Reliable group communication with JGroups 3.x

by Bela Ban (Red Hat, Inc.) and Vladimir Blagojevic (Red Hat, Inc.)

Preface

This is the JGroups manual. It provides information about:

1. Installation and configuration
2. Using JGroups (the API)
3. Configuration of the JGroups protocols

The focus is on how to *use* JGroups, not on how JGroups is implemented.

Here are a couple of points I want to abide by throughout this book:

1. I like brevity. I will strive to describe concepts as clearly as possible (for a non-native English speaker) and will refrain from saying more than I have to to make a point.
2. I like simplicity. Keep It Simple and Stupid. This is one of the biggest goals I have both in writing this manual and in writing JGroups. It is easy to explain simple concepts in complex terms, but it is hard to explain a complex system in simple terms. I'll try to do the latter.

So, how did it all start?

I spent 1998-1999 at the Computer Science Department at Cornell University as a post-doc, in Ken Birman's group. Ken is credited with inventing the group communication paradigm, especially the Virtual Synchrony model. At the time they were working on their third generation group communication prototype, called Ensemble. Ensemble followed Horus (written in C by Robbert VanRenesse), which followed ISIS (written by Ken Birman, also in C). Ensemble was written in OCaml, developed at INRIA, and is a functional language and related to ML. I never liked the OCaml language, which in my opinion has a hideous syntax. Therefore I never got warm with Ensemble either.

However, Ensemble had a Java interface (implemented by a student in a semester project) which allowed me to program in Java and use Ensemble underneath. The Java part would require that an Ensemble process was running somewhere on the same machine, and would connect to it via a bidirectional pipe. The student had developed a simple protocol for talking to the Ensemble engine, and extended the engine as well to talk back to Java.

However, I still needed to compile and install the Ensemble runtime for each different platform, which is exactly why Java was developed in the first place: portability.

Therefore I started writing a simple framework (now `JChannel`), which would allow me to treat Ensemble as just another group communication transport, which could be replaced at any time by a pure Java solution. And soon I found myself working on a pure Java implementation of the group communication transport (now: `ProtocolStack`). I figured that a pure Java implementation would have a much bigger impact that something written in Ensemble. In the end I didn't spend much time writing scientific papers that nobody would read anyway (I guess I'm not a good scientist, at

least not a theoretical one), but rather code for JGroups, which could have a much bigger impact. For me, knowing that real-life projects/products are using JGroups is much more satisfactory than having a paper accepted at a conference/journal.

That's why, after my time was up, I left Cornell and academia altogether, and started a job in the telecom industry in Silicon Valley.

At around that time (May 2000), SourceForge had just opened its site, and I decided to use it for hosting JGroups. This was a major boost for JGroups because now other developers could work on the code. From then on, the page hit and download numbers for JGroups have steadily risen.

In the fall of 2002, Sacha Labourey contacted me, letting me know that JGroups was being used by JBoss for their clustering implementation. I joined JBoss in 2003 and have been working on JGroups and JBossCache. My goal is to make JGroups the most widely used clustering software in Java ...

I want to thank all contributors to JGroups, present and past, for their work. Without you, this project would never have taken off the ground.

I also want to thank Ken Birman and Robbert VanRenesse for many fruitful discussions of all aspects of group communication in particular and distributed systems in general.

I want to dedicate this manual to Jeannette and Michelle.

Bela Ban, San Jose, Aug 2002, Kreuzlingen Switzerland 2011

Overview

Group communication uses the terms *group* and *member*. Members are part of a group. In the more common terminology, a member is a *node* and a group is a *cluster*. We use these terms interchangeably.

A node is a process, residing on some host. A cluster can have one or more nodes belonging to it. There can be multiple nodes on the same host, and all may or may not be part of the same cluster. Nodes can of course also run on different hosts.

JGroups is toolkit for reliable group communication. Processes can join a group, send messages to all members or single members and receive messages from members in the group. The system keeps track of the members in every group, and notifies group members when a new member joins, or an existing member leaves or crashes. A group is identified by its name. Groups do not have to be created explicitly; when a process joins a non-existing group, that group will be created automatically. Processes of a group can be located on the same host, within the same LAN, or across a WAN. A member can be part of multiple groups.

The architecture of JGroups is shown in *Figure 1.1, "The architecture of JGroups"*.

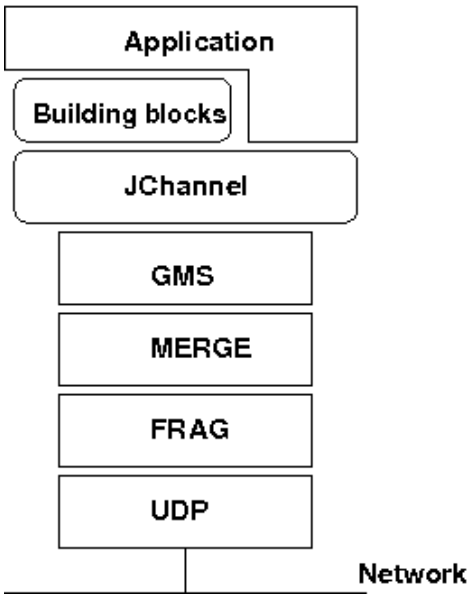

Figure 1.1. The architecture of JGroups

It consists of 3 parts: (1) the Channel used by application programmers to build reliable group communication applications, (2) the building blocks, which are layered on top of the channel and provide a higher abstraction level and (3) the protocol stack, which implements the properties specified for a given channel.

This document describes how to install and *use* JGroups, ie. the Channel API and the building blocks. The targeted audience is application programmers who want to use JGroups to build reliable distributed programs that need group communication.

A channel is connected to a protocol stack. Whenever the application sends a message, the channel passes it on to the protocol stack, which passes it to the topmost protocol. The protocol processes the message and the passes it down to the protocol below it. Thus the message is handed from protocol to protocol until the bottom (transport) protocol puts it on the network. The same happens in the reverse direction: the transport protocol listens for messages on the network. When a message is received it will be handed up the protocol stack until it reaches the channel. The channel then invokes the receive() callback in the application to deliver the message.

When an application connects to the channel, the protocol stack will be started, and when it disconnects the stack will be stopped. When the channel is closed, the stack will be destroyed, releasing its resources.

The following three sections give an overview of channels, building blocks and the protocol stack.

1.1. Channel

To join a group and send messages, a process has to create a *channel* and connect to it using the group name (all channels with the same name form a group). The channel is the handle to the group. While connected, a member may send and receive messages to/from all other group members. The client leaves a group by disconnecting from the channel. A channel can be reused: clients can connect to it again after having disconnected. However, a channel allows only 1 client to be connected at a time. If multiple groups are to be joined, multiple channels can be created and connected to. A client signals that it no longer wants to use a channel by closing it. After this operation, the channel cannot be used any longer.

Each channel has a unique address. Channels always know who the other members are in the same group: a list of member addresses can be retrieved from any channel. This list is called a *view*. A process can select an address from this list and send a unicast message to it (also to itself), or it may send a multicast message to all members of the current view (also including itself). Whenever a process joins or leaves a group, or when a crashed process has been detected, a new *view* is sent to all remaining group members. When a member process is suspected of having crashed, a *suspicion message* is received by all non-faulty members. Thus, channels receive regular messages, and view and suspicion notifications.

The properties of a channel are typically defined in an XML file, but JGroups also allows for configuration through simple strings, URIs, DOM trees or even programmatically.

The Channel API and its related classes is described in *Chapter 3, API*.

1.2. Building Blocks

Channels are simple and primitive. They offer the bare functionality of group communication, and have been designed after the simple model of sockets, which are widely used and well understood. The reason is that an application can make use of just this small subset of JGroups, without having to include a whole set of sophisticated classes, that it may not even need. Also, a somewhat minimalistic interface is simple to understand: a client needs to know about 5 methods to be able to create and use a channel.

Channels provide asynchronous message sending/reception, somewhat similar to UDP. A message sent is essentially put on the network and the send() method will return immediately. Conceptual *requests*, or *responses* to previous requests, are received in undefined order, and the application has to take care of matching responses with requests.

JGroups offers building blocks that provide more sophisticated APIs on top of a Channel. Building blocks either create and use channels internally, or require an existing channel to be specified when creating a building block. Applications communicate directly with the building block, rather than the channel. Building blocks are intended to save the application programmer from having to write tedious and recurring code, e.g. request-response correlation, and thus offer a higher level of abstraction to group communication.

Building blocks are described in *Chapter 4, Building Blocks*.

1.3. The Protocol Stack

The protocol stack containins a number of protocol layers in a bidirectional list. All messages sent and received over the channel have to pass through all protocols. Every layer may modify, reorder, pass or drop a message, or add a header to a message. A fragmentation layer might break up a message into several smaller messages, adding a header with an id to each fragment, and re-assemble the fragments on the receiver's side.

The composition of the protocol stack, i.e. its protocols, is determined by the creator of the channel: an XML file defines the protocols to be used (and the parameters for each protocol). The configuration is then used to create the stack.

Knowledge about the protocol stack is not necessary when only *using* channels in an application. However, when an application wishes to ignore the default properties for a protocol stack, and configure their own stack, then knowledge about what the individual layers are supposed to do is needed.

Installation and Configuration

The installation refers to version 3.x of JGroups. Refer to the installation instructions (INSTALL.html) that are shipped with the JGroups version you downloaded for details.

The JGroups JAR can be downloaded from *SourceForge* [http://sourceforge.net/projects/javagroups/files/JGroups/]. It is named jgroups-x.y.z, where x=major, y=minor and z=patch version, for example jgroups-3.0.0.Final.jar. The JAR is all that's needed to get started using JGroups; it contains all core, demo and (selected) test classes, the sample XML configuration files and the schema.

The source code is hosted on *GitHub* [https://github.com/belaban/jgroups]. To build JGroups, ANT is currently used. In *Section 2.3, "Building JGroups (source distribution only)"* we'll show how to build JGroups from source.

2.1. Requirements

- JGroups 3.x requires JDK 6 or higher.
- There is no JNI code present so JGroups should run on all platforms.
- If you want to generate HTML-based test reports from the unittests, then xalan.jar needs to be in the CLASSPATH (also available in the lib directory)

2.2. Structure of the source version

The source version consists of the following directories and files:

- src: the sources
- tests: unit and stress tests
- lib: JARs needed to either run the unit tests, or build the manual etc. No JARs from here are required at runtime !
- conf: configuration files needed by JGroups, plus default protocol stack definitions
- doc: documentation

2.3. Building JGroups (source distribution only)

1. Download the sources from *GitHub* [https://github.com/belaban/jgroups], either via 'git clone', or the *download link* [https://github.com/belaban/JGroups/archives/master] into a directory "JGroups", e.g. /home/bela/JGroups.

2. Change to the JGroups directory
3. On UNIX systems use `build.sh`, on Windows `build.bat`: **`$> ./build.sh`**
4. This will compile all Java files (into the `classes` directory).
5. To generate the JARs: **`$> ./build.sh jar`**
6. This will generate the following JAR files in the `dist` directory:
   - `jgroups-3.x.y.jar` - the JGroups JAR
   - `jgroups-sources.jar` - the source code for the core classes and demos.
7. Now add the following directories to the classpath:
   a. `JGroups/classes`
   b. `JGroups/conf`
   c. All needed JAR files in `JGroups/lib` . To build from sources, the two Ant JARs are required. To run unit tests, the JUnit (and possibly Xalan) JARs are needed.
8. To generate JavaDocs simple run **`$> ./build.sh javadoc`** and the Javadoc documentation will be generated in the `dist/javadoc` directory
9. Note that - if you already have Ant installed on your system - you do not need to use build.sh or build.bat, simply invoke ant on the build.xml file. To be able to invoked ant from any directory below the root directory, place **`ANT_ARGS="-find build.xml -emacs"`** into the `.antrc` file in your home directory.
10. For more details on Ant see *http://jakarta.apache.org/ant/*.

2.4. Testing your Setup

To see whether your system can find the JGroups classes, execute the following command:

```
java org.jgroups.Version
```

or

```
java -jar jgroups-all.jar
```

You should see the following output (more or less) if the class is found:

```
$  java org.jgroups.Version

   Version:      3.0.0.Beta1
```

2.5. Running a Demo Program

To test whether JGroups works okay on your machine, run the following command twice:

```
java -Djava.net.preferIPv4Stack=true org.jgroups.demos.Draw
```

2 whiteboard windows should appear as shown in *Figure 2.1, "Screenshot of 2 Draw instances"*.

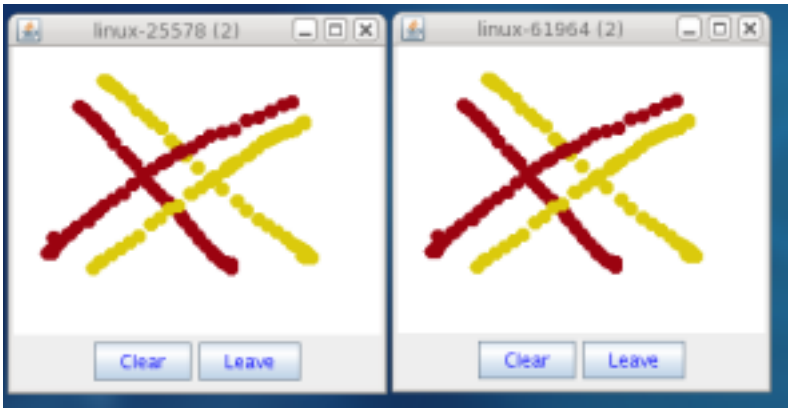

Figure 2.1. Screenshot of 2 Draw instances

If you started them simultaneously, they could initially show a membership of 1 in their title bars. After some time, both windows should show 2. This means that the two instances found each other and formed a cluster.

When drawing in one window, the second instance should also be updated. As the default group transport uses IP multicast, make sure that - if you want start the 2 instances in different subnets - IP multicast is enabled. If this is not the case, the 2 instances won't 'find' each other and the example won't work.

You can change the properties of the demo to for example use a different transport if multicast doesn't work (it should always work on the same machine). Please consult the documentation to see how to do this.

State transfer (see the section in the API later) can also be tested by passing the -state flag to Draw.

If the 2 instances find each other and form a cluster, you can skip ahead to the next chapter ("Writing a simple application").

2.6. Using IP Multicasting without a network connection

Sometimes there isn't a network connection (e.g. DSL modem is down), or we want to multicast only on the local machine. For this the loopback interface (typically lo) can be configured, e.g.

```
route add -net 224.0.0.0 netmask 240.0.0.0 dev lo
```

This means that all traffic directed to the 224.0.0.0 network will be sent to the loopback interface, which means it doesn't need any network to be running. Note that the 224.0.0.0 network is a placeholder for all multicast addresses in most UNIX implementations: it will catch *all* multicast traffic. This is an undocumented feature of `/sbin/route` and may not work across all UNIX flavors. The above instructions may also work for Windows systems, but this hasn't been tested. Note that not all systems allow multicast traffic to use the loopback interface.

Typical home networks have a gateway/firewall with 2 NICs: the first (eth0) is connected to the outside world (Internet Service Provider), the second (eth1) to the internal network, with the gateway firewalling/masquerading traffic between the internal and external networks. If no route for multicast traffic is added, the default will be to use the fdefault gateway, which will typically direct the multicast traffic towards the ISP. To prevent this (e.g. ISP drops multicast traffic, or latency is too high), we recommend to add a route for multicast traffic which goes to the internal network (e.g. eth1).

2.7. It doesn't work !

Make sure your machine is set up correctly for IP multicast. There are 2 test programs that can be used to detect this: McastReceiverTest and McastSenderTest. Start McastReceiverTest, e.g.

```
java org.jgroups.tests.McastReceiverTest
```

Then start McastSenderTest:

```
java org.jgroups.tests.McastSenderTest
```

If you want to bind to a specific network interface card (NIC), use -bind_addr 192.168.0.2, where 192.168.0.2 is the IP address of the NIC to which you want to bind. Use this parameter in both sender and receiver.

You should be able to type in the McastSenderTest window and see the output in the McastReceiverTest. If not, try to use -ttl 32 in the sender. If this still fails, consult a system administrator to help you setup IP multicast correctly. If you are the system administrator, look for another job :-)

Other means of getting help: there is a public forum on *JIRA* [http://jira.jboss.com/jira/browse/JGRP] for questions. Also consider subscribing to the javagroups-users mailing list to discuss such and other problems.

2.8. Problems with IPv6

Another source of problems might be the use of IPv6, and/or misconfiguration of `/etc/hosts`. If you communicate between an IPv4 and an IPv6 host, and they are not able to find each other, try the `-Djava.net.preferIP4Stack=true` property, e.g.

```
java -Djava.net.preferIPv4Stack=true org.jgroups.demos.Draw -props /home/
bela/udp.xml
```

The JDK uses IPv6 by default, although is has a dual stack, that is, it also supports IPv4. *Here's* [http://java.sun.com/j2se/1.4/docs/guide/net/ipv6_guide/] more details on the subject.

2.9. Wiki

There is a wiki which lists FAQs and their solutions at *http://www.jboss.org/wiki/Wiki.jsp?page=JGroups*. It is frequently updated and a useful companion to this user's guide.

2.10. I have discovered a bug !

If you think that you discovered a bug, submit a bug report on *JIRA* [http://jira.jboss.com/jira/browse/JGRP] or send email to the jgroups-users mailing list if you're unsure about it. Please include the following information:

- Version of JGroups (java org.jgroups.Version)
- Platform (e.g. Solaris 8)
- Version of JDK (e.g. JDK 1.4.2_07)
- Stack trace. Use kill -3 PID on UNIX systems or CTRL-BREAK on windows machines
- Small program that reproduces the bug

2.11. Supported classes

JGroups project has been around since 1998. Over this time, some of the JGroups classes have been used in experimental phases and have never been matured enough to be used in today's production releases. However, they were not removed since some people used them in their products.

The following tables list unsupported and experimental classes. These classes are not actively maintained, and we will not work to resolve potential issues you might find. Their final faith is not yet determined; they might even be removed altogether in the next major release. Weight your risks if you decide to use them anyway.

2.11.1. Experimental

Table 2.1. Experimental

Package	Class
org.jgroups.util	HashedTimingWheel
org.jgroups.blocks	GridOutputStream

Package	Class
org.jgroups.blocks	GridInputStream
org.jgroups.blocks	GridFile
org.jgroups.blocks	ReplCache
org.jgroups.blocks	PartitionedHashMap
org.jgroups.blocks	Cache
org.jgroups.blocks	GridFilesystem
org.jgroups.client	StompConnection
org.jgroups.protocols	COUNTER
org.jgroups.protocols	FD_ICMP
org.jgroups.protocols	STOMP
org.jgroups.protocols	BSH
org.jgroups.protocols	TUNNEL
org.jgroups.protocols	BPING
org.jgroups.protocols	HTOTAL
org.jgroups.protocols	SHUFFLE
org.jgroups.protocols	SWIFT_PING
org.jgroups.protocols	TCP_NIO
org.jgroups.protocols	DAISYCHAIN
org.jgroups.protocols	PRIO
org.jgroups.protocols.tom	TOA
org.jgroups.protocols	RATE_LIMITER
org.jgroups.protocols	RSVP

2.11.2. Unsupported

Table 2.2. Unsupported

Package	Class
org.jgroups.util	HashedTimingWheel
org.jgroups.blocks	ReplicatedHashMap
org.jgroups.blocks	ReplCache
org.jgroups.blocks	ReplicatedTree
org.jgroups.blocks	PartitionedHashMap
org.jgroups.blocks	Cache
org.jgroups.protocols	FD_SIMPLE

Package	Class
org.jgroups.protocols	DELAY_JOIN_REQ
org.jgroups.protocols	SIZE
org.jgroups.protocols	DISCARD
org.jgroups.protocols	EXAMPLE
org.jgroups.protocols	HDRS
org.jgroups.protocols	HTOTAL
org.jgroups.protocols	FD_PING
org.jgroups.protocols	TCP_NIO
org.jgroups.protocols	DISCARD_PAYLOAD
org.jgroups.protocols	DUPL
org.jgroups.protocols	DELAY
org.jgroups.protocols	TRACE

API

This chapter explains the classes available in JGroups that will be used by applications to build reliable group communication applications. The focus is on creating and using channels.

Information in this document may not be up-to-date, but the nature of the classes in JGroups described here is the same. For the most up-to-date information refer to the Javadoc-generated documentation in the `doc/javadoc` directory.

All of the classes discussed here are in the `org.jgroups` package unless otherwise mentioned.

3.1. Utility classes

The `org.jgroups.util.Util` class contains useful common functionality which cannot be assigned to any other package.

3.1.1. objectToByteBuffer(), objectFromByteBuffer()

The first method takes an object as argument and serializes it into a byte buffer (the object has to be serializable or externalizable). The byte array is then returned. This method is often used to serialize objects into the byte buffer of a message. The second method returns a reconstructed object from a buffer. Both methods throw an exception if the object cannot be serialized or unserialized.

3.1.2. objectToStream(), objectFromStream()

The first method takes an object and writes it to an output stream. The second method takes an input stream and reads an object from it. Both methods throw an exception if the object cannot be serialized or unserialized.

3.2. Interfaces

These interfaces are used with some of the APIs presented below, therefore they are listed first.

3.2.1. MessageListener

The `MessageListener` interface below provides callbacks for message reception and for providing and setting the state:

```
public interface MessageListener {
    void receive(Message msg);
    void getState(OutputStream output) throws Exception;
    void setState(InputStream input) throws Exception;
```

```
}
```

Method `receive()` is be called whenever a message is received. The `getState()` and `setState()` methods are used to fetch and set the group state (e.g. when joining). Refer to *Section 3.8.11, "Getting the group's state"* for a discussion of state transfer.

3.2.2. MembershipListener

The `MembershipListener` interface is similar to the `MessageListener` interface above: every time a new view, a suspicion message, or a block event is received, the corresponding method of the class implementing `MembershipListener` will be called.

```
public interface MembershipListener {
    public void viewAccepted(View new_view);
    public void suspect(Object suspected_mbr);
    public void block();
    public void unblock();
}
```

Oftentimes the only callback that needs to be implemented will be `viewAccepted()` which notifies the receiver that a new member has joined the group or that an existing member has left or crashed. The `suspect()` callback is invoked by JGroups whenever a member if suspected of having crashed, but not yet excluded [1].

The `block()` method is called to notify the member that it will soon be blocked sending messages. This is done by the FLUSH protocol, for example to ensure that nobody is sending messages while a state transfer or view installation is in progress. When block() returns, any thread sending messages will be blocked, until FLUSH unblocks the thread again, e.g. after the state has been transferred successfully.

Therefore, block() can be used to send pending messages or complete some other work. Note that block() should be brief, or else the entire FLUSH protocol is blocked.

The `unblock()` method is called to notify the member that the FLUSH protocol has completed and the member can resume sending messages. If the member did not stop sending messages on block(), FLUSH simply blocked them and will resume, so no action is required from a member. Implementation of the unblock() callback is optional.

[1] It could be that the member is suspected falsely, in which case the next view would still contain the suspected member (there is no `unsuspect()` method

Use of MessageListener and MembershipListener

Note that it is oftentimes simpler to extend ReceiverAdapter (see below) and implement the needed callbacks than to implement all methods of both of these interfaces, as most callbacks are not needed.

3.2.3. Receiver

```
public interface Receiver extends MessageListener, MembershipListener;
```

A Receiver can be used to receive messages and view changes; receive() will be invoked as soon as a message has been received, and viewAccepted() will be called whenever a new view is installed.

3.2.4. ReceiverAdapter

This class implements Receiver with no-op implementations. When implementing a callback, we can simply extend ReceiverAdapter and overwrite receive() in order to not having to implement all callbacks of the interface.

`ReceiverAdapter` looks as follows:

```
public class ReceiverAdapter implements Receiver {
    public void receive(Message msg) {}
    public void getState(OutputStream output) throws Exception {}
    public void setState(InputStream input) throws Exception {}
    public void viewAccepted(View view) {}
    public void suspect(Address mbr) {}
    public void block() {}
    public void unblock() {}
}
```

A ReceiverAdapter is the recommended way to implement callbacks.

Sending messages in callbacks

Note that anything that could block should *not* be done in a callback. This includes sending of messages; if we have FLUSH on the stack, and send a message in a viewAccepted() callback, then the following happens: the FLUSH protocol blocks all (multicast) messages before installing a view, then installs the view, then

unblocks. However, because installation of the view triggers the viewAccepted() callback, sending of messages inside of viewAccepted() will block. This in turn blocks the viewAccepted() thread, so the flush will never return !

If we need to send a message in a callback, the sending should be done on a separate thread, or a timer task should be submitted to the timer.

3.2.5. ChannelListener

```
public interface ChannelListener {
    void channelConnected(Channel channel);
    void channelDisconnected(Channel channel);
    void channelClosed(Channel channel);
}
```

A class implementing `ChannelListener` can use the `Channel.addChannelListener()` method to register with a channel to obtain information about state changes in a channel. Whenever a channel is closed, disconnected or opened, the corresponding callback will be invoked.

3.3. Address

Each member of a group has an address, which uniquely identifies the member. The interface for such an address is Address, which requires concrete implementations to provide methods such as comparison and sorting of addresses. JGroups addresses have to implement the following interface:

```
public interface Address extends Externalizable, Comparable, Cloneable {
    int size();
}
```

For marshalling purposes, size() needs to return the number of bytes an instance of an address implementation takes up in serialized form.

Please never use implementations of Address directly; Address should always be used as an opaque identifier of a cluster node !

Actual implementations of addresses are often generated by the bottommost protocol layer (e.g. UDP or TCP). This allows for all possible sorts of addresses to be used with JGroups.

Since an address uniquely identifies a channel, and therefore a group member, it can be used to send messages to that group member, e.g. in Messages (see next section).

The default implementation of Address is `org.jgroups.util.UUID`. It uniquely identifies a node, and when disconnecting and reconnecting to a cluster, a node is given a new UUID on reconnection.

UUIDs are never shown directly, but are usually shown as a logical name (see *Section 3.8.2, "Giving the channel a logical name"*). This is a name given to a node either via the user or via JGroups, and its sole purpose is to make logging output a bit more readable.

UUIDs maps to IpAddresses, which are IP addresses and ports. These are eventually used by the transport protocol to send a message.

3.4. Message

Data is sent between members in the form of messages (`org.jgroups.Message`). A message can be sent by a member to a *single member*, or to *all members* of the group of which the channel is an endpoint. The structure of a message is shown in *Figure 3.1, "Structure of a message"*.

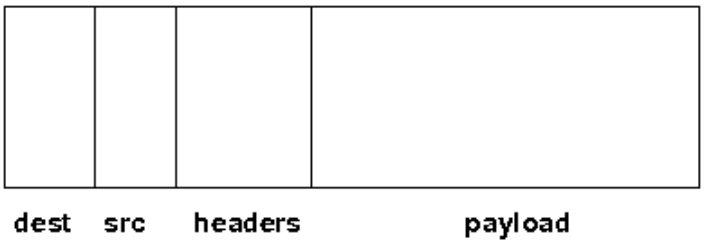

Figure 3.1. Structure of a message

A message has 5 fields:

Destination address
: The address of the receiver. If `null`, the message will be sent to all current group members. Message.getDest() returns the destination address of a message.

Source address
: The address of the sender. Can be left `null`, and will be filled in by the transport protocol (e.g. UDP) before the message is put on the network. Message.getSrc() returns the source address, ie. the address of the sender of a message.

Flags
: This is one byte used for flags. The currently recognized flags are OOB, DONT_BUNDLE, NO_FC, NO_RELIABILITY, NO_TOTAL_ORDER, NO_RELAY and RSVP. For OOB, see the discussion on the concurrent stack (*Section 5.4, "The concurrent stack"*). For the use of flags see *Section 5.13, "Tagging messages with flags"*.

Payload
: The actual data (as a byte buffer). The Message class contains convenience methods to set a serializable object and to retrieve it again, using serialization to convert the object to/from

a byte buffer. A message also has an offset and a length, if the buffer is only a subrange of a larger buffer.

Headers
: A list of headers that can be attached to a message. Anything that should not be in the payload can be attached to a message as a header. Methods `putHeader()` , `getHeader()` and `removeHeader()` of Message can be used to manipulate headers.

  Note that headers are only used by protocol implementers; headers should not be added or removed by application code !

A message is similar to an IP packet and consists of the payload (a byte buffer) and the addresses of the sender and receiver (as Addresses). Any message put on the network can be routed to its destination (receiver address), and replies can be returned to the sender's address.

A message usually does not need to fill in the sender's address when sending a message; this is done automatically by the protocol stack before a message is put on the network. However, there may be cases, when the sender of a message wants to give an address different from its own, so that for example, a response should be returned to some other member.

The destination address (receiver) can be an Address, denoting the address of a member, determined e.g. from a message received previously, or it can be `null`, which means that the message will be sent to all members of the group. A typical multicast message, sending string `"Hello"` to all members would look like this:

```
Message msg=new Message(null, "Hello");
channel.send(msg);
```

3.5. Header

A header is a custom bit of information that can be added to each message. JGroups uses headers extensively, for example to add sequence numbers to each message (NAKACK and UNICAST), so that those messages can be delivered in the order in which they were sent.

3.6. Event

Events are means by which JGroups protcols can talk to each other. Contrary to Messages, which travel over the network between group members, events only travel up and down the stack.

Headers and events

Headers and events are only used by protocol implementers; they are not needed by application code !

3.7. View

A view (`org.jgroups.View`) is a list of the current members of a group. It consists of a `ViewId`, which uniquely identifies the view (see below), and a list of members. Views are installed in a channel automatically by the underlying protocol stack whenever a new member joins or an existing one leaves (or crashes). All members of a group see the same sequence of views.

Note that the first member of a view is the *coordinator* (the one who emits new views). Thus, whenever the membership changes, every member can determine the coordinator easily and without having to contact other members, by picking the first member of a view.

The code below shows how to send a (unicast) message to the first member of a view (error checking code omitted):

```
View view=channel.getView();
Address first=view.getMembers().get(0);
Message msg=new Message(first, "Hello world");
channel.send(msg);
```

Whenever an application is notified that a new view has been installed (e.g. by `Receiver.viewAccepted()`, the view is already set in the channel. For example, calling `Channel.getView()` in a `viewAccepted()` callback would return the same view (or possibly the next one in case there has already been a new view !).

3.7.1. ViewId

The ViewId is used to uniquely number views. It consists of the address of the view creator and a sequence number. ViewIds can be compared for equality and put in a hashmaps as they implement equals() and hashCode(). [2]

3.7.2. MergeView

Whenever a group splits into subgroups, e.g. due to a network partition, and later the subgroups merge back together, a MergeView instead of a View will be received by the application. The MergeView is a subclass of View and contains as additional instance variable the list of views that were merged. As an example if the group denoted by view `V1:(p,q,r,s,t)` split into subgroups `V2:(p,q,r)` and `V2:(s,t)`, the merged view might be `V3:(p,q,r,s,t)`. In this case the MergeView would contains a list of 2 views: `V2:(p,q,r)` and `V2:(s,t)`.

[2] Note that the latter 2 methods only take the ID into account.

3.8. JChannel

In order to join a group and send messages, a process has to create a channel. A channel is like a socket. When a client connects to a channel, it gives the the name of the group it would like to join. Thus, a channel is (in its connected state) always associated with a particular group. The protocol stack takes care that channels with the same group name find each other: whenever a client connects to a channel given group name G, then it tries to find existing channels with the same name, and joins them, resulting in a new view being installed (which contains the new member). If no members exist, a new group will be created.

A state transition diagram for the major states a channel can assume are shown in *Figure 3.2, "Channel states"*.

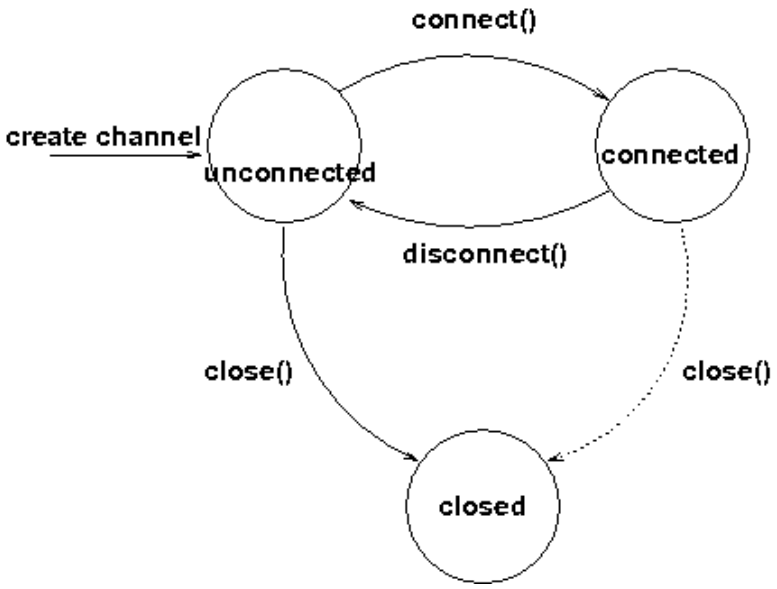

Figure 3.2. Channel states

When a channel is first created, it is in the unconnected state. An attempt to perform certain operations which are only valid in the connected state (e.g. send/receive messages) will result in an exception. After a successful connection by a client, it moves to the connected state. Now the channel will receive messages from other members and may send messages to other members or to the group, and it will get notified when new members join or leave. Getting the local address of a channel is guaranteed to be a valid operation in this state (see below). When the channel is disconnected, it moves back to the unconnected state. Both a connected and unconnected channel may be closed, which makes the channel unusable for further operations. Any attempt to do so will result in an exception. When a channel is closed directly from a connected state, it will first be disconnected, and then closed.

The methods available for creating and manipulating channels are discussed now.

3.8.1. Creating a channel

A channel is created using one of its public constructors (e.g. `new JChannel()`).

The most frequently used constructor of `JChannel` looks as follows:

```
public JChannel(String props) throws Exception;
```

The *props* argument points to an XML file containing the configuration of the protocol stack to be used. This can be a String, but there are also other constructors which take for example a DOM element or a URL (see the javadoc for details).

The code sample below shows how to create a channel based on an XML configuration file:

```
JChannel ch=new JChannel("/home/bela/udp.xml");
```

If the props argument is null, the default properties will be used. An exception will be thrown if the channel cannot be created. Possible causes include protocols that were specified in the property argument, but were not found, or wrong parameters to protocols.

For example, the Draw demo can be launched as follows:

```
java org.javagroups.demos.Draw -props file:/home/bela/udp.xml
```

or

```
java org.javagroups.demos.Draw -props http://www.jgroups.org/udp.xml
```

In the latter case, an application downloads its protocol stack specification from a server, which allows for central administration of application properties.

A sample XML configuration looks like this (edited from udp.xml):

```
<config xmlns="urn:org:jgroups"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="urn:org:jgroups  http://www.jgroups.org/schema/
JGroups-3.0.xsd">
    <UDP
         mcast_port="${jgroups.udp.mcast_port:45588}"
         tos="8"
         ucast_recv_buf_size="20M"
         ucast_send_buf_size="640K"
         mcast_recv_buf_size="25M"
         mcast_send_buf_size="640K"
         loopback="true"
         discard_incompatible_packets="true"
         max_bundle_size="64K"
         max_bundle_timeout="30"
```

```
        ip_ttl="${jgroups.udp.ip_ttl:2}"
        enable_bundling="true"
        enable_diagnostics="true"
        thread_naming_pattern="cl"

        timer_type="new"
        timer.min_threads="4"
        timer.max_threads="10"
        timer.keep_alive_time="3000"
        timer.queue_max_size="500"

        thread_pool.enabled="true"
        thread_pool.min_threads="2"
        thread_pool.max_threads="8"
        thread_pool.keep_alive_time="5000"
        thread_pool.queue_enabled="true"
        thread_pool.queue_max_size="10000"
        thread_pool.rejection_policy="discard"

        oob_thread_pool.enabled="true"
        oob_thread_pool.min_threads="1"
        oob_thread_pool.max_threads="8"
        oob_thread_pool.keep_alive_time="5000"
        oob_thread_pool.queue_enabled="false"
        oob_thread_pool.queue_max_size="100"
        oob_thread_pool.rejection_policy="Run"/>

    <PING timeout="2000"
            num_initial_members="3"/>
    <MERGE2 max_interval="30000"
            min_interval="10000"/>
    <FD_SOCK/>
    <FD_ALL/>
    <VERIFY_SUSPECT timeout="1500"  />
    <BARRIER />
    <pbcast.NAKACK use_stats_for_retransmission="false"
                   exponential_backoff="0"
                   use_mcast_xmit="true"
                   retransmit_timeout="300,600,1200"
                   discard_delivered_msgs="true"/>
    <UNICAST timeout="300,600,1200"/>
    <pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000"
                   max_bytes="4M"/>
    <pbcast.GMS print_local_addr="true" join_timeout="3000"
                view_bundling="true"/>
    <UFC max_credits="2M"
         min_threshold="0.4"/>
    <MFC max_credits="2M"
         min_threshold="0.4"/>
```

```
    <FRAG2 frag_size="60K"  />
    <pbcast.STATE_TRANSFER />
</config>
```

A stack is wrapped by <config> and </config> elements and lists all protocols from bottom (UDP) to top (STATE_TRANSFER). Each element defines one protocol.

Each protocol is implemented as a Java class. When a protocol stack is created based on the above XML configuration, the first element ("UDP") becomes the bottom-most layer, the second one will be placed on the first, etc: the stack is created from the bottom to the top.

Each element has to be the name of a Java class that resides in the `org.jgroups.protocols` package. Note that only the base name has to be given, not the fully specified class name (`UDP` instead of `org.jgroups.protocols.UDP`). If the protocol class is not found, JGroups assumes that the name given is a fully qualified classname and will therefore try to instantiate that class. If this does not work an exception is thrown. This allows for protocol classes to reside in different packages altogether, e.g. a valid protocol name could be `com.sun.eng.protocols.reliable.UCAST`.

Each layer may have zero or more arguments, which are specified as a list of name/value pairs in parentheses directly after the protocol name. In the example above, UDP is configured with some options, one of them being the IP multicast port (mcast_port) which is set to 45588, or to the value of the system property jgroups.udp.mcast_port, if set.

Note that all members in a group have to have the same protocol stack.

3.8.1.1. Programmatic creation

Usually, channels are created by passing the name of an XML configuration file to the JChannel() constructor. On top of this declarative configuration, JGroups provides an API to create a channel programmatically. The way to do this is to first create a JChannel, then an instance of ProtocolStack, then add all desired protocols to the stack and finally calling init() on the stack to set it up. The rest, e.g. calling JChannel.connect() is the same as with the declarative creation.

An example of how to programmatically create a channel is shown below (copied from ProgrammaticChat):

```
public class ProgrammaticChat {

    public static void main(String[] args) throws Exception {
        JChannel ch=new JChannel(false);         // (1)
        ProtocolStack stack=new ProtocolStack(); // (2)
        ch.setProtocolStack(stack);
        stack.addProtocol(new UDP().setValue("bind_addr",
```

```
                                        InetAddress.getByName("192.168.1.5")))
                .addProtocol(new PING())
                .addProtocol(new MERGE2())
                .addProtocol(new FD_SOCK())
                .addProtocol(new FD_ALL().setValue("timeout", 12000)
                                         .setValue("interval", 3000))
                .addProtocol(new VERIFY_SUSPECT())
                .addProtocol(new BARRIER())
                .addProtocol(new NAKACK())
                .addProtocol(new UNICAST2())
                .addProtocol(new STABLE())
                .addProtocol(new GMS())
                .addProtocol(new UFC())
                .addProtocol(new MFC())
                .addProtocol(new FRAG2());       // (3)
        stack.init();                            // (4)

        ch.setReceiver(new ReceiverAdapter() {
            public void viewAccepted(View new_view) {
                System.out.println("view: " + new_view);
            }

            public void receive(Message msg) {
                Address sender=msg.getSrc();
                System.out.println(msg.getObject() + " [" + sender + "]");
            }
        });

        ch.connect("ChatCluster");


        for(;;) {
            String line=Util.readStringFromStdin(": ");
            ch.send(null, line);
        }
    }

}
```

First a JChannel is created (1). The 'false' argument tells the channel not to create a ProtocolStack. This is needed because we will create one ourselves later and set it in the channel (2).

Next, all protocols are added to the stack (3). Note that the order is from bottom (transport protocol) to top. So UDP as transport is added first, then PING and so on, until FRAG2, which is the top protocol. Every protocol can be configured via setters, but there is also a generic setValue(String attr_name, Object value), which can be used to configure protocols as well, as shown in the example.

Once the stack is configured, we call ProtocolStack.init() to link all protocols correctly and to call init() in every protocol instance (4). After this, the channel is ready to be used and all subsequent actions (e.g. connect()) can be executed. When the init() method returns, we have essentially the equivalent of new JChannel(config_file).

3.8.2. Giving the channel a logical name

A channel can be given a logical name which is then used instead of the channel's address in toString(). A logical name might show the function of a channel, e.g. "HostA-HTTP-Cluster", which is more legible than a UUID `3c7e52ea-4087-1859-e0a9-77a0d2f69f29`.

For example, when we have 3 channels, using logical names we might see a view "{A,B,C}", which is nicer than "{`56f3f99e-2fc0-8282-9eb0-866f542ae437`, `ee0be4af-0b45-8ed6-3f6e-92548bfa5cde`, `9241a071-10ce-a931-f675-ff2e3240e1ad`} !"

If no logical name is set, JGroups generates one, using the hostname and a random number, e.g. linux-3442. If this is not desired and the UUIDs should be shown, use system property -Djgroups.print_uuids=true.

The logical name can be set using:

```
public void setName(String logical_name);
```

This must be done *before* connecting a channel. Note that the logical name stays with a channel until the channel is destroyed, whereas a UUID is created on each connection.

When JGroups starts, it prints the logical name and the associated physical address(es):

```
-------------------------------------------------------------------
GMS: address=mac-53465, cluster=DrawGroupDemo, physical
 address=192.168.1.3:49932
-------------------------------------------------------------------
```

The logical name is mac-53465 and the physical address is 192.168.1.3:49932. The UUID is not shown here.

3.8.3. Generating custom addresses

Since 2.12 address generation is pluggable. This means that an application can determine what kind of addresses it uses. The default address type is UUID, and since some protocols use UUID, it is recommended to provide custom classes as *subclasses of UUID*.

This can be used to for example pass additional data around with an address, for example information about the location of the node to which the address is assigned. Note that methods equals(), hashCode() and compare() of the UUID super class should not be changed.

To use custom addresses, an implementation of `org.jgroups.stack.AddressGenerator` has to be written.

For any class CustomAddress, it will need to get registered with the ClassConfigurator in order to marshal it correctly:

```
class CustomAddress extends UUID {
    static {
        ClassConfigurator.add((short)8900, CustomAddress.class);
    }
}
```

Note

Note that the ID should be chosen such that it doesn't collide with any IDs defined in jg-magic-map.xml.

Set the address generator in JChannel: setAddressGenerator(AddressGenerator). This has to be done *before* the channel is connected.

An example of a subclass is `org.jgroups.util.PayloadUUID`, and there are 2 more shipped with JGroups.

3.8.4. Joining a cluster

When a client wants to join a cluster, it *connects* to a channel giving the name of the cluster to be joined:

```
public void connect(String cluster) throws Exception;
```

The cluster name is the name of the cluster to be joined. All channels that call connect() with the same name form a cluster. Messages sent on any channel in the cluster will be received by all members (including the one who sent it [3]).

The connect() method returns as soon as the cluster has been joined successfully. If the channel is in the closed state (see *Figure 3.2, "Channel states"*), an exception will be thrown. If there are no other members, i.e. no other member has connected to a cluster with this name, then a new cluster is created and the member joins it as first member. The first member of a cluster becomes its

[3] Local delivery can be turned off using `setDiscardOwnMessages(true)`.

coordinator. A coordinator is in charge of installing new views whenever the membership changes [4].

3.8.5. Joining a cluster and getting the state in one operation

Clients can also join a cluster and fetch cluster state *in one operation*. The best way to conceptualize the connect and fetch state connect method is to think of it as an invocation of the regular connect() and getState() methods executed in succession. However, there are several advantages of using the connect and fetch state connect method over the regular connect. First of all, the underlying message exchange is heavily optimized, especially if the flush protocol is used. But more importantly, from a client's perspective, the connect() and fetch state operations become one atomic operation.

```
public void connect(String cluster, Address target, long timeout) throws Exception;
```

Just as in a regular connect(), the cluster name represents a cluster to be joined. The target parameter indicates a cluster member to fetch the state from. A null target indicates that the state should be fetched from the cluster coordinator. If the state should be fetched from a particular member other than the coordinator, clients can simply provide the address of that member. The timeout paremeter bounds the entire join and fetch operation. An exception will be thrown if the timeout is exceeded.

3.8.6. Getting the local address and the cluster name

Method `getAddress()` returns the address of the channel. The address may or may not be available when a channel is in the unconnected state.

```
public Address getAddress();
```

Method `getClusterName()` returns the name of the cluster which the member joined.

```
public String getClusterName();
```

Again, the result is undefined if the channel is in the disconnected or closed state.

3.8.7. Getting the current view

The following method can be used to get the current view of a channel:

[4] This is managed internally however, and an application programmer does not need to be concerned about it.

```
public View getView();
```

This method returns the current view of the channel. It is updated every time a new view is installed (viewAccepted() callback).

Calling this method on an unconnected or closed channel is implementation defined. A channel may return null, or it may return the last view it knew of.

3.8.8. Sending messages

Once the channel is connected, messages can be sent using one of the `send()` methods:

```
public void send(Message msg) throws Exception;
public void send(Address dst, Serializable obj) throws Exception;
public void send(Address dst, byte[] buf) throws Exception;
public void send(Address dst, byte[] buf, int off, int len) throws Exception;
```

The first `send()` method has only one argument, which is the message to be sent. The message's destination should either be the address of the receiver (unicast) or null (multicast). When the destination is null, the message will be sent to all members of the cluster (including itself).

The remainaing `send()` methods are helper methods; they take either a byte[] buffer or a serializable, create a Message and call send(Message).

If the channel is not connected, or was closed, an exception will be thrown upon attempting to send a message.

Here's an example of sending a message to all members of a cluster:

```
Map data; // any serializable data
channel.send(null, data);
```

The null value as destination address means that the message will be sent to all members in the cluster. The payload is a hashmap, which will be serialized into the message's buffer and unserialized at the receiver. Alternatively, any other means of generating a byte buffer and setting the message's buffer to it (e.g. using Message.setBuffer()) also works.

Here's an example of sending a unicast message to the first member (coordinator) of a group:

```
```

```
Map data;
Address receiver=channel.getView().getMembers().get(0);
channel.send(receiver, "hello world");
```

The sample code determines the coordinator (first member of the view) and sends it a "hello world" message.

3.8.8.1. Discarding one's own messages

Sometimes, it is desirable not to have to deal with one's own messages, ie. messages sent by oneself. To do this, `JChannel.setDiscardOwnMessages(boolean flag)` can be set to true (false by default). This means that every cluster node will receive a message sent by P, but P itself won't.

Note that this method replaces the old JChannel.setOpt(LOCAL, false) method, which was removed in 3.0.

3.8.8.2. Synchronous messages

While JGroups guarantees that a message will eventually be delivered at all non-faulty members, sometimes this might take a while. For example, if we have a retransmission protocol based on negative acknowledgments, and the last message sent is lost, then the receiver(s) will have to wait until the stability protocol notices that the message has been lost, before it can be retransmitted.

This can be changed by setting the Message.RSVP flag in a message: when this flag is encountered, the message send blocks until all members have acknowledged reception of the message (of course excluding members which crashed or left meanwhile).

This also serves as another purpose: if we send an RSVP-tagged message, then - when the send() returns - we're guaranteed that all messages sent *before* will have been delivered at all members as well. So, for example, if P sends message 1-10, and marks 10 as RSVP, then, upon JChannel.send() returning, P will know that all members received messages 1-10 from P.

Note that since RSVP'ing a message is costly, and might block the sender for a while, it should be used sparingly. For example, when completing a unit of work (ie. member P sending N messages), and P needs to know that all messages were received by everyone, then RSVP could be used.

To use RSVP, 2 things have to be done:

First, the RSVP protocol has to be in the config, somewhere above the reliable transmission protocols such as NAKACK or UNICAST(2), e.g.:

```
<config>
    <UDP/>
    <PING />
```

```
    <FD_ALL/>
    <pbcast.NAKACK use_mcast_xmit="true"
                   discard_delivered_msgs="true"/>
    <UNICAST timeout="300,600,1200"/>
    <RSVP />
    <pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000"
                   max_bytes="4M"/>
    <pbcast.GMS print_local_addr="true" join_timeout="3000"
                view_bundling="true"/>
    ...
</config>
```

Secondly, the message we want to get ack'ed must be tagged with RSVP:

```
Message msg=new Message(null, null, "hello world");
msg.setFlag(Message.RSVP);
ch.send(msg);
```

Here, we send a message to all cluster members (dest = null). (Note that RSVP also works for sending a message to a unicast destination). Method send() will return as soon as it has received acks from all current members. If there are 4 members A, B, C and D, and A has received acks from itself, B and C, but D's ack is missing and D crashes before the timeout kicks in, then this will nevertheless make send() return, as if D had actually sent an ack.

If the `timeout` property if greater than 0, and we don't receive all acks within timeout milliseconds, a TimeoutException will be thrown (if RSVP.throw_exception_on_timeout is true). The application can choose to catch this (runtime) exception and do something with it, e.g. retry.

The configuration of RSVP is described here: *Section 7.6.5, "RSVP"*.

Note

RSVP was added in version 3.1.

3.8.9. Receiving messages

Method `receive()` in ReceiverAdapter (or Receiver) can be overridden to receive messages, views, and state transfer callbacks.

```
public void receive(Message msg);
```

A Receiver can be registered with a channel using JChannel.setReceiver(). All received messages, view changes and state transfer requests will invoke callbacks on the registered Receiver:

```
JChannel ch=new JChannel();
ch.setReceiver(new ReceiverAdapter() {
    public void receive(Message msg) {
        System.out.println("received message " + msg);
    }
    public void viewAccepted(View view) {
        System.out.println("received view " + new_view);
    }
});
ch.connect("MyCluster");
```

3.8.10. Receiving view changes

As shown above, the `viewAccepted()` callback of ReceiverAdapter can be used to get callbacks whenever a cluster membership change occurs. The receiver needs to be set via `JChannel.setReceiver(Receiver)`.

As discussed in *Section 3.2.4, "ReceiverAdapter"*, code in callbacks must avoid anything that takes a lot of time, or blocks; JGroups invokes this callback as part of the view installation, and if this user code blocks, the view installation would block, too.

3.8.11. Getting the group's state

A newly joined member may want to retrieve the state of the cluster before starting work. This is done with `getState()`:

```
public void getState(Address target, long timeout) throws Exception;
```

This method returns the state of one member (usually of the oldest member, the coordinator). The target parameter can usually be null, to ask the current coordinator for the state. If a timeout (ms) elapses before the state is fetched, an exception will be thrown. A timeout of 0 waits until the entire state has been transferred.

Note

The reason for not directly returning the state as a result of `getState()` is that the state has to be returned in the correct position relative to other messages.

> Returning it directly would violate the FIFO properties of a channel, and state transfer would not be correct !

To participate in state transfer, both state provider and state requester have to implement the following callbacks from ReceiverAdapter (Receiver):

```
public void getState(OutputStream output) throws Exception;
public void setState(InputStream input) throws Exception;
```

Method getState() is invoked on the *state provider* (usually the coordinator). It needs to write its state to the output stream given. Note that output doesn't need to be closed when done (or when an exception is thrown); this is done by JGroups.

The setState() method is invoked on the *state requester*; this is the member which called JChannel.getState(). It needs to read its state from the input stream and set its internal state to it. Note that input doesn't need to be closed when done (or when an exception is thrown); this is done by JGroups.

In a cluster consisting of A, B and C, with D joining the cluster and calling Channel.getState(), the following sequence of callbacks happens:

- D calls JChannel.getState(). The state will be retrieved from the oldest member, A
- A's getState() callback is called. A writes its state to the output stream passed as a parameter to getState().
- D's setState() callback is called with an input stream as argument. D reads the state from the input stream and sets its internal state to it, overriding any previous data.
- D: JChannel.getState() returns. Note that this will only happen *after* the state has been transferred successfully, or a timeout elapsed, or either the state provider or requester throws an exception. Such an exception will be re-thrown by getState(). This could happen for instance if the state provider's getState() callback tries to stream a non-serializable class to the output stream.

The following code fragment shows how a group member participates in state transfers:

```
public void getState(OutputStream output) throws Exception {
    synchronized(state) {
        Util.objectToStream(state, new DataOutputStream(output));
    }
}
```

```
public void setState(InputStream input) throws Exception {
    List<String> list;
    list=(List<String>)Util.objectFromStream(new DataInputStream(input));
    synchronized(state) {
        state.clear();
        state.addAll(list);
    }
    System.out.println(list.size() + " messages in chat history):");
    for(String str: list)
        System.out.println(str);
    }
}
```

This code is the Chat example from the JGroups tutorial and the state here is a list of strings.

The getState() implementation synchronized on the state (so no incoming messages can modify it during the state transfer), and uses the JGroups utility method objectToStream().

Performance when writing to an output stream

If a lot of smaller fragments are written to an output stream, it is best to wrap the output stream into a BufferedOutputStream, e.g.

```
Util.objectToStream(state,
                    new BufferedOutputStream(
                        new DataOutputStream(output)));
```

The setState() implementation also uses the Util.objectFromStream() utility method to read the state from the input stream and assign it to its internal list.

3.8.11.1. State transfer protocols

In order to use state transfer, a state transfer protocol has to be included in the configuration. This can either be STATE_TRANSFER, STATE, or STATE_SOCK. More details on the protocols can be found at *Chapter 7, List of Protocols*.

3.8.11.1.1. STATE_TRANSFER

The is the original state transfer protocol, which used to transfer byte[] buffers. It still does that, but is internally converted to call the getState() and setState() callbacks which use input and output streams.

Note that, because byte[] buffers are converted into input and output streams, this protocol should not be used for transfer of large states.

For details see *Section 7.12.1, "pbcast.STATE_TRANSFER"*.

3.8.11.1.2. STATE

This is the STREAMING_STATE_TRANSFER protocol, renamed in 3.0. It sends the entire state across from the provider to the requester in (configurable) chunks, so that memory consumption is minimal.

For details see *Section 7.12.3, "pbcast.STATE"*.

3.8.11.1.3. STATE_SOCK

Same as STREAMING_STATE_TRANSFER, but a TCP connection between provider and requester is used to transfer the state.

For details see *Section 7.12.4, "STATE_SOCK"*.

3.8.12. Disconnecting from a channel

Disconnecting from a channel is done using the following method:

```
public void disconnect();
```

It will have no effect if the channel is already in the disconnected or closed state. If connected, it will leave the cluster. This is done (transparently for a channel user) by sending a leave request to the current coordinator. The latter will subsequently remove the leaving node from the view and install a new view in all remaining members.

After a successful disconnect, the channel will be in the unconnected state, and may subsequently be reconnected.

3.8.13. Closing a channel

To destroy a channel instance (destroy the associated protocol stack, and release all resources), method `close()` is used:

```
public void close();
```

Closing a connected channel disconnects the channel first.

The close() method moves the channel to the closed state, in which no further operations are allowed (most throw an exception when invoked on a closed channel). In this state, a channel

instance is not considered used any longer by an application and -- when the reference to the instance is reset -- the channel essentially only lingers around until it is garbage collected by the Java runtime system.

Building Blocks

Building blocks are layered on top of channels, and can be used instead of channels whenever a higher-level interface is required.

Whereas channels are simple socket-like constructs, building blocks may offer a far more sophisticated interface. In some cases, building blocks offer access to the underlying channel, so that -- if the building block at hand does not offer a certain functionality -- the channel can be accessed directly. Building blocks are located in the `org.jgroups.blocks` package.

4.1. MessageDispatcher

Channels are simple patterns to *asynchronously* send and receive messages. However, a significant number of communication patterns in group communication require *synchronous* communication. For example, a sender would like to send a message to the group and wait for all responses. Or another application would like to send a message to the group and wait only until the majority of the receivers have sent a response, or until a timeout occurred.

`MessageDispatcher` provides blocking (and non-blocking) request sending and response correlation. It offers synchronous (as well as asynchronous) message sending with request-response correlation, e.g. matching one or multiple responses with the original request.

An example of using this class would be to send a request message to all cluster members, and block until all responses have been received, or until a timeout has elapsed.

Contrary to *Section 4.2, "RpcDispatcher"*, MessageDispatcher deals with *sending message requests and correlating message responses*, while RpcDispatcher deals with *invoking method calls and correlating responses*. RpcDispatcher extends MessageDispatcher, and offers an even higher level of abstraction over MessageDispatcher.

RpcDispatcher is essentially a way to invoke remote procedure calls (RCs) across a cluster.

Both MessageDispatcher and RpcDispatcher sit on top of a channel; therefore an instance of `MessageDispatcher` is created with a channel as argument. It can now be used in both *client and server role*: a client sends requests and receives responses and a server receives requests and sends responses. `MessageDispatcher` allows for an application to be both at the same time. To be able to serve requests in the server role, the `RequestHandler.handle()` method has to be implemented:

```
Object handle(Message msg) throws Exception;
```

The `handle()` method is called whenever a request is received. It must return a value (must be serializable, but can be null) or throw an exception. The returned value will be sent to the sender, and exceptions are also propagated to the sender.

Before looking at the methods of MessageDispatcher, let's take a look at RequestOptions first.

4.1.1. RequestOptions

Every message sending in MessageDispatcher or request invocation in RpcDispatcher is governed by an instance of RequestOptions. This is a class which can be passed to a call to define the various options related to the call, e.g. a timeout, whether the call should block or not, the flags (see *Section 5.13, "Tagging messages with flags"*) etc.

The various options are:

- Response mode: this determines whether the call is blocking and - if yes - how long it should block. The modes are:
  - GET_ALL: block until responses from all members (minus the suspected ones) have been received.
  - GET_NONE: wait for none. This makes the call non-blocking
  - GET_FIRST: block until the first response (from anyone) has been received
  - GET_MAJORITY: block until a majority of members have responded
- Timeout: number of milliseconds we're willing to block. If the call hasn't terminated after the timeout elapsed, a TimeoutException will be thrown. A timeout of 0 means to wait forever. The timeout is ignored if the call is non-blocking (mode=GET_NONE)
- Anycasting: if set to true, this means we'll use unicasts to individual members rather than sending multicasts. For example, if we have have TCP as transport, and the cluster is {A,B,C,D,E}, and we send a message through MessageDispatcher where dests={C,D}, and we do *not* want to send the request to everyone, then we'd set anycasting=true. This will send the request to C and D only, as unicasts, which is better if we use a transport such as TCP which cannot use IP multicasting (sending 1 packet to reach all members).
- Response filter: A RspFilter allows for filtering of responses and user-defined termination of a call. For example, if we expect responses from 10 members, but can return after having received 3 non-null responses, a RspFilter could be used. See *Section 4.2.2, "Response filters"* for a discussion on response filters.
- Scope: a short, defining a scope. This allows for concurrent delivery of messages from the same sender. See *Section 5.4.4, "Scopes: concurrent message delivery for messages from the same sender"* for a discussion on scopes.
- Flags: the various flags to be passed to the message, see *Section 5.13, "Tagging messages with flags"* for details.
- Exclusion list: here we can pass a list of members (addresses) that should be excluded. For example, if the view is A,B,C,D,E, and we set the exclusion list to A,C then the caller will wait

for responses from everyone except A and C. Also, every recipient that's in the exclusion list will discard the message.

An example of how to use RequestOptions is:

```
RpcDispatcher disp;
RequestOptions opts=new RequestOptions(Request.GET_ALL)
                    .setFlags(Message.NO_FC).setFlags(Message.DONT_BUNDLE);
Object val=disp.callRemoteMethod(target, method_call, opts);
```

The methods to send requests are:

```
public <T> RspList<T>
       castMessage(final Collection<Address> dests,
                   Message msg,
                   RequestOptions options) throws Exception;
public <T> NotifyingFuture<RspList<T>>
       castMessageWithFuture(final Collection<Address> dests,
                             Message msg,
                             RequestOptions options) throws Exception;
public <T> T sendMessage(Message msg,
                         RequestOptions opts) throws Exception;
public <T> NotifyingFuture<T>
       sendMessageWithFuture(Message msg,
                             RequestOptions options) throws Exception;
```

`castMessage()` sends a message to all members defined in *`dests`*. If *`dests`* is null, the message will be sent to all members of the current cluster. Note that a possible destination set in the message will be overridden. If a message is sent synchronously (defined by options.mode) then *`options.timeout`* defines the maximum amount of time (in milliseconds) to wait for the responses.

`castMessage()` returns a RspList, which contains a map of addresses and Rsps; there's one Rsp per member listed in *`dests`*.

A Rsp instance contains the response value (or null), an exception if the target handle() method threw an exception, whether the target member was suspected, or not, and so on. See the example below for more details.

`castMessageWithFuture()` returns immediately, with a future. The future can be used to fetch the response list (now or later), and it also allows for installation of a callback which will be invoked whenever the future is done. See *Section 4.2.1.1, "Asynchronous calls with futures"* for details on how to use NotifyingFutures.

`sendMessage()` allows an application programmer to send a unicast message to a single cluster member and receive the response. The destination of the message has to be non-null (valid address of a member). The *`mode`* argument is ignored (it is by default set to `ResponseMode.GET_FIRST`) unless it is set to `GET_NONE` in which case the request becomes asynchronous, ie. we will not wait for the response.

`sendMessageWithFuture()` returns immediately with a future, which can be used to fetch the result.

One advantage of using this building block is that failed members are removed from the set of expected responses. For example, when sending a message to 10 members and waiting for all responses, and 2 members crash before being able to send a response, the call will return with 8 valid responses and 2 marked as failed. The return value of `castMessage()` is a `RspList` which contains all responses (not all methods shown):

```
public class RspList<T> implements Map<Address,Rsp> {
    public boolean isReceived(Address sender);
    public int     numSuspectedMembers();
    public List<T> getResults();
    public List<Address> getSuspectedMembers();
    public boolean isSuspected(Address sender);
    public Object  get(Address sender);
    public int     size();
}
```

`isReceived()` checks whether a response from *`sender`* has already been received. Note that this is only true as long as no response has yet been received, and the member has not been marked as failed. `numSuspectedMembers()` returns the number of members that failed (e.g. crashed) during the wait for responses. `getResults()` returns a list of return values. `get()` returns the return value for a specific member.

4.1.2. Requests and target destinations

When a non-null list of addresses is passed (as the destination list) to MessageDispatcher.castMessage() or RpcDispatcher.callRemoteMethods(), then this does *not* mean that only the members included in the list will receive the message, but rather it means that we'll only wait for responses from those members, if the call is blocking.

If we want to restrict the reception of a message to the destination members, there are a few ways to do this:

- If we only have a few destinations to send the message to, use several unicasts.
- Use anycasting. E.g. if we have a membership of {A,B,C,D,E,F}, but only want A and C to receive the message, then set the destination list to A and C and enable anycasting in the

RequestOptions passed to the call (see above). This means that the transport will send 2 unicasts.

- Use exclusion lists. If we have a membership of {A,B,C,D,E,F}, and want to send a message to almost all members, but exclude D and E, then we can define an exclusion list: this is done by settting the destination list to null (= send to all members), or to {A,B,C,D,E,F} and set the exclusion list in the RequestOptions passed to the call to D and E.

4.1.3. Example

This section shows an example of how to use a `MessageDispatcher`.

```
public class MessageDispatcherTest implements RequestHandler {
    Channel            channel;
    MessageDispatcher  disp;
    RspList            rsp_list;
    String             props; // to be set by application programmer

    public void start() throws Exception {
        channel=new JChannel(props);
        disp=new MessageDispatcher(channel, null, null, this);
        channel.connect("MessageDispatcherTestGroup");

        for(int i=0; i < 10; i++) {
            Util.sleep(100);
            System.out.println("Casting message #" + i);
            rsp_list=disp.castMessage(null,
                new Message(null, null, new String("Number #" + i)),
                ResponseMode.GET_ALL, 0);
            System.out.println("Responses:\n" +rsp_list);
        }
        channel.close();
        disp.stop();
    }

    public Object handle(Message msg) throws Exception {
        System.out.println("handle(): " + msg);
        return "Success !";
    }

    public static void main(String[] args) {
        try {
            new MessageDispatcherTest().start();
        }
        catch(Exception e) {
            System.err.println(e);
        }
```

```
    }
}
```

The example starts with the creation of a channel. Next, an instance of `MessageDispatcher` is created on top of the channel. Then the channel is connected. The `MessageDispatcher` will from now on send requests, receive matching responses (client role) and receive requests and send responses (server role).

We then send 10 messages to the group and wait for all responses. The *`timeout`* argument is 0, which causes the call to block until all responses have been received.

The `handle()` method simply prints out a message and returns a string. This will be sent back to the caller as a response value (in Rsp.value). Has the call thrown an exception, Rsp.exception would be set instead.

Finally both the `MessageDispatcher` and channel are closed.

4.2. RpcDispatcher

`RpcDispatcher` is derived from `MessageDispatcher`. It allows a programmer to invoke remote methods in all (or single) cluster members and optionally wait for the return value(s). An application will typically create a channel first, and then create an `RpcDispatcher` on top of it. RpcDispatcher can be used to invoke remote methods (client role) and at the same time be called by other members (server role).

Compared to`MessageDispatcher`, no `handle()` method needs to be implemented. Instead the methods to be called can be placed directly in the class using regular method definitions (see example below). The methods will get invoked using reflection.

To invoke remote method calls (unicast and multicast) the following methods are used:

```
public <T> RspList<T>
       callRemoteMethods(Collection<Address> dests,
                         String method_name,
                         Object[] args,
                         Class[] types,
                         RequestOptions options) throws Exception;
public <T> RspList<T>
       callRemoteMethods(Collection<Address> dests,
                         MethodCall method_call,
                         RequestOptions options) throws Exception;
public <T> NotifyingFuture<RspList<T>>
       callRemoteMethodsWithFuture(Collection<Address> dests,
                                   MethodCall method_call,
                                   RequestOptions options) throws Exception;
```

```
public <T> T callRemoteMethod(Address dest,
                              String method_name,
                              Object[] args,
                              Class[] types,
                              RequestOptions options) throws Exception;
public <T> T callRemoteMethod(Address dest,
                              MethodCall call,
                              RequestOptions options) throws Exception;
public <T> NotifyingFuture<T>
       callRemoteMethodWithFuture(Address dest,
                                  MethodCall call,
                                  RequestOptions options) throws Exception;
```

The family of `callRemoteMethods()` methods is invoked with a list of receiver addresses. If null, the method will be invoked in all cluster members (including the sender). Each call takes the target members to invoke it on (null mean invoke on all cluster members), a method and a RequestOption.

The method can be given as (1) the method name, (2) the arguments and (3) the argument types, or a `MethodCall` (containing a java.lang.reflect.Method and argument) can be given instead.

As with `MessageDispatcher`, a RspList or a future to a RspList is returned.

The family of `callRemoteMethod()` methods takes almost the same parameters, except that there is only one destination address instead of a list. If the *`dest`* argument is null, the call will fail.

The `callRemoteMethod()` calls return the actual result (or type T), or throw an exception if the method threw an exception on the target member.

Java's Reflection API is used to find the correct method in the target member according to the method name and number and types of supplied arguments. There is a runtime exception if a method cannot be resolved.

Note that we could also use method IDs and the `MethodLookup` interface to resolve methods, which is faster and has every RPC carry less data across the wire. To see how this is done, have a look at some of the MethodLookup implementations, e.g. in RpcDispatcherSpeedTest.

4.2.1. Example

The code below shows an example of using RpcDispatcher:

```
public class RpcDispatcherTest {
    JChannel           channel;
    RpcDispatcher      disp;
    RspList            rsp_list;
```

```
    String             props; // set by application

    public static int print(int number) throws Exception {
        return number * 2;
    }

    public void start() throws Exception {
       MethodCall call=new MethodCall(getClass().getMethod("print", int.class));
        RequestOptions opts=new RequestOptions(ResponseMode.GET_ALL, 5000);
        channel=new JChannel(props);
        disp=new RpcDispatcher(channel, this);
        channel.connect("RpcDispatcherTestGroup");

        for(int i=0; i < 10; i++) {
            Util.sleep(100);
            rsp_list=disp.callRemoteMethods(null,
                                            "print",
                                            new Object[]{i},
                                            new Class[]{int.class},
                                            opts);
            // Alternative: use a (prefabricated) MethodCall:
            // call.setArgs(i);
            // rsp_list=disp.callRemoteMethods(null, call, opts);
            System.out.println("Responses: " + rsp_list);
        }
        channel.close();
        disp.stop();
    }

    public static void main(String[] args) throws Exception {
        new RpcDispatcherTest().start();
    }
}
```

Class `RpcDispatcher` defines method `print()` which will be called subsequently. The entry point `start()` creates a channel and an `RpcDispatcher` which is layered on top. Method `callRemoteMethods()` then invokes the remote `print()` in all cluster members (also in the caller). When all responses have been received, the call returns and the responses are printed.

As can be seen, the `RpcDispatcher` building block reduces the amount of code that needs to be written to implement RPC-based group communication applications by providing a higher abstraction level between the application and the primitive channels.

4.2.1.1. Asynchronous calls with futures

When invoking a synchronous call, the calling thread is blocked until the response (or responses) has been received.

A *Future* allows a caller to return immediately and grab the result(s) later. In 2.9, two new methods, which return futures, have been added to RpcDispatcher:

```
public NotifyingFuture<RspList>
       callRemoteMethodsWithFuture(Collection<Address> dests,
                                   MethodCall method_call,
                                   RequestOptions options) throws Exception;
public <T> NotifyingFuture<T>
       callRemoteMethodWithFuture(Address dest,
                                  MethodCall call,
                                  RequestOptions options) throws Exception;
```

A NotifyingFuture extends java.util.concurrent.Future, with its regular methods such as isDone(), get() and cancel(). NotifyingFuture adds setListener<FutureListener> to get notified when the result is available. This is shown in the following code:

```
NotifyingFuture<RspList> future=dispatcher.callRemoteMethodsWithFuture(...);
future.setListener(new FutureListener() {
    void futureDone(Future<T> future) {
        System.out.println("result is " + future.get());
    }
});
```

4.2.2. Response filters

Response filters allow application code to hook into the reception of responses from cluster members and can let the request-response execution and correlation code know (1) wether a response is acceptable and (2) whether more responses are needed, or whether the call (if blocking) can return. The `RspFilter` interface looks as follows:

```
public interface RspFilter {
    boolean isAcceptable(Object response, Address sender);
    boolean needMoreResponses();
}
```

`isAcceptable()` is given a response value and the address of the member which sent the response, and needs to decide whether the response is valid (should return true) or not (should return false).

`needMoreResponses()` determine whether a call returns or not.

The sample code below shows how to use a RspFilter:

```
public void testResponseFilter() throws Exception {
    final long timeout = 10 * 1000 ;

    RequestOptions opts;
    opts=new RequestOptions(ResponseMode.GET_ALL,
                            timeout, false,
                            new RspFilter() {
                                int num=0;
                                public boolean isAcceptable(Object response,
                                                            Address sender) {
                             boolean retval=((Integer)response).intValue() > 1;
                                    if(retval)
                                        num++;
                                    return retval;
                                }
                                public boolean needMoreResponses() {
                                    return num < 2;
                                }
                            });

    RspList rsps=disp1.callRemoteMethods(null, "foo", null, null, opts);
    System.out.println("responses are:\n" + rsps);
    assert rsps.size() == 3;
    assert rsps.numReceived() == 2;
}
```

Here, we invoke a cluster wide RPC (dests=null), which blocks (mode=GET_ALL) for 10 seconds max (timeout=10000), but also passes an instance of RspFilter to the call (in options).

The filter accepts all responses whose value is greater than 2, and returns as soon as it has received 2 responses which satisfy the above condition.

Be careful with RspFilters

If we have a RspFilter which doesn't terminate the call even if responses from all members have been received, we might block forever (if no timeout was given) ! For example, if we have 10 members, and every member returns 1 or 2 as return

value of foo() in the above code, then `isAcceptable()` would always return false, therefore never incrementing 'num', and `needMoreResponses()` would always return true; this would never terminate the call if it wasn't for the timeout of 10 seconds !

This will be fixed in 3.1; a blocking call will always return if we've received as many responses as we have members in 'dests', regardless of what the RspFilter says.

4.3. ReplicatedHashMap

This class was written as a demo of how state can be shared between nodes of a cluster. It has never been heavily tested and is therefore not meant to be used in production.

A `ReplicatedHashMap` uses a concurrent hashmap internally and allows to create several instances of hashmaps in different processes. All of these instances have exactly the same state at all times. When creating such an instance, a cluster name determines which cluster of replicated hashmaps will be joined. The new instance will then query the state from existing members and update itself before starting to service requests. If there are no existing members, it will simply start with an empty state.

Modifications such as `put()`, `clear()` or `remove()` will be propagated in orderly fashion to all replicas. Read-only requests such as `get()` will only be invoked on the local hashmap.

Since both keys and values of a hashtable will be sent across the network, they have to be serializable. Putting a non-serializable value in the map will result in an exception at marshalling time.

A `ReplicatedHashMap` allows to register for notifications, e.g. when data is added removed. All listeners will get notified when such an event occurs. Notification is always local; for example in the case of removing an element, first the element is removed in all replicas, which then notify their listener(s) of the removal (after the fact).

`ReplicatedHashMap` allow members in a group to share common state across process and machine boundaries.

4.4. ReplCache

ReplCache is a distributed cache which - contrary to ReplicatedHashMap - doesn't replicate its values to all cluster members, but just to selected backups.

A `put(K,V,R)` method has a *replication count R* which determines on how many cluster members key K and value V should be stored. When we have 10 cluster members, and R=3, then K and V will be stored on 3 members. If one of those members goes down, or leaves the cluster, then a different member will be told to store K and V. ReplCache tries to always have R cluster members store K and V.

A replication count of -1 means that a given key and value should be stored on *all* cluster members.

The mapping between a key K and the cluster member(s) on which K will be stored is always deterministic, and is computed using a *consistent hash function*.

Note that this class was written as a demo of how state can be shared between nodes of a cluster. It has never been heavily tested and is therefore not meant to be used in production.

4.5. Cluster wide locking

In 2.12, a new distributed locking service was added, replacing DistributedLockManager. The new service is implemented as a protocol and is used via org.jgroups.blocks.locking.LockService.

LockService talks to the locking protocol via events. The main abstraction of a distributed lock is an implementation of java.util.concurrent.locks.Lock. All lock methods are supported, however, conditions are not fully supported, and still need some more testing (as of July 2011).

Below is an example of how LockService is typically used:

```
// locking.xml needs to have a locking protocol
JChannel ch=new JChannel("/home/bela/locking.xml");
LockService lock_service=new LockService(ch);
ch.connect("lock-cluster");
Lock lock=lock_service.getLock("mylock");
lock.lock();
try {
    // do something with the locked resource
}
finally {
    lock.unlock();
}
```

In the example, we create a channel, then a LockService, then connect the channel. If the channel's configuration doesn't include a locking protocol, an exception will be thrown. Then we grab a lock named "mylock", which we lock and subsequently unlock. If another member P had already acquired "mylock", we'd block until P released the lock, or P left the cluster or crashed.

Note that the owner of a lock is always a given thread in a cluster, so the owner is the JGroups address and the thread ID. This means that different threads inside the same JVM trying to access the same named lock will compete for it. If thread-22 grabs the lock first, then thread-5 will block until thread-23 releases the lock.

JGroups includes a demo (org.jgroups.demos.LockServiceDemo), which can be used to interactively experiment with distributed locks. LockServiceDemo -h dumps all command line options.

Currently (Jan 2011), there are 2 protocols which provide locking: *Section 7.14.10.2, "PEER_LOCK"* and *Section 7.14.10.1, "CENTRAL_LOCK"*. The locking protocol has to be placed at or towards the top of the stack (close to the channel).

4.5.1. Locking and merges

The following scenario is susceptible to network partitioning and subsequent merging: we have a cluster view of {A,B,C,D} and then the cluster splits into {A,B} and {C,D}. Assume that B and D now acquire a lock "mylock". This is what happens (with the locking protocol being CENTRAL_LOCK):

- There are 2 coordinators: A for {A,B} and C for {C,D}
- B successfully acquires "mylock" from A
- D successfully acquires "mylock" from C
- The partitions merge back into {A,B,C,D}. Now, only A is the coordinator, but C ceases to be a coordinator
- Problem: D still holds a lock which should actually be invalid !

There is no easy way (via the Lock API) to 'remove' the lock from D. We could for example simply release D's lock on "mylock", but then there's no way telling D that the lock it holds is actually stale !

Therefore the recommended solution here is for nodes to listen to MergeView changes if they expect merging to occur, and re-acquire all of their locks after a merge, e.g.:

```
Lock l1, l2, l3;
LockService lock_service;
...
public void viewAccepted(View view) {
    if(view instanceof MergeView) {
        new Thread() {
            public void run() {
                lock_service.unlockAll();
                // stop all access to resources protected by l1, l2 or l3
                // every thread needs to re-acquire the locks it holds
            }
        }.start
    }
}
```

4.6. Cluster wide task execution

In 2.12, a distributed execution service was added. The new service is implemented as a protocol and is used via org.jgroups.blocks.executor.ExecutionService.

`ExecutionService` extends java.util.concurrent.ExecutorService and distributes tasks submitted to it across the cluster, trying to distribute the tasks to the cluster members as evenly as possible. When a cluster member leaves or dies, the tasks is was processing are re-distributed to other members in the cluster.

ExecutionService talks to the executing protocol via events. The main abstraction is an implementation of java.util.concurrent.ExecutorService. All methods are supported. The restrictions are however that the Callable or Runnable must be Serializable, Externalizable or Streamable. Also the result produced from the future needs to be Serializable, Externalizable or Streamable. If the Callable or Runnable are not, then an IllegalArgumentException is immediately thrown. If a result is not, then a NotSerializableException with the name of the class will be returned to the Future as an exception cause.

Below is an example of how ExecutionService is typically used:

```
// executing.xml needs to have a locking protocol
JChannel ch=new JChannel("/home/bela/executing.xml");
ExecutionService exec_service =new ExecutionService(ch);
ch.connect("exec-cluster");
Future<Value> future = exec_service.submit(new MyCallable());
try {
    Value value = future.get();
    // Do something with value
}
catch (InterruptedException e) {
    e.printStackTrace();
}
catch (ExecutionException e) {
    e.getCause().printStackTrace();
}
```

In the example, we create a channel, then an ExecutionService, then connect the channel. Then we submit our callable giving us a Future. Then we wait for the future to finish returning our value and do something with it. If any exception occurs we print the stack trace of that exception.

The `ExecutionService` follows the Producer-Consumer Pattern very closely. The `ExecutionService` is used as the Producer for this Pattern. Therefore the service only passes tasks off to be handled and doesn't do anything with the actual invocation of those tasks. There is a separate class that can was written specifically as a consumer, which can be ran on any node of the cluster. This class is `ExecutionRunner` and implements java.lang.Runnable. A user is required to run one or more instances of a `ExecutionRunner` on a node of the cluster. By having a thread run one of these runners, that thread has no volunteered to be able to run any task that is submitted to the cluster via an `ExecutionService`. This allows for any node in the cluster to participate or not participate in the running of these tasks and also any node can optionally run

more than 1 `ExecutionRunner` if this node has additional capacity to do so. A runner will run indefinately until the thread that is currently running it is interrupted. If a task is running when the runner is interrupted the task will be interrupted.

Below is an example of how simple it is to have a single node start and allow for 10 distributed tasks to be executed simultaneously on it:

```
int runnerCount = 10;
// locking.xml needs to have a locking protocol
JChannel ch=new JChannel("/home/bela/executing.xml");
ch.connect("exec-cluster");

ExecutionRunner runner = new ExecutionRunner(ch);

ExecutorService service = Executors.newFixedThreadPool(runnerCount);
for (int i = 0; i < runnerCount; ++i) {
   // If you want to stop the runner hold onto the future
   // and cancel with interrupt.
   service.submit(runner);
}
```

In the example, we create a channel, then connect the channel, then an ExecutionRunner. Then we create a java.util.concurrent.ExecutorService that is used to start 10 threads that each thread runs the ExecutionRunner. This allows for this node to have 10 threads actively accept and work on requests submitted via any ExecutionService in the cluster.

Since an ExecutionService does not allow for non serializable class instances to be sent across as tasks there are 2 utility classes provided to get around this problem. For users that are used to using a CompletionService with an Executor there is an equivalent ExecutionCompletionService provided that allows for a user to have the same functionality. It would have been preferred to allow for the same ExecutorCompletionService to be used, but due to it's implementation using a non serializable object the ExecutionCompletionService was implemented to be used instead in conjunction with an ExecutorService. Also utility class was designed to help users to submit tasks which use a non serializable class. The Executions class contains a method serializableCallable which allows for a user to pass a constructor of a class that implements Callable and it's arguments to then return to a user a Callable that will upon running will automatically create and object from the constructor passing the provided arguments to it and then will call the call method on the object and return it's result as a normal callable. All the arguments provided must still be serializable and the return object as detailed previously.

JGroups includes a demo (org.jgroups.demos.ExecutionServiceDemo), which can be used to interactively experiment with a distributed sort algorithm and performance. This is for demonstration purposes and performance should not be assumed to be better than local. ExecutionServiceDemo -h dumps all command line options.

Currently (July 2011), there is 1 protocol which provide executions: *Section 7.14.11, "CENTRAL_EXECUTOR"*. The executing protocol has to be placed at or towards the top of the stack (close to the channel).

4.7. Cluster wide atomic counters

Cluster wide counters provide named counters (similar to AtomicLong) which can be changed atomically. 2 nodes incrementing the same counter with initial value 10 will see 11 and 12 as results, respectively.

To create a named counter, the following steps have to be taken:

1. Add protocol COUNTER to the top of the stack configuration
2. Create an instance of CounterService
3. Create a new or get an existing named counter
4. Use the counter to increment, decrement, get, set, compare-and-set etc the counter

In the first step, we add COUNTER to the top of the protocol stack configuration:

```
<config>
    ...
    <MFC max_credits="2M"
         min_threshold="0.4"/>
    <FRAG2 frag_size="60K"  />
    <COUNTER bypass_bundling="true" timeout="5000"/>
</config>
```

Configuration of the COUNTER protocol is described in *Section 7.14.12, "COUNTER"*.

Next, we create a CounterService, which is used to create and delete named counters:

```
ch=new JChannel(props);
CounterService counter_service=new CounterService(ch);
ch.connect("counter-cluster");
Counter counter=counter_service.getOrCreateCounter("mycounter", 1);
```

In the sample code above, we create a channel first, then create the CounterService referencing the channel. Then we connect the channel and finally create a new named counter "mycounter",

with an initial value of 1. If the counter already exists, the existing counter will be returned and the initial value will be ignored.

CounterService doesn't consume any messages from the channel over which it is created; instead it grabs a reference to the COUNTER protocols and invokes methods on it directly. This has the advantage that CounterService is non-intrusive: many instances can be created over the same channel. CounterService even co-exists with other services which use the same mechanism, e.g. LockService or ExecutionService (see above).

The returned counter instance implements interface Counter:

```
package org.jgroups.blocks.atomic;

public interface Counter {

    public String getName();

    /**
     * Gets the current value of the counter
     * @return The current value
     */
    public long get();

    /**
     * Sets the counter to a new value
     * @param new_value The new value
     */
    public void set(long new_value);

    /**
     * Atomically updates the counter using a CAS operation
     *
     * @param expect The expected value of the counter
     * @param update The new value of the counter
     * @return True if the counter could be updated, false otherwise
     */
    public boolean compareAndSet(long expect, long update);

    /**
     * Atomically increments the counter and returns the new value
     * @return The new value
     */
    public long incrementAndGet();

    /**
     * Atomically decrements the counter and returns the new value
     * @return The new value
```

```
     */
    public long decrementAndGet();


    /**
     * Atomically adds the given value to the current value.
     *
     * @param delta the value to add
     * @return the updated value
     */
    public long addAndGet(long delta);
}
```

4.7.1. Design

The design of COUNTER is described in details in *CounterService* [https://github.com/belaban/JGroups/blob/master/doc/design/CounterService.txt].

In a nutshell, in a cluster the current coordinator maintains a hashmap of named counters. Members send requests (increment, decrement etc) to it, and the coordinator atomically applies the requests and sends back responses.

The advantage of this centralized approach is that - regardless of the size of a cluster - every request has a constant execution cost, namely a network round trip.

A crash or leaving of the coordinator is handled as follows. The coordinator maintains a version for every counter value. Whenever the counter value is changed, the version is incremented. For every request that modifies a counter, both the counter value and the version are returned to the requester. The requester caches all counter values and associated versions in its own local cache.

When the coordinator leaves or crashes, the next-in-line member becomes the new coordinator. It then starts a reconciliation phase, and discards all requests until the reconciliation phase has completed. The reconciliation phase solicits all members for their cached values and versions. To reduce traffic, the request also carries all version numbers with it.

Clients return values whose versions are higher than the ones shipped by the new coordinator. The new coordinator waits for responses from all members or timeout milliseconds. Then it updates its own hashmap with values whose versions are higher than its own. Finally, it stops discarding requests and sends a resend message to all clients in order to resend any requests that might be pending.

There's another edge case that also needs to be covered: if a client P updates a counter, and both P and the coordinator crash, then the update is lost. To reduce the chances of this happening, COUNTER can be enabled to replicate all counter changes to one or more backup coordinators. The num_backups property defines the number of such backups. Whenever a counter was changed in the current coordinator, it also updates the backups (asynchronously). 0 disables this.

Advanced Concepts

This chapter discusses some of the more advanced concepts of JGroups with respect to using it and setting it up correctly.

5.1. Using multiple channels

When using a fully virtual synchronous protocol stack, the performance may not be great because of the larger number of protocols present. For certain applications, however, throughput is more important than ordering, e.g. for video/audio streams or airplane tracking. In the latter case, it is important that airplanes are handed over between control domains correctly, but if there are a (small) number of radar tracking messages (which determine the exact location of the plane) missing, it is not a problem. The first type of messages do not occur very often (typically a number of messages per hour), whereas the second type of messages would be sent at a rate of 10-30 messages/second. The same applies for a distributed whiteboard: messages that represent a video or audio stream have to be delivered as quick as possible, whereas messages that represent figures drawn on the whiteboard, or new participants joining the whiteboard have to be delivered according to a certain order.

The requirements for such applications can be solved by using two separate channels: one for control messages such as group membership, floor control etc and the other one for data messages such as video/audio streams (actually one might consider using one channel for audio and one for video). The control channel might use virtual synchrony, which is relatively slow, but enforces ordering and retransmission, and the data channel might use a simple UDP channel, possibly including a fragmentation layer, but no retransmission layer (losing packets is preferred to costly retransmission).

5.2. Sharing a transport between multiple channels in a JVM

A transport protocol (UDP, TCP) has all the resources of a stack: the default thread pool, the OOB thread pool and the timer thread pool. If we run multiple channels in the same JVM, instead of creating 4 separate stacks with a separate transport each, we can create the transport protocol as a *singleton* protocol, shared by all 4 stacks.

If those transports happen to be the same (all 4 channels use UDP, for example), then we can share them and only create 1 instance of UDP. That transport instance is created and started only once; when the first channel is created, and is deleted when the last channel is closed.

If we have 4 channels inside of a JVM (as is the case in an application server such as JBoss), then we have 12 separate thread pools (3 per transport, 4 transports). Sharing the transport reduces this to 3.

Each channel created over a shared transport has to join a different cluster. An exception will be thrown if a channel sharing a transport tries to connect to a cluster to which another channel over the same transport is already connected.

This is needed to multiplex and de-multiplex messages between the shared transport and the different stacks running over it; when we have 3 channels (C1 connected to "cluster-1", C2 connected to "cluster-2" and C3 connected to "cluster-3") sending messages over the same shared transport, the cluster name with which the channel connected is used to multiplex messages over the shared transport: a header with the cluster name ("cluster-1") is added when C1 sends a message.

When a message with a header of "cluster-1" is received by the shared transport, it is used to demultiplex the message and dispatch it to the right channel (C1 in this example) for processing.

How channels can share a single transport is shown in *Figure 5.1, "A shared transport"*.

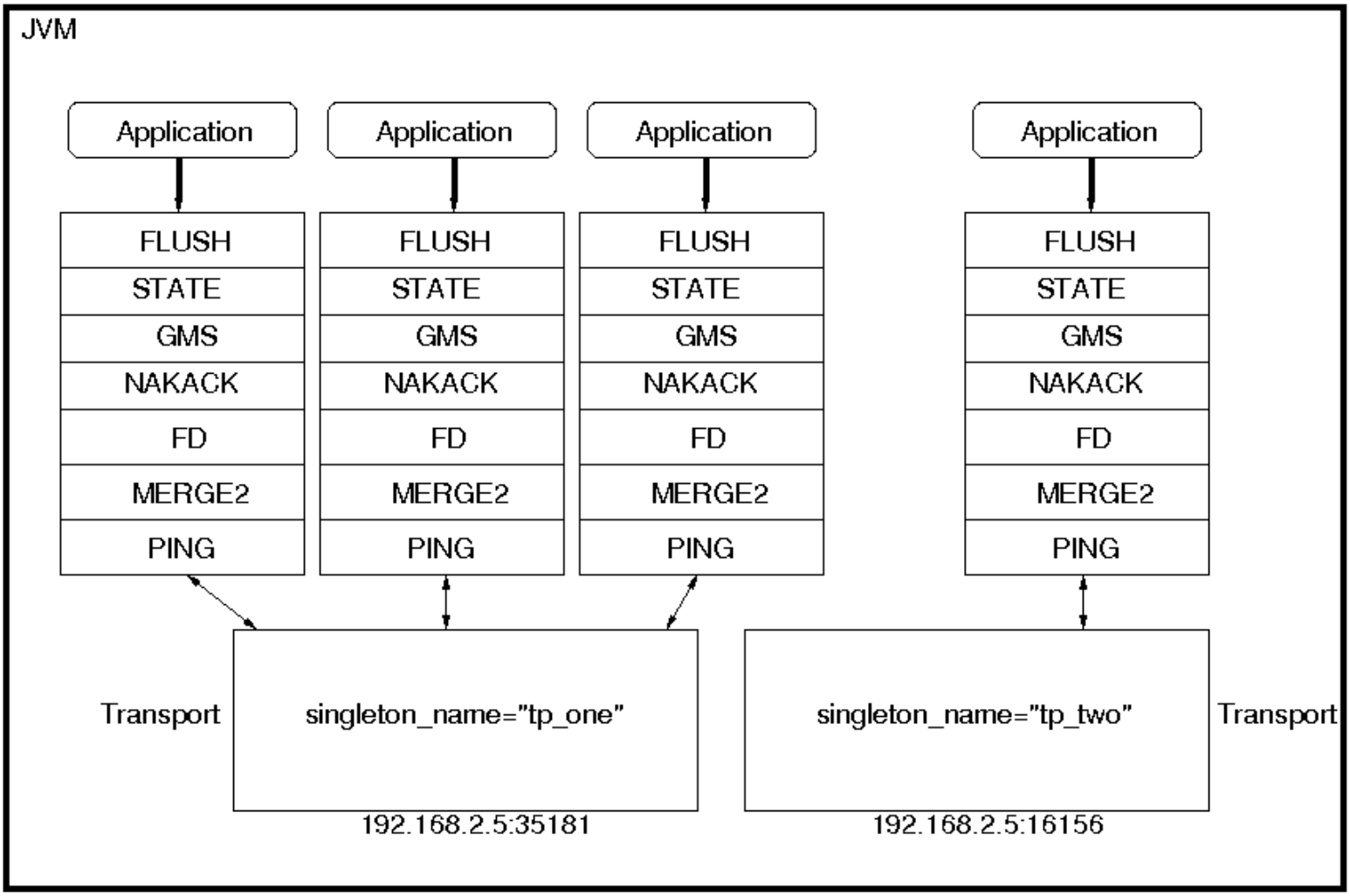

Figure 5.1. A shared transport

Here we see 4 channels which share 2 transports. Note that first 3 channels which share transport "tp_one" have the same protocols on top of the shared transport. This is *not* required; the protocols

above "tp_one" could be different for each of the 3 channels as long as all applications residing on the same shared transport have the same requirements for the transport's configuration.

The "tp_two" transport is used by the application on the right side.

Note that the physical address of a shared channel is the same for all connected channels, so all applications sharing the first transport have physical address 192.168.2.5:35181.

To use shared transports, all we need to do is to add a property "singleton_name" to the transport configuration. All channels with the same singleton name will be shared:

```
<UDP ...
    singleton_name="tp_one" ...
/>
```

All channels using this configuration will now shared transport "tp_one". The channel on the right will have a different configuration, with singleton_name="tp_two".

5.3. Transport protocols

A *transport protocol* refers to the protocol at the bottom of the protocol stack which is responsible for sending messages to and receiving messages from the network. There are a number of transport protocols in JGroups. They are discussed in the following sections.

A typical protocol stack configuration using UDP is:

```
<config xmlns="urn:org:jgroups"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="urn:org:jgroups  http://www.jgroups.org/schema/
JGroups-3.0.xsd">
    <UDP
         mcast_port="${jgroups.udp.mcast_port:45588}"
         tos="8"
         ucast_recv_buf_size="20M"
         ucast_send_buf_size="640K"
         mcast_recv_buf_size="25M"
         mcast_send_buf_size="640K"
         loopback="true"
         discard_incompatible_packets="true"
         max_bundle_size="64K"
         max_bundle_timeout="30"
         ip_ttl="${jgroups.udp.ip_ttl:2}"
         enable_bundling="true"
         enable_diagnostics="true"
```

```
         thread_naming_pattern="cl"

         timer_type="new"
         timer.min_threads="4"
         timer.max_threads="10"
         timer.keep_alive_time="3000"
         timer.queue_max_size="500"

         thread_pool.enabled="true"
         thread_pool.min_threads="2"
         thread_pool.max_threads="8"
         thread_pool.keep_alive_time="5000"
         thread_pool.queue_enabled="true"
         thread_pool.queue_max_size="10000"
         thread_pool.rejection_policy="discard"

         oob_thread_pool.enabled="true"
         oob_thread_pool.min_threads="1"
         oob_thread_pool.max_threads="8"
         oob_thread_pool.keep_alive_time="5000"
         oob_thread_pool.queue_enabled="false"
         oob_thread_pool.queue_max_size="100"
         oob_thread_pool.rejection_policy="Run"/>

    <PING timeout="2000"
            num_initial_members="3"/>
    <MERGE2 max_interval="30000"
            min_interval="10000"/>
    <FD_SOCK/>
    <FD_ALL/>
    <VERIFY_SUSPECT timeout="1500"  />
    <BARRIER />
    <pbcast.NAKACK use_mcast_xmit="true"
                   retransmit_timeout="300,600,1200"
                   discard_delivered_msgs="true"/>
    <UNICAST timeout="300,600,1200"/>
    <pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000"
                   max_bytes="4M"/>
    <pbcast.GMS print_local_addr="true" join_timeout="3000"
                view_bundling="true"/>
    <UFC max_credits="2M"
         min_threshold="0.4"/>
    <MFC max_credits="2M"
         min_threshold="0.4"/>
    <FRAG2 frag_size="60K"  />
    <pbcast.STATE_TRANSFER />
</config>
```

In a nutshell the properties of the protocols are:

UDP
: This is the transport protocol. It uses IP multicasting to send messages to the entire cluster, or individual nodes. Other transports include TCP and TUNNEL.

PING
: This is the discovery protocol. It uses IP multicast (by default) to find initial members. Once found, the current coordinator can be determined and a unicast JOIN request will be sent to it in order to join the cluster.

MERGE2
: Will merge sub-clusters back into one cluster, kicks in after a network partition healed.

FD_SOCK
: Failure detection based on sockets (in a ring form between members). Generates notification if a member fails

FD / FD_ALL
: Failure detection based on heartbeat are-you-alive messages. Generates notification if a member fails

VERIFY_SUSPECT
: Double-checks whether a suspected member is really dead, otherwise the suspicion generated from protocol below is discarded

BARRIER
: Needed to transfer state; this will block messages that modify the shared state until a digest has been taken, then unblocks all threads. *Not needed if no state transfer protocol is present.*

pbcast.NAKACK
: Ensures (a) message reliability and (b) FIFO. Message reliability guarantees that a message will be received. If not, the receiver(s) will request retransmission. FIFO guarantees that all messages from sender P will be received in the order P sent them

UNICAST
: Same as NAKACK for unicast messages: messages from sender P will not be lost (retransmission if necessary) and will be in FIFO order (conceptually the same as TCP in TCP/IP)

pbcast.STABLE
: Deletes messages that have been seen by all members (distributed message garbage collection)

pbcast.GMS
: Membership protocol. Responsible for joining/leaving members and installing new views.

UFC
: Unicast Flow Control. Provides flow control between 2 members.

MFC
: Multicast Flow Control. Provides flow control between a sender and all cluster members.

FRAG2
: Fragments large messages into smaller ones and reassembles them back at the receiver side. For both multicast and unicast messages

STATE_TRANSFER
: Ensures that state is correctly transferred from an existing member (usually the coordinator) to a new member.

5.3.1. Message bundling

Message bundling is beneficial when sending many small messages; it queues them until they have accumulated a certain size, or until a timeout has elapsed. Then, the queued messages are assembled into a larger message, and that message is then sent. At the receiver, the large message is disassembled and the smaller messages are sent up the stack.

When sending many smaller messages, the ratio between payload and message headers might be small; say we send a "hello" string: the payload here is 7 bytes, whereas the addresses and headers (depending on the stack configuration) might be 30 bytes. However, if we bundle (say) 100 messages, then the payload of the large message is 700 bytes, but the header is still 30 bytes. Thus, we're able to send more actual data across the wire with one large message than many smaller ones.

Message bundling is conceptually similar to TCP's Nagling algorithm.

A sample configuration is shown below:

```
<UDP
    enable_bundling="true"
    max_bundle_size="64K"
    max_bundle_timeout="30"
/>
```

Here, bundling is enabled (the default). The max accumulated size is 64'000 bytes and we wait for 30 ms max. If at time T0, we're sending 10 smaller messages with an accumulated size of 2'000 bytes, but then send no more messages, then the timeout will kick in after 30 ms and the messages will get packed into a large message M and M will be sent. If we send 1000 messages of 100 bytes each, then - after exceeding 64'000 bytes (after ca. 64 messages) - we'll send the large message, and this might have taken only 3 ms.

5.3.1.1. Message bundling and performance

While message bundling is good when sending many small messages asynchronously, it can be bad when invoking synchronous RPCs: say we're invoking 10 synchronous (blocking) RPCs across the cluster with an RpcDispatcher (see *Section 4.2, "RpcDispatcher"*), and the payload of the marshalled arguments of one call is less than 64K.

Because the RPC is blocking, we'll wait until the call has returned before invoking the next RPC.

For each RPC, the request takes up to 30 ms, and each response will also take up to 30 ms, for a total of 60 ms *per call*. So the 10 blocking RPCs would take a total of 600 ms !

This is clearly not desirable. However, there's a simple solution: we can use message flags (see *Section 5.13, "Tagging messages with flags"*) to override the default bundling behavior in the transport:

```
RpcDispatcher disp;
RequestOptions opts=new RequestOptions(ResponseMode.GET_ALL, 5000)
                        .setFlags(Message.DONT_BUNDLE);
RspList rsp_list=disp.callRemoteMethods(null,
                                        "print",
                                        new Object[]{i},
                                        new Class[]{int.class},
                                        opts);
```

The `RequestOptions.setFlags(Message.DONT_BUNDLE)` call tags the message with the DONT_BUNDLE flag. When the message is to be sent by the transport, it will be sent immediately, regardless of whether bundling is enabled in the transport.

Using the DONT_BUNDLE flag to invoke `print()` will take a few milliseconds for 10 blocking RPCs versus 600 ms without the flag.

An alternative to setting the DONT_BUNDLE flag is to use futures to invoke 10 blocking RPCs:

```
List<Future<RspList>> futures=new ArrayList<Future<RspList>>();
for(int i=0; i < 10; i++) {
    Future<RspList> future=disp.callRemoteMethodsWithFuture(...);
    futures.add(future);
}

for(Future<RspList> future: futures) {
    RspList rsp_list=future.get();
    // do something with the response
}
```

Here we use `callRemoteMethodsWithFuture()` which (although the call is blocking!) returns immediately, with a future. After invoking the 10 calls, we then grab the results by fetching them from the futures.

Compared to the few milliseconds above, this will take ca 60 ms (30 for the request and 30 for the responses), but this is still better than the 600 ms we get when not using the DONT_BUNDLE flag. Note that, if the accumulated size of the 10 requests exceeds `max_bundle_size`, the large message would be sent immediately, so this might even be faster than 30 ms for the request.

5.3.2. UDP

UDP uses *IP multicast* for sending messages to all members of a cluster, and *UDP datagrams* for unicast messages (sent to a single member). When started, it opens a unicast and multicast socket: the unicast socket is used to send/receive unicast messages, while the multicast socket sends/receives multicast messages. The physical address of the channel will be the address and port number of the *unicast* socket.

5.3.2.1. Using UDP and plain IP multicasting

A protocol stack with UDP as transport protocol is typically used with clusters whose members run on the same host or are distributed across a LAN. Note that before running instances *in different subnets*, an admin has to make sure that IP multicast is enabled across subnets. It is often the case that IP multicast is not enabled across subnets. Refer to section *Section 2.7, "It doesn't work !"* for running a test program that determines whether members can reach each other via IP multicast. If this does not work, the protocol stack cannot use UDP with IP multicast as transport. In this case, the stack has to either use UDP without IP multicasting, or use a different transport such as TCP.

5.3.2.2. Using UDP without IP multicasting

The protocol stack with UDP and PING as the bottom protocols use IP multicasting by default to send messages to all members (UDP) and for discovery of the initial members (PING). However, if multicasting cannot be used, the UDP and PING protocols can be configured to send multiple unicast messages instead of one multicast message [1].

To configure UDP to use multiple unicast messages to send a group message instead of using IP multicasting, the *`ip_mcast`* property has to be set to `false`.

If we disable ip_mcast, we now also have to change the discovery protocol (PING). Because PING requires IP multicasting to be enabled in the transport, we cannot use it. Some of the alternatives are TCPPING (static list of member addresses), TCPGOSSIP (external lookup service), FILE_PING (shared directory), BPING (using broadcasts) or JDBC_PING (using a shared database).

[1] Although not as efficient (and using more bandwidth), it is sometimes the only possibility to reach group members.

See *Section 7.3, "Initial membership discovery"* for details on configuration of different discovery protocols.

5.3.3. TCP

TCP is a replacement for UDP as transport in cases where IP multicast cannot be used. This may be the case when operating over a WAN, where routers might discard IP multicast packets. Usually, UDP is used as transport in LANs, while TCP is used for clusters spanning WANs.

The properties for a typical stack based on TCP might look like this (edited for brevity):

```
<TCP bind_port="7800" />
<TCPPING timeout="3000"
       initial_hosts="${jgroups.tcpping.initial_hosts:HostA[7800],HostB[7801]}"
         port_range="1"
         num_initial_members="3"/>
<VERIFY_SUSPECT timeout="1500"  />
<pbcast.NAKACK use_mcast_xmit="false"
               retransmit_timeout="300,600,1200,2400,4800"
               discard_delivered_msgs="true"/>
<pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000"
               max_bytes="400000"/>
<pbcast.GMS print_local_addr="true" join_timeout="3000"
               view_bundling="true"/>
```

TCP
: The transport protocol, uses TCP (from TCP/IP) to send unicast and multicast messages. In the latter case, it sends multiple unicast messages.

TCPPING
: Discovers the initial membership to determine coordinator. Join request will then be sent to coordinator.

VERIFY_SUSPECT
: Double checks that a suspected member is really dead

pbcast.NAKACK
: Reliable and FIFO message delivery

pbcast.STABLE
: Distributed garbage collection of messages seen by all members

pbcast.GMS
: Membership services. Takes care of joining and removing new/old members, emits view changes

When using TCP, each message to all of the cluster members is sent as multiple unicast messages (one to each member). Due to the fact that IP multicasting cannot be used to discover the initial members, another mechanism has to be used to find the initial membership. There are a number of alternatives (see *Section 7.3, "Initial membership discovery"* for a discussion of all discovery protocols):

- TCPPING: uses a list of well-known group members that it solicits for initial membership
- TCPGOSSIP: this requires a GossipRouter (see below), which is an external process, acting as a lookup service. Cluster members register with under their cluster name, and new members query the GossipRouter for initial cluster membership information.

The next two section illustrate the use of TCP with both TCPPING and TCPGOSSIP.

5.3.3.1. Using TCP and TCPPING

A protocol stack using TCP and TCPPING looks like this (other protocols omitted):

```
<TCP bind_port="7800" /> +
<TCPPING initial_hosts="HostA[7800],HostB[7800]" port_range="2"
         timeout="3000" num_initial_members="3" />
```

The concept behind TCPPING is that some selected cluster members assume the role of well-known hosts from which the initial membership information can be retrieved. In the example, *`HostA`* and *`HostB`* are designated members that will be used by TCPPING to lookup the initial membership. The property *`bind_port`* in `TCP` means that each member should try to assign port 7800 for itself. If this is not possible it will try the next higher port (`7801`) and so on, until it finds an unused port.

`TCPPING` will try to contact both *`HostA`* and *`HostB`*, starting at port `7800` and ending at port `7800 + port_range`, in the above example ports `7800` - `7802`. Assuming that at least one of *`HostA`* or *`HostB`* is up, a response will be received. To be absolutely sure to receive a response, it is recommended to add all the hosts on which members of the cluster will be running to the configuration.

5.3.3.2. Using TCP and TCPGOSSIP

`TCPGOSSIP` uses one or more GossipRouters to (1) register itself and (2) fetch information about already registered cluster members. A configuration looks like this:

```
<TCP />
<TCPGOSSIP initial_hosts="HostA[5555],HostB[5555]" num_initial_members="3" />
```

The `initial_hosts` property is a comma-delimited list of GossipRouters. In the example there are two GossipRouters on HostA and HostB, at port `5555`.

A member always registers with all GossipRouters listed, but fetches information from the first available GossipRouter. If a GossipRouter cannot be accessed, it will be marked as failed and removed from the list. A task is then started, which tries to periodically reconnect to the failed process. On reconnection, the failed GossipRouter is marked as OK, and re-inserted into the list.

The advantage of having multiple GossipRouters is that, as long as at least one is running, new members will always be able to retrieve the initial membership.

Note that the GossipRouter should be started before any of the members.

5.3.4. TUNNEL

Firewalls are usually placed at the connection to the internet. They shield local networks from outside attacks by screening incoming traffic and rejecting connection attempts to host inside the firewalls by outside machines. Most firewall systems allow hosts inside the firewall to connect to hosts outside it (outgoing traffic), however, incoming traffic is most often disabled entirely.

Tunnels are host protocols which encapsulate other protocols by multiplexing them at one end and demultiplexing them at the other end. Any protocol can be tunneled by a tunnel protocol.

The most restrictive setups of firewalls usually disable *all* incoming traffic, and only enable a few selected ports for outgoing traffic. In the solution below, it is assumed that one TCP port is enabled for outgoing connections to the GossipRouter.

JGroups has a mechanism that allows a programmer to tunnel a firewall. The solution involves a GossipRouter, which has to be outside of the firewall, so other members (possibly also behind firewalls) can access it.

The solution works as follows. A channel inside a firewall has to use protocol TUNNEL instead of UDP or TCP as transport. The recommended discovery protocol is PING. Here's a configuration:

```
<TUNNEL gossip_router_hosts="HostA[12001]" />
<PING />
```

`TUNNEL` uses a GossipRouter (outside the firewall) running on HostA at port `12001` for tunneling. Note that it is not recommended to use TCPGOSSIP for discovery if TUNNEL is used (use PING instead). TUNNEL accepts one or multiple GossipRouters tor tunneling; they can be listed as a comma delimited list of host[port] elements specified in property gossip_router_hosts.

`TUNNEL` establishes a TCP connection to the *GossipRouter* process (outside the firewall) that accepts messages from members and passes them on to other members. This connection is

initiated by the host inside the firewall and persists as long as the channel is connected to a group. A GossipRouter will use the *same connection* to send incoming messages to the channel that initiated the connection. This is perfectly legal, as TCP connections are fully duplex. Note that, if GossipRouter tried to establish its own TCP connection to the channel behind the firewall, it would fail. But it is okay to reuse the existing TCP connection, established by the channel.

Note that `TUNNEL` has to be given the hostname and port of the GossipRouter process. This example assumes a GossipRouter is running on HostA at port`12001`. TUNNEL accepts one or multiple router hosts as a comma delimited list of host[port] elements specified in property gossip_router_hosts.

Any time a message has to be sent, TUNNEL forwards the message to GossipRouter, which distributes it to its destination: if the message's destination field is null (send to all group members), then GossipRouter looks up the members that belong to that group and forwards the message to all of them via the TCP connections they established when connecting to GossipRouter. If the destination is a valid member address, then that member's TCP connection is looked up, and the message is forwarded to it [2] .

A GossipRouter is not a single point of failure. In a setup with multiple gossip routers, the routers do not communicate among themselves, and a single point of failure is avoided by having each channel simply connect to multiple available routers. In case one or more routers go down, the cluster members are still able to exchange messages through any of the remaining available router instances, if there are any.

For each send invocation, a channel goes through a list of available connections to routers and attempts to send the message on each connection until it succeeds. If a message can not be sent on any of the connections, an exception is raised. The default policy for connection selection is random. However, we provide an plug-in interface for other policies as well.

The GossipRouter configuration is static and is not updated for the lifetime of the channel. A list of available routers has to be provided in the channel's configuration file.

To tunnel a firewall using JGroups, the following steps have to be taken:

1. Check that a TCP port (e.g. 12001) is enabled in the firewall for outgoing traffic
2. Start the GossipRouter:

```
java org.jgroups.stack.GossipRouter -port 12001
```

3. Configure the TUNNEL protocol layer as instructed above.
4. Create a channel

The general setup is shown in *Figure 5.2, “Tunneling a firewall”* .

[2] To do so, GossipRouter maintains a mapping between cluster names and member addresses, and TCP connections.

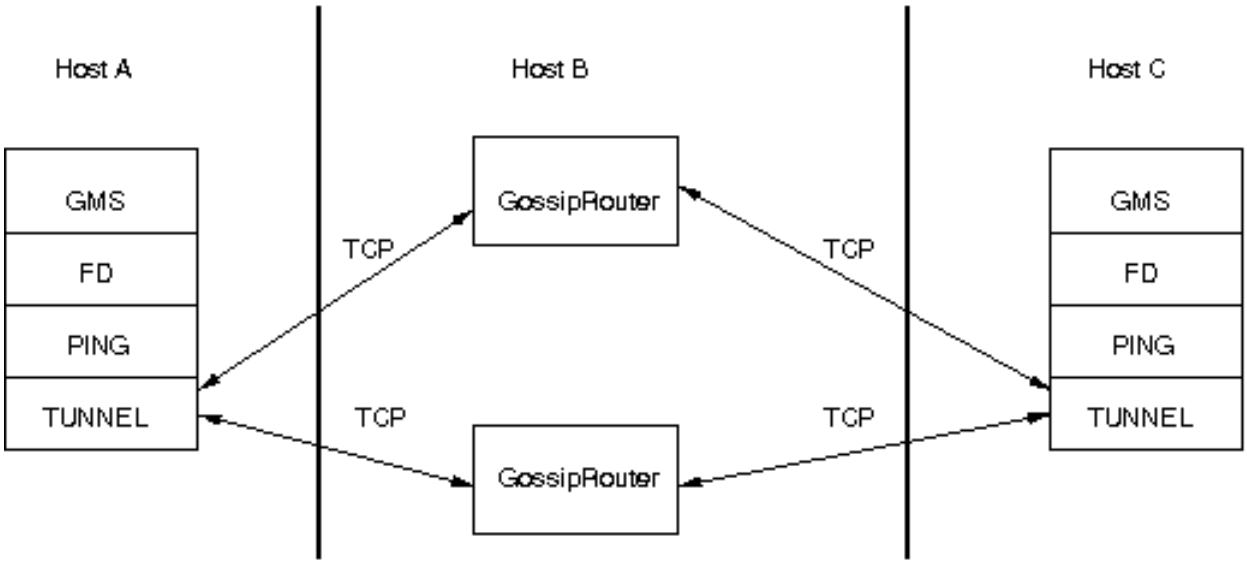

Figure 5.2. Tunneling a firewall

First, the GossipRouter process is created on host B. Note that host B should be outside the firewall, and all channels in the same group should use the same GossipRouter process. When a channel on host A is created, its `TCPGOSSIP` protocol will register its address with the GossipRouter and retrieve the initial membership (assume this is C). Now, a TCP connection with the GossipRouter is established by A; this will persist until A crashes or voluntarily leaves the group. When A multicasts a message to the cluster, GossipRouter looks up all cluster members (in this case, A and C) and forwards the message to all members, using their TCP connections. In the example, A would receive its own copy of the multicast message it sent, and another copy would be sent to C.

This scheme allows for example *Java applets* , which are only allowed to connect back to the host from which they were downloaded, to use JGroups: the HTTP server would be located on host B and the gossip and GossipRouter daemon would also run on that host. An applet downloaded to either A or C would be allowed to make a TCP connection to B. Also, applications behind a firewall would be able to talk to each other, joining a group.

However, there are several drawbacks: first, having to maintain a TCP connection for the duration of the connection might use up resources in the host system (e.g. in the GossipRouter), leading to scalability problems, second, this scheme is inappropriate when only a few channels are located behind firewalls, and the vast majority can indeed use IP multicast to communicate, and finally, it is not always possible to enable outgoing traffic on 2 ports in a firewall, e.g. when a user does not 'own' the firewall.

5.4. The concurrent stack

The concurrent stack (introduced in 2.5) provides a number of improvements over previous releases, which has some deficiencies:

- Large number of threads: each protocol had by default 2 threads, one for the up and one for the down queue. They could be disabled per protocol by setting up_thread or down_thread to false. In the new model, these threads have been removed.
- Sequential delivery of messages: JGroups used to have a single queue for incoming messages, processed by one thread. Therefore, messages from different senders were still processed in FIFO order. In 2.5 these messages can be processed in parallel.
- Out-of-band messages: when an application doesn't care about the ordering properties of a message, the OOB flag can be set and JGroups will deliver this particular message without regard for any ordering.

5.4.1. Overview

The architecture of the concurrent stack is shown in *Figure 5.3, "The concurrent stack"*. The changes were made entirely inside of the transport protocol (TP, with subclasses UDP, TCP and TCP_NIO). Therefore, to configure the concurrent stack, the user has to modify the config for (e.g.) UDP in the XML file.

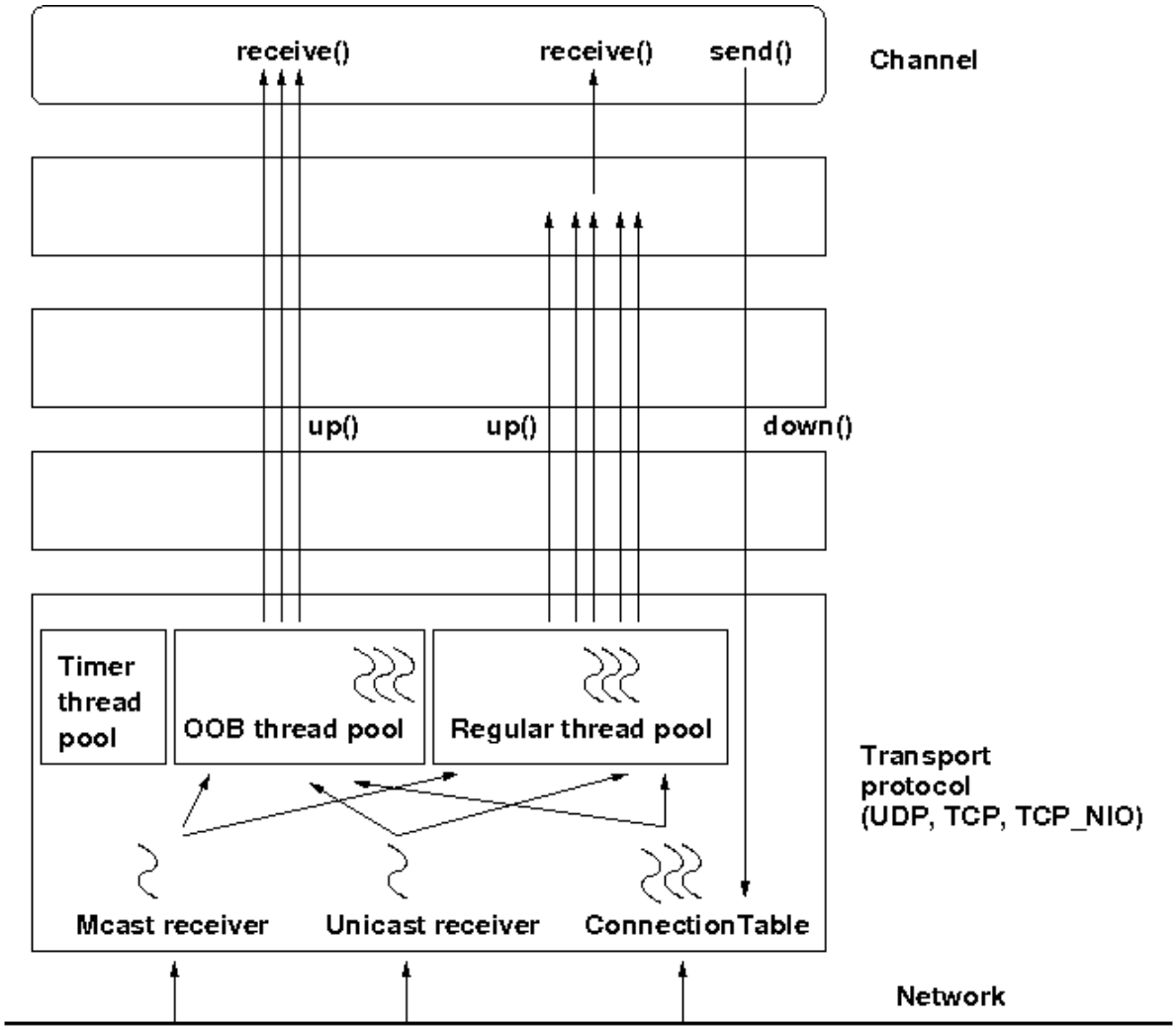

Figure 5.3. The concurrent stack

The concurrent stack consists of 2 thread pools (java.util.concurrent.Executor): the out-of-band (OOB) thread pool and the regular thread pool. Packets are received by multicast or unicast receiver threads (UDP) or a ConnectionTable (TCP, TCP_NIO). Packets marked as OOB (with Message.setFlag(Message.OOB)) are dispatched to the OOB thread pool, and all other packets are dispatched to the regular thread pool.

When a thread pool is disabled, then we use the thread of the caller (e.g. multicast or unicast receiver threads or the ConnectionTable) to send the message up the stack and into the application. Otherwise, the packet will be processed by a thread from the thread pool, which sends the message up the stack. When all current threads are busy, another thread might be created, up to the maximum number of threads defined. Alternatively, the packet might get queued up until a thread becomes available.

The point of using a thread pool is that the receiver threads should only receive the packets and forward them to the thread pools for processing, because unmarshalling and processing is slower than simply receiving the message and can benefit from parallelization.

5.4.1.1. Configuration

Note that this is preliminary and names or properties might change

We are thinking of exposing the thread pools programmatically, meaning that a developer might be able to set both threads pools programmatically, e.g. using something like TP.setOOBThreadPool(Executor executor).

Here's an example of the new configuration:

```
<UDP
     thread_naming_pattern="cl"

     thread_pool.enabled="true"
     thread_pool.min_threads="1"
     thread_pool.max_threads="100"
     thread_pool.keep_alive_time="20000"
     thread_pool.queue_enabled="false"
     thread_pool.queue_max_size="10"
     thread_pool.rejection_policy="Run"

     oob_thread_pool.enabled="true"
     oob_thread_pool.min_threads="1"
     oob_thread_pool.max_threads="4"
     oob_thread_pool.keep_alive_time="30000"
     oob_thread_pool.queue_enabled="true"
     oob_thread_pool.queue_max_size="10"
     oob_thread_pool.rejection_policy="Run"/>
```

The attributes for the 2 thread pools are prefixed with thread_pool and oob_thread_pool respectively.

The attributes are listed below. The roughly correspond to the options of a java.util.concurrent.ThreadPoolExecutor in JDK 5.

Table 5.1. Attributes of thread pools

Name	Description
thread_naming_pattern	Determines how threads are named that are running from thread pools in concurrent stack. Valid values include any combination of "cl" letters, where "c" includes the cluster name and "l" includes local address of the channel. The default is "cl"

Name	Description
enabled	Whether of not to use a thread pool. If set to false, the caller's thread is used.
min_threads	The minimum number of threads to use.
max_threads	The maximum number of threads to use.
keep_alive_time	Number of milliseconds until an idle thread is placed back into the pool
queue_enabled	Whether of not to use a (bounded) queue. If enabled, when all minimum threads are busy, work items are added to the queue. When the queue is full, additional threads are created, up to max_threads. When max_threads have been reached (and the queue is full), the rejection policy is consulted.
max_size	The maximum number of elements in the queue. Ignored if the queue is disabled
rejection_policy	Determines what happens when the thread pool (and queue, if enabled) is full. The default is to run on the caller's thread. "Abort" throws an runtime exception. "Discard" discards the message, "DiscardOldest" discards the oldest entry in the queue. Note that these values might change, for example a "Wait" value might get added in the future.

5.4.2. Elimination of up and down threads

By removing the 2 queues/protocol and the associated 2 threads, we effectively reduce the number of threads needed to handle a message, and thus context switching overhead. We also get clear and unambiguous semantics for Channel.send(): now, all messages are sent down the stack on the caller's thread and the send() call only returns once the message has been put on the network. In addition, an exception will only be propagated back to the caller if the message has not yet been placed in a retransmit buffer. Otherwise, JGroups simply logs the error message but keeps retransmitting the message. Therefore, if the caller gets an exception, the message should be re-sent.

On the receiving side, a message is handled by a thread pool, either the regular or OOB thread pool. Both thread pools can be completely eliminated, so that we can save even more threads and thus further reduce context switching. The point is that the developer is now able to control the threading behavior almost completely.

5.4.3. Concurrent message delivery

Up to version 2.5, all messages received were processed by a single thread, even if the messages were sent by different senders. For instance, if sender A sent messages 1,2 and 3, and B sent message 34 and 45, and if A's messages were all received first, then B's messages 34 and 35 could only be processed after messages 1-3 from A were processed !

Now, we can process messages from different senders in parallel, e.g. messages 1, 2 and 3 from A can be processed by one thread from the thread pool and messages 34 and 35 from B can be processed on a different thread.

As a result, we get a speedup of almost N for a cluster of N if every node is sending messages and we configure the thread pool to have at least N threads. There is actually a unit test (ConcurrentStackTest.java) which demonstrates this.

5.4.4. Scopes: concurrent message delivery for messages from the same sender

In the previous paragraph, we showed how the concurrent stack delivers messages from different senders concurrently. But all (non-OOB) messages from the same sender P are delivered in the order in which P sent them. However, this is not good enough for certain types of applications.

Consider the case of an application which replicates HTTP sessions. If we have sessions X, Y and Z, then updates to these sessions are delivered in the order in which there were performed, e.g. X1, X2, X3, Y1, Z1, Z2, Z3, Y2, Y3, X4. This means that update Y1 has to wait until updates X1-3 have been delivered. If these updates take some time, e.g. spent in lock acquisition or deserialization, then all subsequent messages are delayed by the sum of the times taken by the messages ahead of them in the delivery order.

However, in most cases, updates to different web sessions should be completely unrelated, so they could be delivered concurrently. For instance, a modification to session X should not have any effect on session Y, therefore updates to X, Y and Z can be delivered concurrently.

One solution to this is out-of-band (OOB) messages (see next paragraph). However, OOB messages do not guarantee ordering, so updates X1-3 could be delivered as X1, X3, X2. If this is not wanted, but messages pertaining to a given web session should all be delivered concurrently between sessions, but ordered *within* a given session, then we can resort to *scoped messages*.

Scoped messages apply only to *regular* (non-OOB) messages, and are delivered concurrently between scopes, but ordered within a given scope. For example, if we used the sessions above (e.g. the jsessionid) as scopes, then the delivery could be as follows ('->' means sequential, '||' means concurrent):

```
X1 -> X2 -> X3 -> X4 || Y1 -> Y2 -> Y3 || Z1 -> Z2 -> Z3
```

This means that all updates to X are delivered in parallel to updates to Y and updates to Z. However, within a given scope, updates are delivered in the order in which they were performed, so X1 is delivered before X2, which is deliverd before X3 and so on.

Taking the above example, using scoped messages, update Y1 does *not* have to wait for updates X1-3 to complete, but is processed immediately.

To set the scope of a message, use method Message.setScope(short).

Scopes are implemented in a separate protocol called *Section 7.14.4, "SCOPE"*. This protocol has to be placed somewhere above ordering protocols like UNICAST or NAKACK (or SEQUENCER for that matter).

Uniqueness of scopes

Note that scopes should be *as unique as possible*. Compare this to hashing: the fewer collisions there are, the better the concurrency will be. So, if for example, two web sessions pick the same scope, then updates to those sessions will be delivered in the order in which they were sent, and not concurrently. While this doesn't cause erraneous behavior, it defies the purpose of SCOPE.

Also note that, if multicast and unicast messages have the same scope, they will be delivered in sequence. So if A multicasts messages to the group with scope 25, and A also unicasts messages to B with scope 25, then A's multicasts and unicasts will be delivered in order at B ! Again, this is correct, but since multicasts and unicasts are unrelated, might slow down things !

5.4.5. Out-of-band messages

OOB messages completely ignore any ordering constraints the stack might have. Any message marked as OOB will be processed by the OOB thread pool. This is necessary in cases where we don't want the message processing to wait until all other messages from the same sender have been processed, e.g. in the heartbeat case: if sender P sends 5 messages and then a response to a heartbeat request received from some other node, then the time taken to process P's 5 messages might take longer than the heartbeat timeout, so that P might get falsely suspected ! However, if the heartbeat response is marked as OOB, then it will get processed by the OOB thread pool and therefore might be concurrent to its previously sent 5 messages and not trigger a false suspicion.

The 2 unit tests UNICAST_OOB_Test and NAKACK_OOB_Test demonstrate how OOB messages influence the ordering, for both unicast and multicast messages.

5.4.6. Replacing the default and OOB thread pools

In 2.7, there are 3 thread pools and 4 thread factories in TP:

Table 5.2. Thread pools and factories in TP

Name	Description
Default thread pool	This is the pool for handling incoming messages. It can be fetched using getDefaultThreadPool() and replaced using setDefaultThreadPool(). When setting a thread pool, the old thread pool (if any) will be shutdown and all of it tasks cancelled first
OOB thread pool	This is the pool for handling incoming OOB messages. Methods to get and set it are getOOBThreadPool() and setOOBThreadPool()
Timer thread pool	This is the thread pool for the timer. The max number of threads is set through the timer.num_threads property. The timer thread pool cannot be set, it can only be retrieved using getTimer(). However, the thread factory of the timer can be replaced (see below)
Default thread factory	This is the thread factory (org.jgroups.util.ThreadFactory) of the default thread pool, which handles incoming messages. A thread pool factory is used to name threads and possibly make them daemons. It can be accessed using getDefaultThreadPoolThreadFactory() and setDefaultThreadPoolThreadFactory()
OOB thread factory	This is the thread factory for the OOB thread pool. It can be retrieved using getOOBThreadPoolThreadFactory() and set using method setOOBThreadPoolThreadFactory()
Timer thread factory	This is the thread factory for the timer thread pool. It can be accessed using getTimerThreadFactory() and setTimerThreadFactory()
Global thread factory	The global thread factory can get used (e.g. by protocols) to create threads which don't live in the transport, e.g. the FD_SOCK server socket handler thread. Each protocol has a method getTransport(). Once the TP is obtained, getThreadFactory() can be called to get the

Name	Description
	global thread factory. The global thread factory can be replaced with setThreadFactory()

Note

Note that thread pools and factories should be replaced after a channel has been created and before it is connected (`JChannel.connect()`).

5.4.7. Sharing of thread pools between channels in the same JVM

In 2.7, the default and OOB thread pools can be shared between instances running inside the same JVM. The advantage here is that multiple channels running within the same JVM can pool (and therefore save) threads. The disadvantage is that thread naming will not show to which channel instance an incoming thread belongs to.

Note that we can not just shared thread pools between JChannels within the same JVM, but we can also share entire transports. For details see *Section 5.2, "Sharing a transport between multiple channels in a JVM"*.

5.5. Using a custom socket factory

JGroups creates all of its sockets through a SocketFactory, which is located in the transport (TP) or TP.ProtocolAdapter (in a shared transport). The factory has methods to create sockets (Socket, ServerSocket, DatagramSocket and MulticastSocket) [3], closen sockets and list all open sockets. Every socket creation method has a service name, which could be for example "jgroups.fd_sock.srv_sock". The service name is used to look up a port (e.g. in a config file) and create the correct socket.

To provide one's own socket factory, the following has to be done: if we have a non-shared transport, the code below creates a SocketFactory implementation and sets it in the transport:

```
JChannel ch;
MySocketFactory factory; // e.g. extends DefaultSocketFactory
ch=new JChannel("config.xml");
ch.setSocketFactory(new MySocketFactory());
ch.connect("demo");
```

[3] Currently, SocketFactory does not support creation of NIO sockets / channels.

If a shared transport is used, then we have to set 2 socket factories: 1 in the shared transport and one in the TP.ProtocolAdapter:

```
JChannel c1=new JChannel("config.xml"), c2=new JChannel("config.xml");

TP transport=c1.getProtocolStack().getTransport();
transport.setSocketFactory(new MySocketFactory("transport"));

c1.setSocketFactory(new MySocketFactory("first-cluster"));
c2.setSocketFactory(new MySocketFactory("second-cluster"));

c1.connect("first-cluster");
c2.connect("second-cluster");
```

First, we grab one of the channels to fetch the transport and set a SocketFactory in it. Then we set one SocketFactory per channel that resides on the shared transport. When JChannel.connect() is called, the SocketFactory will be set in TP.ProtocolAdapter.

5.6. Handling network partitions

Network partitions can be caused by switch, router or network interface crashes, among other things. If we have a cluster {A,B,C,D,E} spread across 2 subnets {A,B,C} and {D,E} and the switch to which D and E are connected crashes, then we end up with a network partition, with subclusters {A,B,C} and {D,E}.

A, B and C can ping each other, but not D or E, and vice versa. We now have 2 coordinators, A and D. Both subclusters operate independently, for example, if we maintain a shared state, subcluster {A,B,C} replicate changes to A, B and C.

This means, that if during the partition, some clients access {A,B,C}, and others {D,E}, then we end up with different states in both subclusters. When a partition heals, the merge protocol (e.g. MERGE2) will notify A and D that there were 2 subclusters and merge them back into {A,B,C,D,E}, with A being the new coordinator and D ceasing to be coordinator.

The question is what happens with the 2 diverged substates ?

There are 2 solutions to merging substates: first we can attempt to create a new state from the 2 substates, and secondly we can shut down all members of the *non primary partition*, such that they have to re-join and possibly reacquire the state from a member in the primary partition.

In both cases, the application has to handle a MergeView (subclass of View), as shown in the code below:

```
```

```
public void viewAccepted(View view) {
    if(view instanceof MergeView) {
        MergeView tmp=(MergeView)view;
        Vector<View> subgroups=tmp.getSubgroups();
        // merge state or determine primary partition
        // run in a separate thread !
    }
}
```

It is essential that the merge view handling code run on a separate thread if it needs more than a few milliseconds, or else it would block the calling thread.

The MergeView contains a list of views, each view represents a subgroups and has the list of members which formed this group.

5.6.1. Merging substates

The application has to merge the substates from the various subgroups ({A,B,C} and {D,E}) back into one single state for {A,B,C,D,E}. This task *has* to be done by the application because JGroups knows nothing about the application state, other than it is a byte buffer.

If the in-memory state is backed by a database, then the solution is easy: simply discard the in-memory state and fetch it (eagerly or lazily) from the DB again. This of course assumes that the members of the 2 subgroups were able to write their changes to the DB. However, this is often not the case, as connectivity to the DB might have been severed by the network partition.

Another solution could involve tagging the state with time stamps. On merging, we could compare the time stamps for the substates and let the substate with the more recent time stamps win.

Yet another solution could increase a counter for a state each time the state has been modified. The state with the highest counter wins.

Again, the merging of state can only be done by the application. Whatever algorithm is picked to merge state, it has to be deterministic.

5.6.2. The primary partition approach

The primary partition approach is simple: on merging, one subgroup is designated as the *primary partition* and all others as non-primary partitions. The members in the primary partition don't do anything, whereas the members in the non-primary partitions need to drop their state and re-initialize their state from fresh state obtained from a member of the primary partition.

The code to find the primary partition needs to be deterministic, so that all members pick the *same* primary partition. This could be for example the first view in the MergeView, or we could sort all members of the new MergeView and pick the subgroup which contained the new coordinator (the one from the consolidated MergeView). Another possible solution could be to pick the

largest subgroup, and, if there is a tie, sort the tied views lexicographically (all Addresses have a compareTo() method) and pick the subgroup with the lowest ranked member.

Here's code which picks as primary partition the first view in the MergeView, then re-acquires the state from the *new* coordinator of the combined view:

```
public static void main(String[] args) throws Exception {
    final JChannel ch=new JChannel("/home/bela/udp.xml");
    ch.setReceiver(new ExtendedReceiverAdapter() {
        public void viewAccepted(View new_view) {
            handleView(ch, new_view);
        }
    });
    ch.connect("x");
    while(ch.isConnected())
        Util.sleep(5000);
    }

    private static void handleView(JChannel ch, View new_view) {
        if(new_view instanceof MergeView) {
            ViewHandler handler=new ViewHandler(ch, (MergeView)new_view);
            // requires separate thread as we don't want to block JGroups
            handler.start();
        }
    }

    private static class ViewHandler extends Thread {
        JChannel ch;
        MergeView view;

        private ViewHandler(JChannel ch, MergeView view) {
            this.ch=ch;
            this.view=view;
        }

        public void run() {
            Vector<View> subgroups=view.getSubgroups();
            View tmp_view=subgroups.firstElement(); // picks the first
            Address local_addr=ch.getLocalAddress();
            if(!tmp_view.getMembers().contains(local_addr)) {
                System.out.println("Not member of the new primary partition ("
                                   + tmp_view + "), will re-acquire the state");
                try {
                    ch.getState(null, 30000);
                }
                catch(Exception ex) {
                }
```

```
            }
            else {
                System.out.println("Not member of the new primary partition ("
                                   + tmp_view + "), will do nothing");
            }
        }
}
```

The handleView() method is called from viewAccepted(), which is called whenever there is a new view. It spawns a new thread which gets the subgroups from the MergeView, and picks the first subgroup to be the primary partition. Then, if it was a member of the primary partition, it does nothing, and if not, it reaqcuires the state from the coordinator of the primary partition (A).

The downside to the primary partition approach is that work (= state changes) on the non-primary partition is discarded on merging. However, that's only problematic if the data was purely in-memory data, and not backed by persistent storage. If the latter's the case, use state merging discussed above.

It would be simpler to shut down the non-primary partition as soon as the network partition is detected, but that a non trivial problem, as we don't know whether {D,E} simply crashed, or whether they're still alive, but were partitioned away by the crash of a switch. This is called a *split brain syndrome*, and means that none of the members has enough information to determine whether it is in the primary or non-primary partition, by simply exchanging messages.

5.6.3. The Split Brain syndrome and primary partitions

In certain situations, we can avoid having multiple subgroups where every subgroup is able to make progress, and on merging having to discard state of the non-primary partitions.

If we have a fixed membership, e.g. the cluster always consists of 5 nodes, then we can run code on a view reception that determines the primary partition. This code

- assumes that the primary partition has to have at least 3 nodes
- any cluster which has less than 3 nodes doesn't accept modfications. This could be done for shared state for example, by simply making the {D,E} partition read-only. Clients can access the {D,E} partition and read state, but not modify it.
- As an alternative, clusters without at least 3 members could shut down, so in this case D and E would leave the cluster.

The algorithm is shown in pseudo code below:

```
On initialization:
    - Mark the node as read-only
```

```
On view change V:
    - If V has >= N members:
        - If not read-write: get state from coord and switch to read-write
    - Else: switch to read-only
```

Of course, the above mechanism requires that at least 3 nodes are up at any given time, so upgrades have to be done in a staggered way, taking only one node down at a time. In the worst case, however, this mechanism leaves the cluster read-only and notifies a system admin, who can fix the issue. This is still better than shutting the entire cluster down.

5.7. Flushing: making sure every node in the cluster received a message

When sending messages, the properties of the default stacks (udp.xml, tcp.xml) are that all messages are delivered reliably to all (non-crashed) members. However, there are no guarantees with respect to the view in which a message will get delivered. For example, when a member A with view V1={A,B,C} multicasts message M1 to the group and D joins at about the same time, then D may or may not receive M1, and there is no guarantee that A, B and C receive M1 in V1 or V2={A,B,C,D}.

To change this, we can turn on virtual synchrony (by adding FLUSH to the top of the stack), which guarantees that

- A message M sent in V1 will be delivered in V1. So, in the example above, M1 would get delivered in view V1; by A, B and C, but not by D.
- The set of messages seen by members in V1 is the same for all members before a new view V2 is installed. This is important, as it ensures that all members in a given view see the same messages. For example, in a group {A,B,C}, C sends 5 messages. A receives all 5 messages, but B doesn't. Now C crashes before it can retransmit the messages to B. FLUSH will now ensure, that before installing V2={A,B} (excluding C), B gets C's 5 messages. This is done through the flush protocol, which has all members reconcile their messages before a new view is installed. In this case, A will send C's 5 messages to B.

Sometimes it is important to know that every node in the cluster received all messages up to a certain point, even if there is no new view being installed. To do this (initiate a manual flush), an application programmer can call Channel.startFlush() to start a flush and Channel.stopFlush() to terminate it.

Channel.startFlush() flushes all pending messages out of the system. This stops all senders (calling Channel.down() during a flush will block until the flush has completed)[4]. When startFlush() returns, the caller knows that (a) no messages will get sent anymore until stopFlush() is called and (b) all members have received all messages sent before startFlush() was called.

[4]Note that block() will be called in a Receiver when the flush is about to start and unblock() will be called when it ends

Channel.stopFlush() terminates the flush protocol, no blocked senders can resume sending messages.

Note that the FLUSH protocol has to be present on top of the stack, or else the flush will fail.

5.8. Large clusters

This section is a collection of best practices and tips and tricks for running large clusters on JGroups. By large clusters, we mean several hundred nodes in a cluster. These recommendations are captured in `udp-largecluster.xml` which is shipped with JGroups.

Note

This is work-in-progress, and `udp-largecluster.xml` is likely to see changes in the future.

5.8.1. Reducing chattiness

When we have a chatty protocol, scaling to a large number of nodes might be a problem: too many messages are sent and - because they are generated in addition to the regular traffic - this can have a negative impact on the cluster. A possible impact is that more of the regular messages are dropped, and have to be retransmitted, which impacts performance. Or heartbeats are dropped, leading to false suspicions. So while the negative effects of chatty protocols may not be seen in small clusters, they *will* be seen in large clusters !

5.8.1.1. Failure detection protocols

Failure detection protocols determine when a member is unresponsive, and subsequently *suspect* it. Usually (FD, FD_ALL), messages (heartbeats) are used to determine the health of a member, but we can also use TCP connections (FD_SOCK) to connect to a member P, and suspect P when the connection is closed.

Heartbeating requires messages to be sent around, and we need to be careful to limit the number of messages sent by a failure detection protocol (1) to detect crashed members and (2) when a member has been suspected. The following sections discuss how to configure FD_ALL, FD and FD_SOCK, the most commonly used failure detection protocols, for use in large clusters.

5.8.1.1.1. FD_SOCK

FD_SOCK is discussed in detail in *Section 7.5.3, “FD_SOCK”*.

5.8.1.1.2. FD

FD uses a ring topology, where every member sends heartbeats to its neighbor only. We recommend to use this protocol only when TCP is the transport, as it generates a lot of traffic in large clusters.

For details see *Section 7.5.1, “FD”*.

5.8.1.1.3. FD_ALL

FD_ALL has every member periodically multicast a heartbeat, and everyone updates internal tables of members and their last heartbeat received. When a member hasn't received a heartbeat from any given member for more than `timeout` ms, that member will get suspected.

FD_ALL is the recommended failure detection protocol when the transport provides IP multicasting capabilities (UDP).

For details see *Section 7.5.2, “FD_ALL”*.

5.9. STOMP support

STOMP is a JGroups protocol which implements the *STOMP* [http://stomp.codehaus.org] protocol. Currently (as of Aug 2011), transactions and acks are not implemented.

Adding the STOMP protocol to a configuration means that

- Clients written in different languages can subscribe to destinations, send messages to destinations, and receive messages posted to (subscribed) destinations. This is similar to JMS topics.
- Clients don't need to join any cluster; this allows for light weight clients, and we can run many of them.
- Clients can access a cluster from a remote location (e.g. across a WAN).
- STOMP clients can send messages to cluster members, and vice versa.

The location of a STOMP protocol in a stack is shown in *Figure 5.4, “STOMP in a protocol stack”*.

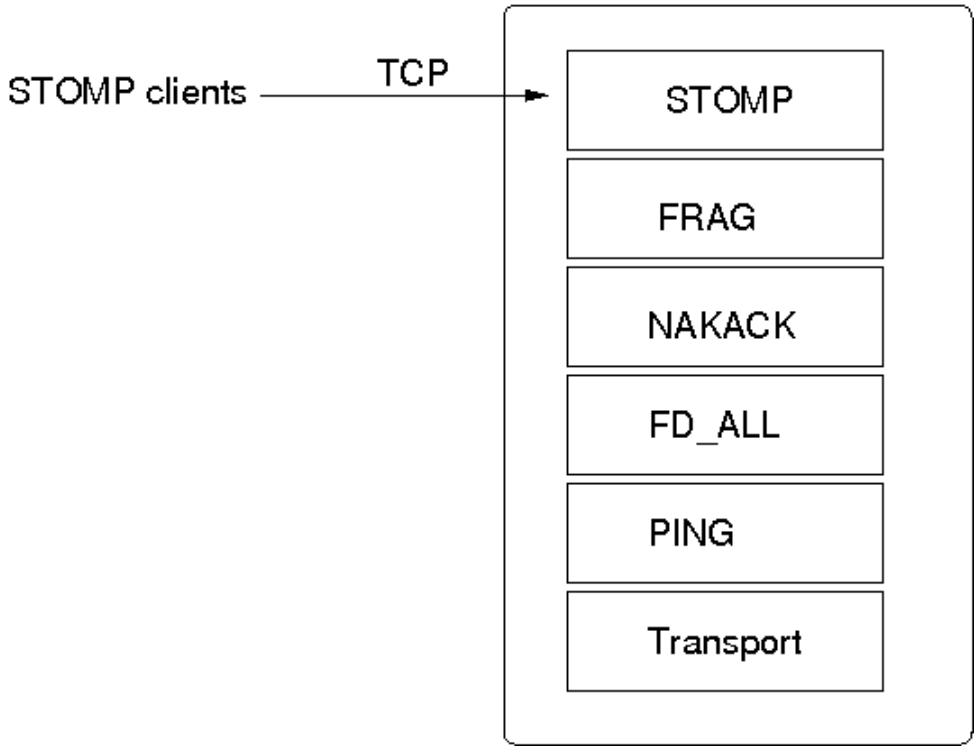

Figure 5.4. STOMP in a protocol stack

The STOMP protocol should be near the top of the stack.

A STOMP instance listens on a TCP socket for client connections. The port and bind address of the server socket can be defined via properties.

A client can send SUBSCRIBE commands for various destinations. When a SEND for a given destination is received, STOMP adds a header to the message and broadcasts it to all cluster nodes. Every node then in turn forwards the message to all of its connected clients which have subscribed to the same destination. When a destination is not given, STOMP simply forwards the message to *all* connected clients.

Traffic can be generated by clients and by servers. In the latter case, we could for example have code executing in the address space of a JGroups (server) node. In the former case, clients use the SEND command to send messages to a JGroups server and receive messages via the MESSAGE command. If there is code on the server which generates messages, it is important that both client and server code agree on a marshalling format, e.g. JSON, so that they understand each other's messages.

Clients can be written in any language, as long as they understand the STOMP protocol. Note that the JGroups STOMP protocol implementation sends additional information (e.g. INFO) to clients; non-JGroups STOMP clients should simply ignore them.

JGroups comes with a STOMP client (org.jgroups.client.StompConnection) and a demo (StompDraw). Both need to be started with the address and port of a JGroups cluster node. Once they have been started, the JGroups STOMP protocol will notify clients of cluster changes, which

is needed so client can failover to another JGroups server node when a node is shut down. E.g. when a client connects to C, after connection, it'll get a list of endpoints (e.g. A,B,C,D). When C is terminated, or crashes, the client automatically reconnects to any of the remaining nodes, e.g. A, B, or D. When this happens, a client is also re-subscribed to the destinations it registered for.

The JGroups STOMP protocol can be used when we have clients, which are either not in the same network segment as the JGroups server nodes, or which don't want to become full-blown JGroups server nodes. *Figure 5.5, "STOMP architecture"* shows a typical setup.

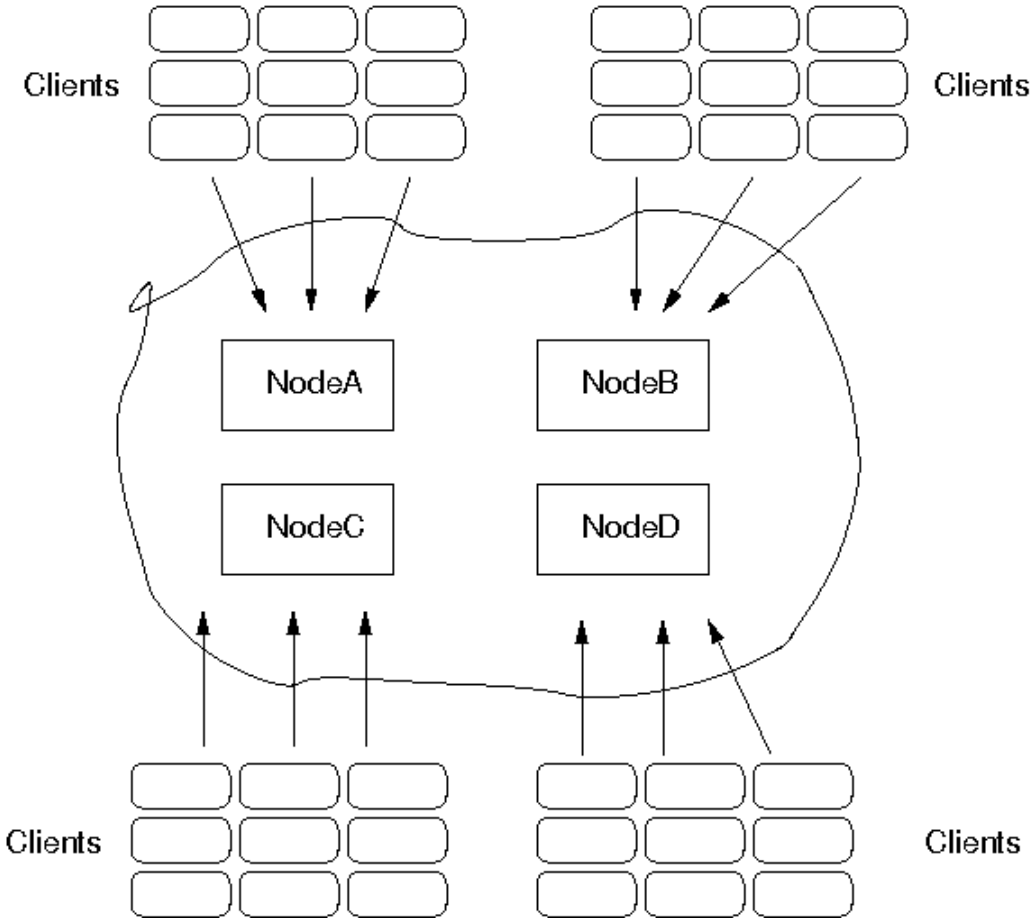

Figure 5.5. STOMP architecture

There are 4 nodes in a cluster. Say the cluster is in a LAN, and communication is via IP multicasting (UDP as transport). We now have clients which do not want to be part of the cluster themselves, e.g. because they're in a different geographic location (and we don't want to switch the main cluster to TCP), or because clients are frequently started and stopped, and therefore the cost of startup and joining wouldn't be amortized over the lifetime of a client. Another reason could be that clients are written in a different language, or perhaps, we don't want a large cluster, which could be the case if we for example have 10 JGroups server nodes and 1000 clients connected to them.

In the example, we see 9 clients connected to every JGroups cluster node. If a client connected to node A sends a message to destination /topics/chat, then the message is multicast from node

A to all other nodes (B, C and D). Every node then forwards the message to those clients which have previously subscribed to /topics/chat.

When node A crashes (or leaves) the JGroups STOMP clients (org.jgroups.client.StompConnection) simply pick another server node and connect to it.

For more information about STOMP see the blog entry at *http://belaban.blogspot.com/2010/10/stomp-for-jgroups.html*.

5.10. Bridging between remote clusters

In 2.12, the RELAY protocol was added to JGroups (for the properties see *Section 7.14.5, "RELAY"*). It allows for bridging of remote clusters. For example, if we have a cluster in New York (NYC) and another one in San Francisco (SFO), then RELAY allows us to bridge NYC and SFO, so that multicast messages sent in NYC will be forwarded to SFO and vice versa.

The NYC and SFO clusters could for example use IP multicasting (UDP as transport), and the bridge could use TCP as transport. The SFO and NYC clusters don't even need to use the same cluster name.

Figure 5.6, "Relaying between different clusters" shows how the two clusters are bridged.

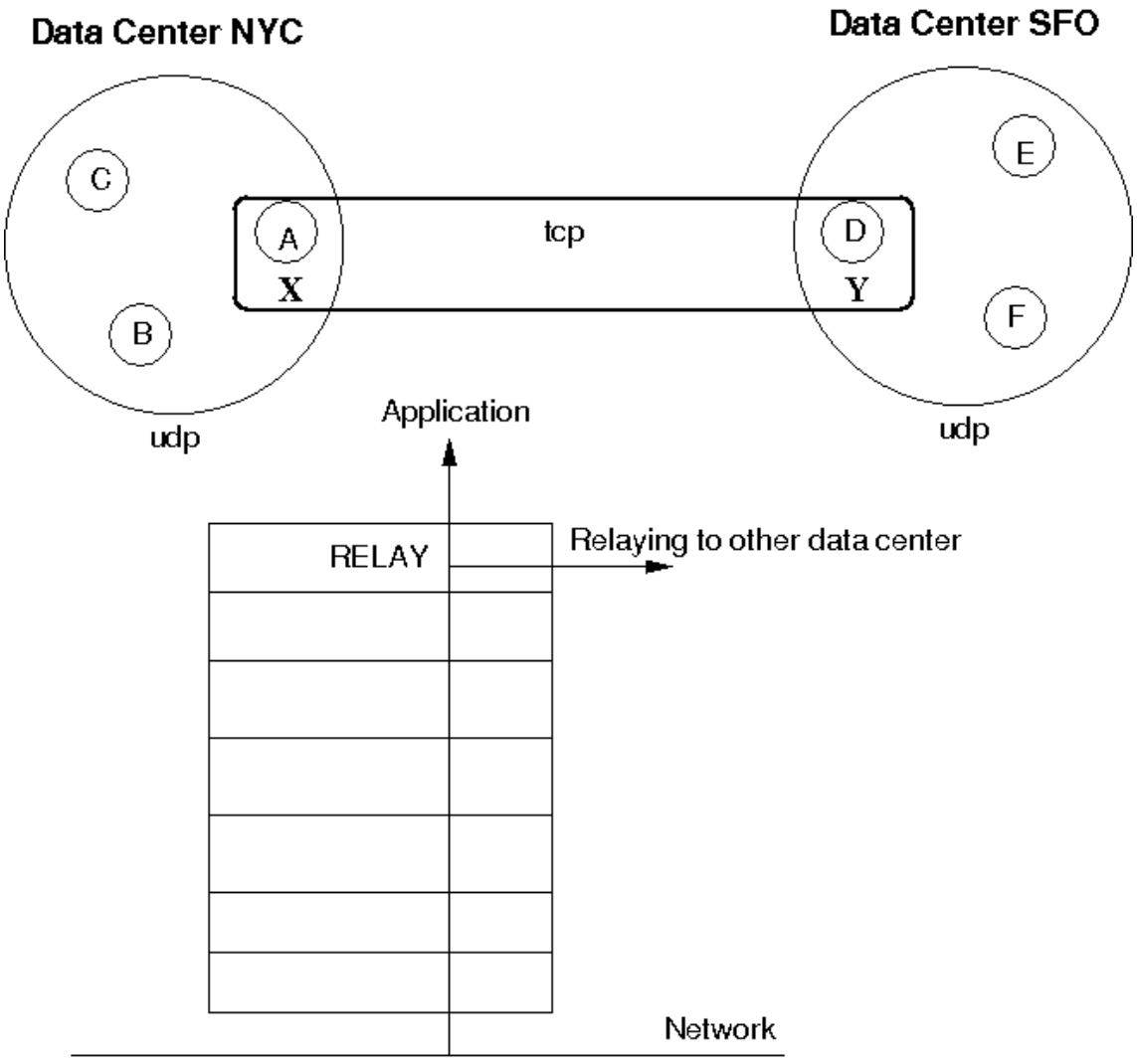

Figure 5.6. Relaying between different clusters

The cluster on the left side with nodes A (the coordinator), B and C is called "NYC" and use IP multicasting (UDP as transport). The cluster on the right side ("SFO") has nodes D (coordinator), E and F.

The bridge between the local clusters NYC and SFO is essentially another cluster with the coordinators (A and D) of the local clusters as members. The bridge typically uses TCP as transport, but any of the supported JGroups transports could be used (including UDP, if supported across a WAN, for instance).

Only a coordinator relays traffic between the local and remote cluster. When A crashes or leaves, then the next-in-line (B) takes over and starts relaying.

Relaying is done via the RELAY protocol added to the top of the stack. The bridge is configured with the bridge_props property, e.g. bridge_props="/home/bela/tcp.xml". This creates a JChannel inside RELAY.

Note that property "site" must be set in both subclusters. In the example above, we could set site="nyc" for the NYC subcluster and site="sfo" for the SFO ubcluster.

The design is described in detail in JGroups/doc/design/RELAY.txt (part of the source distribution). In a nutshell, multicast messages received in a local cluster are wrapped and forwarded to the remote cluster by a relay (= the coordinator of a local cluster). When a remote cluster receives such a message, it is unwrapped and put onto the local cluster.

JGroups uses subclasses of UUID (PayloadUUID) to ship the site name with an address. When we see an address with site="nyc" on the SFO side, then RELAY will forward the message to the SFO subcluster, and vice versa. When C multicasts a message in the NYC cluster, A will forward it to D, which will re-broadcast the message on its local cluster, with the sender being D. This means that the sender of the local broadcast will appear as D (so all retransmit requests got to D), but the original sender C is preserved in the header. At the RELAY protocol, the sender will be replaced with the original sender (C) having site="nyc". When node F wants to reply to the sender of the multicast, the destination of the message will be C, which is intercepted by the RELAY protocol and forwarded to the current relay (D). D then picks the correct destination (C) and sends the message to the remote cluster, where A makes sure C (the original sender) receives it.

An important design goal of RELAY is to be able to have completely autonomous clusters, so NYC doesn't for example have to block waiting for credits from SFO, or a node in the SFO cluster doesn't have to ask a node in NYC for retransmission of a missing message.

5.10.1. Views

RELAY presents a *global view* to the application, e.g. a view received by nodes could be {D,E,F,A,B,C}. This view is the same on all nodes, and a global view is generated by taking the two local views, e.g. A|5 {A,B,C} and D|2 {D,E,F}, comparing the coordinators' addresses (the UUIDs for A and D) and concatenating the views into a list. So if D's UUID is greater than A's UUID, we first add D's members into the global view ({D,E,F}), and then A's members.

Therefore, we'll always see all of A's members, followed by all of D's members, or the other way round.

To see which nodes are local and which ones remote, we can iterate through the addresses (PayloadUUID) and use the site (PayloadUUID.getPayload()) name to for example differentiate between "nyc" and "sfo".

5.10.2. Configuration

To setup a relay, we need essentially 3 XML configuration files: 2 to configure the local clusters and 1 for the bridge.

To configure the first local cluster, we can copy udp.xml from the JGroups distribution and add RELAY on top of it: <RELAY bridge_props="/home/bela/tcp.xml" />. Let's say we call this config relay.xml.

The second local cluster can be configured by copying relay.xml to relay2.xml. Then change the mcast_addr and/or mcast_port, so we actually have 2 different cluster in case we run instances

of both clusters in the same network. Of course, if the nodes of one cluster are run in a different network from the nodes of the other cluster, and they cannot talk to each other, then we can simply use the same configuration.

The 'site' property needs to be configured in relay.xml and relay2.xml, and it has to be different. For example, relay.xml could use site="nyc" and relay2.xml could use site="sfo".

The bridge is configured by taking the stock tcp.xml and making sure both local clusters can see each other through TCP.

5.11. Relaying between multiple sites (RELAY2)

Note

RELAY2 was added to JGroups in the 3.2 release.

Similar to *Section 5.10, "Bridging between remote clusters"*, RELAY2 provides clustering between sites. However, the differences to RELAY are:

- Clustering can be done between *multiple sites*. Currently (3.2), sites have to be directly reachable. In 3.3, hierarchical setups of sites will be implemented.
- Virtual (global) views are not provided anymore. If we have clusters SFO={A,B,C} and LON={X,Y,Z}, then both clusters are completed autonomous and don't know about each other's existence.
- Not only unicasts, but also multicasts can be routed between sites (configurable).

To use RELAY2, it has to be placed at the top of the configuration, e.g.:

```
<relay.RELAY2 site="LON" config="/home/bela/relay2.xml"
              relay_multicasts="true" />
<FORWARD_TO_COORD />
```

The above configuration has a site name which will be used to route messages between sites. To do that, addresses contain the site-ID, so we always know which site the address is from. E.g. an address A1:LON in the SFO site is not local, but will be routed to the remote site SFO.

The FORWARD_TO_COORD protocol is optional, but since it guarantees reliable message forwarding to the local site master, it is recommended. It makes sure that - if a local coordinator (site master) crashes or leaves while a message is being forwarded to it - the message will be forwarded to the next coordinator once elected.

The relay_multicasts property determines whether or not multicast messages (with dest = null) are relayed to the other sites, or not. When we have a site LON, connected to sites SFO and NYC, if a multicast message is sent in site LON, and relay_multicasts is true, then all members of sites SFO and NYC will receive the message.

The config property points to an XML file which defines the setup of the sites, e.g.:

```
<RelayConfiguration xmlns="urn:jgroups:relay:1.0">

    <sites>
        <site name="lon" id="0">
            <bridges>
                <bridge config="/home/bela/global.xml" name="global"/>
            </bridges>
        </site>

        <site name="nyc" id="1">
            <bridges>
                <bridge config="/home/bela/global.xml" name="global"/>
            </bridges>
        </site>

        <site name="sfo" id="2">
            <bridges>
                <bridge name="global" config="/home/bela/global.xml"/>
            </bridges>
        </site>
    </sites>
</RelayConfiguration>
```

Note

The configuration as shown above might change in 3.3, when hierarchical routing will be added.

This defines 3 sites LON, SFO and NYC. All the sites are connected to a global cluster (bus) "global" (defined by /home/bela/global.xml). All inter-site traffic will be sent via this global cluster (which has to be accessible by all of the sites). Intra-site traffic is sent via the cluster that's defined by the configuration of which RELAY2 is the top protocol.

The above configuration is not mandatory, ie. instead of a global cluster, we could define separate clusters between LON and SFO and LON and NYC. However, in such a setup, due to lack of

hierarchical routing, NYC and SFO wouldn't be able to send each other messages; only LON would be able to send message to SFO and NYC.

5.11.1. Relaying of multicasts

If relay_multicasts is true then any multicast received by the *site master* of a site (ie. the coordinator of the local cluster, responsible for relaying of unicasts and multicasts) will relay the multicast to all connected sites. This means that - beyond setting relay_multicasts - nothing has to be done in order to relay multicasts across all sites.

A recipient of a multicast message which originated from a different site will see that the sender's address is not a UUID, but a subclass (SiteUUID) which is the UUID plus the site suffix, e.g. A1:SFO. Since a SiteUUID is a subclass of a UUID, both types can be mixed and matched, placed into hashmaps or lists, and they implement compareTo() and equals() correctly.

When a reply is to be sent to the originator of the multicast message, Message.getSrc() provides the target address for the unicast response message. This is also a SiteUUID, but the sender of the response neither has to know nor take any special action to send the response, as JGroups takes care of routing the response back to the original sender.

5.11.2. Relaying of unicasts

As discussed above, relaying of unicasts is done transparently. However, if we don't have a target address (e.g. as a result of reception of a multicast), there is a special address *SiteMaster* which identifies the site master; the coordinator of a local cluster responsible for relaying of messages.

Class SiteMaster is created with the name of a site, e.g. new SiteMaster("LON"). When a unicast with destination SiteMaster("LON") is sent, then we relay the message to the *current* site master of LON. If the site master changes, messages will get relayed to a different node, which took over the role of the site master from the old (perhaps crashed) site master.

Sometimes only certain members of a site should become site masters; e.g. the more powerful boxes (as routing needs some additional CPU power), or multi-homed hosts which are connected to the external network (over which the sites are connected with each other).

To do this, RELAY2 can generate special addresses which contain the knowledge about whether a member should be skipped when selecting a site master from a view, or not. If can_become_site_master is set to false in RELAY2, then the selection process will skip that member. However, if all members in a given view are marked with can_become_site_master=false, then the first member of the view will get picked.

When we have all members in a view marked with can_become_site_master=false, e.g. {B,C,D}, then B is the site master. If we now start a member A with can_become_site_master=true, then B will stop being the site master and A will become the new site master.

5.11.3. Invoking RPCs across sites

Invoking RPCs across sites is more or less transparent, except for the case when we cannot reach a member of a remote site. If we want to invoke method foo() in A1, A2 (local) and SiteMaster("SFO"), we could write the following code:

```
List<Address> dests=new ArrayList<Address>(view.getMembers());
dests.add(new SiteMaster("SFO"));
RspList<Object> rsps;
rsps=disp.callRemoteMethods(dests, call,
           new RequestOptions(ResponseMode.GET_ALL, 5000).setAnycasting(true));
for(Rsp rsp: rsps.values()) {
    if(rsp.wasUnreachable())
        System.out.println("<< unreachable: " + rsp.getSender());
    else
       System.out.println("<< " + rsp.getValue() + " from " + rsp.getSender());
}
```

First, we add the members (A1 and A2) of the current (local) view to the destination set. Then we add the special address SiteMaster("SFO") which acts as a placeholder for the current coordinator of the SFO site.

Next, we invoke the call with dests as target set and block until responses from all A1, A2 and SiteMaster("SFO") have been received, or until 5 seconds have elapsed.

Next, we check the response list. And here comes the bit that's new in 3.2: if a site is unreachable, a Rsp has an additional field "unreachable", which means that we could not reach the site master of SFO for example. Note that this is not necessarily an error, as a site maybe currently down, but the caller now has the option of checking on this new status field.

5.11.4. Configuration

Let's configure an example which consists of 3 sites SFO, LON and NYC and 2 members in each site. First we define the configuration for the local cluster (site) SFO. To do this, we could for example copy udp.xml from the JGroups distro (and name it sfo.xml) and add RELAY2 to the top (as shown above). RELAY2's config property points to relay2.xml as shown above as well. The relay2.xml file defines a global cluster with global.xml, which uses TCP and MPING for the global cluster (copy for example tcp.xml to create global.xml)

Now copy sfo.xml to lon.xml and nyc.xml. The RELAY2 configuration stays the same for lon.xml and nyc.xml, but the multicast address and/or multicast port has to be changed in order to create 3 separate local clusters. Therefore, modify both lon.xml and nyc.xml and change mcast_port and / or mcast_addr in UDP to use separate values, so the clusters don't interfere with each other.

To test whether we have 3 different clusters, start the Draw application (shipped with JGroups):

```
java -Djava.net.preferIPv4Stack=true org.jgroups.demos.Draw -props ./sfo.xml
 -name sfo1
java -Djava.net.preferIPv4Stack=true org.jgroups.demos.Draw -props ./sfo.xml
 -name sfo2
java -Djava.net.preferIPv4Stack=true org.jgroups.demos.Draw -props ./lon.xml
 -name lon1
java -Djava.net.preferIPv4Stack=true org.jgroups.demos.Draw -props ./lon.xml
 -name lon2
java -Djava.net.preferIPv4Stack=true org.jgroups.demos.Draw -props ./nyc.xml
 -name nyc1
java -Djava.net.preferIPv4Stack=true org.jgroups.demos.Draw -props ./nyc.xml
 -name nyc2
```

We should now have 3 local clusters (= sites) of 2 instances each. When RELAY2.relay_multicasts is true, if you draw in one instance, we should see the drawing in all 6 instances. This means that relaying of multicasting between sites works. If this doesn't work, run a few Draw instances on global.xml, to see if they find each other.

Note that the first member of each cluster always joins the global cluster (defined by global.xml) too. This is necessary to relay messages between sites.

To test unicasts between sites, you can use the org.jgroups.demos.RelayDemoRpc program: start it as follows:

```
java -Djava.net.preferIPv4Stack=true org.jgroups.demos.RelayDemoRpc -
props ./sfo.xml -name sfo1
```

Start 2 instances in 3 sites and then use

```
mcast lon sfo nyc
```

to invoke RPCs on all local members and site masters SFO, NYC and LON. If one of the sites is down, you'll get a message stating the site is unreachable.

5.12. Daisychaining

Daisychaining refers to a way of disseminating messages sent to the entire cluster.

The idea behind it is that it is inefficient to broadcast a message in clusters where IP multicasting is not available. For example, if we only have TCP available (as is the case in most clouds today), then we have to send a broadcast (or group) message N-1 times. If we want to broadcast M to a cluster of 10, we send the same message 9 times.

Example: if we have {A,B,C,D,E,F}, and A broadcasts M, then it sends it to B, then to C, then to D etc. If we have a 1 GB switch, and M is 1GB, then sending a broadcast to 9 members takes 9

seconds, even if we parallelize the sending of M. This is due to the fact that the link to the switch only sustains 1GB / sec. (Note that I'm conveniently ignoring the fact that the switch will start dropping packets if it is overloaded, causing TCP to retransmit, slowing things down)...

Let's introduce the concept of a round. A round is the time it takes to send or receive a message. In the above example, a round takes 1 second if we send 1 GB messages. In the existing N-1 approach, it takes X * (N-1) rounds to send X messages to a cluster of N nodes. So to broadcast 10 messages a the cluster of 10, it takes 90 rounds.

Enter DAISYCHAIN.

The idea is that, instead of sending a message to N-1 members, we only send it to our neighbor, which forwards it to its neighbor, and so on. For example, in {A,B,C,D,E}, D would broadcast a message by forwarding it to E, E forwards it to A, A to B, B to C and C to D. We use a time-to-live field, which gets decremented on every forward, and a message gets discarded when the time-to-live is 0.

The advantage is that, instead of taxing the link between a member and the switch to send N-1 messages, we distribute the traffic more evenly across the links between the nodes and the switch. Let's take a look at an example, where A broadcasts messages m1 and m2 in cluster {A,B,C,D}, '-->' means sending:

5.12.1. Traditional N-1 approach

- Round 1: A(m1) --> B
- Round 2: A(m1) --> C
- Round 3: A(m1) --> D
- Round 4: A(m2) --> B
- Round 5: A(m2) --> C
- Round 6: A(m2) --> D

It takes 6 rounds to broadcast m1 and m2 to the cluster.

5.12.2. Daisychaining approach

- Round 1: A(m1) --> B
- Round 2: A(m2) --> B || B(m1) --> C
- Round 3: B(m2) --> C || C(m1) --> D
- Round 4: C(m2) --> D

In round 1, A send m1 to B.

In round 2, A sends m2 to B, but B also forwards m1 (received in round 1) to C.

In round 3, A is done. B forwards m2 to C and C forwards m1 to D (in parallel, denoted by '||').

In round 4, C forwards m2 to D.

5.12.3. Switch usage

Let's take a look at this in terms of switch usage: in the N-1 approach, A can only send 125MB/sec, no matter how many members there are in the cluster, so it is constrained by the link capacity to the switch. (Note that A can also receive 125MB/sec in parallel with today's full duplex links).

So the link between A and the switch gets hot.

In the daisychaining approach, link usage is more even: if we look for example at round 2, A sending to B and B sending to C uses 2 different links, so there are no constraints regarding capacity of a link. The same goes for B sending to C and C sending to D.

In terms of rounds, the daisy chaining approach uses X + (N-2) rounds, so for a cluster size of 10 and broadcasting 10 messages, it requires only 18 rounds, compared to 90 for the N-1 approach !

5.12.4. Performance

To measure performance of DAISYCHAIN, a performance test (test.Perf) was run, with 4 nodes connected to a 1 GB switch; and every node sending 1 million 8K messages, for a total of 32GB received by every node. The config used was tcp.xml.

The N-1 approach yielded a throughput of 73 MB/node/sec, and the daisy chaining approach 107MB/node/sec !

5.12.5. Configuration

DAISYCHAIN can be placed directly on top of the transport, regardless of whether it is UDP or TCP, e.g.

```
<TCP .../>
<DAISYCHAIN .../>
<TCPPING .../>
```

5.13. Tagging messages with flags

A message can be tagged with a selection of *flags*, which alter the way certain protocols treat the message. This is done as follows:

```
```

```
Message msg=new Message();
msg.setFlag(Message.OOB);
msg.setFlag(Message.NO_FC);
```

Here we tag the message to be OOB (out of band) and to bypass flow control.

The advantage of tagging messages is that we don't need to change the configuration, but instead can override it on a per-message basis.

The available flags are:

Message.OOB
: This tags a message as out-of-band, which will get it processed by the out-of-band thread pool at the receiver's side. Note that an OOB message does not provide any ordering guarantees, although OOB messages are reliable (no loss) and are delivered only once. See *Section 5.4.5, "Out-of-band messages"* for details.

Message.DONT_BUNDLE
: This flag causes the transport not to bundle the message, but to send it immediately.

  See *Section 5.3.1.1, "Message bundling and performance"* for a discussion of the DONT_BUNDLE flag with respect to performance of blocking RPCs.

Message.NO_FC
: This flag bypasses any flow control protocol (see *Section 7.9, "Flow control"*) for a discussion of flow control protocols.

Message.SCOPED
: This flag is set automatically when `Message.setScope()` is called. See *Section 5.4.4, "Scopes: concurrent message delivery for messages from the same sender"* for a discussion on scopes.

Message.NO_RELIABILITY
: When sending unicast or multicast messages, some protocols (UNICAST, NAKACK) add sequence numbers to the messages in order to (1) deliver them reliably and (2) in order.

  If we don't want reliability, we can tag the message with flag NO_RELIABILITY. This means that a message tagged with this flag may not be received, may be received more than once, or may be received out of order.

  A message tagged with NO_RELIABILITY will simply bypass reliable protocols such as UNICAST and NAKACK.

  For example, if we send multicast message M1, M2 (NO_RELIABILITY), M3 and M4, and the starting sequence number is #25, then M1 will have seqno #25, M3 will have #26 and M4 will have #27. We can see that we don't allocate a seqno for M2 here.

Message.NO_TOTAL_ORDER
: If we use a total order configuration with SEQUENCER (*Section 7.11.1, "SEQUENCER"*), then we can bypass SEQUENCER (if we don't need total order for a given message) by tagging the message with `NO_TOTAL_ORDER`.

Message.NO_RELAY
: If we use RELAY (see *Section 5.10, "Bridging between remote clusters"*) and don't want a message to be relayed to the other site(s), then we can tag the message with NO_RELAY.

Message.RSVP
: When this flag is set, a message send will block until the receiver (unicast) or receivers (multicast) have acked reception of the message, or until a timeout occurs. See *Section 3.8.8.2, "Synchronous messages"* for details.

5.14. Performance tests

There are a number of performance tests shipped with JGroups. The section below discusses MPerf, which is a replacement for (static) perf.Test. This change was done in 3.1.

5.14.1. MPerf

MPerf is a test which measures multicast performance. This doesn't mean *IP multicast* performance, but *point-to-multipoint* performance. Point-to-multipoint means that we measure performance of one-to-many messages; in other words, messages sent to all cluster members.

Compared to the old perf.Test, MPerf is dynamic; it doesn't need a setup file to define the number of senders, number of messages to be sent and message size. Instead, all the configuration needed by an instance of MPerf is an XML stack configuration, and configuration changes done in one member are automatically broadcast to all other members.

MPerf can be started as follows:

```
java -cp $CLASSPATH -Djava.net.preferIPv4Stack=true
 org.jgroups.tests.perf.MPerf -props ./fast.xml
```

This assumes that we're using IPv4 addresses (otherwise IPv6 addresses are used) and the JGroups JAR on CLASSPATH.

A screen shot of MPerf looks like this (could be different, depending on the JGroups version):

```
[linux]/home/bela$ mperf.sh -props ./fast.xml -name B

----------------------- MPerf -----------------------
Date: Mon Dec 12 15:33:21 CET 2011
```

```
Run by: bela
JGroups version: 3.1.0.Alpha1

-------------------------------------------------------------------
GMS: address=B, cluster=mperf, physical address=192.168.1.5:46614
-------------------------------------------------------------------
** [A|9] [A, B]
num_msgs=1000000
msg_size=1000
num_threads=1
[1] Send [2] View
[3] Set num msgs (1000000) [4] Set msg size (1KB) [5] Set threads (1)
[6] New config (./fast.xml)
[x] Exit this [X] Exit all
```

We're starting MPerf with `-props ./fast.xml` and `-name B`. The -props option points to a JGroups configuration file, and -name gives the member the name "B".

MPerf can then be run by pressing [1]. In this case, every member in the cluster (in the example, we have members A and B) will send 1 million 1K messages. Once all messages have been received, MPerf will write a summary of the performance results to stdout:

```
[1] Send [2] View
[3] Set num msgs (1000000) [4] Set msg size (1KB) [5] Set threads (1)
[6] New config (./fast.xml)
[x] Exit this [X] Exit all
1
-- sending 1000000 msgs
++ sent 100000
-- received 200000 msgs (1410 ms, 141843.97 msgs/sec, 141.84MB/sec)
++ sent 200000
-- received 400000 msgs (1326 ms, 150829.56 msgs/sec, 150.83MB/sec)
++ sent 300000
-- received 600000 msgs (1383 ms, 144613.16 msgs/sec, 144.61MB/sec)
++ sent 400000
-- received 800000 msgs (1405 ms, 142348.75 msgs/sec, 142.35MB/sec)
++ sent 500000
-- received 1000000 msgs (1343 ms, 148920.33 msgs/sec, 148.92MB/sec)
++ sent 600000
-- received 1200000 msgs (1700 ms, 117647.06 msgs/sec, 117.65MB/sec)
++ sent 700000
-- received 1400000 msgs (1399 ms, 142959.26 msgs/sec, 142.96MB/sec)
++ sent 800000
-- received 1600000 msgs (1359 ms, 147167.03 msgs/sec, 147.17MB/sec)
++ sent 900000
-- received 1800000 msgs (1689 ms, 118413.26 msgs/sec, 118.41MB/sec)
++ sent 1000000
```

```
-- received 2000000 msgs (1519 ms, 131665.57 msgs/sec, 131.67MB/sec)

Results:

B: 2000000 msgs, 2GB received, msgs/sec=137608.37, throughput=137.61MB
A: 2000000 msgs, 2GB received, msgs/sec=137959.58, throughput=137.96MB

================================================================================
 Avg/node:    2000000 msgs, 2GB received, msgs/sec=137788.49,
 throughput=137.79MB
 Avg/cluster: 4000000 msgs, 4GB received, msgs/sec=275576.99,
 throughput=275.58MB
================================================================================


[1] Send [2] View
[3] Set num msgs (1000000) [4] Set msg size (1KB) [5] Set threads (1) [6]
 New config (./fast.xml)
[x] Exit this [X] Exit all
```

In the sample run above, we see member B's screen. B sends 1 million messages and waits for its 1 million and the 1 million messages from B to be received before it dumps some stats to stdout. The stats include the number of messages and bytes received, the time, the message rate and throughput averaged over the 2 members. It also shows the aggregated performance over the entire cluster.

In the sample run above, we got an average 137MB of data per member per second, and an aggregated 275MB per second for the entire cluster (A and B in this case).

Parameters such as the number of messages to be sent, the message size and the number of threads to be used to send the messages can be configured by pressing the corresponding numbers. After pressing return, the change will be broadcast to all cluster members, so that we don't have to go to each member and apply the same change. Also, new members started, will fetch the current configuration and apply it.

For example, if we set the message size in A to 2000 bytes, then the change would be sent to B, which would apply it as well. If we started a third member C, it would also have a configuration with a message size of 2000.

Another feature is the ability to restart all cluster members with a new configuration. For example, if we modified `./fast.xml`, we could select [6] to make all cluster members disconnect and close their existing channels and start a new channel based on the modified fast.xml configuration.

The new configuration file doesn't even have to be accessible on all cluster members; only on the member which makes the change. The file contents will be read by that member, converted into a byte buffer and shipped to all cluster members, where the new channel will then be created with the byte buffer (converted into an input stream) as config.

Being able to dynamically change the test parameters and the JGroups configuration makes MPerf suited to be run in larger clusters; unless a new JGroups version is installed, MPerf will never have to be restarted manually.

5.15. Ergonomics

Ergonomics is similar to the dynamic setting of optimal values for the JVM, e.g. garbage collection, memory sizes etc. In JGroups, ergonomics means that we try to dynamically determine and set optimal values for protocol properties. Examples are thread pool size, flow control credits, heartbeat frequency and so on.

There is an `ergonomics` property which can be enabled or disabled for every protocol. The default is true. To disable it, set it to false, e.g.:

```
<UDP... />
<PING ergonomics="false"/>
```

Here we leave ergonomics enabled for UDP (the default is true), but disable it for PING.

Ergonomics is work-in-progress, and will be implemented over multiple releases.

5.16. Supervising a running stack

SUPERVISOR (*Section 7.14.13, "SUPERVISOR"*) provides a rule based fault detection and correction protocol. It allows for rules to be installed, which are periodically invoked. When invoked, a condition can be checked and corrective action can be taken to fix the problem. Essentially, SUPERVISOR acts like a human administrator, except that condition checking and action triggering is done automatically.

An example of a rule is org.jgroups.protocols.rules.CheckFDMonitor: invoked periodically, it checks if the monitor task in FD is running when the membership is 2 or more and - if not - restarts it. The sections below show how to write the rule and how to invoke it.

All rules to be installed in SUPERVISOR are listed in an XML file, e.g. rules.xml:

```
<rules xmlns="urn:jgroups:rules:1.0">
     <rule name="rule1" class="org.jgroups.protocols.rules.CheckFDMonitorRule"
           interval="1000"/>
</rules>
```

There is only one rule "rule1" present, which is run every second. The name of the class implementing the rule is "org.jgroups.protocols.rules.CheckFDMonitorRule", and its implementation is:

```
public class CheckFDMonitor extends Rule {
    protected FD fd;

    public String name() {return "sample";}

    public String description() {
        return "Starts FD.Monitor if membership > 1 and monitor isn't running";
    }

    public void init() {
        super.init();
        fd=(FD)sv.getProtocolStack().findProtocol(FD.class);
        if(fd == null) {
            log.info("FD was not found, uninstalling myself (sample)");
            sv.uninstallRule("sample");
        }
    }

    public boolean eval() {
        return sv.getView() != null && sv.getView().size() > 1
            && !fd.isMonitorRunning();
    }

    public String condition() {
        View view=sv.getView();
        return "Membership is " + (view != null? view.size() : "n/a") +
            ", FD.Monitor running=" + fd.isMonitorRunning();
    }

    public void trigger() throws Throwable {
      System.out.println(sv.getLocalAddress() + ": starting failure detection");
        fd.startFailureDetection();
    }
}
```

CheckFDMonitor extends abstract class Rule which sets a reference to SUPERVISOR and the log when the rule has been installed.

Method name() needs to return a unique name by which the rule can be uninstalled later if necessary.

Description() should provide a meaningful description (used by JMX).

In init(), a reference to FD is set by getting the protocol stack from the SUPERVISOR (sv). If not found, e.g. because there is no FD protocol present in a given stack, the rule uninstalls itself.

Method eval() is called every second. It checks that the monitor task in FD is running (when the membership is 2 or more) and, if not, returns true. In that case, method trigger() will get called by the code in the Rule superclass and it simply restarts the stopped monitor task.

Note that rules can be installed and uninstalled dynamically at runtime, e.g. via probe.sh:

```
probe.sh op=SUPERVISOR.installRule["myrule",
 1000,"org.jgroups.protocols.rules.CheckFDMonitor"]
```

installs rule CheckFDMonitor as "myrule" into the running system, and this rule will be run every 1000 ms.

```
probe.sh op=SUPERVISOR.uninstallRule["myrule"]
```

uninstalls "myrule" again.

```
probe.sh op=SUPERVISOR.dumpRules
```

dumps a list of currently installed rules to stdout.

5.17. Probe

Probe is the Swiss Army Knife for JGroups; it allows to fetch information about the members running in a cluster, get and set properties of the various protocols, and invoke methods in all cluster members.

Probe can even insert protocols into running cluster members, or remove/replace existing protocols. Note that this doesn't make sense though with *stateful* protocols such as NAKACK. But this feature is helpful, it could be used for example to insert a diagnostics or stats protocol into a running system. When done, the protocol can be removed again.

Of course, probe could be used to do crazy things: one could insert the BSH (BeanShell, a Java interpreter) protocol into all cluster members, then connect to the individual nodes (listening on a TCP port), and inject Java code into the running JVM ! To prevent this, probing can be disabled (see below). The default configurations have it *enabled*.

Probe is a script (`probe.sh` in the `bin` directory of the source distribution) that can be invoked on any of the hosts in same network in which a cluster is running.

Note

Probe currently requires IP multicasting to be enabled in a network, in order to discover the cluster members in a network. It *can* be used with TCP as transport, but still requires multicasting.

The probe.sh script essentially calls `org.jgroups.tests.Probe` which is part of the JGroups JAR.

The way probe works is that every stack has an additional multicast socket that by default listens on 224.0.75.75:7500 for diagnostics requests from probe. The configuration is located in the transport protocol (e.g. UDP), and consists of the following properties:

Table 5.3. Properties for diagnostics / probe

Name	Description
enable_diagnostics	Whether or not to enable diagnostics (default: true). When enabled, this will create a MulticastSocket and we have one additional thread listening for probe requests. When disabled, we'll have neither the thread nor the socket created.
diagnostics_addr	The multicast address which the MulticastSocket should join. The default is `"224.0.75.75"` for IPv4 and `"ff0e::0:75:75" for IPv6`.
diagnostics_port	The port on which the MulticastSocket should listen. The default is 7500.

Probe is extensible; by implementing a `ProbeHandler` and registering it with the transport (`TP.registerProbeHandler()`), any protocol, or even *applications* can register functionality to be invoked via probe. Refer to the javadoc for details.

To get information about the cluster members running in the local network, we can use the following probe command (note that probe could also be invoked as `java -classpath $CP org.jgroups.tests.Probe $*`):

```
[linux]/home/bela/JGroups$ probe.sh

-- send probe on /224.0.75.75:7500


#1 (149 bytes):
local_addr=A [1a1f543c-2332-843b-b523-8d7653874de7]
cluster=DrawGroupDemo
```

```
view=[A|1] [A, B]
physical_addr=192.168.1.5:43283
version=3.0.0.Beta1


#2 (149 bytes):
local_addr=B [88588976-5416-b054-ede9-0bf8d4b56c02]
cluster=DrawGroupDemo
view=[A|1] [A, B]
physical_addr=192.168.1.5:35841
version=3.0.0.Beta1



2 responses (2 matches, 0 non matches)
[linux]/home/bela/JGroups$
```

This gets us 2 responses, from A and B. "A" and "B" are the logical names, but we also see the UUIDs. They're both in the same cluster ("DrawGroupDemo") and both have the same view (`[A|1] [A, B]`). The physical address and the version of both members is also shown.

Note that `probe.sh -help` lists the command line options.

To fetch all of the JMX information from all protocols, we can invoke

```
probe jmx
```

However, this dumps all of the JMX attributes from all protocols of all cluster members, so make sure to pipe the output into a file and awk and sed it for legibility !

However, we can also JMX information from a specific protocol, e.g. FRAG2 (slightly edited>:

```
[linux]/home/bela$ probe.sh  jmx=FRAG2

-- send probe on /224.0.75.75:7500


#1 (318 bytes):
local_addr=B [88588976-5416-b054-ede9-0bf8d4b56c02]
cluster=DrawGroupDemo
physical_addr=192.168.1.5:35841
jmx=FRAG2={id=5, level=off, num_received_msgs=131, frag_size=60000,
           num_sent_msgs=54, stats=true, num_sent_frags=0,
           name=FRAG2, ergonomics=true, num_received_frags=0}

view=[A|1] [A, B]
version=3.0.0.Beta1
```

```
#2 (318 bytes):
local_addr=A [1a1f543c-2332-843b-b523-8d7653874de7]
cluster=DrawGroupDemo
physical_addr=192.168.1.5:43283
jmx=FRAG2={id=5, level=off, num_received_msgs=131, frag_size=60000,
           num_sent_msgs=77, stats=true, num_sent_frags=0,
           name=FRAG2, ergonomics=true, num_received_frags=0}

view=[A|1] [A, B]
version=3.0.0.Beta1




2 responses (2 matches, 0 non matches)
[linux]/home/bela$
```

We can also get information about specific properties in a given protocol:

```
[linux]/home/bela$ probe.sh  jmx=NAKACK.xmit

-- send probe on /224.0.75.75:7500


#1 (443 bytes):
local_addr=A [1a1f543c-2332-843b-b523-8d7653874de7]
cluster=DrawGroupDemo
physical_addr=192.168.1.5:43283
jmx=NAKACK={xmit_table_max_compaction_time=600000, xmit_history_max_size=50,
            xmit_rsps_sent=0, xmit_reqs_received=0, xmit_table_num_rows=5,
            xmit_reqs_sent=0, xmit_table_resize_factor=1.2,
            xmit_from_random_member=false, xmit_table_size=78,
            xmit_table_msgs_per_row=10000, xmit_rsps_received=0}

view=[A|1] [A, B]
version=3.0.0.Beta1


#2 (443 bytes):
local_addr=B [88588976-5416-b054-ede9-0bf8d4b56c02]
cluster=DrawGroupDemo
physical_addr=192.168.1.5:35841
jmx=NAKACK={xmit_table_max_compaction_time=600000, xmit_history_max_size=50,
            xmit_rsps_sent=0, xmit_reqs_received=0, xmit_table_num_rows=5,
```

```
             xmit_reqs_sent=0, xmit_table_resize_factor=1.2,
             xmit_from_random_member=false, xmit_table_size=54,
             xmit_table_msgs_per_row=10000, xmit_rsps_received=0}

view=[A|1] [A, B]
version=3.0.0.Beta1




2 responses (2 matches, 0 non matches)
[linux]/home/bela$
```

This returns all JMX attributes that start with `"xmit"` in all NAKACK protocols of all cluster members. We can also pass a list of attributes:

```
[linux]/home/bela$ probe.sh  jmx=NAKACK.missing,xmit

-- send probe on /224.0.75.75:7500


#1 (468 bytes):
local_addr=A [1a1f543c-2332-843b-b523-8d7653874de7]
cluster=DrawGroupDemo
physical_addr=192.168.1.5:43283
jmx=NAKACK={xmit_table_max_compaction_time=600000, xmit_history_max_size=50,
             xmit_rsps_sent=0, xmit_reqs_received=0, xmit_table_num_rows=5,
             xmit_reqs_sent=0, xmit_table_resize_factor=1.2,
             xmit_from_random_member=false, xmit_table_size=78,
             missing_msgs_received=0, xmit_table_msgs_per_row=10000,
             xmit_rsps_received=0}

view=[A|1] [A, B]
version=3.0.0.Beta1


#2 (468 bytes):
local_addr=B [88588976-5416-b054-ede9-0bf8d4b56c02]
cluster=DrawGroupDemo
physical_addr=192.168.1.5:35841
jmx=NAKACK={xmit_table_max_compaction_time=600000, xmit_history_max_size=50,
             xmit_rsps_sent=0, xmit_reqs_received=0, xmit_table_num_rows=5,
             xmit_reqs_sent=0, xmit_table_resize_factor=1.2,
             xmit_from_random_member=false, xmit_table_size=54,
             missing_msgs_received=0, xmit_table_msgs_per_row=10000,
             xmit_rsps_received=0}
```

```
view=[A|1] [A, B]
version=3.0.0.Beta1




2 responses (2 matches, 0 non matches)
[linux]/home/bela$
```

This returns all attributes of NAKACK that start with `"xmit"` or `"missing"`.

To invoke an operation, e.g. to set the logging level in all UDP protocols from "warn" to "trace", we can use `probe.sh op=UPD.setLevel["trace"]`. This raises the logging level in all UDP protocols of all cluster members, which is useful to diagnose a running system.

Operation invocation uses reflection, so any method defined in any protocol can be invoked. This is a powerful tool to get diagnostics information from a running cluster.

For further information, refer to the command line options of probe (`probe.sh -h`).

Writing protocols

This chapter discusses how to write custom protocols

6.1. Writing user defined headers

Headers are mainly used by protocols, to ship additional information around with a message, without having to place it into the payload buffer, which is often occupied by the application already. However, headers can also be used by an application, e.g. to add information to a message, without having to squeeze it into the payload buffer.

A header has to extend org.jgroups.Header, have an empty public constructor and implement the `Streamable` interface (writeTo() and readFrom() methods).

A header should also override size(), which returns the total number of bytes taken up in the output stream when an instance is marshalled using Streamable. Streamable is an interface for efficient marshalling with methods

```
public interface Streamable {

    /** Write the entire state of the current object (including superclasses)
        to outstream. Note that the output stream must not be closed */
    void writeTo(DataOutput out) throws IOException;

    /** Read the state of the current object (including superclasses) from
        instream. Note that the input stream must not be closed */
    void readFrom(DataInput in) throws IOException,
                                       IllegalAccessException,
                                       InstantiationException;
}
```

Method writeTo() needs to write all relevant instance variables to the output stream and readFrom() needs to read them back in.

It is important that size() returns the correct number of bytes, because some components (such a message bundling in the transport) depend on this, as they need to measure the exact number of bytes before sending a message. If size() returns fewer bytes than what will actually be written to the stream, then it is possible that (if we use UDP with a 65535 bytes maximum) the datagram packet is dropped by UDP !

The final requirement is to add the newly created header class to jg-magic-map.xml (in the ./conf directory), or - if this is not a JGroups internal protocol - to add the class to ClassConfigurator. This can be done with method

```
ClassConfigurator.getInstance().put(1899, MyHeader.class)
```

.

The code below shows how an application defines a custom header, MyHeader, and uses it to attach additional information to message sent (to itself):

```
public class bla {

    public static void main(String[] args) throws Exception {
        JChannel ch=new JChannel();
        ch.connect("demo");
        ch.setReceiver(new ReceiverAdapter() {
            public void receive(Message msg) {
                MyHeader hdr=(MyHeader)msg.getHeader("x");
                System.out.println("-- received " + msg +
                                   ", header is " + hdr);
            }
        });

        ClassConfigurator.getInstance().add((short)1900, MyHeader.class);

        int cnt=1;
        for(int i=0; i < 5; i++) {
            Message msg=new Message();
            msg.putHeader((short)1900, new MyHeader(cnt++));
            ch.send(msg);
        }
        ch.close();
    }


    public static class MyHeader extends Header implements Streamable {
        int counter=0;

        public MyHeader() {
        }

        private MyHeader(int counter) {
            this.counter=counter;
        }

        public String toString() {
            return "counter=" + counter;
        }
```

```
         public int size() {
             return Global.INT_SIZE;
         }

        public void writeTo(DataOutputStream out) throws IOException {
             out.writeInt(counter);
        }

        public void readFrom(DataInputStream in) throws IOException,
                                                        IllegalAccessException,
                                                        InstantiationException {
             counter=in.readInt();
        }
    }
}
```

The MyHeader class has an empty public constructor and implements the writeExternal() and readExternal() methods with no-op implementations.

The state is represented as an integer counter. Method size() returns 4 bytes (Global.INT_SIZE), which is the number of bytes written by writeTo() and read by readFrom().

Before sending messages with instances of MyHeader attached, the program registers the MyHeader class with the ClassConfigurator. The example uses a magic number of 1900, but any number greater than 1024 can be used. If the magic number was already taken, an IllegalAccessException would be thrown.

The final part is adding an instance of MyHeader to a message using Message.putHeader(). The first argument is a name which has to be unique across all headers for a given message. Usually, protocols use the protocol name (e.g. "UDP", "NAKACK"), so these names should not be used by an application. The second argument is an instance of the header.

Getting a header is done through Message.getHeader() which takes the name as argument. This name of course has to be the same as the one used in putHeader().

List of Protocols

This chapter describes the most frequently used protocols, and their configuration. *Ergonomics* (*Section 5.15, "Ergonomics"*) strives to reduce the number of properties that have to be configured, by dynamically adjusting them at run time, however, this is not yet in place.

Meanwhile, we recommend that users should copy one of the predefined configurations (shipped with JGroups), e.g. `udp.xml` or `tcp.xml`, and make only minimal changes to it.

This section is work in progress; we strive to update the documentation as we make changes to the code.

7.1. Properties availabe in every protocol

The table below lists properties that are available in all protocols, as they're defined in the superclass of all protocols, `org.jgroups.stack.Protocol`.

Table 7.1. Properties of `org.jgroups.stack.Protocol`

Name	Description
stats	Whether the protocol should collect protocol-specific runtime statistics. What those statistics are (or whether they even exist) depends on the particular protocol. See the org.jgroups.stack.Protocol javadoc for the available API related to statistics. Default is true.
ergonomics	Turns on ergonomics. See *Section 5.15, "Ergonomics"* for details.
id	Gives the protocol a different ID if needed so we can have multiple instances of it in the same stack

7.2. Transport

`TP` is the base class for all transports, e.g. UDP and TCP. All of the properties defined here are inherited by the subclasses. The properties for `TP` are:

Table 7.2. Properties

Name	Description
bind_addr	The bind address which should be used by this transport. The following special values are also recognized: GLOBAL, SITE_LOCAL, LINK_LOCAL and NON_LOOPBACK

Name	Description
bind_interface_str	The interface (NIC) which should be used by this transport
bind_port	The port to which the transport binds. Default of 0 binds to any (ephemeral) port
bundler_capacity	The max number of elements in a bundler if the bundler supports size limitations
bundler_type	The type of bundler used. Has to be "old" (default) or "new"
diagnostics_addr	Address for diagnostic probing. Default is 224.0.75.75
diagnostics_bind_interfaces	Comma delimited list of interfaces (IP addresses or interface names) that the diagnostics multicast socket should bind to
diagnostics_passcode	Authorization passcode for diagnostics. If specified every probe query will be authorized
diagnostics_port	Port for diagnostic probing. Default is 7500
diagnostics_ttl	TTL of the diagnostics multicast socket
discard_incompatible_packets	Discard packets with a different version if true
enable_bundling	Enable bundling of smaller messages into bigger ones. Default is true
enable_diagnostics	Switch to enable diagnostic probing. Default is true
enable_unicast_bundling	Enable bundling of smaller messages into bigger ones for unicast messages. Default is false
external_addr	Use "external_addr" if you have hosts on different networks, behind firewalls. On each firewall, set up a port forwarding rule (sometimes called "virtual server") to the local IP (e.g. 192.168.1.100) of the host then on each host, set "external_addr" TCP transport parameter to the external (public IP) address of the firewall.
external_port	Used to map the internal port (bind_port) to an external port. Only used if > 0
log_discard_msgs	whether or not warnings about messages from different groups are logged

Name	Description
log_discard_msgs_version	whether or not warnings about messages from members with a different version are discarded
logical_addr_cache_expiration	Time (in ms) after which entries in the logical address cache marked as removable are removed
logical_addr_cache_max_size	Max number of elements in the logical address cache before eviction starts
loopback	Messages to self are looped back immediately if true
max_bundle_size	Maximum number of bytes for messages to be queued until they are sent
max_bundle_timeout	Max number of milliseconds until queued messages are sent
oob_thread_pool.keep_alive_time	Timeout in ms to remove idle threads from the OOB pool
oob_thread_pool.max_threads	Max thread pool size for the OOB thread pool
oob_thread_pool.min_threads	Minimum thread pool size for the OOB thread pool
oob_thread_pool_enabled	Switch for enabling thread pool for OOB messages. Default=true
oob_thread_pool_queue_enabled	Use queue to enqueue incoming OOB messages
oob_thread_pool_queue_max_size	Maximum queue size for incoming OOB messages. Default is 500
oob_thread_pool_rejection_policy	Thread rejection policy. Possible values are Abort, Discard, DiscardOldest and Run. Default is Discard
physical_addr_max_fetch_attempts	Max number of attempts to fetch a physical address (when not in the cache) before giving up
port_range	The range of valid ports, from bind_port to end_port. 0 only binds to bind_port and fails if taken
receive_interfaces	Comma delimited list of interfaces (IP addresses or interface names) to receive multicasts on
receive_on_all_interfaces	If true, the transport should use all available interfaces to receive multicast messages

Name	Description
singleton_name	If assigned enable this transport to be a singleton (shared) transport
suppress_time_different_cluster_warnings	Time during which identical warnings about messages from a member from a different cluster will be suppressed. 0 disables this (every warning will be logged). Setting the log level to ERROR also disables this.
suppress_time_different_version_warnings	Time during which identical warnings about messages from a member with a different version will be suppressed. 0 disables this (every warning will be logged). Setting the log level to ERROR also disables this.
thread_naming_pattern	Thread naming pattern for threads in this channel. Valid values are "pcl": "p": includes the thread name, e.g. "Incoming thread-1", "UDP ucast receiver", "c": includes the cluster name, e.g. "MyCluster", "l": includes the local address of the current member, e.g. "192.168.5.1:5678"
thread_pool.keep_alive_time	Timeout in milliseconds to remove idle thread from regular pool
thread_pool.max_threads	Maximum thread pool size for the regular thread pool
thread_pool.min_threads	Minimum thread pool size for the regular thread pool
thread_pool_enabled	Switch for enabling thread pool for regular messages. Default true
thread_pool_queue_enabled	Use queue to enqueue incoming regular messages. Default is true
thread_pool_queue_max_size	Maximum queue size for incoming OOB messages. Default is 500
thread_pool_rejection_policy	Thread rejection policy. Possible values are Abort, Discard, DiscardOldest and Run
tick_time	Tick duration in the HashedTimingWheel timer. Only applicable if timer_type is "wheel"
timer.keep_alive_time	Timeout in ms to remove idle threads from the timer pool
timer.max_threads	Max thread pool size for the timer thread pool

Name	Description
timer.min_threads	Minimum thread pool size for the timer thread pool
timer_queue_max_size	Max number of elements on a timer queue
timer_rejection_policy	Timer rejection policy. Possible values are Abort, Discard, DiscardOldest and Run
timer_type	Type of timer to be used. Valid values are "old" (DefaultTimeScheduler, used up to 2.10), "new" or "new2" (TimeScheduler2), "new3" (TimeScheduler3) and "wheel". Note that this property might disappear in future releases, if one of the 3 timers is chosen as default timer
wheel_size	Number of ticks in the HashedTimingWheel timer. Only applicable if timer_type is "wheel"
who_has_cache_timeout	Timeout (in ms) to determine how long to wait until a request to fetch the physical address for a given logical address will be sent again. Subsequent requests for the same physical address will therefore be spaced at least who_has_cache_timeout ms apart

`bind_addr` can be set to the address of a network interface, e.g. `192.168.1.5`. It can also be set for the entire stack using system property `-Djgroups.bind_addr`, which overrrides the XML value (if given).

The following special values are also recognized for `bind_addr`:

GLOBAL
: Picks a global IP address if available. If not, falls back to a SITE_LOCAL IP address.

SITE_LOCAL
: Picks a site local (non routable) IP address, e.g. from the `192.168.0.0` or `10.0.0.0` address range.

LINK_LOCAL
: Picks a link-local IP address, from `169.254.1.0` through `169.254.254.255`.

NON_LOOPBACK
: Picks *any* non loopback address.

LOOPBACK
: Pick a loopback address, e.g. `127.0.0.1`.

An example of setting the bind address in UDP to use a site local address is:

```
<UDP bind_addr="SITE_LOCAL" />
```

This will pick any address of any interface that's site-local, e.g. a `192.168.x.x` or `10.x.x.x` address.

7.2.1. UDP

UDP uses IP multicast for sending messages to all members of a group and UDP datagrams for unicast messages (sent to a single member). When started, it opens a unicast and multicast socket: the unicast socket is used to send/receive unicast messages, whereas the multicast socket sends and receives multicast messages. The channel's physical address will be the address and port number of the unicast socket.

A protocol stack with UDP as transport protocol is typically used with clusters whose members run in the same subnet. If running across subnets, an admin has to ensure that IP multicast is enabled across subnets. It is often the case that IP multicast is not enabled across subnets. In such cases, the stack has to either use UDP without IP multicasting or other transports such as TCP.

Table 7.3. Properties

Name	Description
disable_loopback	If true, disables IP_MULTICAST_LOOP on the MulticastSocket (for sending and receiving of multicast packets). IP multicast packets send on a host P will therefore not be received by anyone on P. Use with caution.
ip_mcast	Multicast toggle. If false multiple unicast datagrams are sent instead of one multicast. Default is true
ip_ttl	The time-to-live (TTL) for multicast datagram packets. Default is 8
max_bundle_size	Maximum number of bytes for messages to be queued until they are sent
mcast_group_addr	The multicast address used for sending and receiving packets. Default is 228.8.8.8
mcast_port	The multicast port used for sending and receiving packets. Default is 7600
mcast_recv_buf_size	Receive buffer size of the multicast datagram socket. Default is 500'000 bytes

Name	Description
mcast_send_buf_size	Send buffer size of the multicast datagram socket. Default is 100'000 bytes
tos	Traffic class for sending unicast and multicast datagrams. Default is 8
ucast_recv_buf_size	Receive buffer size of the unicast datagram socket. Default is 64'000 bytes
ucast_send_buf_size	Send buffer size of the unicast datagram socket. Default is 100'000 bytes

7.2.2. TCP

Specifying TCP in your protocol stack tells JGroups to use TCP to send messages between cluster members. Instead of using a multicast bus, the cluster members create a mesh of TCP connections.

For example, while UDP sends 1 IP multicast packet when sending a message to a cluster of 10 members, TCP needs to send the message 9 times. It sends the same message to the first member, to the second member, and so on (excluding itself as the message is looped back internally).

This is slow, as the cost of sending a group message is O(n) with TCP, where it is O(1) with UDP. As the cost of sending a group message with TCP is a function of the cluster size, it becomes higher with larger clusters.

Note

We recommend to use UDP for larger clusters, whenever possible

Table 7.4. Properties

Name	Description
client_bind_addr	The address of a local network interface which should be used by client sockets to bind to. The following special values are also recognized: GLOBAL, SITE_LOCAL, LINK_LOCAL and NON_LOOPBACK
client_bind_port	The local port a client socket should bind to. If 0, an ephemeral port will be picked.
conn_expire_time	Max time connection can be idle before being reaped (in ms)
defer_client_bind_addr	If true, client sockets will not explicitly bind to bind_addr but will defer to the native socket

Name	Description
linger	SO_LINGER in msec. Default of -1 disables it
peer_addr_read_timeout	Max time to block on reading of peer address
reaper_interval	Reaper interval in msec. Default is 0 (no reaping)
recv_buf_size	Receiver buffer size in bytes
send_buf_size	Send buffer size in bytes
send_queue_size	Max number of messages in a send queue
sock_conn_timeout	Max time allowed for a socket creation in connection table
tcp_nodelay	Should TCP no delay flag be turned on
use_send_queues	Should separate send queues be used for each connection

7.2.3. TUNNEL

TUNNEL was described in *Section 5.3.4, "TUNNEL"*.

Table 7.5. Properties (experimental)

Name	Description
gossip_router_hosts	A comma-separated list of GossipRouter hosts, e.g. HostA[12001],HostB[12001]
reconnect_interval	Interval in msec to attempt connecting back to router in case of torn connection. Default is 5000 msec
tcp_nodelay	Should TCP no delay flag be turned on

7.3. Initial membership discovery

The task of the discovery is to find an initial membership, which is used to determine the current coordinator. Once a coordinator is found, the joiner sends a JOIN request to the coord.

Discovery is also called periodically by MERGE2 (see *Section 7.4.1, "MERGE2"*), to see if we have diverging cluster membership information.

7.3.1. Discovery

`Discovery` is the superclass for all discovery protocols and therefore its properties below can be used in any subclass.

Discovery sends a discovery request, and waits for `num_initial_members` discovery responses, or `timeout` ms, whichever occurs first, before returning. Note that `break_on_coord_rsp="true"` will return as soon as we have a response from a coordinator.

Table 7.6. Properties

Name	Description
break_on_coord_rsp	Return from the discovery phase as soon as we have 1 coordinator response
force_sending_discovery_rsps	Always sends a discovery response, no matter what
num_initial_members	Minimum number of initial members to get a response from
num_initial_srv_members	Minimum number of server responses (PingData.isServer()=true). If this value is greater than 0, we'll ignore num_initial_members
return_entire_cache	Whether or not to return the entire logical-physical address cache mappings on a discovery request, or not.
stagger_timeout	If greater than 0, we'll wait a random number of milliseconds in range [0..stagger_timeout] before sending a discovery response. This prevents traffic spikes in large clusters when everyone sends their discovery response at the same time
timeout	Timeout to wait for the initial members
use_disk_cache	If a persistent disk cache (PDC) is present, combine the discovery results with the contents of the disk cache before returning the results

7.3.2. PING

Initial (dirty) discovery of members. Used to detect the coordinator (oldest member), by mcasting PING requests to an IP multicast address.

Each member responds with a packet {C, A}, where C=coordinator's address and A=own address. After N milliseconds or M replies, the joiner determines the coordinator from the responses, and sends a JOIN request to it (handled by GMS). If nobody responds, we assume we are the first member of a group.

Unlike TCPPING, PING employs dynamic discovery, meaning that the member does not have to know in advance where other cluster members are.

`PING` uses the IP multicasting capabilities of the transport to send a discovery request to the cluster. It therefore requires UDP as transport.

7.3.3. TCPPING

TCPPING is used with TCP as transport, and uses a static list of cluster members's addresses. See *Section 5.3.3.1, "Using TCP and TCPPING"* for details.

Table 7.7. Properties

Name	Description
initial_hosts	Comma delimited list of hosts to be contacted for initial membership
max_dynamic_hosts	max number of hosts to keep beyond the ones in initial_hosts
port_range	Number of additional ports to be probed for membership. A port_range of 0 does not probe additional ports. Example: initial_hosts=A[7800] port_range=0 probes A:7800, port_range=1 probes A:7800 and A:7801

Note

It is recommended to include the addresses of *all* cluster members in `initial_hosts`.

7.3.4. TCPGOSSIP

TCPGOSSIP uses an external GossipRouter to discover the members of a cluster. See *Section 5.3.3.2, "Using TCP and TCPGOSSIP"* for details.

Table 7.8. Properties

Name	Description
initial_hosts	Comma delimited list of hosts to be contacted for initial membership
reconnect_interval	Interval (ms) by which a disconnected stub attempts to reconnect to the GossipRouter
sock_conn_timeout	Max time for socket creation. Default is 1000 msec
sock_read_timeout	Max time in milliseconds to block on a read. 0 blocks forever

7.3.5. MPING

MPING (=Multicast PING) uses IP multicast to discover the initial membership. It can be used with all transports, but usually is used in combination with TCP. TCP usually requires TCPPING, which has to list all cluster members explicitly, but MPING doesn't have this requirement. The typical use case for this is when we want TCP as transport, but multicasting for discovery so we don't have to define a static list of initial hosts in TCPPING

MPING uses its own multicast socket for discovery. Properties `bind_addr` (can also be set via `-Djgroups.bind_addr=`), `mcast_addr` and `mcast_port` can be used to configure it.

Note that MPING requires a separate thread listening on the multicast socket for discovery requests.

Table 7.9. Properties

Name	Description
bind_addr	Bind address for multicast socket. The following special values are also recognized: GLOBAL, SITE_LOCAL, LINK_LOCAL and NON_LOOPBACK
bind_interface_str	The interface (NIC) which should be used by this transport
ip_ttl	Time to live for discovery packets. Default is 8
mcast_addr	Multicast address to be used for discovery
mcast_port	Multicast port for discovery packets. Default is 7555
receive_interfaces	List of interfaces to receive multicasts on
receive_on_all_interfaces	If true, the transport should use all available interfaces to receive multicast messages. Default is false
send_interfaces	List of interfaces to send multicasts on
send_on_all_interfaces	Whether send messages are sent on all interfaces. Default is false

7.3.6. FILE_PING

This uses a shared directory into which all members write their addresses. New joiners read all addresses from this directory (which needs to be shared, e.g. via NFS or SMB) and ping each of the elements of the resulting set of members. When a member leaves, it deletes its corresponding file.

FILE_PING can be used instead of GossipRouter in cases where no external process is desired.

7.3.7. JDBC_PING

This uses a shared Database into which all members write their addresses. New joiners read all addresses from this Database and ping each of the elements of the resulting set of members. When a member leaves, it deletes its corresponding record.

JDBC_PING is an alternative to S3_PING by using Amazon RDS instead of S3.

Table 7.10. Properties

Name	Description
connection_driver	The JDBC connection driver name
connection_password	The JDBC connection password
connection_url	The JDBC connection URL
connection_username	The JDBC connection username
datasource_jndi_name	To use a DataSource registered in JNDI, specify the JNDI name here. This is an alternative to all connection_* configuration options: if this property is not empty, then all connection relatedproperties must be empty.
delete_single_sql	SQL used to delete a row. Customizable, but keep the order of parameters and pick compatible types: 1)Own Address, as String 2)Cluster name, as String
initialize_sql	If not empty, this SQL statement will be performed at startup.Customize it to create the needed table on those databases which permit table creation attempt without loosing data, such as PostgreSQL and MySQL (using IF NOT EXISTS). To allow for creation attempts, errors performing this statement will be loggedbut not considered fatal. To avoid any DDL operation, set this to an empty string.
insert_single_sql	SQL used to insert a new row. Customizable, but keep the order of parameters and pick compatible types: 1)Own Address, as String 2)Cluster name, as String 3)Serialized PingData as byte[]
select_all_pingdata_sql	SQL used to fetch all node's PingData. Customizable, but keep the order of parameters and pick compatible types: only one parameter needed, String compatible, representing the Cluster name. Must return a

Name	Description
	byte[], the Serialized PingData as it was stored by the insert_single_sql statement

7.3.8. BPING

BPING uses UDP broadcasts to discover other nodes. The default broadcast address (dest) is 255.255.255.255, and should be replaced with a subnet specific broadcast, e.g. 192.168.1.255.

Table 7.11. Properties (experimental)

Name	Description
bind_port	Port for discovery packets
dest	Target address for broadcasts. This should be restricted to the local subnet, e.g. 192.168.1.255
port_range	Sends discovery packets to ports 8555 to (8555+port_range)

7.3.9. RACKSPACE_PING

RACKSPACE_PING uses Rackspace Cloud Files Storage to discover initial members. Each node writes a small object in a shared Rackspace container. New joiners read all addresses from the container and ping each of the elements of the resulting set of members. When a member leaves, it deletes its corresponding object.

This objects are stored under a container called 'jgroups', and each node will write an object name after the cluster name, plus a "/" followed by the address, thus simulating a hierarchical structure.

Table 7.12. Properties

Name	Description
apiKey	Rackspace API access key
container	Name of the root container
region	Rackspace region, either UK or US
username	Rackspace username

7.3.10. S3_PING

S3_PING uses Amazon S3 to discover initial members. New joiners read all addresses from this bucket and ping each of the elements of the resulting set of members. When a member leaves, it deletes its corresponding file.

It's designed specifically for members running on Amazon EC2, where multicast traffic is not allowed and thus MPING or PING will not work. When Amazon RDS is preferred over S3, or if a shared database is used, an alternative is to use JDBC_PING.

Each instance uploads a small file to an S3 bucket and each instance reads the files out of this bucket to determine the other members.

There are three different ways to use S3_PING, each having its own tradeoffs between security and ease-of-use. These are described in more detail below:

- Private buckets, Amazon AWS credentials given to each instance
- Public readable and writable buckets, no credentials given to each instance
- Public readable but private writable buckets, pre-signed URLs given to each instance

Pre-signed URLs are the most secure method since writing to buckets still requires authorization and you don't have to pass Amazon AWS credentials to every instance. However, they are also the most complex to setup.

Here's a configuration example for private buckets with credentials given to each instance:

```
<S3_PING location="my_bucket" access_key="access_key"
         secret_access_key="secret_access_key" timeout="2000"
         num_initial_members="3"/>
```

Here's an example for public buckets with no credentials:

```
<S3_PING location="my_bucket"
         timeout="2000" num_initial_members="3"/>
```

And finally, here's an example for public readable buckets with pre-signed URLs:

```
<S3_PING     pre_signed_put_url="http://s3.amazonaws.com/my_bucket/DemoCluster/
node1?
AWSAccessKeyId=access_key&Expires=1316276200&Signature=it1cUUtgCT9ZJyCJDj2xTAcRTFg
%3D"
          pre_signed_delete_url="http://s3.amazonaws.com/my_bucket/DemoCluster/
node1?AWSAccessKeyId=access_key&Expires=1316276200&Signature=u4IFPRq
%2FL6%2FAohykIW4QrKjR23g%3D"
         timeout="2000" num_initial_members="3"/>
```

Table 7.13. Properties

Name	Description
access_key	The access key to AWS (S3)
pre_signed_delete_url	When non-null, we use this pre-signed URL for DELETEs
pre_signed_put_url	When non-null, we use this pre-signed URL for PUTs
prefix	When non-null, we set location to prefix-UUID
secret_access_key	The secret access key to AWS (S3)

7.3.11. SWIFT_PING

SWIFT_PING uses Openstack Swift to discover initial members. Each node writes a small object in a shared container. New joiners read all addresses from the container and ping each of the elements of the resulting set of members. When a member leaves, it deletes its corresponding object.

These objects are stored under a container called 'jgroups' (by default), and each node will write an object name after the cluster name, plus a "/" followed by the address, thus simulating a hierarchical structure.

Currently only Openstack Keystone authentication is supported. Here is a sample configuration block:

```
<SWIFT_PING timeout="2000"
    num_initial_members="3"
    auth_type="keystone_v_2_0"
    auth_url="http://localhost:5000/v2.0/tokens"
    username="demo"
    password="password"
    tenant="demo" />
```

Table 7.14. Properties (experimental)

Name	Description
auth_type	Authentication type
auth_url	Authentication url
container	Name of the root container
password	Password
tenant	Openstack Keystone tenant name

Name	Description
username	Username

7.3.12. AWS_PING

This is a protocol written by Meltmedia, which uses the AWS API. It is not part of JGroups, but can be downloaded at *https://metmedia.github.com/jgroups-aws*.

7.3.13. PDC - Persistent Discovery Cache

The Persistent Discovery Cache can be used to cache the results of the discovery process persistently. E.g. if we have TCPPING.initial_hosts configured to include only members A and B, but have a lot more members, then other members can bootstrap themselves and find the right coordinator even when neither A nor B are running.

An example of a TCP-based stack configuration is:

```
<TCP />
<PDC cache_dir="/tmp/jgroups"  />
<TCPPING timeout="2000" num_initial_members="20"
         initial_hosts="192.168.1.5[7000]" port_range="0"
         return_entire_cache="true"
         use_disk_cache="true" />
```

Table 7.15. Properties

Name	Description
cache_dir	The absolute path of the directory for the disk cache. The mappings will be stored as individual files in this directory

7.4. Merging after a network partition

7.4.1. MERGE2

If a cluster gets split for some reasons (e.g. network partition), this protocol merges the subclusters back into one cluster. It is only run by the coordinator (the oldest member in a cluster), which periodically multicasts its presence and view information. If another coordinator (for the same cluster) receives this message, it will initiate a merge process. Note that this merges subgroups `{A,B}` and `{C,D,E}` back into `{A,B,C,D,E}`, but it does *not merge state*. The application has to handle the callback to merge state. See *Section 5.6, "Handling network partitions"* for suggestion on merging states.

Following a merge, the coordinator of the merged group can shift from the typical case of "the coordinator is the member who has been up the longest." During the merge process, the coordinators of the various subgroups need to reach a common decision as to who the new coordinator is. In order to ensure a consistent result, each coordinator combines the addresses of all the members in a list and then sorts the list. The first member in the sorted list becomes the coordinator. The sort order is determined by how the address implements the interface. Then JGroups compares based on the UUID. So, take a hypothetical case where two machines were running, with one machine running three separate cluster members and the other two members. If communication between the machines were cut, the following subgroups would form: `{A,B}` and `{C,D,E}` Following the merge, the new view would be: `{C,D,A,B,E}`, with C being the new coordinator.

Note that "A", "B" and so on are just logical names, attached to UUIDs, but the actual sorting is done on the actual UUIDs.

Table 7.16. Properties

Name	Description
inconsistent_view_threshold	Number of inconsistent views with only 1 coord after a MERGE event is sent up
max_interval	Maximum time in ms between runs to discover other clusters
merge_fast	When receiving a multicast message, checks if the sender is member of the cluster. If not, initiates a merge. Generates a lot of traffic for large clusters when there is a lot of merging
merge_fast_delay	The delay (in milliseconds) after which a merge fast execution is started
min_interval	Minimum time in ms between runs to discover other clusters

7.4.2. MERGE3

MERGE3 was added in JGroups 3.1.

In MERGE3, all members periodically send an INFO message with their address (UUID), logical name, physical address and ViewId. The ViewId (*Section 3.7.1, "ViewId"*) is used to see if we have diverging views among the cluster members: periodically, every coordinator looks at the INFO messages received so far and checks if there are any inconsistencies.

When inconsistencies are found, the merge leader will be the member with the lowest address (UUID). This is deterministic, and therefore we should at most times only have 1 merge going on.

The merge leader then asks the senders of the inconsistent ViewIds for their full Views. Once received, it simply passes a MERGE event up the stack, where the merge will be handled (by GMS) in exactly the same way as if MERGE2 has generated the MERGE event.

The advantages of MERGE3 compared to MERGE2 are:

- Sending of INFO messages is spread out over time, preventing messgage peaks which might cause packet loss. This is especially important in large clusters.
- Only 1 merge should be running at any time. Competing merges, as happening with MERGE2, slow down the merge process, and don't scale to large clusters.
- An INFO message carries the logical name and physical address of a member. Compared to MERGE2, this allows us to immediately send messages to newly merged members, and not have to solicit this information first.
- On the downside, MERGE3 has constant (small) traffic by all members.
- MERGE3 was written for an IP multicast capable transport (UDP), but it also works with other transports (such as TCP), although it isn't as efficient on TCP as on UDP.

Table 7.17. Properties

Name	Description
max_interval	Interval (in milliseconds) when the next info message will be sent. A random value is picked from range [1..max_interval]
max_participants_in_merge	The max number of merge participants to be involved in a merge. 0 sets this to unlimited.
min_interval	Minimum time in ms before sending an info message

7.5. Failure Detection

The task of failure detection is to probe members of a group and see whether they are alive. When a member is suspected (= deemed dead), then a SUSPECT message is sent to all nodes of the cluster. It is not the task of the failure detection layer to exclude a crashed member (this is done by the group membership protocol, GMS), but simply to notify everyone that a node in the cluster is suspected of having crashed.

The SUSPECT message is handled by the GMS protocol of the current coordinator only; all other members ignore it.

7.5.1. FD

Failure detection based on heartbeat messages. If reply is not received without `timeout` ms, `max_tries` times, a member is declared suspected, and will be excluded by GMS

Each member send a message containing a "FD" - HEARTBEAT header to its neighbor to the right (identified by the ping_dest address). The heartbeats are sent by the inner class Monitor.

When the neighbor receives the HEARTBEAT, it replies with a message containing a "FD" - HEARTBEAT_ACK header. The first member watches for "FD" - HEARTBEAT_ACK replies from its neigbor. For each received reply, it resets the last_ack timestamp (sets it to current time) and num_tries counter (sets it to 0). The same Monitor instance that sends heartbeats whatches the difference between current time and last_ack. If this difference grows over timeout, the Monitor cycles several more times (until max_tries) is reached) and then sends a SUSPECT message for the neighbor's address. The SUSPECT message is sent down the stack, is addressed to all members, and is as a regular message with a FdHeader.SUSPECT header.

Table 7.18. Properties

Name	Description
max_tries	Number of times to send an are-you-alive message
msg_counts_as_heartbeat	Treat messages received from members as heartbeats. Note that this means we're updating a value in a hashmap every time a message is passing up the stack through FD, which is costly.
timeout	Timeout to suspect a node P if neither a heartbeat nor data were received from P.

7.5.2. FD_ALL

Failure detection based on simple heartbeat protocol. Every member periodically multicasts a heartbeat. Every member also maintains a table of all members (minus itself). When data or a heartbeat from P are received, we reset the timestamp for P to the current time. Periodically, we check for expired members, and suspect those.

Example: <FD_ALL interval="3000" timeout="10000"/>

In the example above, we send a heartbeat every 3 seconds and suspect members if we haven't received a heartbeat (or traffic) for more than 10 seconds. Note that since we check the timestamps every 'interval' milliseconds, we will suspect a member after roughly 4 * 3s == 12 seconds. If we set the timeout to 8500, then we would suspect a member after 3 * 3 secs == 9 seconds.

Table 7.19. Properties

Name	Description
interval	Interval at which a HEARTBEAT is sent to the cluster
msg_counts_as_heartbeat	Treat messages received from members as heartbeats. Note that this means we're

Name	Description
	updating a value in a hashmap every time a message is passing up the stack through FD_ALL, which is costly. Default is false
timeout	Timeout after which a node P is suspected if neither a heartbeat nor data were received from P
timeout_check_interval	Interval at which the HEARTBEAT timeouts are checked

7.5.3. FD_SOCK

Failure detection protocol based on a ring of TCP sockets created between cluster members. Each member in a cluster connects to its neighbor (the last member connects to the first), thus forming a ring. Member B is suspected when its neighbor A detects abnormally closing of its TCP socket (presumably due to a node B crash). However, if a member B is about to leave gracefully, it lets its neighbor A know, so that it does not become suspected.

If you are using a multi NIC machine note that JGroups versions prior to 2.2.8 have FD_SOCK implementation that does not assume this possibility. Therefore JVM can possibly select NIC unreachable to its neighbor and setup FD_SOCK server socket on it. Neighbor would be unable to connect to that server socket thus resulting in immediate suspecting of a member. Suspected member is kicked out of the group, tries to rejoin, and thus goes into join/leave loop. JGroups version 2.2.8 introduces srv_sock_bind_addr property so you can specify network interface where FD_SOCK TCP server socket should be bound. This network interface is most likely the same interface used for other JGroups traffic. JGroups versions 2.2.9 and newer consult bind.address system property or you can specify network interface directly as FD_SOCK bind_addr property.

Table 7.20. Properties

Name	Description
bind_addr	The NIC on which the ServerSocket should listen on. The following special values are also recognized: GLOBAL, SITE_LOCAL, LINK_LOCAL and NON_LOOPBACK
bind_interface_str	The interface (NIC) which should be used by this transport
client_bind_port	Start port for client socket. Default value of 0 picks a random port
external_addr	Use "external_addr" if you have hosts on different networks, behind firewalls. On each firewall, set up a port forwarding rule (sometimes called "virtual server") to the local IP (e.g. 192.168.1.100) of the host then on

Name	Description
	each host, set "external_addr" TCP transport parameter to the external (public IP) address of the firewall.
external_port	Used to map the internal port (bind_port) to an external port. Only used if > 0
get_cache_timeout	Timeout for getting socket cache from coordinator. Default is 1000 msec
keep_alive	Whether to use KEEP_ALIVE on the ping socket or not. Default is true
num_tries	Number of attempts coordinator is solicited for socket cache until we give up. Default is 3
port_range	Number of ports to probe for start_port and client_bind_port
sock_conn_timeout	Max time in millis to wait for ping Socket.connect() to return
start_port	Start port for server socket. Default value of 0 picks a random port
suspect_msg_interval	Interval for broadcasting suspect messages. Default is 5000 msec

7.5.4. FD_PING

FD_PING uses a script or command that is run with 1 argument (the host to be pinged) and needs to return 0 (success) or 1 (failure). The default command is /sbin/ping (ping.exe on Windows), but this is user configurable and can be replaced with any user-provided script or executable.

7.5.5. FD_ICMP

Uses InetAddress.isReachable() to determine whether a host is up or not. Note that this is only available in JDK 5, so reflection is used to determine whether InetAddress provides such a method. If not, an exception will be thrown at protocol initialization time.

The problem with InetAddress.isReachable() is that it may or may not use ICMP in its implementation ! For example, an implementation might try to establish a TCP connection to port 9 (echo service), and - if the echo service is not running - the host would be suspected, although a real ICMP packet would *not* have suspected the host ! Please check your JDK/OS combo before running this protocol.

Table 7.21. Properties (experimental)

Name	Description
bind_addr	The NIC on which the ServerSocket should listen on. The following special values are

Name	Description
	also recognized: GLOBAL, SITE_LOCAL, LINK_LOCAL and NON_LOOPBACK
bind_interface_str	The interface (NIC) which should be used by this transport
ttl	

7.5.6. VERIFY_SUSPECT

Verifies that a suspected member is really dead by pinging that member one last time before excluding it, and dropping the suspect message if the member does respond.

VERIFY_SUSPECT tries to minimize false suspicions.

The protocol works as follows: it catches SUSPECT events traveling up the stack. The it verifies that the suspected member is really dead. If yes, it passes the SUSPECT event up the stack, otherwise it discards it. VERIFY_SUSPECT Has to be placed somewhere above the failure detection protocol and below the GMS protocol (receiver of the SUSPECT event). Note that SUSPECT events may be reordered by this protocol.

Table 7.22. Properties

Name	Description
bind_addr	Interface for ICMP pings. Used if use_icmp is true The following special values are also recognized: GLOBAL, SITE_LOCAL, LINK_LOCAL and NON_LOOPBACK
bind_interface_str	The interface (NIC) which should be used by this transport
num_msgs	Number of verify heartbeats sent to a suspected member
timeout	Number of millisecs to wait for a response from a suspected member
use_icmp	Use InetAddress.isReachable() to verify suspected member instead of regular messages

7.6. Reliable message transmission

7.6.1. pbcast.NAKACK

NAKACK provides reliable delivery and FIFO (= First In First Out) properties for messages sent to all nodes in a cluster.

Reliable delivery means that no message sent by a sender will ever be lost, as all messages are numbered with sequence numbers (by sender) and retransmission requests are sent to the sender of a message[1] if that sequence number is not received.

FIFO order means that all messages from a given sender are received in exactly the order in which they were sent.

NAKACK is a Lossless and FIFO delivery of multicast messages, using negative acks. E.g. when receiving P:1, P:3, P:4, a receiver delivers only P:1, and asks P for retransmission of message 2, queuing P3-4. When P2 is finally received, the receiver will deliver P2-4 to the application.

Table 7.23. Properties

Name	Description
become_server_queue_size	Size of the queue to hold messages received after creating the channel, but before being connected (is_server=false). After becoming the server, the messages in the queue are fed into up() and the queue is cleared. The motivation is to avoid retransmissions (see https://issues.jboss.org/browse/JGRP-1509 for details). 0 disables the queue.
discard_delivered_msgs	Should messages delivered to application be discarded
exponential_backoff	The first value (in milliseconds) to use in the exponential backoff. Enabled if greater than 0
log_discard_msgs	discards warnings about promiscuous traffic
log_not_found_msgs	If true, trashes warnings about retransmission messages not found in the xmit_table (used for testing)
max_msg_batch_size	Max number of messages to be removed from a NakReceiverWindow. This property might get removed anytime, so don't use it !
max_rebroadcast_timeout	Timeout to rebroadcast messages. Default is 2000 msec
print_stability_history_on_failed_xmit	Should stability history be printed if we fail in retransmission. Default is false
retransmit_timeouts	Timeout before requesting retransmissions
suppress_time_non_member_warnings	Time during which identical warnings about messages from a non member will be

[1] Note that NAKACK can also be configured to send retransmission requests for M to *anyone* in the cluster, rather than only to the sender of M.

Name	Description
	suppressed. 0 disables this (every warning will be logged). Setting the log level to ERROR also disables this.
use_mcast_xmit	Retransmit retransmit responses (messages) using multicast rather than unicast
use_mcast_xmit_req	Use a multicast to request retransmission of missing messages
use_range_based_retransmitter	Whether to use the old retransmitter which retransmits individual messages or the new one which uses ranges of retransmitted messages. Default is true. Note that this property will be removed in 3.0; it is only used to switch back to the old (and proven) retransmitter mechanism if issues occur
xmit_from_random_member	Ask a random member for retransmission of a missing message. Default is false
xmit_stagger_timeout	Number of milliseconds to delay the sending of an XMIT request. We pick a random number in the range [1 .. xmit_req_stagger_timeout] and add this to the scheduling time of an XMIT request. When use_mcast_xmit is enabled, if a number of members drop messages from the same member, then chances are that, if staggering is enabled, somebody else already sent the XMIT request (via mcast) and we can cancel the XMIT request once we receive the missing messages. For unicast XMIT responses (use_mcast_xmit=false), we still have an advantage by not overwhelming the receiver with XMIT requests, all at the same time. 0 disabless staggering.
xmit_table_max_compaction_time	Number of milliseconds after which the matrix in the retransmission table is compacted (only for experts)
xmit_table_msgs_per_row	Number of elements of a row of the matrix in the retransmission table (only for experts). The capacity of the matrix is xmit_table_num_rows * xmit_table_msgs_per_row
xmit_table_num_rows	Number of rows of the matrix in the retransmission table (only for experts)

Name	Description
xmit_table_resize_factor	Resize factor of the matrix in the retransmission table (only for experts)

7.6.2. NAKACK2

NAKACK2 was introduced in 3.1 and is a successor to NAKACK (at some point it will replace NAKACK). It has the same properties as NAKACK, but its implementation is faster and uses less memory, plus it creates fewer tasks in the timer.

Some of the properties of NAKACK were deprecated in NAKACK2, but were not removed so people can simply change from NAKACK to NAKACK2 by changing the protocol name in the config.

Table 7.24. Properties

Name	Description
become_server_queue_size	Size of the queue to hold messages received after creating the channel, but before being connected (is_server=false). After becoming the server, the messages in the queue are fed into up() and the queue is cleared. The motivation is to avoid retransmissions (see https://issues.jboss.org/browse/JGRP-1509 for details). 0 disables the queue.
discard_delivered_msgs	Should messages delivered to application be discarded
log_discard_msgs	discards warnings about promiscuous traffic
log_not_found_msgs	If true, trashes warnings about retransmission messages not found in the xmit_table (used for testing)
max_msg_batch_size	Max number of messages to be removed from a RingBuffer. This property might get removed anytime, so don't use it !
max_rebroadcast_timeout	Timeout to rebroadcast messages. Default is 2000 msec
print_stability_history_on_failed_xmit	Should stability history be printed if we fail in retransmission. Default is false
suppress_time_non_member_warnings	Time during which identical warnings about messages from a non member will be suppressed. 0 disables this (every warning will

Name	Description
	be logged). Setting the log level to ERROR also disables this.
use_mcast_xmit	Retransmit retransmit responses (messages) using multicast rather than unicast
use_mcast_xmit_req	Use a multicast to request retransmission of missing messages
xmit_from_random_member	Ask a random member for retransmission of a missing message. Default is false
xmit_interval	Interval (in milliseconds) at which missing messages (from all retransmit buffers) are retransmitted
xmit_table_max_compaction_time	Number of milliseconds after which the matrix in the retransmission table is compacted (only for experts)
xmit_table_msgs_per_row	Number of elements of a row of the matrix in the retransmission table (only for experts). The capacity of the matrix is xmit_table_num_rows * xmit_table_msgs_per_row
xmit_table_num_rows	Number of rows of the matrix in the retransmission table (only for experts)
xmit_table_resize_factor	Resize factor of the matrix in the retransmission table (only for experts)

7.6.3. UNICAST

UNICAST provides reliable delivery and FIFO (= First In First Out) properties for point-to-point messages between one sender and one receiver.

Reliable delivery means that no message sent by a sender will ever be lost, as all messages are numbered with sequence numbers (by sender) and retransmission requests are sent to the sender of a message if that sequence number is not received.

FIFO order means that all messages from a given sender are received in exactly the order in which they were sent.

UNICAST uses *positive acks* for retransmission; sender A keeps sending message M until receiver B delivers M and sends an ack(M) to A, or until B leaves the cluster or A crashes.

Although JGroups attempts to send acks selectively, UNICAST will still see a lot of acks on the wire. If this is not desired, use UNICAST2 (see *Section 7.6.4, “UNICAST2”*).

On top of a reliable transport, such as TCP, UNICAST is not really needed. However, concurrent delivery of messages from the same sender is prevented by UNICAST by acquiring a lock on the

sender's retransmission table, so unless concurrent delivery is desired, UNICAST should not be removed from the stack even if TCP is used.

Table 7.25. Properties

Name	Description
conn_expiry_timeout	Time (in milliseconds) after which an idle incoming or outgoing connection is closed. The connection will get re-established when used again. 0 disables connection reaping
max_msg_batch_size	Max number of messages to be removed from a retransmit window. This property might get removed anytime, so don't use it !
max_retransmit_time	Max number of milliseconds we try to retransmit a message to any given member. After that, the connection is removed. Any new connection to that member will start with seqno #1 again. 0 disables this
segment_capacity	Size (in bytes) of a Segment in the segments table. Only for experts, do not use !
timeout	n/a
xmit_interval	Interval (in milliseconds) at which messages in the send windows are resent
xmit_table_max_compaction_time	Number of milliseconds after which the matrix in the retransmission table is compacted (only for experts)
xmit_table_msgs_per_row	Number of elements of a row of the matrix in the retransmission table (only for experts). The capacity of the matrix is xmit_table_num_rows * xmit_table_msgs_per_row
xmit_table_num_rows	Number of rows of the matrix in the retransmission table (only for experts)
xmit_table_resize_factor	Resize factor of the matrix in the retransmission table (only for experts)

7.6.4. UNICAST2

UNICAST2 provides lossless, ordered, communication between 2 members. Contrary to UNICAST, it uses *negative acks* (similar to NAKACK) rather than positive acks. This reduces the communication overhead required for sending an ack for every message.

Negative acks have sender A simply send messages without retransmission, and receivers never ack messages, until they detect a gap: for instance, if A sends messages 1,2,4,5, then B delivers

1 and 2, but queues 4 and 5 because it is missing message 3 from A. B then asks A to retransmit 3. When 3 is received, messages 3, 4 and 5 can be delivered to the application.

Compared to a positive ack scheme as used in UNICAST, negative acks have the advantage that they generate less traffic: if all messages are received in order, we never need to do retransmission.

Table 7.26. Properties

Name	Description
conn_expiry_timeout	Time (in milliseconds) after which an idle incoming or outgoing connection is closed. The connection will get re-established when used again. 0 disables connection reaping
exponential_backoff	The first value (in milliseconds) to use in the exponential backoff. Enabled if greater than 0
log_not_found_msgs	If true, trashes warnings about retransmission messages not found in the xmit_table (used for testing)
max_bytes	Max number of bytes before a stability message is sent to the sender
max_msg_batch_size	Max number of messages to be removed from a NakReceiverWindow. This property might get removed anytime, so don't use it !
max_retransmit_time	Max number of milliseconds we try to retransmit a message to any given member. After that, the connection is removed. Any new connection to that member will start with seqno #1 again. 0 disables this
max_stable_msgs	Max number of STABLE messages sent for the same highest_received seqno. A value < 1 is invalid
stable_interval	Max number of milliseconds before a stability message is sent to the sender(s)
timeout	list of timeouts
use_range_based_retransmitter	Whether to use the old retransmitter which retransmits individual messages or the new one which uses ranges of retransmitted messages. Default is true. Note that this property will be removed in 3.0; it is only used to switch back to the old (and proven) retransmitter mechanism if issues occur

Name	Description
xmit_interval	Interval (in milliseconds) at which missing messages (from all retransmit buffers) are retransmitted
xmit_table_automatic_purging	If enabled, the removal of a message from the retransmission table causes an automatic purge (only for experts)
xmit_table_max_compaction_time	Number of milliseconds after which the matrix in the retransmission table is compacted (only for experts)
xmit_table_msgs_per_row	Number of elements of a row of the matrix in the retransmission table (only for experts). The capacity of the matrix is xmit_table_num_rows * xmit_table_msgs_per_row
xmit_table_num_rows	Number of rows of the matrix in the retransmission table (only for experts)
xmit_table_resize_factor	Resize factor of the matrix in the retransmission table (only for experts)

7.6.5. RSVP

The RSVP protocol is not a reliable delivery protocol per se, but augments reliable protocols such as NAKACK, UNICAST or UNICAST2. It should be placed somewhere *above* these in the stack.

Table 7.27. Properties (experimental)

Name	Description
ack_on_delivery	When true, we pass the message up to the application and only then send an ack. When false, we send an ack first and only then pass the message up to the application.
resend_interval	Interval (in milliseconds) at which we resend the RSVP request. Needs to be < timeout. 0 disables it.
throw_exception_on_timeout	Whether an exception should be thrown when the timeout kicks in, and we haven't yet received all acks. An exception would be thrown all the way up to JChannel.send()
timeout	Max time in milliseconds to block for an RSVP'ed message (0 blocks forever).

7.7. Message stability

To serve potential retransmission requests, a member has to store received messages until it is known that every member in the cluster has received them. Message stability for a given message M means that M has been seen by everyone in the cluster.

The stability protocol periodically (or when a certain number of bytes have been received) initiates a consensus protocol, which multicasts a stable message containing the highest message numbers for a given member. This is called a digest.

When everyone has received everybody else's stable messages, a digest is computed which consists of the minimum sequence numbers of all received digests so far. This is the stability vector, and contain only message sequence numbers that have been seen by everyone.

This stability vector is the broadcast to the group and everyone can remove messages from their retransmission tables whose sequence numbers are smaller than the ones received in the stability vector. These messages can then be garbage collected.

7.7.1. STABLE

STABLE garbage collects messages that have been seen by all members of a cluster. Each member has to store all messages because it may be asked to retransmit. Only when we are sure that all members have seen a message can it be removed from the retransmission buffers. STABLE periodically gossips its highest and lowest messages seen. The lowest value is used to compute the min (all lowest seqnos for all members), and messages with a seqno below that min can safely be discarded.

Note that STABLE can also be configured to run when N bytes have been received. This is recommended when sending messages at a high rate, because sending stable messages based on time might accumulate messages faster than STABLE can garbage collect them.

Table 7.28. Properties

Name	Description
cap	Max percentage of the max heap (-Xmx) to be used for max_bytes. Only used if ergonomics is enabled. 0 disables setting max_bytes dynamically.
desired_avg_gossip	Average time to send a STABLE message. Default is 20000 msec
max_bytes	Maximum number of bytes received in all messages before sending a STABLE message is triggered .If ergonomics is enabled, this value is computed as max(MAX_HEAP * cap, N * max_bytes) where N = number of members

Name	Description
stability_delay	Delay before stability message is sent. Default is 6000 msec

7.8. Group Membership

Group membership takes care of joining new members, handling leave requests by existing members, and handling SUSPECT messages for crashed members, as emitted by failure detection protocols. The algorithm for joining a new member is essentially:

```
- loop
- find initial members (discovery)
- if no responses:
    - become singleton group and break out of the loop
- else:
    - determine the coordinator (oldest member) from the responses
    - send JOIN request to coordinator
    - wait for JOIN response
    - if JOIN response received:
        - install view and break out of the loop
    - else
        - sleep for 5 seconds and continue the loop
```

7.8.1. pbcast.GMS

Table 7.29. Properties

Name	Description
flushInvokerClass	
handle_concurrent_startup	Temporary switch. Default is true and should not be changed
join_timeout	Join timeout
leave_timeout	Leave timeout
log_collect_msgs	Logs failures for collecting all view acks if true
log_view_warnings	Logs warnings for reception of views less than the current, and for views which don't include self
max_bundling_time	Max view bundling timeout if view bundling is turned on. Default is 50 msec

Name	Description
max_join_attempts	Number of join attempts before we give up and become a singleton. Zero means 'never give up'.
merge_timeout	Timeout (in ms) to complete merge
num_prev_mbrs	Max number of old members to keep in history. Default is 50
num_prev_views	Number of views to store in history
print_local_addr	Print local address of this member after connect. Default is true
print_physical_addrs	Print physical address(es) on startup
resume_task_timeout	Timeout to resume ViewHandler
use_flush_if_present	Use flush for view changes. Default is true
view_ack_collection_timeout	Time in ms to wait for all VIEW acks (0 == wait forever. Default is 2000 msec
view_bundling	View bundling toggle

7.8.1.1. Joining a new member

Consider the following situation: a new member wants to join a group. The prodedure to do so is:

- Multicast an (unreliable) discovery request (ping)
- Wait for n responses or m milliseconds (whichever is first)
- Every member responds with the address of the coordinator
- If the initial responses are > 0: determine the coordinator and start the JOIN protocolg
- If the initial response are 0: become coordinator, assuming that no one else is out there

However, the problem is that the initial mcast discovery request might get lost, e.g. when multiple members start at the same time, the outgoing network buffer might overflow, and the mcast packet might get dropped. Nobody receives it and thus the sender will not receive any responses, resulting in an initial membership of 0. This could result in multiple coordinators, and multiple subgroups forming. How can we overcome this problem ? There are two solutions:

1. Increase the timeout, or number of responses received. This will only help if the reason of the empty membership was a slow host. If the mcast packet was dropped, this solution won't help
2. Add the MERGE2 or MERGE3 protocol. This doesn't actually prevent multiple initial cordinators, but rectifies the problem by merging different subgroups back into one. Note that this might involve state merging which needs to be done by the application.

7.9. Flow control

Flow control takes care of adjusting the rate of a message sender to the rate of the slowest receiver over time. If a sender continuously sends messages at a rate that is faster than the receiver(s), the receivers will either queue up messages, or the messages will get discarded by the receiver(s), triggering costly retransmissions. In addition, there is spurious traffic on the cluster, causing even more retransmissions.

Flow control throttles the sender so the receivers are not overrun with messages.

Note that flow control can be bypassed by setting message flag Message.NO_FC. See *Section 5.13, "Tagging messages with flags"* for details.

The properties for `FlowControl` are shown below and can be used in MFC and UFC:

Table 7.30. Properties

Name	Description
ignore_synchronous_response	Does not block a down message if it is a result of handling an up message in thesame thread. Fixes JGRP-928
max_block_time	Max time (in milliseconds) to block. Default is 5000 msec
max_block_times	Max times to block for the listed messages sizes (Message.getLength()). Example: "1000:10,5000:30,10000:500"
max_credits	Max number of bytes to send per receiver until an ack must be received to proceed
min_credits	Computed as max_credits x min_theshold unless explicitly set
min_threshold	The threshold (as a percentage of max_credits) at which a receiver sends more credits to a sender. Example: if max_credits is 1'000'000, and min_threshold 0.25, then we send ca. 250'000 credits to P once we've got only 250'000 credits left for P (we've received 750'000 bytes from P)

7.9.1. FC

FC uses a credit based system, where each sender has `max_credits` credits and decrements them whenever a message is sent. The sender blocks when the credits fall below 0, and only resumes sending messages when it receives a replenishment message from the receivers.

The receivers maintain a table of credits for all senders and decrement the given sender's credits as well, when a message is received.

When a sender's credits drops below a threshold, the receiver will send a replenishment message to the sender. The threshold is defined by `min_bytes` or `min_threshold`.

Table 7.31. Properties

Name	Description
ignore_synchronous_response	Does not block a down message if it is a result of handling an up message in thesame thread. Fixes JGRP-928
max_block_time	Max time (in milliseconds) to block. Default is 5000 msec
max_block_times	Max times to block for the listed messages sizes (Message.getLength()). Example: "1000:10,5000:30,10000:500"
max_credits	Max number of bytes to send per receiver until an ack must be received to proceed. Default is 500000 bytes
min_credits	Computed as max_credits x min_theshold unless explicitly set
min_threshold	The threshold (as a percentage of max_credits) at which a receiver sends more credits to a sender. Example: if max_credits is 1'000'000, and min_threshold 0.25, then we send ca. 250'000 credits to P once we've received 250'000 bytes from P

Note

FC has been deprecated, use MFC and UFC instead.

7.9.2. MFC and UFC

In 2.10, FC was separated into MFC (Multicast Flow Control) and Unicast Flow Control (UFC). The reason was that multicast flow control should not be impeded by unicast flow control, and vice versa. Also, performance for the separate implementations could be increased, plus they can be individually omitted. For example, if no unicast flow control is needed, UFC can be left out of the stack configuration.

7.9.2.1. MFC

`MFC` has currently no properties other than those inherited by `FlowControl` (see above).

7.9.2.2. UFC

`UFC` has currently no properties other than those inherited by `FlowControl` (see above).

7.10. Fragmentation

7.10.1. FRAG and FRAG2

FRAG and FRAG2 fragment large messages into smaller ones, send the smaller ones, and at the receiver side, the smaller fragments will get assembled into larger messages again, and delivered to the application. FRAG and FRAG2 work for both unicast and multicast messages.

The difference between FRAG and FRAG2 is that FRAG2 does 1 less copy than FRAG, so it is the recommended fragmentation protocol. FRAG serializes a message to know the exact size required (including headers), whereas FRAG2 only fragments the payload (excluding the headers), so it is faster.

The properties of FRAG2 are:

Table 7.32. Properties

Name	Description
frag_size	The max number of bytes in a message. Larger messages will be fragmented

Contrary to FRAG, FRAG2 does not need to serialize a message in order to break it into smaller fragments: it looks only at the message's buffer, which is a byte array anyway. We assume that the size addition for headers and src and dest addresses is minimal when the transport finally has to serialize the message, so we add a constant (by default 200 bytes). Because of the efficiency gained by not having to serialize the message just to determine its size, FRAG2 is generally recommended over FRAG.

7.11. Ordering

7.11.1. SEQUENCER

SEQUENCER provider total order for multicast (=group) messages by forwarding messages to the current coordinator, which then sends the messages to the cluster on behalf of the original sender. Because it is always the same sender (whose messages are delivered in FIFO order), a global (or total) order is established.

Sending members add every forwarded message M to a buffer and remove M when they receive it. Should the current coordinator crash, all buffered messages are forwarded to the new coordinator.

Table 7.33. Properties

Name	Description
delivery_table_max_size	Size of the set to store received seqnos (for duplicate checking)
threshold	Number of acks needed before going from ack-mode to normal mode. 0 disables this, which means that ack-mode is always on

7.11.2. Total Order Anycast (TOA)

A total order anycast is a totally ordered message sent to a subset of the cluster members. TOA intercepts messages with an AnycastMessage (carrying a list of addresses) and handles sending of the message in total order. Say the cluster is {A,B,C,D,E} and the Anycast is to {B,C}.

Skeen's algorithm is used to send the message: B and C each maintain a logical clock (a counter). When a message is to be sent, TOA contacts B and C and asks them for their counters. B and C return their counters (incrementing them for the next request).

The originator of the message then sets the message's ID to be the max of all returned counters and sends the message. Receivers then deliver the messages in order of their IDs.

The main use of TOA is currently in Infinispan's transactional caches with partial replication: it is used to apply transactional modifications in total order, so that no two-phase commit protocol has to be run and no locks have to be acquired.

As shown in *"Exploiting Total Order Multicast in Weakly Consistent Transactional Caches"* [http://www.cloudtm.eu/home/Publications], when we have many conflicts by different transactions modifying the same keys, TOM fares better than 2PC.

Note that TOA is experimental (as of 3.1).

7.12. State Transfer

7.12.1. pbcast.STATE_TRANSFER

STATE_TRANSFER is the existing transfer protocol, which transfers byte[] buffers around. However, at the state provider's side, JGroups creates an output stream over the byte[] buffer, and passes the ouput stream to the `getState(OutputStream)` callback, and at the state requester's side, an input stream is created and passed to the `setState(InputStream)` callback.

This allows us to continue using STATE_TRANSFER, until the new state transfer protocols are going to replace it (perhaps in 4.0).

In order to transfer application state to a joining member of a cluster, STATE_TRANSFER has to load entire state into memory and send it to a joining member. The major limitation of this approach is that for state transfers that are very large this would likely result in memory exhaustion.

For large state transfer use either the STATE or STATE_SOCK protocol. However, if the state is small, STATE_TRANSFER is okay.

7.12.2. StreamingStateTransfer

`StreamingStateTransfer` is the superclass of STATE and STATE_SOCK (see below). Its properties are:

Table 7.34. Properties

Name	Description
buffer_size	Size (in bytes) of the state transfer buffer
max_pool	Maximum number of pool threads serving state requests
pool_thread_keep_alive	Keep alive for pool threads serving state requests

7.12.3. pbcast.STATE

7.12.3.1. Overview

STATE was renamed from (2.x) STREAMING_STATE_TRANSFER, and refactored to extend a common superclass `StreamingStateTransfer`. The other state transfer protocol extending `StreamingStateTransfer` is STATE_SOCK (see *Section 3.8.11.1.3, "STATE_SOCK"*).

`STATE` uses a *streaming approach* to state transfer; the state provider writes its state to the output stream passed to it in the `getState(OutputStream)` callback, which chunks the stream up into chunks that are sent to the state requester in separate messages.

The state requester receives those chunks and feeds them into the input stream from which the state is read by the `setState(InputStream)` callback.

The advantage compared to STATE_TRANSFER is that state provider and requester only need small (transfer) buffers to keep a part of the state in memory, whereas STATE_TRANSFER needs to copy the *entire* state into memory.

If we for example have a list of 1 million elements, then STATE_TRANSFER would have to create a byte[] buffer out of it, and return the byte[] buffer, whereas a streaming approach could iterate through the list and write each list element to the output stream. Whenever the buffer capacity is reached, we'd then send a message and the buffer would be reused to receive more data.

7.12.3.2. Configuration

STATE has currently no properties other than those inherited by `StreamingStateTransfer` (see above).

7.12.4. STATE_SOCK

STATE_SOCK is also a streaming state transfer protocol, but compared to STATE, it doesn't send the chunks as messages, but uses a TCP socket connection between state provider and requester to transfer the state.

The state provider creates a server socket at a configurable bind address and port, and the address and port are sent back to a state requester in the state response. The state requester then establishes a socket connection to the server socket and passes the socket's input stream to the `setState(InputStream)` callback.

7.12.4.1. Configuration

The configuration options of STATE_SOCK are listed below:

Table 7.35. Properties

Name	Description
bind_addr	The interface (NIC) used to accept state requests. The following special values are also recognized: GLOBAL, SITE_LOCAL, LINK_LOCAL and NON_LOOPBACK
bind_interface_str	The interface (NIC) which should be used by this transport
bind_port	The port listening for state requests. Default value of 0 binds to any (ephemeral) port
external_addr	Use "external_addr" if you have hosts on different networks, behind firewalls. On each firewall, set up a port forwarding rule (sometimes called "virtual server") to the local IP (e.g. 192.168.1.100) of the host then on each host, set "external_addr" TCP transport parameter to the external (public IP) address of the firewall.
external_port	Used to map the internal port (bind_port) to an external port. Only used if > 0

7.12.5. BARRIER

BARRIER is used by some of the state transfer protocols, as it lets existing threads complete and blocks new threads to get both the digest and state in one go.

In 3.1, a new mechanism for state transfer will be implemented, eliminating the need for BARRIER. Until then, BARRIER should be used when one of the state transfer protocols is used. BARRIER is part of every default stack which contains a state transfer protocol.

Table 7.36. Properties

Name	Description
max_close_time	Max time barrier can be closed. Default is 60000 ms

7.13. pbcast.FLUSH

Flushing forces group members to send all their pending messages prior to a certain event. The process of flushing acquiesces the cluster so that state transfer or a join can be done. It is also called the stop-the-world model as nobody will be able to send messages while a flush is in process. Flush is used in:

- State transfer

  When a member requests state transfer, it tells everyone to stop sending messages and waits for everyone's ack. Then it have received everyone's asks, the application asks the coordinator for its state and ships it back to the requester. After the requester has received and set the state successfully, the requester tells everyone to resume sending messages.

- View changes (e.g.a join). Before installing a new view V2, flushing ensures that all messages *sent* in the current view V1 are indeed *delivered* in V1, rather than in V2 (in all non-faulty members). This is essentially Virtual Synchrony.

FLUSH is designed as another protocol positioned just below the channel, on top of the stack (e.g. above STATE_TRANSFER). The STATE_TRANSFER and GMS protocols request a flush by sending an event up the stack, where it is handled by the FLUSH protcol. Another event is sent back by the FLUSH protocol to let the caller know that the flush has completed. When done (e.g. view was installed or state transferred), the protocol sends a message, which will allow everyone in the cluster to resume sending.

A channel is notified that the FLUSH phase has been started by the `Receiver.block()` callback.

Read more about flushing in *Section 5.7, “Flushing: making sure every node in the cluster received a message”*.

Table 7.37. Properties

Name	Description
bypass	When set, FLUSH is bypassed, same effect as if FLUSH wasn't in the config at all
enable_reconciliation	Reconciliation phase toggle. Default is true
end_flush_timeout	Timeout to wait for UNBLOCK after STOP_FLUSH is issued. Default is 2000 msec

Name	Description
retry_timeout	Retry timeout after an unsuccessful attempt to quiet the cluster (first flush phase). Default is 3000 msec
start_flush_timeout	Timeout (per atttempt) to quiet the cluster during the first flush phase. Default is 2000 msec
timeout	Max time to keep channel blocked in flush. Default is 8000 msec

7.14. Misc

7.14.1. Statistics

STATS exposes various statistics, e.g. number of received multicast and unicast messages, number of bytes sent etc. It should be placed directly over the transport

7.14.2. Security

JGroups provides protocols to encrypt cluster traffic (ENCRYPT), and to make sure that only authorized members can join a cluster (AUTH).

7.14.2.1. ENCRYPT

A detailed description of ENCRYPT is found in the JGroups source (`JGroups/doc/ENCRYPT.html`). Encryption by default only encrypts the message body, but doesn't encrypt message headers. To encrypt the entire message (including all headers, plus destination and source addresses), the property `encrypt_entire_message` has to be set to true. Also, ENCRYPT has to be below any protocols whose headers we want to encrypt, e.g.

```
<config ... >
    <UDP />
    <PING />
    <MERGE2 />
    <FD />
    <VERIFY_SUSPECT />
    <ENCRYPT encrypt_entire_message="false"
             sym_init="128" sym_algorithm="AES/ECB/PKCS5Padding"
             asym_init="512" asym_algorithm="RSA"/>
    <pbcast.NAKACK />
    <UNICAST />
    <pbcast.STABLE />
    <FRAG2 />
    <pbcast.GMS />
</config>
```

Note that ENCRYPT sits below NAKACK and UNICAST, so the sequence numbers for these 2 protocols will be encrypted. Had ENCRYPT been placed below UNICAST but above NAKACK, then only UNICAST's headers (including sequence numbers) would have been encrypted, but not NAKACKs.

Note that it doesn't make too much sense to place ENCRYPT even lower in the stack, because then almost all traffic (even merge or discovery traffic) will be encrypted, which may be somewhat of a performance drag.

When we encrypt an entire message, we have to marshal the message into a byte buffer first and then encrypt it. This entails marshalling and copying of the byte buffer, which is not so good performance wise...

7.14.2.1.1. Using a key store

ENCRYPT uses store type JCEKS (for details between JKS and JCEKS see here), however `keytool` uses JKS, therefore a keystore generated with keytool will not be accessible.

To generate a keystore compatible with JCEKS, use the following command line options to keytool:

```
keytool -genseckey -alias myKey -keypass changeit -storepass changeit  -
keyalg Blowfish -keysize 56 -keystore defaultStore.keystore -storetype
 JCEKS
```

ENCRYPT could then be configured as follows:

```
<ENCRYPT key_store_name="defaultStore.keystore"
         store_password="changeit"
         alias="myKey"/>
```

Note that defaultStore.keystore will have to be found in the claspath.

Table 7.38. Properties

Name	Description
alias	Alias used for recovering the key. Change the default
asymAlgorithm	Cipher engine transformation for asymmetric algorithm. Default is RSA

Name	Description
asymInit	Initial public/private key length. Default is 512
asymProvider	Cryptographic Service Provider. Default is Bouncy Castle Provider
encrypt_entire_message	
keyPassword	Password for recovering the key. Change the default
keyStoreName	File on classpath that contains keystore repository
storePassword	Password used to check the integrity/unlock the keystore. Change the default
symAlgorithm	Cipher engine transformation for symmetric algorithm. Default is AES
symInit	Initial key length for matching symmetric algorithm. Default is 128
symProvider	Cryptographic Service Provider. Default is Bouncy Castle Provider

7.14.2.2. AUTH

AUTH is used to provide a layer of authentication to JGroups. This allows you to define pluggable security that defines if a node should be allowed to join a cluster. AUTH sits below the GMS protocol and listens for JOIN REQUEST messages. When a JOIN REQUEST is received it tries to find an AuthHeader object, inside of which should be an implementation of the AuthToken object.

AuthToken is an abstract class, implementations of which are responsible for providing the actual authentication mechanism. Some basic implementations of AuthToken are provide in the org.jgroups.auth package (SimpleToken, MD5Token and X509Token). Effectivly all these implementations do is encrypt a string (found in the jgroups config) and pass that on the JOIN REQUEST.

When authentication is successful, the message is simply passed up the stack to the GMS protocol. When it fails, the AUTH protocol creates a JOIN RESPONSE message with a failure string and passes it back down the stack. This failure string informs the client of the reason for failure. Clients will then fail to join the group and will throw a SecurityException. If this error string is null then authentication is considered to have passed.

For more information refer to the wiki at *http://community.jboss.org/wiki/JGroupsAUTH*.

Table 7.39. Properties

Name	Description
auth_class	n/a

7.14.3. COMPRESS

COMPRESS compresses messages larger than `min_size`, and uncompresses them at the receiver's side. Property `compression_level` determines how thorough the compression algorith should be (0: no compression, 9: highest compression).

Table 7.40. Properties

Name	Description
compression_level	Compression level 0-9 (0=no compression, 9=best compression). Default is 9
min_size	Minimal payload size of a message (in bytes) for compression to kick in. Default is 500 bytes
pool_size	Number of inflaters/deflaters for concurrent processing. Default is 2

7.14.4. SCOPE

As discussed in *Section 5.4.4, "Scopes: concurrent message delivery for messages from the same sender"*, the SCOPE protocol is used to deliver updates to different scopes concurrently. It has to be placed somewhere above UNICAST and NAKACK.

SCOPE has a separate thread pool. The reason why the default thread pool from the transport wasn't used is that the default thread pool has a different purpose. For example, it can use a queue to which all incoming messages are added, which would defy the purpose of concurrent delivery in SCOPE. As a matter of fact, using a queue would most likely delay messages get sent up into SCOPE !

Also, the default pool's rejection policy might not be "run", so the SCOPE implementation would have to catch rejection exceptions and engage in a retry protocol, which is complex and wastes resources.

The configuration of the thread pool is shown below. If you expect *concurrent* messages to N *different* scopes, then the max pool size would ideally be set to N. However, in most cases, this is not necessary as (a) the messages might not be to different scopes or (b) not all N scopes might get messages at the same time. So even if the max pool size is a bit smaller, the cost of this is slight delays, in the sense that a message for scope Y might wait until the thread processing message for scope X is available.

To remove unused scopes, an expiry policy is provided: expiration_time is the number of milliseconds after which an idle scope is removed. An idle scope is a scope which hasn't seen any messages for expiration_time milliseconds. The expiration_interval value defines the number of milliseconds at which the expiry task runs. Setting both values to 0 disables expiration; it would then have to be done manually (see *Section 5.4.4, "Scopes: concurrent message delivery for messages from the same sender"* for details).

Table 7.41. Properties

Name	Description
expiration_interval	Interval in milliseconds at which the expiry task tries to remove expired scopes
expiration_time	Time in milliseconds after which an expired scope will get removed. An expired scope is one to which no messages have been added in max_expiration_time milliseconds. 0 never expires scopes
thread_naming_pattern	Thread naming pattern for threads in this channel. Default is cl
thread_pool.keep_alive_time	Timeout in milliseconds to remove idle thread from regular pool
thread_pool.max_threads	Maximum thread pool size for the regular thread pool
thread_pool.min_threads	Minimum thread pool size for the regular thread pool

7.14.5. RELAY

RELAY bridges traffic between seperate clusters, see *Section 5.10, "Bridging between remote clusters"* for details.

Table 7.42. Properties

Name	Description
bridge_name	Name of the bridge cluster
bridge_props	Properties of the bridge cluster (e.g. tcp.xml)
present_global_views	Drops views received from below and instead generates global views and passes them up. A global view consists of the local view and the remote view, ordered by view ID. If true, no protocolwhich requires (local) views can sit on top of RELAY
relay	If set to false, don't perform relaying. Used e.g. for backup clusters; unidirectional replication from one cluster to another, but not back. Can be changed at runtime
site	Description of the local cluster, e.g. "nyc". This is added to every address, so itshould be short. This is a mandatory property and must be set

7.14.6. RELAY2

RELAY2 provides clustering between different sites (local clusters), for multicast and unicast messages. See *Section 5.11, "Relaying between multiple sites (RELAY2)"* for details.

Table 7.43. Properties

Name	Description
async_relay_creation	If true, the creation of the relay channel (and the connect()) are done in the background. Async relay creation is recommended, so the view callback won't be blocked
can_become_site_master	Whether or not this node can become the site master. If false, and we become the coordinator, we won't start the bridge(s)
config	Name of the relay configuration
enable_address_tagging	Whether or not we generate our own addresses in which we use can_become_site_master. If this property is false, can_become_site_master is ignored
forward_sleep	The time (in milliseconds) to sleep between forward attempts
fwd_queue_max_size	Max number of messages in the foward queue. Messages are added to the forward queue when the status of a route went from UP to UNKNOWN and the queue is flushed when the status goes to UP (resending all queued messages) or DOWN (sending SITE-UNREACHABLE messages to the senders)
max_forward_attempts	The number of tries to forward a message to a remote site
relay_multicasts	Whether or not to relay multicast (dest=null) messages
site	Name of the site (needs to be defined in the configuration)
site_down_timeout	Number of millisconds to wait when the status for a site changed from UP to UNKNOWN before that site is declared DOWN. A site that's DOWN triggers immediate sending of a SITE-UNREACHABLE message back to the sender of a message to that site

Name	Description
warn_when_ftc_missing	If true, logs a warning if the FORWARD_TO_COORD protocol is not found. This property might get deprecated soon

7.14.7. STOMP

STOMP is discussed in *Section 5.9, "STOMP support"*. The properties for it are shown below:

Table 7.44. Properties (experimental)

Name	Description
bind_addr	The bind address which should be used by the server socket. The following special values are also recognized: GLOBAL, SITE_LOCAL, LINK_LOCAL and NON_LOOPBACK
endpoint_addr	If set, then endpoint will be set to this address
exact_destination_match	If set to false, then a destination of /a/b match /a/b/c, a/b/d, a/b/c/d etc
forward_non_client_generated_msgs	Forward received messages which don't have a StompHeader to clients
port	Port on which the STOMP protocol listens for requests
send_info	If true, information such as a list of endpoints, or views, will be sent to all clients (via the INFO command). This allows for example intelligent clients to connect to a different server should a connection be closed.

7.14.8. DAISYCHAIN

The DAISYCHAIN protocol is discussed in *Section 5.12, "Daisychaining"*.

Table 7.45. Properties (experimental)

Name	Description
forward_queue_size	The number of messages in the forward queue. This queue is used to host messages that need to be forwarded by us on behalf of our neighbor
loopback	Loop back multicast messages

Name	Description
send_queue_size	The number of messages in the send queue. This queue is used to host messages that need to be sent

7.14.9. RATE_LIMITER

RATE_LIMITER can be used to set a limit on the data sent per time unit. When sending data, only max_bytes can be sent per time_period milliseconds. E.g. if max_bytes="50M" and time_period="1000", then a sender can only send 50MBytes / sec max.

Table 7.46. Properties (experimental)

Name	Description
max_bytes	Max number of bytes to be sent in time_period ms. Blocks the sender if exceeded until a new time period has started
time_period	Number of milliseconds during which max_bytes bytes can be sent

7.14.10. Locking protocols

There are currently 2 locking protocols: org.jgroups.protocols.CENTRAL_LOCK and org.jgroups.protocols.PEER_LOCK. Both extend `Locking`, which has the following properties:

Table 7.47. Properties

Name	Description
bypass_bundling	bypasses message bundling if set

7.14.10.1. CENTRAL_LOCK

CENTRAL_LOCK has the current coordinator of a cluster grants locks, so every node has to communicate with the coordinator to acquire or release a lock. Lock requests by different nodes for the same lock are processed in the order in which they are received.

A coordinator maintains a lock table. To prevent losing the knowledge of who holds which locks, the coordinator can push lock information to a number of backups defined by num_backups. If num_backups is 0, no replication of lock information happens. If num_backups is greater than 0, then the coordinator pushes information about acquired and released locks to all backup nodes. Topology changes might create new backup nodes, and lock information is pushed to those on becoming a new backup node.

The advantage of CENTRAL_LOCK is that all lock requests are granted in the same order across the cluster, which is not the case with PEER_LOCK.

Table 7.48. Properties

Name	Description
num_backups	Number of backups to the coordinator. Server locks get replicated to these nodes as well

7.14.10.2. PEER_LOCK

PEER_LOCK acquires a lock by contacting all cluster nodes, and lock acquisition is only successful if all non-faulty cluster nodes (peers) grant it.

Unless a total order configuration is used (e.g. org.jgroups.protocols.SEQUENCER based), lock requests for the same resource from different senders may be received in different order, so deadlocks can occur. Example:

- Nodes A and B
- A and B call lock(X) at the same time
- A receives L(X,A) followed by L(X,B): locks X(A), queues L(X,B)
- B receives L(X,B) followed by L(X,A): locks X(B), queues L(X,A)

To acquire a lock, we need lock grants from both A and B, but this will never happen here. To fix this, either add SEQUENCER to the configuration, so that all lock requests are received in the same global order at both A and B, or use java.util.concurrent.locks.Lock.tryLock(long,javaTimeUnit) with retries if a lock cannot be acquired.

7.14.11. CENTRAL_EXECUTOR

CENTRAL_EXECUTOR is an implementation of Executing which is needed by the ExecutionService.

Table 7.49. Properties

Name	Description
bypass_bundling	bypasses message bundling if set

Table 7.50. Properties

Name	Description
num_backups	Number of backups to the coordinator. Queue State gets replicated to these nodes as well

7.14.12. COUNTER

COUNTER is the implementation of cluster wide counters, used by the CounterService.

Table 7.51. Properties (experimental)

Name	Description
bypass_bundling	Bypasses message bundling if true
num_backups	Number of backup coordinators. Modifications are asynchronously sent to all backup coordinators
reconciliation_timeout	Number of milliseconds to wait for reconciliation responses from all current members
timeout	Request timeouts (in ms). If the timeout elapses, a Timeout (runtime) exception will be thrown

7.14.13. SUPERVISOR

SUPERVISOR is a protocol which runs rules which periodically (or event triggered) check conditions and take corrective action if a condition is not met. Example: org.jgroups.protocols.rules.CheckFDMonitor is a rule which periodically checks if FD's monitor task is running when the cluster size is > 1. If not, the monitor task is started.

The SUPERVISOR is explained in more detail in *Section 5.16, “Supervising a running stack”*

Table 7.52. Properties

Name	Description
config	Location of an XML file listing the rules to be installed