**
    * `net.sf.jgcs.NotJoinedException` – if the member is not in a group.

- **getMemberAddress**
  `java.net.SocketAddress` **getMemberAddress**`( int` **rank** `)`
  - **Description**
    Gets the socket address of the member that has the given rank.
  - **Parameters**
    * `rank` – the rank of the member.
  - **Returns** – the socket address of the member.

- **getMemberRank**
  `int` **getMemberRank**`( java.net.SocketAddress` **peer** `)`
  - **Description**
    Gets the member rank that has the given socket address, or null if there is no matching rank.
  - **Parameters**
    * `peer` – the socket address of the member.
  - **Returns** – the rank of the member.

- **getMembershipID**
  `MembershipID` **getMembershipID**`( )`
  - **Description**
    Gets the current membership ID.
  - **Returns** – the current membership ID.

- **getMembershipList**
  `java.util.List` **getMembershipList**`( )`
  - **Description**
    Gets the current view of the membership.
  - **Returns** – the current view of the membership.

4.2.4 Interface MembershipID

This class defines a MembershipID. It represents an ID of the membership, that must change and grow on every view change, according to the java.lang.Comparable interface.

4.2.5 See also

- `java.lang.Comparable`

Declaration public interface MembershipID
extends java.lang.Comparable

4.2.6 Interface MembershipListener

This class defines a MembershipListener. This listener must be used to receive membership, when the control session used implements the MembershipSession or BlockSession interfaces.

Declaration public interface MembershipListener

Methods

- **onExcluded**
  `void` **onExcluded**`( )`
  - **Description**
    Notification from the membership to indicate that the registered member does not belong to the group any more. This should happen when the member lost intermediate views (for instance, when using primary views) and lost some messages. After receiving this notification, the member may try to rejoin again.
- **onMembershipChange**
  `void` **onMembershipChange**`( )`
  - **Description**
    Notification of a MembershipChange. This should happen due to joining, leaving or failure of group members, but also because of merging or partitioning of memberships. The new membership can be retrieved from the MembershipSession.

4.2.7 Interface MembershipSession

This class defines a MembershipSession. This session should be implemented when the underlying toolkit provides extended view synchrony semantics.

Declaration public interface MembershipSession
extends net.sf.jgcs.ControlSession

All known subinterfaces BlockSession (in 4.2.2, page 33)

Methods

- **getMembership**
  `Membership` **getMembership** `( )`
  `throws net.sf.jgcs.NotJoinedException`
  - **Description**
    Gets the current Membership.
  - **Returns** – a membership.
  - **Throws**
    * `net.sf.jgcs.NotJoinedException` – if the member is not joined
- **getMembershipID**
  `MembershipID` **getMembershipID** `( )`
  `throws net.sf.jgcs.NotJoinedException`
  - **Description**
    Gets the current membership ID
  - **Returns** – the current membership ID
  - **Throws**
    * `net.sf.jgcs.NotJoinedException` – if the member is not joined
- **setMembershipListener**
  `void` **setMembershipListener** `( MembershipListener` **listener** `)`
  - **Description**
    Registers a listener for the membership changes.
  - **Parameters**
    * `listener` – the listener to register.

5 Samples

5.1 Third party configurator

This sample shows how to setup a group communication toolkit that was previously configured using a Naming and Directory Interface.

The sample uses virtual synchrony and implements all the listeners used to receive messages, exceptions and membership notifications.

```
public class JNDITest implements MessageListener, ControlListener,
MembershipListener, BlockListener, Runnable {

        private static final int NUM_MESSAGES=10;
        private ControlSession control;
        private DataSession data;
        private Context ctx;
        private Service service;

        public JNDITest(Context x) throws JGCSException, NamingException {
                this.ctx=x;
```

The first object to lookup is the protocol factory. This object represents the toolkit that will be used by this application.

```
                ProtocolFactory pf = (ProtocolFactory) x.lookup("myProto");
```

A protocol can now be created. This object represents an instance of the toolkit that will be used for group communication.

```
                Protocol p = pf.createProtocol();
```

The application must also lookup a GroupConfiguration object that represents a configuration of the group communication.

```
                GroupConfiguration g = (GroupConfiguration) x.lookup("myGroup");
```

A service object is needed to send messages. The application may use different services for different messages, if it need to send messages with different qualities of service.

```
                service = (Service) ctx.lookup("myService");
```

Using the configuration object provided by a the configuration process and the previously created protocol, instances of data and control sessions can now be created. A data session will be used to send and receive messages. The control session will be used to join the group and receive notifications concerning the other elements of the group.

```
                this.control = p.openControlSession(g);
                this.data = p.openDataSession(g);
```

The listeners must be set before the application starts using the group communication toolkit.

```
                data.setMessageListener(this);
                control.setControlListener(this);
                if (control instanceof MembershipSession)
                        ((MembershipSession) control).setMembershipListener(this);
                if (control instanceof BlockSession)
                        ((BlockSession) control).setBlockListener(this);
        }
```

This method will run after the creation of the class. At this point, all the necessary objects were already retrieved from the lookup service. The application joins the group, sends some messages and finally leaves the group.

```
	public void run() {
		try {
			control.join();
			for (int i = 0; i < NUM_MESSAGES; i++) {
				Thread.sleep(1000);
```

A new message object must be created using the data session.

```
				Message message = data.createMessage();
				message.setPayload("hello world!".getBytes());
```

The message is sent to the group using the service previously retrieved from the lookup service.

```
				data.multicast(message, service, null);
			}
			Thread.sleep(5000);
```

All resources should be freed in the control and data sessions.

```
			control.leave();
			data.close();
		} catch(Exception e) {
			e.printStackTrace();
		}
	}
```

This method represents the message listener. Every time that a message is sent to the group, it is received in this callback by all elements of the group. The application can return an object to identify this particular message in the future, but this feature is not used at the moment. This feature is discussed in other sample.

```
	public Object onMessage(Message msg) {
		System.out.println("Message from "+msg.getSenderAddress()
				+": "+new String(msg.getPayload()));
		return null;
	}
```

These call backs are used to notify the application that some member has joined, left or failed. This is not necessary if the application is using a Membership or Block sessions.

```
	public void onJoin(SocketAddress peer) {
		System.out.println("-- JOIN: " + peer);
	}

	public void onLeave(SocketAddress peer) {
		System.out.println("-- LEAVE: " + peer);
	}

	public void onFailed(SocketAddress peer) {
		System.out.println("-- FAILED: " + peer);
	}
```

This notification is issued every time that the group membership changes. It is only used if the membership extentions were implemented and may be used instead of the previous call backs. The new membership may be retrieved from the membership session.

```
        public void onMembershipChange() {
                try {
                        System.out.println("-- NEW MEMBERSHIP: " +
                                ((MembershipSession) control).getMembership());
                } catch (NotJoinedException e) {
                        e.printStackTrace();
                        data.close();
                }
        }
```

This call back notifies the application that the group will block and a new membership will be received. The application must flush any pending messages at this time and call the blockOk method from the control session. The membership will not be received if the application do not call this method.

```
        public void onBlock() {
                try {
                        ((BlockSession) control).blockOk();
                } catch (JGCSException e) {
                        e.printStackTrace();
                }
        }
```

This call back is used to notify the application that it was removed from the group.

```
        public void onExcluded() {
                System.out.println("-- REMOVED from group.");
        }

        public static void main(String[] args) {
                try {
                        Context x = new InitialContext();
                        Runnable test = new JNDITest(x);
                        test.run();
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

5.2 Early deliveries

This sample shows how to send and receive messages using a toolkit that was configured to make early deliveries and service notifications.

The sample uses extended virtual synchrony and implements all the listeners used to receive messages, exceptions and membership notifications.

```
public class EarlyDeliveryTest implements MembershipListener,
        ServiceListener, Runnable {

        private static final int NUM_MESSAGES=10;
        private ControlSession control;
        private DataSession data;
        private Context ctx;
        private Service uniformService;

        public EarlyDeliveryTest(Context x)
        throws JGCSException, NamingException {
                this.ctx=x;
```

The startup is similar to the other sample.

```
                ProtocolFactory pf = (ProtocolFactory) x.lookup("myProto");
                Protocol p = pf.createProtocol();
                GroupConfiguration g = (GroupConfiguration) x.lookup("myGroup");
                uniformService = (Service) ctx.lookup("myService");
                this.control = p.openControlSession(g);
                this.data = p.openDataSession(g);
```

The application must register it self on both message and service listeners.

```
                data.setMessageListener(this);
                data.setServiceListener(this);
                if (control instanceof MembershipSession)
                        ((MembershipSession) control).setMembershipListener(this);
        }

        public void run() {
                try {
                        control.join();
                        for (int i = 0; i < NUM_MESSAGES; i++) {
                                Thread.sleep(1000);
                                Message message = data.createMessage();
                                message.setPayload("hello world!".getBytes());
```

The message is sent to the group using the service previously retrieved from the lookup service.

```
                                data.multicast(message, uniformService, null);
                        }
                        Thread.sleep(5000);
                        control.leave();
                        data.close();
                } catch(Exception e) {
                        e.printStackTrace();
                }
        }
```

This method represents the message listener. Every time that a message is sent to the group, it is received in this callback by all elements of the group. In this example, the application assumes that the group communication was configured to make early deliveries of messages. This means that the message payload is delivered before the requested service is ensured. In this example, the application may process the message but will not show that message until the required service is received.

Note that the processing of these messages is not complex (in this example) but some applications can have complex processing based on the message contents, that can be done before printing results to the user or writing results to physical storage.

```
        public Object onMessage(Message msg) {
                String messageToPrint = "Message␣from␣"+msg.getSenderAddress()
                        +":␣"+new String(msg.getPayload());
                return messageToPrint;
        }
```

This method represents the service listener. For each message, several services can be provided. These services have an order relation. This example only prints the messages that have already the uniform property.

```
        public void onServiceEnsured(Object context, Service service) {
                try {
                        if(service.compare(uniformService) >= 0){
                                String messageToPrint = (String) context;
                                System.out.println(messageToPrint);
                        }
                } catch (UnsupportedServiceException e) {
                        e.printStackTrace();
                }
        }

        public void onMembershipChange() {
                try {
                        System.out.println("--␣NEW␣MEMBERSHIP:␣" +
                                ((MembershipSession) control).getMembership());
                } catch (NotJoinedException e) {
                        e.printStackTrace();
                        data.close();
                }
        }

        public void onExcluded() {
                System.out.println("--␣REMOVED␣from␣group.");
        }

        public static void main(String[] args) {
                try {
                        Context x = new InitialContext();
                        Runnable test = new EarlyDeliveryTest(x);
                        test.run();
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }

}
```

A License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

a. **"Adaptation"** means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

b. **"Collection"** means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined above) for the purposes of this License.

c. **"Distribute"** means to make available to the public the original and copies of the Work through sale or other transfer of ownership.

d. **"Licensor"** means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

e. **"Original Author"** means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

f. **"Work"** means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

g. **"You"** means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

h. **"Publicly Perform"** means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

i. **"Reproduce"** means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections; and,

b. to Distribute and Publicly Perform the Work including as incorporated in Collections.

c. For the avoidance of doubt:

   i. **Non-waivable Compulsory License Schemes.** In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

   ii. **Waivable Compulsory License Schemes.** In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,

   iii. **Voluntary License Schemes.** The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

   The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats, but otherwise you have no rights to make Adaptations. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(b), as requested.

b. If You Distribute, or Publicly Perform the Work or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work. The credit required by this Section 4(b) may be implemented in any reasonable manner; provided, however, that in the case of a Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

c. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

b. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

c. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

d. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

e. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.