

# Reliable Multicasting with the JGroups Toolkit

\$Revision: 1.9 \$

Copyright © 1998-2008 Bela BanCopyright © 2006-2008 Red Hat Inc

---

# Table of Contents

Foreword .....	vi
Acknowledgments .....	viii
1. Overview .....	1
1.1. Channel .....	2
1.2. Building Blocks .....	3
1.3. The Protocol Stack .....	3
1.4. Header .....	3
1.5. Event .....	4
2. Installation and Configuration .....	5
2.1. Requirements .....	5
2.2. Installing the binary distribution .....	5
2.3. Installing the source distribution .....	6
2.4. Building JGroups (source distribution only) .....	6
2.5. Testing your Setup .....	7
2.6. Running a Demo Program .....	7
2.7. Using IP Multicasting without a network connection .....	8
2.8. It doesn't work ! .....	9
2.9. The instances still don't find each other ! .....	9
2.10. Problems with IPv6 .....	10
2.11. Wiki .....	10
2.12. I have discovered a bug ! .....	10
3. API .....	12
3.1. Utility classes .....	12
3.1.1. objectToByteBuffer(), objectFromByteBuffer() .....	12
3.1.2. printMessage() .....	12
3.1.3. activeThreads() .....	12
3.1.4. printMembers() .....	12
3.2. Interfaces .....	12
3.2.1. Transport .....	13
3.2.2. MessageListener .....	13
3.2.3. ExtendedMessageListener .....	13
3.2.4. MembershipListener .....	14
3.2.5. ExtendedMembershipListener .....	14
3.2.6. ChannelListener .....	14
3.2.7. Receiver .....	15
3.2.8. ExtendedReceiver .....	15
3.3. Address .....	15
3.4. Message .....	16
3.5. View .....	17
3.5.1. ViewId .....	17
3.5.2. MergeView .....	18
3.6. Membership .....	18
3.7. Channel .....	18
3.7.1. Creating a channel .....	19

3.7.1.1. Using XML to define a protocol stack .....	20
3.7.2. Setting options .....	22
3.7.3. Connecting to a channel .....	22
3.7.4. Connecting to a channel and getting the state in one operation .....	23
3.7.5. Getting the local address and the group name .....	23
3.7.6. Getting the current view .....	24
3.7.7. Sending a message .....	24
3.7.8. Receiving a message .....	25
3.7.9. Using a Receiver to receive messages .....	27
3.7.10. Peeking at a message .....	27
3.7.11. Getting the group's state .....	28
3.7.12. Getting the state with a Receiver .....	29
3.7.13. Partial state transfer .....	29
3.7.14. Streaming state transfer .....	30
3.7.15. Disconnecting from a channel .....	32
3.7.16. Closing a channel .....	32
4. Building Blocks .....	33
4.1. PullPushAdapter .....	33
4.1.1. Example .....	33
4.2. MessageDispatcher .....	34
4.2.1. Example .....	36
4.3. RpcDispatcher .....	37
4.3.1. Example .....	38
4.4. DistributedHashtable .....	39
4.5. ReplicatedHashtable .....	39
4.6. DistributedTree .....	39
4.7. NotificationBus .....	40
5. Advanced Concepts .....	42
5.1. Using multiple channels .....	42
5.2. Using the Multiplexer to run multiple building blocks over the same channel .....	42
5.2.1. The Multiplexer API .....	43
5.2.1.1. Batching state transfers .....	44
5.2.2. Service views .....	45
5.3. Transport protocols .....	46
5.3.1. UDP .....	48
5.3.1.1. Using UDP and plain IP multicasting .....	48
5.3.1.2. Using UDP without IP multicasting .....	48
5.3.2. TCP .....	49
5.3.2.1. Using TCP and TCPPING .....	50
5.3.2.2. Using TCP and TCPGOSSIP .....	51
5.3.3. TUNNEL .....	51
5.3.3.1. Using TUNNEL to tunnel a firewall .....	51
5.4. The concurrent stack .....	54
5.4.1. Overview .....	54
5.4.1.1. Configuration .....	56
5.4.2. Elimination of up and down threads .....	57
5.4.3. Concurrent message delivery .....	57
5.4.4. Out-of-band messages .....	58
5.5. Misc .....	58

5.5.1. Shunning .....	58
5.6. Handling network partitions .....	59
5.6.1. Merging substates .....	59
5.6.2. The primary partition approach .....	60
5.6.3. The Split Brain syndrome and primary partitions .....	61
5.7. Flushing: making sure every node in the cluster received a message .....	62
6. List of Protocols .....	63
6.1. Transport .....	63
6.1.1. UDP .....	63
6.1.2. TCP .....	63
6.1.3. TCP_NIO .....	63
6.1.4. TUNNEL .....	63
6.1.5. JMS .....	63
6.1.6. LOOPBACK .....	63
6.2. Initial membership discovery .....	64
6.2.1. PING .....	64
6.2.2. TCPPING .....	64
6.2.3. TCPGOSSIP .....	64
6.2.4. MPING .....	64
6.3. Merging after a network partition .....	64
6.3.1. MERGE2, MERGE3, MERGEFAST .....	64
6.4. Failure Detection .....	64
6.4.1. FD .....	64
6.4.2. FD_ALL .....	65
6.4.3. FD_SIMPLE .....	66
6.4.4. FD_PING .....	66
6.4.5. FD_ICMP .....	66
6.4.6. FD SOCK .....	67
6.4.7. VERIFY_SUSPECT .....	67
6.5. Reliable message transmission .....	67
6.5.1. pbcaster.NAKACK .....	67
6.5.2. SMACK .....	70
6.5.3. UNICAST .....	70
6.6. Fragmentation .....	70
6.6.1. FRAG and FRAG2 .....	70
6.7. Ordering (FIFO covered by NAKACK) .....	70
6.7.1. Total Order (SEQUENCER) .....	70
6.8. Group Membership .....	70
6.8.1. pbcaster.GMS .....	71
6.8.1.1. Disabling the initial coordinator .....	72
6.9. Security .....	73
6.9.1. ENCRYPT .....	73
6.9.2. AUTH .....	73
6.10. State Transfer .....	73
6.10.1. pbcaster.STATE_TRANSFER .....	73
6.10.2. pbcaster.STREAMING_STATE_TRANSFER .....	73
6.10.2.1. Overview .....	73
6.10.2.2. API .....	73
6.10.2.3. Configuration .....	75

6.10.2.4. Other considerations .....	76
6.11. Flow control .....	76
6.11.1. FC .....	76
6.11.2. SFC .....	77
6.12. Message stability .....	78
6.12.1. STABLE .....	78
6.13. Diagnostics .....	79
6.13.1. PERF .....	79
6.13.2. SIZE .....	79
6.13.3. TRACE .....	79
6.13.4. PRINTOBS .....	79
6.14. Misc .....	79
6.14.1. COMPRESS .....	79
6.14.2. pbcast.FLUSH .....	79
Bibliography .....	81

---

# Foreword

This is the Programmer's and User's Guide for JGroups. It provides information about the following areas:

1. Installation and configuration.
2. Using JGroups.
3. Architecture and implementation of JGroups. Focus on the protocol stack and protocols.

Most of the Installation and User's Guide has been copied from what is freely available on the JGroups web site [1]. However, the focus of this document is to introduce programmers who want to learn more about JGroups to the architecture and implementation of JGroups. I will for example go into the details of the protocol stack, how a message traverses the stack, and how protocols can process it. I will also explain the various design decisions I had to make when designing JGroups, which hopefully leads to a better understanding of why things are the way they are.

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 book 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 for 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, which 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, or within the same network, 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

[1] <http://www.jgroups.org/>

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 than 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 industry: with Fujitsu Network Communications in Silicon Valley.

At around that time (May 2000), SourceForge had just opened its site, and I decided to use it for hosting JGroups. I guess 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 the Java space...

Bela Ban, San Jose, Aug 2002, Kreuzlingen Switzerland 2006

---

# Acknowledgments

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 book to Jeannette and Michelle.



# 1

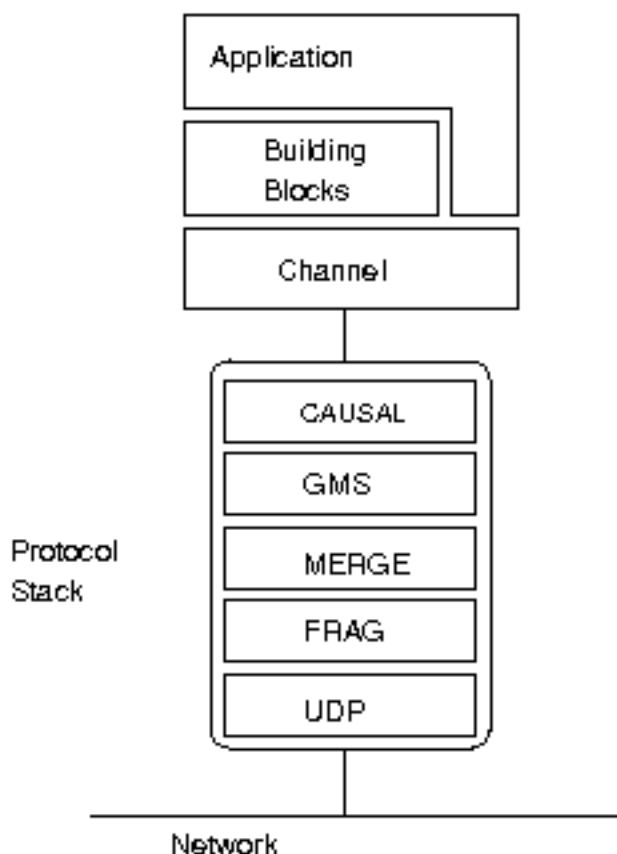
## Overview

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 groups is a *cluster*. We use these words 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.

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. Member 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.



**Figure 1.1.** The architecture of JGroups

It consists of 3 parts: (1) the Channel API 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. Programmers who want to *implement* their own protocols to be used with JGroups should consult the Programmer's Guide for more details about the architecture and implementation of JGroups.

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 then passes it on to the protocol below it. Thus the message is handed from protocol to protocol until the bottom protocol puts it on the network. The same happens in the reverse direction: the bottom (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 stores the message in a queue until the application consumes it.

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. 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, view messages and suspicion messages. A client may choose to turn reception of views and suspicions on/off on a channel basis.

Channels are similar to BSD sockets: messages are stored in a channel until a client removes the next one (pull-principle). When no message is currently available, a client is blocked until the next available message has been received.

There is currently only one implementation of Channel: JChannel.

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 programming.

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

## 1.2. Building Blocks

Channels are simple and primitive. They offer the bare functionality of group communication, and have on purpose been designed after the simple model of BSD 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 12 methods to be able to create and use a channel (and oftentimes will only use 3-4 methods frequently).

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.

Also, an application has to actively *retrieve* messages from a channel (pull-style); it is not notified when a message has been received. Note that pull-style message reception often needs another thread of execution, or some form of event-loop, in which a channel is periodically polled for messages.

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.

Building blocks are described in Chapter 4.

## 1.3. The Protocol Stack

The protocol stack contains a number of protocol layers in a bidirectional list. All messages sent and received over the channel have to pass through the protocol stack. 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 layers, is determined by the creator of the channel: an XML file defines the layers to be used (and the parameters for each layer). This string might be interpreted differently by each channel implementation; in JChannel it is used to create the stack, depending on the protocol names given in the property.

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. Although it is syntactically possible to stack any layer on top of each other (they all have the same interface), this wouldn't make sense semantically in most cases.

## 1.4. 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 de-

livered in the order in which they were sent.

## 1.5. Event

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

## Installation and Configuration

The installation refers to version 2.5 of JGroups. Refer to the installation instructions that are shipped with JGroups for details.

Note that these instructions are also available in the JGroups distribution (`INSTALL.HTML`).

JGroups comes in a binary and a source version: the binary version is `JGroups-2.x.x.bin.zip`, the source version is `JGroups-2.x.x.src.zip`. The binary version contains the JGroups JAR file, plus a number of JARs needed by JGroups. The source version contains all source files, plus several JAR files needed by JGroups, e.g. ANT to build JGroups from source.

### 2.1. Requirements

- JGroups 2.5 requires JDK 5 or higher.
- There is no JNI code present so it 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. Installing the binary distribution

The binary version contains

1. `jgroups-all.jar`: the JGroups library including the demos
2. CREDITS: list of contributors
3. `INSTALL.html`: this file
4. `commons-logging.jar`
5. `log4j.jar`. This JAR is optional, for example if JDK logging is used, we don't need it.

Place the JAR files somewhere in your CLASSPATH, and you're ready to start using JGroups. If you want to use the JGroups JMS protocol (`org.jgroups.protocols.JMS`), then you will also need to place `jms.jar` somewhere in your CLASSPATH.

## 2.3. Installing the source distribution

The source version consists of the following directories and files:

1. `src`: the sources
2. `test`: unit and stress tests
3. `conf`: configuration files needed by JGroups, plus default protocol stack definitions
4. `doc`: documentation
5. `lib`: various JARs needed to build and run JGroups:
  - a. Ant [1] JARs: used to build JGroups. If you already have Ant installed, you won't need these files
  - b. `jms.jar`: JMS library. Needed only if you intend to run the `org.jgroups.protocols.JMS` protocol
  - c. `junit.jar`: to run the JUnit [2] test cases
  - d. `xalan.jar` [3]: to format the output of the JUnit tests using an XSLT converter to HTML
  - e. `commons-logging.jar`
  - f. `log4j.jar`
  - g. etc

## 2.4. Building JGroups (source distribution only)

1. Unzip the source distribution, e.g. `unzip JGroups-2.x.x.src.zip`. This will create the `JGroups-2.x.x` directory (root directory) under the current directory.
2. `cd` to the root directory
3. Modify `build.properties` if you want to use a Java compiler other than `javac` (e.g. `jikes`), or if you want to change the interface JGroups uses for sending and receiving messages
4. On UNIX systems use `build.sh`, on Windows `build.bat`: `$> ./build.sh compile`
5. This will compile all Java files (into the `classes` directory).
6. To generate the JARs: `$> ./build.sh jar`
7. This will generate the following JAR files in the `dist` directory:

[1] <http://jakarta.apache.org/ant/>

[2] <http://www.junit.org/>

[3] <http://xml.apache.org/>

- `jgroups-core.jar` - the core JGroups libraries
  - `jgroups-all.jar` - the complete JGroups libraries including demos and unit tests
8. The `CLASSPATH` now has to be set accordingly: the following directories and/or JARs have to be included:
    - a. `<JGroups rootdir>/classes`
    - b. `<JGroups rootdir>/conf`
    - c. All needed JAR files in `<JGroups rootdir>/lib`. To build from sources, the two Ant JARs are required. To run unit tests, the JUnit (and possibly Xalan) JARs are needed.
  9. To generate JavaDocs simply run `$> ./build.sh javadoc` and the Javadoc documentation will be generated in the `dist/javadoc` directory
  10. 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 invoke `ant` from any directory below the root directory, place `ANT_ARGS="-find build.xml -emacs"` into the `.antrc` file in your home directory.
  11. For more details on Ant see <http://jakarta.apache.org/ant/>.

## 2.5. Testing your Setup

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

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

or (from JGroups 2.2.8 on)

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

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

```
bela@dell /cygdrive/c/JGroups/dist
$ java -jar jgroups-all.jar

Version:      2.6.0 pre-alpha
CVS:          $Id: installation.xml,v 1.5 2007/07/24 16:30:46 belaban Exp $
```

## 2.6. Running a Demo Program

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

```
java org.jgroups.demos.Draw
```

2 whiteboard windows should appear as shown in Figure 2.1.



**Figure 2.1. Screenshot of 2 Draw instances**

Both windows should show 2 in their title bars. This means that the two instances found each other and formed a group.

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 sample 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.

## 2.7. 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).

## 2.8. 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 -mcast_addr 224.10.10.10 -port 5555
```

Then start `McastSenderTest`:

```
java org.jgroups.tests.McastSenderTest -mcast_addr 224.10.10.10 -port 5555
```

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 [5] for questions. Also consider subscribing to the `javagroups-users` mailing list to discuss such and other problems.

## 2.9. The instances still don't find each other !

In this case we have to use a sledgehammer (running only under JDK 1.4. and higher): we can enable the above sender and receiver test to use all available interfaces for sending and receiving. One of them will certainly be the right one... Start the receiver as follows:

```
java org.jgroups.tests.McastReceiverTest1_4 -mcast_addr 228.8.8.8 -use_all_interfaces
```

The multicast receiver uses the 1.4 functionality to list *all available network interfaces and bind to all of them* (including the loopback interface). This means that whichever interface a packet comes in on, we will receive it. Now start the sender:

[5] <http://jira.jboss.com/jira/browse/JGRP>

```
java org.jgroups.tests.McastSenderTest1_4 -mcast_addr 228.8.8.8 -use_all_interfaces
```

The sender will also determine the available network interfaces and send each packet over all interfaces.

This test can be used to find out which network interface to bind to when previously no packets were received. E.g. when you see the following output in the receiver:

```
bash-2.03$ java org.jgroups.tests.McastReceiverTest1_4 -mcast_addr 228.8.8.8 -bind_addr 192.168.168.4
Socket=0.0.0.0/0.0.0.0:5555, bind interface=/192.168.168.4
dd [sender=192.168.168.4:5555]
dd [sender=192.168.168.1:5555]
dd [sender=192.168.168.2:5555]
```

you know that you can bind to any of the 192.168.168.{1,2,4} interfaces to receive your multicast packets. In this case you would need to modify your protocol spec to include `bind_addr=192.168.168.2` in UDP, e.g. `"UDP(mcast_addr=228.8.8.8;bind_addr=192.168.168.2):..."` .

## 2.10. 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 `java.net.preferIPv4Stack=true` property, e.g.

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

JDK 1.4.1 uses IPv6 by default, although it has a dual stack, that is, it also supports IPv4. Here's [6] more details on the subject.

## 2.11. 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.

## 2.12. I have discovered a bug !

If you think that you discovered a bug, submit a bug report on JIRA [8] or send email to [javagroups-developers](mailto:javagroups-developers@jboss.org) if you're unsure about it. Please include the following information:

- Version of JGroups (`java org.jgroups.Version`)

[6] [http://java.sun.com/j2se/1.4/docs/guide/net/ipv6\\_guide/](http://java.sun.com/j2se/1.4/docs/guide/net/ipv6_guide/)

[8] <http://jira.jboss.com/jira/browse/JGRP>

- 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

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 the JGroups toolkit 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 below reside in the `org.jgroups` package unless otherwise mentioned.

### 3.1. Utility classes

The `org.jgroups.util.Util` class contains a collection of useful functionality which cannot be assigned to any particular 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 unseriali

#### 3.1.2. `printMessage()`

Prints the message given as argument in readable format. Returns a string.

#### 3.1.3. `activeThreads()`

Returns a strings containing a pretty-printed list of currently running threads.

#### 3.1.4. `printMembers()`

Given a list of member addresses, pretty-prints the members and returns a string.

### 3.2. Interfaces

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

### 3.2.1. Transport

Interface `Transport` looks as follows:

```
public interface Transport {
    public void send(Message msg) throws Exception;
    public Object receive(long timeout) throws Exception;
}
```

It defines a very small subset of the functionality of a channel, essentially only the methods for sending and receiving messages. There are a number of classes that implement `Transport`, among others `Channel`. Many building blocks (see Chapter 4) require nothing else than a bare-bone facility to send and receive messages; therefore the `Transport` interface was created. It increases the genericness and portability of building blocks: being so simple, the `Transport` interface can easily be ported to a different toolkit, without requiring any modifications to building blocks.

### 3.2.2. MessageListener

Contrary to the pull-style of channels, some building blocks (e.g. `PullPushAdapter`) provide an event-like *push-style* message delivery model. In this case, the entity to be notified of message reception needs to provide a callback to be invoked whenever a message has been received. The `MessageListener` interface below provides a method to do so:

```
public interface MessageListener {
    public void receive(Message msg);
    byte[] getState();
    void setState(byte[] state);
}
```

Method `receive()` will be called when 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 3.7.11 for a discussion of state transfer.

### 3.2.3. ExtendedMessageListener

JGroups release 2.3 introduces `ExtendedMessageListener` enabling partial state transfer (refer to Section 3.7.13) while release 2.4 further expands `ExtendedMessageListener` with streaming state transfer callbacks:

```
public interface ExtendedMessageListener extends MessageListener {
    byte[] getState(String state_id);
    void setState(String state_id, byte[] state);

    /** since JGroups 2.4 *****/
    void getState(OutputStream ostream);
    void getState(String state_id, OutputStream ostream);
    void setState(InputStream istream);
    void setState(String state_id, InputStream istream);
}
```

Method `receive()` will be called when 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 3.7.11 for a discussion of state transfer.

### 3.2.4. 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();
}
```

Oftentimes the only method containing any functionality 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 is suspected of having crashed, but not yet excluded<sup>1</sup>.

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 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. However, sending of messages should be done on a different thread, e.g. the current thread blocks on a mutex, starts a different thread which notifies the mutex once the work has been done.

Note that `block()` should take a small amount of time to complete, otherwise the entire FLUSH protocol is blocked.

### 3.2.5. ExtendedMembershipListener

The `ExtendedMembershipListener` interface extends `MembershipListener`:

```
public interface ExtendedMembershipListener extends MembershipListener {
    public void unblock();
}
```

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.

### 3.2.6. ChannelListener

<sup>1</sup>It could be that the member is suspected falsely, in which case the next view would still contain the suspected member (there is currently no `unsuspect()` method)

```
public interface ChannelListener {
    void channelConnected(Channel channel);
    void channelDisconnected(Channel channel);
    void channelClosed(Channel channel);
    void channelShunned();
    void channelReconnected(Address addr);
}
```

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

### 3.2.7. Receiver

```
public interface Receiver extends MessageListener, MembershipListener {
}
```

A `Receiver` can be used to receive all relevant messages and view changes in push-style; rather than having to pull these events from a channel, they will be dispatched to the receiver as soon as they have been received. This saves one thread (application thread, pulling messages from a channel, or the `PullPushAdapter` thread)

### 3.2.8. ExtendedReceiver

```
public interface ExtendedReceiver extends ExtendedMessageListener, MembershipListener {
}
```

This is a receiver who will be able to handle partial state transfer

### Merging of Extended interfaces with their super interfaces

The Extended- interfaces (`ExtendedMessageListener`, `ExtendedReceiver`) will be merged with their parents in the 3.0 release of JGroups. The reason is that this will create an API backwards incompatibility, which we didn't want to introduce in the 2.x series.

## 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 for comparison and sorting of addresses, and for determination whether the address is a multicast address. JGroups addresses have to implement the following interface:

```
public interface Address extends Externalizable, Comparable, Cloneable {
    boolean isMulticastAddress();
    int compareTo(Object o) throws ClassCastException;
    boolean equals(Object obj);
    int hashCode();
}
```

```

        String toString();
    }

```

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, e.g. ATM.

In JChannel, it is the IP address of the host on which the stack is running and the port on which the stack is receiving incoming messages; it is represented by the concrete class `org.jgroups.stack.Address`. Instances of this class are only used *within* the JChannel protocol stack; *users of a channel see addresses (of any kind) only as Addresses*. 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).

### 3.4. Message

Data is sent between members in the form of messages ( `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 .



**Figure 3.1. Structure of a message**

A message contains 5 fields:

#### Destination address

The address of the receiver. If `null`, the message will be sent to all current group members

#### 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

#### Flags

This is one byte used for flags. The currently recognized flags are OOB, LOW\_PRIO and HIGH\_PRIO. See the discussion on the concurrent stack for OOB.

#### 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.



## 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.

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, null, "Hello".getBytes());
channel.send(msg);
```

## 3.5. View

A `View (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 set 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 there is a comparison function which orders all the members of a group in the same way. Usually, the first member of the list 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.

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

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

Whenever an application is notified that a new view has been installed (e.g. by `MembershipListener.viewAccepted()` or `Channel.receive()`), 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 !).

### 3.5.1. 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 hashtable as they implement equals() and hashCode() methods.

### 3.5.2. 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 class 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 contain a list of 2 views:  $v2:(p,q,r)$  and  $v2:(s,t)$ .

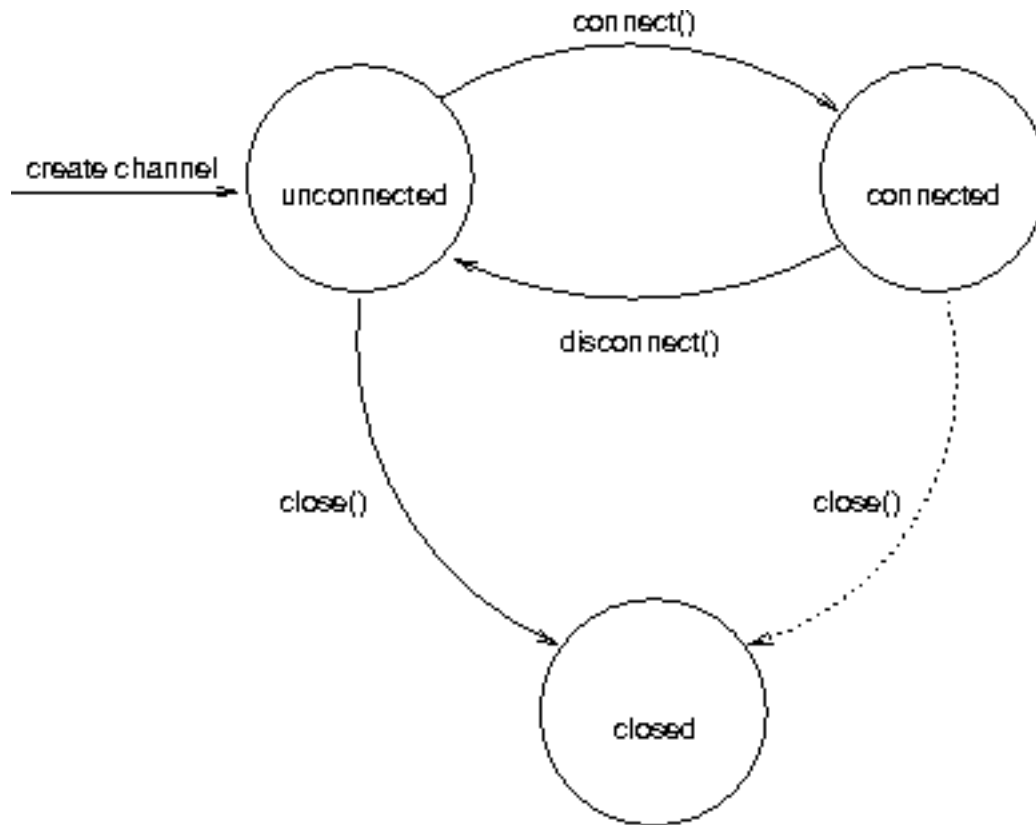
## 3.6. Membership

This class can be used for keeping track of members instead of a Vector class. It adds several functions, such as duplicate elimination, merging with other Membership instances and sorting.

## 3.7. Channel

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 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.



**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 channels will receive messages, views and suspicions from other members and may send messages to other members or to the group. 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.

### 3.7.1. Creating a channel

A channel can be created in two ways: an instance of a subclass of `Channel` is created directly using its public constructor (e.g. `new JChannel()`), or a channel factory is created, which -- upon request -- creates instances of channels. We will only look at the first method of creating channel: by direct instantiation. Note that instantiation may differ between the various channel implementations. As example we will look at `JChannel`.

The public constructor of `JChannel` looks as follows:

```
public JChannel(Object properties) throws ChannelException {}
```

It creates an instance of `JChannel`. The *properties* argument defines the composition of the protocol stack (number and type of layers, parameters for each layer, and their order). For `JChannel`, this has to be a `String`. An example of a channel creation is:

```
String props="UDP(mcast_addr=228.1.2.3;mcast_port=45566;ip_ttl=32):" +
"PING(timeout=3000;num_initial_members=6):" +
"FD(timeout=5000):" +
"VERIFY_SUSPECT(timeout=1500):" +
"pbcast.STABLE(desired_avg_gossip=10000):" +
"pbcast.NAKACK(gc_lag=10;retransmit_timeout=3000):" +
"UNICAST(timeout=5000;min_wait_time=2000):" +
"FRAG:" +
"pbcast.GMS(initial_mbrs_timeout=4000;join_timeout=5000:" +
"shun=false;print_local_addr=false)";

JChannel channel;
try {
    channel=new JChannel(props);
}
catch(Exception ex) {
    // channel creation failed
}
```

The argument is a colon-delimited string of protocols, specified from bottom to top (left to right). The example properties argument will be used to create a protocol stack that uses IP Multicast (UDP) as bottom protocol, the PING protocol to locate the initial members, FD for failure detection, VERIFY\_SUSPECT for double-checking of suspected members, STABLE for garbage collection of messages received by all members, NAKACK for lossless delivery of multicast messages, UNICAST for lossless delivery of unicast messages and GMS for group membership (handling of join or leave requests).

If the properties 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.

### 3.7.1.1. Using XML to define a protocol stack

In version 2.0 of JGroups an XML-based scheme to define protocol stacks was introduced. Instead of specifying a string containing the protocol spec, an URL pointing to a valid protocol stack definition can be given. For example, the Draw demo can be launched as follows:

```
java org.javagroups.demos.Draw -props file:/home/bela/vsync.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>
  <UDP
    mcast_addr="${jgroups.udp.mcast_addr:228.10.10.10}"
    mcast_port="${jgroups.udp.mcast_port:45588}"
    discard_incompatible_packets="true"
    max_bundle_size="60000"
    max_bundle_timeout="30"
    ip_ttl="${jgroups.udp.ip_ttl:2}"
    enable_bundling="true"
    use_concurrent_stack="true"
    thread_pool.enabled="true"
    thread_pool.min_threads="1"
    thread_pool.max_threads="25"
    thread_pool.keep_alive_time="5000"
    thread_pool.queue_enabled="false"
    thread_pool.queue_max_size="100"
    thread_pool.rejection_policy="Run"
    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 timeout="10000" max_tries="5" shun="true"/>
  <VERIFY_SUSPECT timeout="1500" />
  <BARRIER />
  <pbcast.NAKACK
    use_mcast_xmit="false" gc_lag="0"
    retransmit_timeout="300,600,1200,2400,4800"
    discard_delivered_msgs="true"/>
  <UNICAST timeout="300,600,1200,2400,3600"/>
  <pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000"
    max_bytes="400000"/>
  <VIEW_SYNC avg_send_interval="60000" />
  <pbcast.GMS print_local_addr="true" join_timeout="3000"
    shun="false"
    view_bundling="true"/>
  <FC max_credits="20000000"
    min_threshold="0.10"/>
  <FRAG2 frag_size="60000" />
  <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.stack.protocols` package. Note that only the base name has to be given, not the fully specified class name ( UDP instead of

`org.jgroups.stack.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 address (`mcast_addr`) which is set to 228.10.10.10, or to the value of the system property `jgroups.udp.mcast_addr`, if set.

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

### 3.7.2. Setting options

A number of options can be set in a channel. To do so, the following method is used:

```
public void setOpt(int option, Object value);
```

Arguments are the options number and a value. The following options are currently recognized:

`Channel.BLOCK`

The argument is a boolean object. If true, block messages will be received. If this option is set to true, views will also be set to true. Default is false.

`Channel.LOCAL`

Local delivery. The argument is a boolean value. If set to true, a member will receive all messages it sent to itself. Otherwise, all messages sent by itself will be discarded. This option allows to send messages to the group, without receiving a copy. Default is true (members will receive their own copy of messages multicast to the group).

`Channel.AUTO_RECONNECT`

When set to true, a shunned channel will leave the group and then try to automatically re-join. Default is false

`Channel.AUTO_GETSTATE`

When set to true a shunned channel, after reconnection, will attempt to fetch the state from the coordinator. This requires `AUTO_RECONNECT` to be true as well. Default is false.

The equivalent method to get options is `getOpt()` :

```
public Object getOpt(int option);
```

Given an option, the current value of the option is returned.

### 3.7.3. Connecting to a channel

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

```
public void connect(String clustername) throws ChannelClosed;
```

The cluster name is a string, naming the cluster to be joined. All channels that are connected to the same name form a cluster. Messages multicast on any channel in the cluster will be received by all members (including the one who sent it <sup>2</sup>).

The method returns as soon as the group has been joined successfully. If the channel is in the closed state (see Figure 3.2), an exception will be thrown. If there are no other members, i.e. no other client has connected to a group with this name, then a new group is created and the member joined. The first member of a group becomes its *coordinator*. A coordinator is in charge of multicasting new views whenever the membership changes <sup>3</sup>.

### 3.7.4. Connecting to a channel and getting the state in one operation

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

```
public void connect(string cluster_name, address target, string state_id, long timeout)
```

Just as in regular connect method cluster name represents a cluster to be joined. Address parameter indicates a cluster member to fetch state from. Null address parameter indicates that state should be fetched from the cluster coordinator. If state should be fetched from a particular member other than coordinator clients can provide an address of that member. State id used for partial state transfer while timeout bounds entire join and fetch operation.

### 3.7.5. Getting the local address and the group name

Method `getLocalAddress()` returns the local address of the channel. In the case of `JChannel`, the local address is generated by the bottom-most layer of the protocol stack when the stack is connected to. That means that -- depending on the channel implementation -- the local address may or may not be available when a channel is in the unconnected state.

```
public Address getLocalAddress();
```

Method `getClusterName()` returns the name of the cluster in which the channel is a member:

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

<sup>2</sup>Local delivery can be turned on/off using `setOpt()`.

<sup>3</sup>This is managed internally however, and an application programmer does not need to be concerned about it.

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

### 3.7.6. Getting the current view

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

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

This method does *not* retrieve a new view (message) from the channel, but only returns the current view of the channel. The current view is updated every time a view message is received: when method `receive()` is called, and the return value is a view, before the view is returned, it will be installed in the channel, i.e. it will become the current view.

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.

### 3.7.7. Sending a message

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

```
public void send(Message msg) throws ChannelNotConnected, ChannelClosed;
public void send(Address dst, Address src, Object obj) throws ChannelNotConnected, ChannelClosed;
```

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 it is null, the message will be sent to all members of the group (including itself). The source address may be null; if it is, it will be set to the channel's address (so that recipients may generate a response and send it back to the sender).

The second `send()` method is a helper method and uses the former method internally. It requires the address of receiver and sender and an object (which has to be serializable), constructs a `Message` and sends it.

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 (multicast) message to all members of a group:

```
Hashtable data; // any serializable data
try {
    channel.send(null, null, data);
}
catch(Exception ex) {
    // handle errors
}
```

The null value as destination address means that the message will be sent to all members in the group. The sender's address will be filled in by the bottom-most protocol. The payload is a hashtable, which will be serialized into the message's buffer and unserialized at the receiver's end. Alternatively, any other means of generating a byte buffer



and setting the message's buffer to it (e.g. using `Message.setBuffer()`) would also work.

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

```
Address receiver;
Message msg;
Hashtable data;
try {
    receiver=channel.getView().getMembers().first();
    channel.send(receiver, null, data);
}
catch(Exception ex) {
    // handle errors
}
```

It creates a `Message` with a specific address for the receiver (the first member of the group). Again, the sender's address can be left null as it will be filled in by the bottom-most protocol.

### 3.7.8. Receiving a message

Method `receive()` is used to receive messages, views, suspicions and blocks:

```
public Object receive(long timeout) throws ChannelNotConnected, ChannelClosed, Timeout;
```

A channel receives messages asynchronously from the network and stores them in a queue. When `receive()` is called, the next available message from the top of that queue is removed and returned. When there are no messages on the queue, the method will block. If `timeout` is greater than 0, it will wait the specified number of milliseconds for a message to be received, and throw a `TimeoutException` exception if none was received during that time. If the timeout is 0 or negative, the method will wait indefinitely for the next available message.

Depending on the channel options (see Section 3.7.2 ), the following types of objects may be received:

#### Message

A regular message. To send a response to the sender, a new message can be created. Its destination address would be the received message's source address. Method `Message.makeReply()` is a helper method to create a response.

#### View

A view change, signalling that a member has joined, left or crashed. The application may or may not perform some action upon receiving a view change (e.g. updating a GUI object of the membership, or redistributing a load-balanced collaborative task to all members). Note that a longer action, or any action that blocks should be performed in a separate thread. A `MergeView` will be received when 2 or more subgroups merged into one (see Section 3.5.2 for details). Here, a possible state merge by the application needs to be done in a separate thread.

#### SuspectEvent

Notification of a member that is suspected. Method `SuspectEvent.getMember()` retrieves the address of the suspected member. Usually this message will be followed by a view change.

### BlockEvent

The application has to stop sending messages. When the application has stopped sending messages, it needs to acknowledge this message with a `Channel.blockOk()` method.

The `BlockEvent` reception can be used to complete pending tasks, e.g. send pending messages, but once `Channel.blockOk()` has been called, all threads that send messages (calling `Channel.send()` or `Channel.down()`) will be blocked until `FLUSH` unblocks them.

### UnblockEvent

The application can resume sending messages. Any previously messages blocked by `FLUSH` will be unblocked; when the `UnblockEvent` is received the channel has already been unblocked.

### GetStateEvent

Received when the application's current state should be saved (for a later state transfer. A *copy* of the current state should be made (possibly wrapped in a `synchronized` statement and returned calling method `Channel.returnState()` . If state transfer events are not enabled on the channel (default), then this event will never be received. This message will only be received with the Virtual Synchrony suite of protocols (see the Programmer's Guide).

### StreamingGetStateEvent

Received when the application's current state should be provided to a state requesting group member. If state transfer events are not enabled on the channel (default), or if channel is not configured with `pb-cast.STREAMING_STATE_TRANSFER` then this event will never be received.

### SetStateEvent

Received as response to a `getState(s)` method call. The argument contains the state of a single member ( `byte[]` ) or of all members ( `Vector` ). Since the state of a single member could also be a vector, the interpretation of the argument is left to the application.

### StreamingSetStateEvent

Received at state requesting member when the state `InputStream` becomes ready for reading. If state transfer events are not enabled on the channel (default), or if channel is not configured with `pb-cast.STREAMING_STATE_TRANSFER` then this event will never be received.

The caller has to check the type of the object returned. This can be done using the `instanceof` operator, as follows:

```
Object obj;
Message msg;
View v;
obj=channel.receive(0); // wait forever
if(obj instanceof Message)
    msg=(Message)obj;
else if(obj instanceof View)
    v=(View)obj;
else
    ; // don't handle suspicions or blocks
```

If for example views, suspicions and blocks are disabled, then the caller is guaranteed to only receive return values of type `Message` . In this case, the return value can be cast to a `Message` directly, without using the `instanceof` operator.

If the channel is not connected, or was closed, a corresponding exception will be thrown.

The example below shows how to retrieve the "Hello world" string from a message:

```
Message msg; // received above
String s;
try {
    s=(String)msg.getObject(); // error if object not Serializable
    // alternative: s=new String(msg.getBuffer());
}
catch(Exception ex) {
    // handle errors, e.g. casting error above
}
```

The `Message.getObject()` method retrieves the message's byte buffer, converts it into a (serializable) object and returns the object.

### 3.7.9. Using a Receiver to receive messages

Instead of pulling messages from a channel in an application thread, a Receiver can be registered with a channel; all received messages, view changes and state transfer requests will invoke callbacks on the registered Receiver:

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

The `ExtendedReceiverAdapter` class implements all callbacks of `ExtendedReceiver` with no-ops, in the example above we override `receive()` and `viewAccepted()`.

The advantage of using a Receiver is that the application doesn't have to waste 1 thread for pulling messages out of a channel. In addition, the channel doesn't have to maintain an (unbounded) queue of messages/views, which can quickly get large if the receiver cannot process messages fast enough, and the sender keeps sending messages.

#### Note

Note that the `Channel.receive()` method has been deprecated, and will be removed in 3.0. Use the Receiver interface instead and register as a Receiver with `Channel.setReceiver(Receiver r)`.

### 3.7.10. Peeking at a message

Instead of removing the next available message from the channel, `peek()` just returns a reference to the next message, but does not remove it. This is useful when one has to check the type of the next message, e.g. whether it is a regular message, or a view change. The signature of this method is not shown here, it is the same as for `receive()`.

## Note

The `peek()` method has also been deprecated, and will be removed in 3.0.

### 3.7.11. Getting the group's state

A newly joined member may wish to retrieve the state of the group before starting work. This is done with `getState()`. This method returns the state of one member (in most cases, of the oldest member, the coordinator). It returns true or false, depending on whether a valid state could be retrieved. For example, if a member is a singleton, then calling this method would always return false<sup>4</sup>.

The actual state is returned as the return value of one of the subsequent `receive()` calls, in the form of a `SetStateEvent` object. If `getState()` returned true, then a valid state (non-null) will be returned, otherwise a null state will be returned. Alternatively if an application uses `MembershipListener` (see Section 3.2.4 ) instead of pulling messages from a channel, the `getState()` method will be invoked and a copy of the current state should be returned. By the same token, setting a state would be accomplished by JGroups calling the `setState()` method of the state fetcher.

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.

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

```
channel=new JChannel();
channel.connect("TestChannel");
boolean rc=channel.getState(null, 5000);

...

Object state, copy;
Object ret=channel.receive(0);
if(ret instanceof Message)
    ;
else if(ret instanceof GetStateEvent) {
    copy=copyState(state); // make a copy so that other msgs don't change the state
    channel.returnState(Util.objectToByteBuffer(copy));
}
else if(ret instanceof SetStateEvent) {
    SetStateEvent e=(SetStateEvent)ret;
    state=e.getArg();
}
```

A `JChannel` has to be created whose stack includes the `STATE_TRANSFER` or `pbcaster.STATE_TRANSFER` protocols (see Chapter 5 ). Method `getState()` subsequently asks the channel to return the current state. If there is a current state (there may not be any other members in the group !), then true is returned. In this case, one of the subsequent `receive()` method invocations on the channel will return a `SetStateEvent` object which contains the current state. In this case, the caller sets its state to the one received from the channel.

Method `receive()` might return a `GetStateEvent` object, requesting the state of the member to be returned. In this case, *a copy of the current state should be made* and returned using `JChannel.returnState()` . It is important to a) synchronize access to the state when returning it since other accesses may modify it while it is being returned and

<sup>4</sup>A member will *never* retrieve the state from itself !

b) make a copy of the state since other accesses after returning the state may still be able to modify it ! This is possible because the state is not immediately returned, but travels down the stack (in the same address space), and a reference to it could still alter it.

### 3.7.12. Getting the state with a Receiver

As an alternative to handling the `GetStateEvent` and `SetStateEvent` events, and calling `Channel.returnState()`, a Receiver could be used. The example above would look like this:

```
class MyReceiver extends ReceiverAdapter {
    final Map m=new HashMap();
    public byte[] getState() {
        synchronized(m) { // so nobody else can modify the map while we serialize it
            byte[] state=Util.objectToByteBuffer(m);
            return state;
        }
    }

    public void setState(byte[] state) {
        synchronized(m) {
            Map new_m=(Map)Util.objectFromByteBuffer(state);
            m.clear();
            m.addAll(new_m);
        }
    }
}

channel=new JChannel(); // use default properties (has to include pbcast.STATE_TRANSFER p
channel.setReceiver(new MyReceiver());
channel.connect("TestChannel");
boolean rc=channel.getState(null, 5000);
```

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

- D calls `Channel.getState()`. The state will be retrieved from the oldest member, A
- `A.MyReceiver.getState()` is called. A returns a copy of its hashmap
- D: `getState()` returns true
- `D.MyReceiver.setState()` is called with the serialized state. D unserializes the state and sets it

### 3.7.13. Partial state transfer

Partial state transfer means that instead of transferring the entire state, we may want to transfer only a *substate*. For example, with HTTP session replication, a new node in a cluster may want to transfer only the state of a specific session, not *all* HTTP sessions. This can be done with either the pull or push model. The method to call would be `Channel.getState()`, including the ID of the substate (a string). In the pull model, `GetStateEvent` and `SetStateEvent` have an additional member, `state_id`, and in the push model, there are 2 additional `getState()` and `setState()` callbacks. The example below shows partial state transfer for the push model:

```

class MyReceiver extends ExtendedReceiverAdapter {
    final Map m=new HashMap();

    public byte[] getState() {
        return getState(null);
    }

    public byte[] getState(String substate_id) {
        synchronized(m) { // so nobody else can modify the map while we serialize it
            byte[] state=null;
            if(substate_id == null) {
                state=Util.objectToByteBuffer(m);
            }
            else {
                Object value=m.get(substate_id);
                if(value != null) {
                    return Util.objectToByteBuffer(value);
                }
            }
            return state;
        }
    }

    public void setState(byte[] state) {
        setState(null, state);
    }

    public void setState(String substate_id, byte[] state) {
        synchronized(m) {
            if(substate_id != null) {
                Object value=Util.objectFromByteBuffer(state);
                m.put(substate_id, value);
            }
            else {
                Map new_m=(Map)Util.objectFromByteBuffer(state);
                m.clear();
                m.addAll(new_m);
            }
        }
    }
}

channel=new JChannel(); // use default properties (has to include pbcast.STATE_TRANSFER p
channel.setReceiver(new MyReceiver());
channel.connect("TestChannel");
boolean rc=channel.getState(null, "MyID", 5000);

```

The example shows that the `Channel.getState()` method specifies the ID of the substate, in this case "MyID". The `getState(String substate_id)` method checks whether the substate ID is not null, and returns the substate pertaining to the ID, or the entire state if the substate\_id is null. The same goes for setting the substate: if `setState(String substate_id, byte[] state)` has a non-null substate\_id, only that part of the current state will be overwritten, otherwise (if null) the entire state will be overwritten.

### 3.7.14. Streaming state transfer

Streaming state transfer allows transfer of application (partial) state without having to load entire state into memory prior to sending it to a joining member. Streaming state transfer is especially useful if the state is very large (>1Gb), and use of regular state transfer would likely result in `OutOfMemoryException`. Streaming state transfer was introduced in JGroups 2.4. JGroups channel has to be configured with either regular or streaming state transfer.

The JChannel API that invokes state transfer (i.e. JChannel.getState(long timeout, Address member)) remains the same.

Streaming state transfer, just as regular byte based state transfer, can be used in both pull and push mode. Similarly to the current getState and setState methods of org.jgroups.MessageListener, the application interested in streaming state transfer in a push mode would implement streaming getState method(s) by sending/writing state through a provided OutputStream reference and setState method(s) by receiving/reading state through a provided InputStream reference. In order to use streaming state transfer in a push mode, existing ExtendedMessageListener has been expanded to include additional four methods:

```
public interface ExtendedMessageListener {

    /*non-streaming callback methods omitted for clarity*/

    void getState(OutputStream ostream);
    void getState(String state_id, OutputStream ostream);
    void setState(InputStream istream);
    void setState(String state_id, InputStream istream);

}
```

For a pull mode (when application uses channel.receive() to fetch events) two new event classes will be introduced:

- StreamingGetStateEvent
- StreamingSetStateEvent

These two events/classes are very similar to existing GetStateEvent and SetStateEvent but introduce a new field; StreamingGetStateEvent has an OutputStream and StreamingSetStateEvent has an InputStream.

The following code snippet demonstrates how to pull events from a channel, processing StreamingGetStateEvent and sending hypothetical state through a provided OutputStream reference. Handling of StreamingSetStateEvent is analogous to this example:

```
...
Object obj=channel.receive(0);
if(obj instanceof StreamingGetStateEvent) {
    StreamingGetStateEvent evt=(StreamingGetStateEvent)obj;
    OutputStream oos = null;
    try {
        oos = new ObjectOutputStream(evt.getArg());
        oos.writeObject(state);
        oos.flush();
    }
    catch (Exception e) {}
    finally {
        try {
            oos.close();
        }
        catch (IOException e) {
            System.err.println(e);
        }
    }
}
...
}
```

JGroups has a great flexibility with state transfer methodology by allowing application developers to implement both byte based and streaming based state transfers. Application can, for example, implement streaming and byte based state transfer callbacks and then interchange state transfer protocol in channel configuration to use either streaming or byte based state transfer. However, one cannot configure a channel with both state transfers at the same time and then in runtime choose which particular state transfer type to use.

### 3.7.15. 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 remove itself from the group membership. This is done (transparently for a channel user) by sending a leave request to the current coordinator. The latter will subsequently remove the channel's address from its local view and send the new view to all remaining members.

After a successful disconnect, the channel will be in the unconnected state, and may subsequently be re-connected to.

### 3.7.16. Closing a channel

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

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

It 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. Most of them do not even need a channel, all they need is a class that implements interface `Transport` (channels do). This enables them to work on any type of group transport that obeys this interface. Building blocks 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. Only the ones that are relevant for application programmers are discussed below.

### 4.1. PullPushAdapter

This class is a converter (or adapter, as used in [Gamma:1995] between the pull-style of actively receiving messages from the channel and the push-style where clients register a callback which is invoked whenever a message has been received. Clients of a channel do not have to allocate a separate thread for message reception.

A `PullPushAdapter` is always created on top of a class that implements interface `Transport` (e.g. a channel). Clients interested in being called when a message is received can register with the `PullPushAdapter` using method `setListener()`. They have to implement interface `MessageListener`, whose `receive()` method will be called when a message arrives. When a client is interested in getting view, suspicion messages and blocks, then it must additionally register as a `MembershipListener` using method `setMembershipListener()`. Whenever a view, suspicion or block is received, the corresponding method will be called.

Upon creation, an instance of `PullPushAdapter` creates a thread which constantly calls the `receive()` method of the underlying `Transport` instance, blocking until a message is available. When a message is received, if there is a registered message listener, its `receive()` method will be called.

As this class does not implement interface `Transport`, but merely uses it for receiving messages, an underlying object has to be used to send messages (e.g. the channel on top of which an object of this class resides). This is shown in Figure 4.1.

**Figure 4.1.** Class `PullPushAdapter`

As is shown, the thread constantly pulls messages from the channel and forwards them to the registered listeners. An application thus does not have to actively pull for messages, but the `PullPushAdapter` does this for it. Note however, that the application has to *directly* access the channel if it wants to *send* a message.

#### 4.1.1. Example

This section shows sample code for using a `PullPushAdapter`. The example has been shortened for readability (error handling has been removed).

```
public class PullPushTest implements MessageListener {
    Channel        channel;
    PullPushAdapter adapter;
    byte[]         data="Hello world".getBytes();
    String         props; // fetch properties

    public void receive(Message msg) {
        System.out.println("Received msg: " + msg);
    }

    public void start() throws Exception {
        channel=new JChannel(props);
        channel.connect("PullPushTest");
        adapter=new PullPushAdapter(channel);
        adapter.setListener(this);

        for(int i=0; i < 10; i++) {
            System.out.println("Sending msg #" + i);
            channel.send(new Message(null, null, data));
            Thread.currentThread().sleep(1000);
        }
        adapter.stop();
        channel.close();
    }

    public static void main(String args[]) {
        try {
            new PullPushTest().start();
        }
        catch(Exception e) { /* error */ }
    }
}
```

First a channel is created and connected to. Then an instance of `PullPushAdapter` is created with the channel as argument. The constructor of `PullPushAdapter` starts its own thread which continually reads on the channel. Then the `MessageListener` is set, which causes all messages received on the channel to be sent to `receive()`. Then a number of messages are sent via the channel to the entire group. As group messages are also received by the sender, the `receive()` method will be called every time a message is received. Finally the `PullPushAdapter` is stopped and the channel closed. Note that explicitly stopping the `PullPushAdapter` is not actually necessary, a closing the channel would cause the `PullPushAdapter` to terminate anyway.

Note that, compared to the pull-style example, push-style message reception is considerably easier (no separate thread management) and requires less code to program.

### Note

The `PullPushAdapter` has been deprecated, and will be removed in 3.0. Use a `Receiver` implementation instead. The advantage of the `Receiver`-based (push) model is that we save 1 thread.

## 4.2. MessageDispatcher

Channels are simple patterns to *asynchronously* send a receive messages. However, a significant number of com-

munication 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` offers a combination of the above pattern with other patterns. It provides synchronous (as well as asynchronous) message sending with request-response correlation, e.g. matching responses with the original request. It also offers push-style message reception (by internally using the `PullPushAdapter`).

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 send responses. `MessageDispatcher` allows an application to be both at the same time. To be able to serve requests, the `RequestHandler.handle()` method has to be implemented:

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

The `handle()` method is called any time a request is received. It must return a return value (must be serializable, but can be null) or throw an exception. The return value will be returned to the sender (as a null response, see below). The exception will also be propagated to the requester.

The two methods to send requests are:

```
public RspList castMessage(Vector dests, Message msg, int mode, long timeout);  
public Object sendMessage(Message msg, int mode, long timeout)  
    throws TimeoutException;
```

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

The `mode` parameter defines whether the message will be sent synchronously or asynchronously. The following values are valid (from `org.jgroups.blocks.GroupRequest`):

#### GET\_FIRST

Returns the first response received.

#### GET\_ALL

Waits for all responses (minus the ones from suspected members)

#### GET\_MAJORITY

Waits for a majority of all responses (relative to the group size)

#### GET\_ABS\_MAJORITY

Waits for the majority (absolute, computed once)

#### GET\_N

Wait for n responses (may block if n > group size)

#### GET\_NONE

Wait for no responses, return immediately (non-blocking). This make the call asynchronous.

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

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 {
    public boolean isReceived(Address sender);
    public int     numSuspectedMembers();
    public Vector  getResults();
    public Vector  getSuspectedMembers();
    public boolean isSuspected(Address sender);
    public Object  get(Address sender);
    public int     size();
    public Object  elementAt(int i) throws ArrayIndexOutOfBoundsException;
}
```

Method `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.

### 4.2.1. Example

This section describes 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)),
                GroupRequest.GET_ALL, 0);
            System.out.println("Responses:\n" +rsp_list);
        }
        channel.close();
        disp.stop();
    }

    public Object handle(Message msg) {
```

```

        System.out.println("handle(): " + msg);
        return new String("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.

Finally both the `MessageDispatcher` and channel are closed.

### 4.3. RpcDispatcher

This class is derived from `MessageDispatcher`. It allows a programmer to invoke remote methods in all (or single) group members and optionally wait for the return value(s). An application will typically create a channel and layer the `RpcDispatcher` building block on top of it, which allows it to dispatch 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 invoke remote method calls (unicast and multicast) the following methods are used (not all methods shown):

```

public RspList callRemoteMethods(Vector dests, String method_name,
                                int mode, long timeout);
public RspList callRemoteMethods(Vector dests, String method_name,
                                Object arg1, int mode, long timeout);
public Object callRemoteMethod(Address dest, String method_name,
                               int mode, long timeout);
public Object callRemoteMethod(Address dest, String method_name,
                               Object arg1, int mode, long timeout);

```

The family of `callRemoteMethods()` is invoked with a list of receiver addresses. If null, the method will be invoked in all group members (including the sender). Each call takes the name of the method to be invoked and the `mode` and `timeout` parameters, which are the same as for `MessageDispatcher`. Additionally, each method takes zero or more parameters: there are `callRemoteMethods()` methods with up to 3 arguments. As shown in the example above, the first 2 methods take zero and one parameters respectively.

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.

If a sender needs to use more than 3 arguments, it can use the generic versions of `callRemoteMethod()` and `callRemoteMethods()` which use a `MethodCall`<sup>5</sup> instance rather than explicit arguments.

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

(\* Update: these methods are deprecated; must use `MethodCall` argument now \*)

### 4.3.1. Example

The code below shows an example:

```
public class RpcDispatcherTest {
    Channel          channel;
    RpcDispatcher     disp;
    RspList           rsp_list;
    String            props; // set by application

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

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

        for(int i=0; i < 10; i++) {
            Util.sleep(100);
            rsp_list=disp.callRemoteMethods(null, "print",
                new Integer(i), GroupRequest.GET_ALL, 0);
            System.out.println("Responses: " +rsp_list);
        }
        channel.close();
        disp.stop();
    }

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

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

<sup>5</sup>See the Programmer's Guide and the Javadoc documentation for more information about this class.

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.

## 4.4. DistributedHashtable

A `DistributedHashtable` is derived from `java.util.Hashtable` and allows to create several instances of hashtables in different processes. All of these instances have exactly the same state at all times. When creating such an instance, a group name determines which group of hashtables 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 sent to the local copy.

Since both keys and values of a hashtable will be sent across the network as copies, both of them have to be serializable. This allows for example to register remote RMI objects with any local instance of a hashtable, which can subsequently be looked up by another process which can then invoke remote methods (remote RMI objects are serializable). Thus, a distributed naming and registration service can be built in just a couple of lines.

A `DistributedHashtable` allows to register for notifications, e.g. when a new item is set, or an existing one removed. All registered listeners will be 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).

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

## 4.5. ReplicatedHashtable

`ReplicatedHashtable` provides exactly the same methods as `DistributedHashtable`. However, it is implemented differently. Whereas the latter uses synchronous remote group method invocation (similar to `RpcDispatcher`), the former uses asynchronous communication to keep the replicas up-to-date.

## 4.6. DistributedTree

Similar to `DistributedHashtable` this class also provides replication of a data structure across multiple processes. However, a tree structure instead of a hashtable is replicated by `DistributedTree`. Updates are multicast to all group members reliably and in the same order using the underlying channel.

The tree consists of a root and zero or more child nodes. Each node can be either another subtree, or a leaf node. A node has a name and a value. The value can be any object that is serializable. A node in the tree is identified by concatenating all nodes from the root to it, separated with `'/'` characters, e.g.

`/a/b/c.`

New nodes can be added dynamically. Existing nodes (also entire subtrees) can be removed. Values can be at-

tached to an existing node. Whenever the tree is modified events will be sent for which listeners can register. Listeners have to implement interface `DistributedTreeListener`:

```
public interface DistributedTreeListener {
    void nodeAdded(String fqcn, Serializable element);
    void nodeRemoved(String fqcn);
    void nodeModified(String fqcn, Serializable old_element, Serializable new_element);
}
```

The methods provided by `DistributedTree` are listed below (not all methods shown):

```
public class DistributedTree {
    public void add(String fqcn);
    public void add(String fqcn, Serializable element);
    public void remove(String fqcn);
    public boolean exists(String fqcn);
    public Serializable get(String fqcn);
    public void set(String fqcn, Serializable element);
    public Vector getChildrenNames(String fqcn);
}
```

The two `add()` methods add a new node. The first method assigns no value to the node, whereas the second does. Note that it does not matter whether or not parent nodes exists: an addition of `"/a/b/c/d"` to a tree `"/a/b"` would create nodes `"/a/b/c"` and `"/a/b/c/d"`. However, if a value was given, it would be assigned only to the latter.

The `remove()` method removes a node from the tree. If the node is a subtree itself, all nodes under it will be removed recursively. E.g. the removal of `"/"` from `"/a/b"` would trigger 3 `nodeRemoved()` notifications: `"/a/b"`, `"/a"` and `"/"` (in this order)<sup>6</sup>.

The `exists()` method tests whether a given node exists in the tree.

The `get()` method returns either the value associated with the given node, or null if the node cannot be found or there is no value attached.

Method `set()` attaches a value to a given node. It fails if the node does not exist. Use `add(String, Serializable)` instead if the node should be created if not existent.

Method `getChildrenNames()` furnishes a list of the fully qualified names of all children nodes of a given node. This gives a programmer modest navigation possibilities within the tree.

There is a demo application in `org.jgroups.demos.DistributedTreeDemo`.

## 4.7. NotificationBus

This class provides notification sending and handling capability. Also, it allows an application programmer to maintain a local cache which is replicated by all instances. `NotificationBus` also sits on top of a channel, however it creates its channel itself, so the application programmers do not have to provide their own channel. Notification

<sup>6</sup>Assuming that these are the only nodes in the entire tree, e.g. `"/a"` has no other children



consumers can subscribe to receive notifications by calling `setConsumer()` and implementing interface `NotificationBus.Consumer`:

```
public interface Consumer {
    void      handleNotification(Serializable n);
    Serializable getCache();
    void      memberJoined(Address mbr);
    void      memberLeft(Address mbr);
}
```

Method `handleNotification()` is called whenever a notification is received from the channel. A notification is any object that is serializable. Method `getCache()` is called when someone wants to retrieve our state; the state can be returned as a serializable object. The `memberJoined()` and `memberLeft()` callbacks are invoked whenever a member joins or leaves (or crashes).

The most important methods of `NotificationBus` are:

```
public class NotificationBus {
    public void setConsumer(Consumer c);
    public void start() throws Exception;
    public void stop();
    public void sendNotification(Serializable n);
    public Serializable getCacheFromCoordinator(long timeout, int max_tries);
    public Serializable getCacheFromMember(Address mbr, long timeout, int max_tries);
}
```

Method `setConsumer()` allows a consumer to register itself for notifications.

The `start()` and `stop()` methods start and stop the `NotificationBus`.

Method `sendNotification()` sends the serializable object given as argument to all members of the group, invoking their `handleNotification()` methods on reception.

Methods `getCacheFromCoordinator()` and `getCacheFromMember()` provide functionality to fetch the group state from the coordinator (first member in membership list) or any other member (if its address is known). They take as arguments a timeout and a maximum number of unsuccessful attempts until they return null. Typically one of these methods would be called just after creating a new `NotificationBus` to acquire the group state. Note that if these methods are used, then the consumers must implement `Consumer.getCache()`, otherwise the two methods above would always return null.

## 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 stacks: 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).

The `Draw2Channels` demo program (in the `org.jgroups.demos` package) demonstrates how to use two different channels.

### 5.2. Using the Multiplexer to run multiple building blocks over the same channel

In JBoss we have multiple JGroups channels, one for each application (e.g. `JBossCache`, `ClusterPartition` etc).

The goal of the Multiplexer is to combine all stacks with the *same* configuration into one, and have multiple services on top of that same channel.

To do this, we have to introduce multiplexing and demultiplexing functionality, ie. each service will have to have a unique service ID (a string), and when sending a message, the message has to be tagged with that ID. When receiving a message, it will be dispatched to the right destination service based on the ID attached to the message. We require special handling for `VIEW` and `SUSPECT` messages: those need to be dispatched to `*all*` services. State transfer also needs to be handled specially, here we probably have to use thread locals, or change the API (TBD).

When deployed into JBoss, the Multiplexer will be exposed as an MBean, and all services that depend on it will be deployed with dependency injection on the Multiplexer. Of course, the old configuration will still be supported.

The config of the Multiplexer is done via a config file, which lists a number of stacks, each keyed by a name, e.g. "udp", "tcp", "tcp-nio" etc. See `./conf/stacks.xml` for an example. An app is configured with the name of a stack, e.g. "udp", and a reference to the Multiplexer MBean. It will get a proxy channel through which all of its communication will take place. The proxy channel (MuxChannel) will mux/demux messages to the real JGroups channel.

The advantage of the Multiplexer is that we can reduce  $N$  channels into  $M$  where  $M < N$ . This means fewer threads, therefore fewer context switches, less memory consumption and easier configuration and better support.

### 5.2.1. The Multiplexer API

The Multiplexer is actually a JChannelFactory, which is configured with a reference to an XML configuration file, and has a few additional methods to get a Channel. The channel returned is actually an instance of MuxChannel, which transparently forwards all invocations to the underlying JChannel, and performs multiplexing and demultiplexing. Multiple MuxChannels can share the same underlying JChannel, and each message sent by a service over the MuxChannel will add the service's ID to the message (as a header). That ID is then used to demultiplex the message to the correct MuxChannel when received.

The methods of the JChannelFactory are:

```
public Channel createMultiplexerChannel(String stack_name, String id) throws Exception;
public Channel createMultiplexerChannel(String stack_name, String id, boolean register_for_state_transfer);
```

The `stack_name` parameter refers to a channel configuration defined in a separate file (see below).

The `id` parameter is the service ID and has to be unique for all services sitting on the same channel. If an ID is used more than once, when trying to call `createMultiplexerChannel()`, an exception will be thrown.

the `register_for_state_transfer` and `substate_id` parameters are discussed below (in Section 5.2.1.1).

The `stack_name` parameter is a reference to a stack, for example defined in `stacks.xml`. A shortened version of `stacks.xml` is shown below:

```
<protocol_stacks>
  <stack name="fc-fast-minimalthreads" description="Flow control, no up or down threads">
    <config>
      <UDP mcast_port="45566" enable_bundling="true"/>
      ...
      <pbcast.STATE_TRANSFER down_thread="false" up_thread="false"/>
    </config>
  </stack>

  <stack name="sequencer" description="Totally ordered multicast using a sequencer">
    <config>
      // config
    </config>
  </stack>

  <stack name="tcp" description="Using TCP as transport">
```

```

        <config>
          <TCP start_port="7800" loopback="true" send_buf_size="100000" recv_buf_size="100000" />
          <TCPPING timeout="3000" initial_hosts="localhost[7800]" port_range="3" num_initial_hosts="1" />
          <FD timeout="2000" max_tries="4" />
          <VERIFY_SUSPECT timeout="1500" down_thread="false" up_thread="false" />
          <pbcast.NAKACK gc_lag="100" retransmit_timeout="600,1200,2400,4800" />
          <pbcast.STABLE stability_delay="1000" desired_avg_gossip="20000" down_thread="false" up_thread="false" />
          <VIEW_SYNC avg_send_interval="60000" down_thread="false" up_thread="false" />
          <pbcast.GMS print_local_addr="true" join_timeout="5000" shun="true" />
        </config>
      </stack>

      <stack name="discovery" description="Simple UDP-only stack for discovery">
        <config>
          <UDP mcast_port="7609"
              use_incoming_packet_handler="false"
              mcast_addr="228.15.15.15"
              use_outgoing_packet_handler="false"
              ip_ttl="32" />
        </config>
      </stack>
    </protocol_stacks>

```

This file defines 4 configurations: `fc-fast-minimalthreads`, `sequencer`, `tcp` and `discovery`. The first service to call `JChannelFactory.createMultiplexerChannel()` with a `stack_name` of `"tcp"` will create the `JChannel` with the `"tcp"` configuration, all subsequent method calls for the *same* `stack_name` (`"tcp"`) will simply get a `MuxChannel` which has a reference to the same underlying `JChannel`. When a service closes a `MuxChannel`, the underlying `JChannel` will only be closed when there are no more `MuxChannels` referring to it.

For more information on Multiplexing refer to `JGroups/doc/design/Multiplexer.txt`

#### 5.2.1.1. Batching state transfers

Note that this feature is currently *not* used in `JBoss`, because `JBoss` doesn't call all `create()` methods of all dependent beans first, and then all `start()` methods. The call sequence is indeterministic unless all dependent beans are defined in the same XML file, which is unrealistic. We're looking into using a barrier service to provide the guarantee that all `create()` methods are called before all `start()` methods, possibly in `JBoss 5`.

When multiple services are sharing a `JChannel`, and each of the services requires state transfer at a different time, then we need `FLUSH` (see `./doc/design/PartialStateTransfer.txt` for a description of the problem). `FLUSH` is also called the stop-the-world model, and essentially stops everyone in a group from sending messages until the state has been transferred, and then everyone can resume again.

#### Note

The 2.3 release of `JGroups` will *not* have the `FLUSH` protocol integrated, so state transfer for the Multiplexer might be incorrect. 2.4 will have `FLUSH`, so that situation will be corrected. The main reason for putting Multiplexing into 2.3 is that people can start programming against the API, and then use it when `FLUSH` is available.

When multiple services share one `JChannel`, then we have to run the `FLUSH` protocol for every service which requires state, so if we have services A, B, C, D and E running on top of a `JChannel J`, and B,C and E require state, then the `FLUSH` protocol has to be run 3 times, which slows down startup (e.g.) of `JBoss`.

To remedy this, we can *batch* state transfers, so that we suspend everyone from sending messages, then fetch the

states for B, C and E at once, and then resume everyone. Thus, the FLUSH protocol has to be run only once.

To do this, a service has to register with the JChannelFactory when creating the MuxChannel, and know that getState() will be a no-op until the last registered application has called getState(). This works as follows:

- B, C and D register for state transfer
- B calls MuxChannel.getState(). Nothing happens.
- D calls MuxChannel.getState(). Nothing happens.
- E calls MuxChannel.getState(). Now everyone who registered has called getState() and therefore we transfer the state for B, C and E (using partial state transfer).
- At this point B, C and D's setState() will be called, so that they can set the state.

The code below (a snippet from MultiplexerTest) shows how services can register for state transfers. In an MBean (JBoss) environment, the registration could be done in the create() callback, and the actual getState() call in start().

```
public void testStateTransferWithRegistration() throws Exception {
    final String STACK_NAME="fc-fast-minimalthreads";
    Channel ch1, ch2;
    ch1=factory.createMultiplexerChannel(STACK_NAME, "c1", true, null); // register for (entire) state transfer
    ch1.connect("bla"); // will create a new JChannel

    ch2=factory.createMultiplexerChannel(STACK_NAME, "c2", true, null); // register for (entire) state transfer
    ch2.connect("bla"); // will share the JChannel created above (same STACK_NAME)

    boolean rc=ch1.getState(null, 5000); // this will *not* trigger the state transfer protocol
    rc=ch2.getState(null, 5000); // only *this* will trigger the state transfer
}
```

The example above shows that 2 services ("c1" and "c2") share a common JChannel because they use the same stack\_name (STACK\_NAME). It also shows that only the second getState() invocation will actually transfer the 2 states (for "c1" and "c2").

## 5.2.2. Service views

When we have multiple service running on the same channel, then some services might get redeployed or stopped independently from the other services. So we might have a situation where we have services S1, S2 and S3 running on host H1, but on host H2, only services S2 and S3 are running.

The *cluster view* is {H1, H2}, but the *service views* are:

- S1: {H1}
- S2: {H1, H2}
- S3: {H1, H2}

This can also be seen as ordered by hosts:

- H1: {S1, S2, S3}
- H2: {S2, S3}

So here we host H1 running services S1, S2 and S3, whereas H2 is only running S2 and S3. S1 might be in the process of being redeployed on H2, or is simply not running.

A service view is essentially a list of nodes of a cluster on which a given service S is currently running. Service views are always subsets of cluster views. Here's a reason we need service views: consider the example above. Let's say service S1 on H1 wants to make a cluster-wide method invocation on all instances of S1 running on any host. Now, S1 is only running on H1, therefore we have to make the invocation only on S1. However, if we took the cluster view rather than the service view, the invocation would be across H1 and H2, and we'd be waiting for a response from the (non-existent) service S1 on H2 forever !

So, by default, calling `MuxChannel.getView()` will return the service view rather than the cluster view. The cluster view can be retrieved calling `MuxChannel.getClusterView()`.

There are example unit tests in `MultiplexerTest` and `MultiplexerViewTest`. The latter tests service views versus cluster views.

## 5.3. Transport protocols

A *transport protocol* refers to the protocol at the bottom of the protocol stack which is responsible for sending and receiving messages to/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>
  <UDP
    mcast_addr="${jgroups.udp.mcast_addr:228.10.10.10}"
    mcast_port="${jgroups.udp.mcast_port:45588}"
    discard_incompatible_packets="true"
    max_bundle_size="60000"
    max_bundle_timeout="30"
    ip_ttl="${jgroups.udp.ip_ttl:2}"
    enable_bundling="true"
    use_concurrent_stack="true"
    thread_pool.enabled="true"
    thread_pool.min_threads="1"
    thread_pool.max_threads="25"
    thread_pool.keep_alive_time="5000"
    thread_pool.queue_enabled="false"
    thread_pool.queue_max_size="100"
    thread_pool.rejection_policy="Run"
    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 timeout="10000" max_tries="5" shun="true"/>
<VERIFY_SUSPECT timeout="1500" />
<pbcast.NAKACK
        use_mcast_xmit="false" gc_lag="0"
        retransmit_timeout="300,600,1200,2400,4800"
        discard_delivered_msgs="true"/>
<UNICAST timeout="300,600,1200,2400,3600"/>
<pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000"
        max_bytes="400000"/>
<pbcast.GMS print_local_addr="true" join_timeout="3000"
        shun="false"
        view_bundling="true"/>
<FC max_credits="20000000"
        min_threshold="0.10"/>
<FRAG2 frag_size="60000" />
<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, TCP\_NIO and TUNNEL.

#### PING

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 subgroups back into one group, kicks in after a cluster partition.

#### FD\_SOCK

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

#### FD

Failure detection based on heartbeats and are-you-alive messages (in a ring form between members). 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

#### 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.

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. 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/receives multicast messages. The channel's address will be the address and port number of the *unicast* socket.

#### 5.3.1.1. Using UDP and plain IP multicasting

A protocol stack with UDP as transport protocol is typically used with groups whose members run on the same host or are distributed across a LAN. Before running such a stack a programmer has to ensure that IP multicast is enabled across subnets. It is often the case that IP multicast is not enabled across subnets. Refer to section Section 2.8 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 other transports such as TCP.

#### 5.3.1.2. 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<sup>7</sup> (UDP) and to access a well-known server ( *GossipRouter* ) for initial membership information (PING).

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* .

To configure PING to access a GossipRouter instead of using IP multicast the following properties have to be set:

gossip\_host

The name of the host on which GossipRouter is started

gossip\_port

The port on which GossipRouter is listening

gossip\_refresh

The number of milliseconds to wait until refreshing our address entry with the GossipRouter

<sup>7</sup>Although not as efficient (and using more bandwidth), it is sometimes the only possibility to reach group members.



Before any members are started the GossipRouter has to be started, e.g.

```
java org.jgroups.stack.GossipRouter -port 5555 -bindaddress localhost
```

This starts the GossipRouter on the local host on port 5555. The GossipRouter is essentially a lookup service for groups and members. It is a process that runs on a well-known host and port and accepts GET(group) and REGISTER(group, member) requests. The REGISTER request registers a member's address and group with the GossipRouter. The GET request retrieves all member addresses given a group name. Each member has to periodically (*gossip\_refresh*) re-register their address with the GossipRouter, otherwise the entry for that member will be removed (accommodating for crashed members).

The following example shows how to disable the use of IP multicasting and use a GossipRouter instead. Only the bottom two protocols are shown, the rest of the stack is the same as in the previous example:

```
<UDP ip_mcast="false" mcast_addr="224.0.0.35" mcast_port="45566" ip_ttl="32"
    mcast_send_buf_size="150000" mcast_rcv_buf_size="80000" />
<PING gossip_host="localhost" gossip_port="5555" gossip_refresh="15000"
    timeout="2000" num_initial_members="3" />
```

The property *ip\_mcast* is set to *false* in UDP and the gossip properties in PING define the GossipRouter to be on the local host at port 5555 with a refresh rate of 15 seconds. If PING is parameterized with the GossipRouter's address and port, then gossiping is enabled, otherwise it is disabled. If only one parameter is given, gossiping will be *disabled*.

Make sure to run the GossipRouter before starting any members, otherwise the members will not find each other and each member will form its own group<sup>8</sup>.

### 5.3.2. TCP

TCP is a replacement of UDP as bottom layer in cases where IP Multicast based on UDP is not desired. This may be the case when operating over a WAN, where routers will discard IP MCAST. As a rule of thumb UDP is used as transport for LANs, whereas TCP is used for WANs.

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

```
<TCP start_port="7800" />
<TCPPING timeout="3000"
    initial_hosts="{jgroups.tcpping.initial_hosts:localhost[7800],localhost[7801]}"
    port_range="1"
    num_initial_members="3" />
<VERIFY_SUSPECT timeout="1500" />
<pbcast.NAKACK
    use_mcast_xmit="false" gc_lag="0"
    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"
```

<sup>8</sup>This can actually be used to test the MERGE2 protocol: start two members (forming two singleton groups because they don't find each other), then start the GossipRouter. After some time, the two members will merge into one group

```
shun="true"
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

Since TCP already offers some of the reliability guarantees that UDP doesn't, some protocols (e.g. FRAG and UNICAST) are not needed on top of TCP.

When using TCP, each message to the group 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:

- PING with GossipRouter: same solution as described in Section 5.3.1.2 . The *ip\_mcast* property has to be set to *false* . GossipRouter has to be started before the first member is started.
- TCPPING: uses a list of well-known group members that it solicits for initial membership
- TCPGOSSIP: essentially the same as the above PING<sup>9</sup> . The only difference is that TCPGOSSIP allows for multiple GossipRouters instead of only one.

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

### 5.3.2.1. Using TCP and TCPPING

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

```
<TCP start_port="7800" /> +
<TCPPING initial_hosts="HostA[7800],HostB[7800]" port_range="5"
    timeout="3000" num_initial_members="3" />
```

The concept behind TCPPING is that no external daemon such as GossipRouter is needed. Instead some selected<sup>9</sup>PING and TCPGOSSIP will be merged in the future.

group members assume the role of well-known hosts from which 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 *start\_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 - 7804 . Assuming that at least one of *HostA* or *HostB* is up, a response will be received. To be absolutely sure to receive a response all the hosts on which members of the group will be running can be added to the configuration string.

### 5.3.2.2. Using TCP and TCPGOSSIP

As mentioned before TCPGOSSIP is essentially the same as PING with properties *gossip\_host* , *gossip\_port* and *gossip\_refresh* set. However, in TCPGOSSIP these properties are called differently as shown below (only the bottom two protocols are shown):

```
<TCP />
<TCPGOSSIP initial_hosts="localhost[5555],localhost[5556]" gossip_refresh_rate="10000"
            num_initial_members="3" />
```

The *initial\_hosts* properties combines both the host and port of a GossipRouter, and it is possible to specify more than one GossipRouter. In the example there are two GossipRouters at ports 5555 and 5556 on the local host. Also, *gossip\_refresh\_rate* defines how many milliseconds to wait between refreshing the entry with the GossipRouters.

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.

## 5.3.3. TUNNEL

### 5.3.3.1. Using TUNNEL to tunnel a firewall

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 as bottom-most layer in the stack, plus either PING or TCPGOSSIP, as shown below (only the bottom two protocols shown):

```
<TUNNEL router_host="localhost" router_port="12001" />
<TCPGOSSIP initial_hosts="localhost[12001]" gossip_refresh_rate="10000"
  num_initial_members="3" />
```

TCPGOSSIP uses the GossipRouter (outside the firewall) at port 12001 to register its address (periodically) and to retrieve the initial membership for its group.

TUNNEL establishes a TCP connection to the *GossipRouter* process (also 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. 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 the local host at port 12001. Both TUNNEL and TCPGOSSIP (or PING) access the same GossipRouter.

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 connection 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<sup>10</sup>.

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

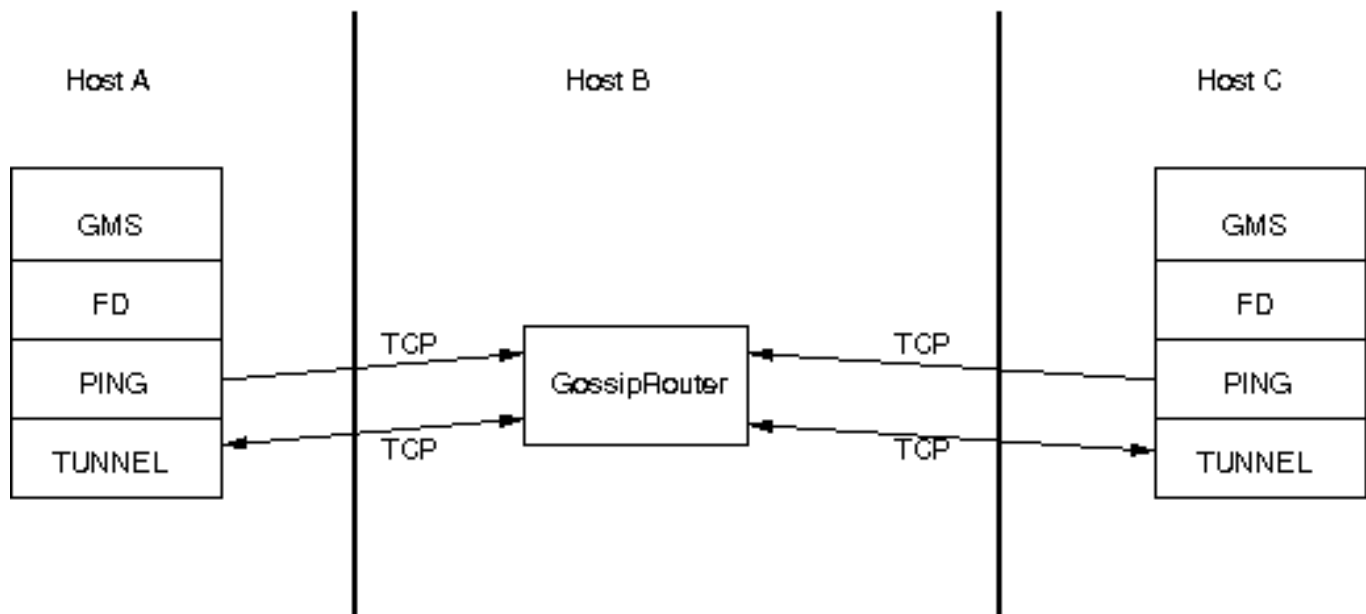
1. Check that a TCP port (e.g. 12001) is enabled in the firewall for outgoing traffic
2. Start the GossipRouter:

```
start 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 5.1 .

<sup>10</sup>To do so, GossipRouter has to maintain a table between groups, member addresses and TCP connections.



**Figure 5.1. 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 TCP-`GOSSIP` 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 group, GossipRouter looks up all group 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, the central GossipRouter process constitute a single point of failure (if host B crashes)<sup>11</sup>, second, 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, third, 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.

## Note

There will be a major overhaul of GossipRouter/TUNNEL in 2.6, where we'll streamline the connection table and possibly introduce new functionality such as connecting to multiple GossipRouters, connecting to both IP multicasting and TCP based clients etc.

<sup>11</sup> Although multiple GossipRouters could be started

## 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 5.2. 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.

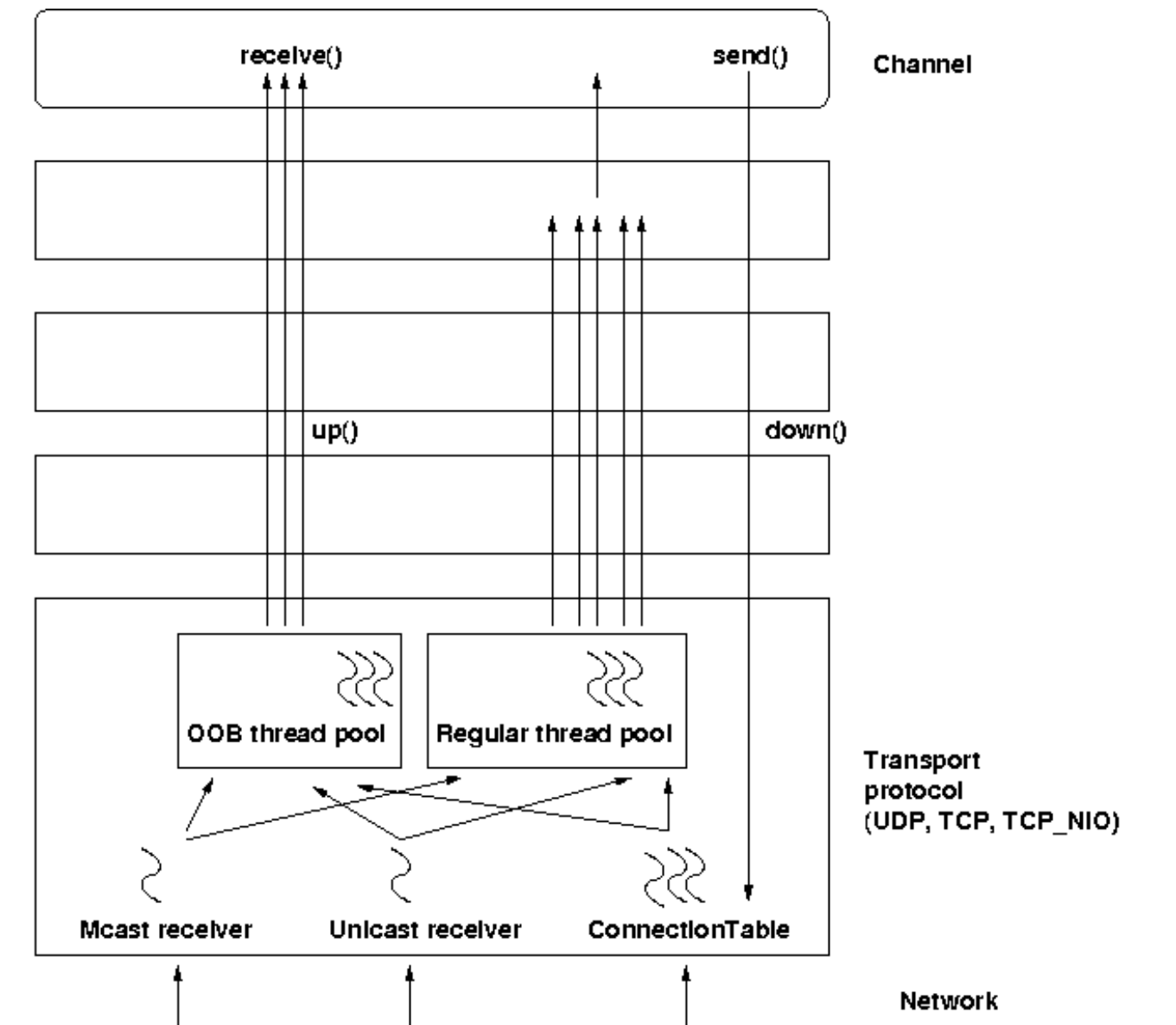


Figure 5.2. The concurrent stack

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
    mcast_addr="228.10.10.10"
    mcast_port="45588"

    use_concurrent_stack="true"

    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 concurrent stack can be completely eliminated by setting `use_concurrent_stack` to false. (Note that this attribute might be removed in a future release).

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

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

**Table 5.1. Attributes of thread pools**

Name	Description
enabled	Whether or 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 removed from the pool



Name	Description
queue_enabled	Whether or 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, 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.
thread_naming_pattern	Determines how threads are named that are running from thread pools in concurrent stack. Valid values include any combination of "pcsl" letters, where p includes a pool name in a compound thread name, c includes cluster name, s includes sender address, and finally l includes local address of the channel. By default, thread names are composed of corresponding pool name and a thread id.

### 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. 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.

### 5.5. Misc

#### 5.5.1. Shunning

Let's say we have 4 members in a group: {A,B,C,D}. When a member (say D) is expelled from the group, e.g. because it didn't respond to are-you-alive messages, and later comes back, then it is shunned. Shunning causes a member to leave the group and re-join, if this is enabled on the Channel. To enable automatic re-connects, the `AUTO_RECONNECT` option has to be set on the Channel:

```
channel.setOpt(Channel.AUTO_RECONNECT, Boolean.TRUE);
```

To enable shunning, set `FD.shun` and `GMS.shun` to true.

Let's look at a more detailed example. Say member D is overloaded, and doesn't respond to are-you-alive messages (done by the failure detection (FD) protocol). It is therefore suspected and excluded. The new view for A, B and C will be {A,B,C}, however for D the view is still {A,B,C,D}. So when D comes back and sends messages to the group, or any individual member, those messages will be discarded, because A,B and C don't see D in their view. D is shunned when A,B or C receive an are-you-alive message from D, or D shuns itself when it receives a view which doesn't include D.

So shunning is always a unilateral decision. However, things may be different if all members exclude each other from the group. For example, say we have a switch connecting A, B, C and D. If someone pulls all plugs on the switch, or powers the switch down, then A, B, C and D will all form singleton groups, that is, each member thinks it's the only member in the group. When the switch goes back to normal, then each member will shun everybody else (a real shun fest :-)). This is clearly not desirable, so in this case shunning should be turned off:

```
<FD timeout="2000" max_tries="3" shun="false"/> ... <pbcast.GMS join_timeout="3000" shun="false"/>
```

## 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 this 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);
        handler.start(); // requires separate thread as we don't want to block JGroups
    }
}

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("I (" + local_addr + ") am not member of the new primary p
        "), will re-acquire the state");
        try {
            ch.getState(null, 30000);
        }
        catch(Exception ex) {
        }
    }
    else {
        System.out.println("I (" + local_addr + ") am member of the new primary parti
        "), 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 reacquires 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 modifications. 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 coordinator 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 re-transmit 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)<sup>12</sup>. 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.

`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.

<sup>12</sup>Note that `block()` will be called in a Receiver when the flush is about to start and `unblock()` will be called when it ends

## List of Protocols

This section is work in progress; we strive to update the documentation as we make changes to the code.

The most important properties are described on the wiki [1]. The idea is that users take one of the predefined configurations (shipped with JGroups) and make only minor changes to it.

For each protocol define:

- Properties provided
- Required services
- Provided services
- Behavior

### 6.1. Transport

#### 6.1.1. UDP

#### 6.1.2. TCP

#### 6.1.3. TCP\_NIO

#### 6.1.4. TUNNEL

#### 6.1.5. JMS

#### 6.1.6. LOOPBACK

[1] <http://www.jboss.org/wiki/Wiki.jsp?page=JGroups>

## 6.2. Initial membership discovery

### 6.2.1. PING

### 6.2.2. TCPPING

### 6.2.3. TCPGOSSIP

### 6.2.4. MPING

## 6.3. Merging after a network partition

### 6.3.1. MERGE2, MERGE3, MERGEFAST

## 6.4. 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.

### 6.4.1. 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 neighbor. 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 watches 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 6.1. Properties**



Name	Description
timeout	Max number of ms to wait for a response, e.g. timeout="2500"
max_tries	Max number of missed responses until a member is declare suspected., e.g. max_tries="5"
shun	<p>Once a member is excluded from the group, and then rejoins (e.g. because it didn't crash, but was just slow, or a router that had crashed came back), it will be excluded (shunned) and then has to rejoin. JGroups allows to configure itself such that shunning leads to automatic rejoins and state transfer (default in JBoss), e.g. shun="true".</p> <p>Automatic rejoins can be enabled by setting channel option AUTO_RECONNECT to true:</p> <pre>channel.setOpt(Channel.AUTO_RECONNECT, true);</pre> <p>Same for automatically fetching the state after automatic reconnection:</p> <pre>channel.setOpt(Channel.AUTO_FETCH_STATE_TRANSFER, true);</pre>

### 6.4.2. 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 re-set 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  $3 * 3s == 9$  seconds.

**Table 6.2. Properties**

Name	Description
timeout	Max number of milliseconds until a member is suspected
interval	Interval (in milliseconds) to multicast heartbeat messages to the cluster

Name	Description
msg_counts_as_heartbeat	If this is true, we treat traffic from P as if P sent a heartbeat, ie. we set the timestamp for P to the current time. Default is true
shun	Once a member is excluded from the group, and then rejoins (e.g. because it didn't crash, but was just slow, or a router that had crashed came back), it will be excluded (shunned) and then has to rejoin. JGroups allows to configure itself such that shunning leads to automatic rejoins and state transfer (default in JBoss). , e.g. shun="true"

### 6.4.3. FD\_SIMPLE

### 6.4.4. 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.

**Table 6.3. Properties**

Name	Description
cmd	The command to be executed, e.g. "/sbin/ping" (or "ping" if found on path)
verbose	Whether or not to show the output of the command. Valid: "true" or "false"

### 6.4.5. 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 6.4. Properties**

Name	Description
bind_addr	The network interface to be used for sending ICMP

Name	Description
	packets, e.g. <code>bind_addr="192.16.8.0.2"</code>

### 6.4.6. FD SOCK

Failure detection protocol based on a ring of TCP sockets created between group members. Each member in a group connects to its neighbor (last member connects to first) thus forming a ring. Member B is suspected when its neighbor A detects abnormally closed 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 6.5. Properties**

Name	Description
<code>bind_addr</code>	The network interface to be used for sending ICMP packets, e.g. <code>bind_addr="192.16.8.0.2"</code>

### 6.4.7. VERIFY\_SUSPECT

## 6.5. Reliable message transmission

### 6.5.1. 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<sup>13</sup> 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.

<sup>13</sup> 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.

**Table 6.6. Properties**

Name	Description
retransmit_timeout	<p>A comma-separated list of milliseconds, e.g. 100,200,400,800,1600. We ask for retransmission of a given message 100ms after it wasn't received, then 200ms, and so on, until we're at 1600ms. From then on, we will send retransmit requests for that message every 100ms, until the message is received, or the original sender crashed.</p> <p>Example: retransmit_timeout="300,600,1200,2400"</p>
exponential_backoff	<p>Value in milliseconds. If greater than 0, exponential backoff for retransmission is enabled. The value is then the initial value, and we'll double it every time we ask for retransmission, until a max value of 15000ms.</p> <p>If the value is greater than 0, exponential backoff is enabled and retransmit_timeout will be ignored.</p> <p>Note that this property is experimental, and may be removed at any time.</p> <p>Example: exponential_backoff="30"</p>
use_stats_for_retransmission	<p>Boolean. If true, we ignore both retransmit_timeout and exponential_backoff, and use statistics that we collect during retransmission to compute the ideal retransmission times, based on actual retransmission times.</p> <p>Note that this property is experimental, and may be removed at any time.</p> <p>Example: use_stats_for_retransmission="true"</p>
use_mcast_xmit	<p>When we get a retransmission request from P for a message M, then we send the retransmitted M to P. However, assuming that many nodes lost M, we might as well send M to the entire cluster, so that we can satisfy many retransmit requests at the same time.</p> <p>Setting this option to true only makes sense for an IP multicast capable transport (e.g. UDP), where we send the retransmitted message one time. Otherwise, e.g. if we use TCP, we send the message N-1 times, one time for each node.</p>

Name	Description
	Example: <code>use_mcast_xmit="true"</code>
use_mcast_xmit_req	<p>Similar to <code>use_mcast_xmit</code>, but for requests: if enabled, we'll send a retransmit request via a multicast.</p> <p><b>Note</b></p> <p>Setting this option to true only makes sense for an IP multicast capable transport (e.g. UDP), where we send the retransmit requests one time. Otherwise, e.g. if we use TCP, we send the message N-1 times, one time for each node !</p> <p>Example: <code>use_mcast_xmit_req="true"</code></p>
xmit_from_random_member	<p>Instead of sending all retransmit requests to the original sender of a message, we could also pick a random member of the cluster. Setting <code>xmit_from_random_member="true"</code> does this. The advantage is that the retransmission load is distributed more equally across the cluster. The down side is that a random member may or may not yet have received a given message, so it may not be able to satisfy the retransmission request. In this case, it may take a few retransmission requests to different members in the cluster to finally get the requested message.</p> <p>Note that <code>discard_delivered_msgs</code> must be set to false if this option is enabled.</p> <p>Example: <code>xmit_from_random_member="true"</code></p>
discard_delivered_msgs	<p>If set to true, messages received from other members are <i>not</i> buffered until stability purges them, but instead discarded immediately. This means that retransmission requests can only be satisfied by the original sender (and <code>xmit_from_random_member</code> won't work). However, since we don't have to wait for stability to kick in and purge messages seen by everyone, we conserve memory.</p> <p>Example: <code>discard_delivered_msgs="true"</code></p>
max_xmit_buf_size	<p>Number of entries to keep in the retransmission tables. Since messages are buffered in the retransmission tables, we have to wait for a given message M to be delivered until all messages preceding M have been delivered as well. Using <code>max_xmit_buf_size</code> makes the retransmission tables <i>bounded</i>, and older</p>

Name	Description
	<p>messages will get discarded if they haven't been received when <code>max_xmit_buf_size</code> elements are exceeded.</p> <p>Note that setting this property to true will cause <i>message loss</i> in the case where a message is lost and not retransmitted because the retransmission buffer is full !</p>

## 6.5.2. SMACK

## 6.5.3. UNICAST

# 6.6. Fragmentation

## 6.6.1. FRAG and FRAG2

# 6.7. Ordering (FIFO covered by NAKACK)

## 6.7.1. Total Order (SEQUENCER)

# 6.8. 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
```

## 6.8.1. pbcast.GMS

**Table 6.7. Properties**

Name	Description
join_timeout	Number of milliseconds to wait for a JOIN response from the coordinator, until we send a new JOIN request. Default=5000
join_retry_timeou	Number of ms to wait before sending a new JOIN request
leave_timeout	Number of ms to wait until a LEAVE response has been received from the coordinator. Once this time has elapsed, we leave anyway.
shun	
merge_leader	
print_local_addr	Whether or not to print to stdout the local address of a newly started member. Default is "true". Example: <div style="border: 1px solid black; padding: 10px; margin-top: 10px;"> <pre>----- GMS: address is 192.1 -----</pre> </div>
merge_timeout	
digest_timeout	
view_ack_collection_timeout	
resume_task_timeout	
disable_initial_coord	
handle_concurrent_startup	Default: "true". Handles concurrent starting of N <i>initial</i> members. Setting it to false is only used for unit tests, where the correctness of the subsequent merge is tested, we don't recommend setting it to false.
num_prev_mbrs	
use_flush	
flush_timeout	
reject_join_from_existing_member	If we receive a JOIN request from P and P is already in the current membership, then we send back a JOIN

Name	Description
	response with an error message when this property is set to true (Channel.connect() will fail). Otherwise, we return the current view
view_bundling	Whether to enable view bundling (default is true). View bundling means that multiple view-affecting requests to GNS, such as JOIN, LEAVE or SUSPECT, are bundled for a number of milliseconds in order to avoid having to generate one view per request.  This is especially interesting if we have many members joining or leaving a group at the same time.
max_bundling_time	Max number of milliseconds to wait for subsequent JOIN/LEAVE/SUSPECT requests (default is 50). Therefore, when 5 JOIN or LEAVE requests are received within 50ms, only 1 view will be generated

#### 6.8.1.1. Disabling the initial coordinator

Consider the following situation: a new member wants to join a group. The procedure 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 protocol
- 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 3 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 MERGE(2) protocol. This doesn't actually prevent multiple initial coordinators, but rectifies the problem by merging different subgroups back into one. Note that this involves state merging which needs to be done by the application.
3. (new) Prevent members from becoming coordinator on initial startup. This solution is applicable when we know which member is going to be the initial coordinator of a fresh group. We don't care about afterwards, then coordinatorship can migrate to another member. In this case, we configure the member that is always



supposed to be started first with `disable_initial_coord=false` (the default) and all other members with `disable_initial_coord=true`. This works as described below.

When the initial membership is received, and is null, and the property `disable_initial_coord` is true, then we just continue in the loop and retry receiving the initial membership (until it is non-null). If the property is false, we are allowed to become coordinator, and will do so. Note that - if a member is started as first member of a group - but its property is set to true, then it will loop until another member whose `disable_initial_coord` property is set to false, is started.

## **6.9. Security**

### **6.9.1. ENCRYPT**

### **6.9.2. AUTH**

## **6.10. State Transfer**

### **6.10.1. pbcast.STATE\_TRANSFER**

### **6.10.2. pbcast.STREAMING\_STATE\_TRANSFER**

#### **6.10.2.1. Overview**

In order to transfer application state to a joining member of a group `pbcast.STATE_TRANSFER` has to load entire state into memory and send it to a joining member. Major limitation of this approach is that the state transfer that is very large (>1Gb) would likely result in `OutOfMemoryException`. In order to alleviate this problem a new state transfer methodology, based on a streaming state transfer, was introduced in JGroups 2.4

Streaming state transfer supports both partial and full state transfer.

Streaming state transfer provides an `InputStream` to a state reader and an `OutputStream` to a state writer. `OutputStream` and `InputStream` abstractions enable state transfer in byte chunks thus resulting in smaller memory requirements. For example, if application state consists a huge DOM tree, whose aggregate size is 2GB (and which has partly been passivated to disk), then the state provider (ie. the coordinator) can simply iterate over the DOM tree (activating the parts which have been passivated out to disk), and write to the `OutputStream` as it traverses the tree. The state receiver will simply read from the `InputStream` and reconstruct the tree on its side, possibly again passivating parts to disk.

Rather than having to provide a 2GB `byte[]` buffer, streaming state transfer transfers the state in chunks of N bytes where N is user configurable.

#### **6.10.2.2. API**

Streaming state transfer, just as regular byte based state transfer, can be used in both pull and push mode. Similarly to the current `getState` and `setState` methods of `org.jgroups.MessageListener`, application interested in streaming state transfer in a push mode would implement streaming `getState` method(s) by sending/writing state through a provided `OutputStream` reference and `setState` method(s) by receiving/reading state through a provided `InputStream` reference. In order to use streaming state transfer in a push mode, existing `ExtendedMessageListener` has been expanded to include additional four methods:

```
public interface ExtendedMessageListener
{

    /*non-streaming callback methods omitted for clarity*/

    /**
     * Allows an application to write a state through a provided OutputStream.
     * An application is obligated to always close the given OutputStream reference.
     *
     * @param ostream the OutputStream
     * @see OutputStream#close()
     */
    public void getState(OutputStream ostream);

    /**
     * Allows an application to write a partial state through a provided OutputStream.
     * An application is obligated to always close the given OutputStream reference.
     *
     * @param state_id id of the partial state requested
     * @param ostream the OutputStream
     * @see OutputStream#close()
     */
    public void getState(String state_id, OutputStream ostream);

    /**
     * Allows an application to read a state through a provided InputStream.
     * An application is obligated to always close the given InputStream reference.
     *
     * @param istream the InputStream
     * @see InputStream#close()
     */
    public void setState(InputStream istream);

    /**
     * Allows an application to read a partial state through a provided InputStream.
     * An application is obligated to always close the given InputStream reference.
     *
     * @param state_id id of the partial state requested
     * @param istream the InputStream
     * @see InputStream#close()
     */
    public void setState(String state_id, InputStream istream);

}
```

For a pull mode (when application uses `channel.receive()` to fetch events) two new event classes will be introduced:

- StreamingGetStateEvent
- StreamingSetStateEvent

These two events/classes are very similar to existing GetStateEvent and SetStateEvent but introduce a new field; StreamingGetStateEvent has an OutputStream and StreamingSetStateEvent has an InputStream.

The following code snippet demonstrates how to pull events from a channel, processing StreamingGetStateEvent and sending hypothetical state through a provided OutputStream reference. Handling of StreamingSetStateEvent is analogous to this example:

```
...
        Object obj=channel.receive(0);
        if(obj instanceof StreamingGetStateEvent) {
            StreamingGetStateEvent evt=(StreamingGetStateEvent)obj;
            OutputStream oos = null;
            try {
                oos = new ObjectOutputStream(evt.getArg());
                oos.writeObject(state);
                oos.flush();
            } catch (Exception e) {}
            finally{
                try {
                    oos.close();
                } catch (IOException e) {
                    System.err.println(e);
                }
            }
        }
        ...
```

API that initiates state transfer on a JChannel level has the following methods:

```
public boolean getState(Address target,long timeout)throws
    ChannelNotConnectedException,ChannelClosedException;
public boolean getState(Address target,String state_id,long timeout)throws
    ChannelNotConnectedException,ChannelClosedException;
```

Introduction of STREAMING\_STATE\_TRANSFER does *not* change the current API.

### 6.10.2.3. Configuration

State transfer type choice is static, implicit and mutually exclusive. JChannel cannot use both STREAMING\_STATE\_TRANSFER and STATE\_TRANSFER in one JChannel configuration.

STREAMING\_STATE\_TRANSFER allows the following configuration parameters:

**Table 6.8. Properties**

Name	Description
bind_addr	The network interface to be used for receiving of state requests, e.g. bind_addr="192.16.8.0.2"
start_port	Port on the bind_addr network interface to be used for

Name	Description
	receiving of state requests, e.g. <code>start_port="4444"</code>
<code>max_pool</code>	Maximum number of threads in the pool providing state requests, default=5, e.g. <code>max_pool="10"</code>
<code>pool_thread_keep_alive</code>	Pool thread keep alive in msec, default=30000, e.g. <code>pool_thread_keep_alive="60000"</code>
<code>use_reading_thread</code>	Use separate thread for reading state, default=false, e.g. <code>use_reading_thread="true"</code>
<code>socket_buffer_size</code>	Chunk size used for state transfer, default=8192 e.g. <code>socket_buffer_size="32768"</code>

#### 6.10.2.4. Other considerations

Threading model used for state writing in a member providing state and state reading in a member receiving a state is tunable. For state provider thread pool is used to spawn threads providing state. Thus member providing state, in a push mode, will be able to concurrently serve N state requests where N is `max_threads` configuration parameter of the thread pool. If there are no further state transfer requests pool threads will be automatically reaped after configurable "`pool_thread_keep_alive`" timeout expires. For a channel operating in the push mode state reader channel can read state by piggybacking on jgroups protocol stack thread or optionally use a separate thread. State reader should use a separate thread if state reading is expensive (eg. large state, serialization) thus potentially affecting liveness of jgroups protocol thread. Since most state transfers are very short (<2-3 sec) by default we do not use a separate thread.

## 6.11. 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.

### 6.11.1. 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 6.9. Properties**

Name	Description
max_credits	Max number of bytes the sender is allowed to send before blocking until replenishments from the receivers are received
min_credits	Min credits in bytes. If the available credits for a sender drop below this value, a receiver will send a replenishment message to the sender
min_threshold	Same as min_credits, but expressed as a percentage of max_credits, e.g. 0.1 (10% of max_credits)
max_block_time	The maximum time in milliseconds a sender can be blocked. After this time has elapsed, and no replenishment has been received, the sender replenishes itself and continues sending. Set it to 0 to prevent this

### 6.11.2. SFC

A simplified version of FC. FC can actually still overrun receivers when the transport's latency is very small. SFC is a simple flow control protocol for group (= multipoint) messages.

Every sender has max\_credits bytes for sending multicast messages to the group.

Every multicast message (we don't consider unicast messages) decrements max\_credits by its size. When max\_credits falls below 0, the sender asks all receivers for new credits and blocks until \*all\* credits have been received from all members.

When the receiver receives a credit request, it checks whether it has received max\_credits bytes from the requester since the last credit request. If yes, it sends new credits to the requester and resets the max\_credits for the requester. Else, it takes a note of the credit request from P and - when max\_credits bytes have finally been received from P - it sends the credits to P and resets max\_credits for P.

The maximum amount of memory for received messages is therefore <number of senders> \* max\_credits.

The relationship with STABLE is as follows: when a member Q is slow, it will prevent STABLE from collecting messages above the ones seen by Q (everybody else has seen more messages). However, because Q will \*not\* send credits back to the senders until it has processed all messages worth max\_credits bytes, the senders will block. This in turn allows STABLE to progress and eventually garbage collect most messages from all senders. Therefore, SFC and STABLE complement each other, with SFC blocking senders so that STABLE can catch up.

**Table 6.10. Properties**

Name	Description
max_credits	Max number of bytes the sender is allowed to send

Name	Description
	before blocking until replenishments from the receivers are received

## 6.12. 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.

### 6.12.1. STABLE

**Table 6.11. Properties**

Name	Description
desired_avg_gossip	Interval in milliseconds at which a stable message is broadcast to the cluster. This is randomized to prevent all members from sending the message at the same time. If set to 0, it is disabled.
max_bytes	Maximum number of bytes received after which a stable message is broadcast to the cluster. A high number means fewer stability rounds which purge more messages. A smaller value means a higher frequency of stability rounds which purge fewer messages. This is similar to garbage collection in the JVM.
stability_delay	When sending a stability message, we wait a randomized time between 1 and stability_delay milliseconds before sending it. If, when about to send the message, a stability message is received, we cancel our own message. This is to prevent everyone from sending the message at the same time.

## 6.13. Diagnostics

### 6.13.1. PERF

### 6.13.2. SIZE

### 6.13.3. TRACE

### 6.13.4. PRINTOBS

## 6.14. Misc

### 6.14.1. COMPRESS

### 6.14.2. 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:

- State transfer

When a member requests state transfer it tells everyone to stop sending messages and waits for everyone's ack. Then it asks the application 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 would ensure 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, e.g. above STATE\_TRANSFER and FC. STATE\_TRANSFER and GMS protocol request flush by sending a SUSPEND event up the stack, where it is handled by the FLUSH protocol. The SUSPEND\_OK ack sent back by the FLUSH protocol lets the caller know that the flush has completed. When done (e.g. view was installed or state transferred), the protocol sends up a RESUME event, which will allow everyone in the cluster to resume sending.

Channel can be notified that FLUSH phase has been started by turning channel block option on. By default it is turned off. If channel blocking is turned on FLUSH notifies application layer that channel has been blocked by

sending `EVENT.BLOCK` event. Channel responds by sending `EVENT.BLOCK_OK` event down to `FLUSH` protocol. We recommend turning on channel block notification only if channel is used in push mode. In push mode application that uses channel can perform block logic by implementing `MembershipListener.block()` callback method.

**Table 6.12. Properties**

Name	Description
timeout	Maximum time that <code>FLUSH.down()</code> will be blocked before being unblocked. Should be sufficient enough to allow large state transfers,default=8000 msec
block_timeout	Maximum amount of time that <code>FLUSH</code> will be waiting for <code>EVENT.BLOCK_OK</code> once <code>Event.BLOCK</code> has been sent to application level, default=10000 msec



---

# Bibliography

[Ensemble:1997] The Ensemble Distributed Communication System , CS Dept Cornell University , 1997 . <http://www.cs.cornell.edu/Info/Projects/Ensemble/index.html> .

[Gamma:1995] Erich Gamma , Richard Helm , Ralph Johnson , and John Vlissides . Design Patterns: Elements of Reusable Object-Oriented Software . Addison-Wesley , 1995 .