JavaGroups User's Guide

Bela Ban
Dept. of Computer Science
Cornell University
bba@cs.cornell.edu

August 16 1999

Contents

Chapter 1

Overview

JavaGroups is toolkit for reliable group communication. Clients 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, it will be created automatically.

To join a group, a process has to create a *channel* and connect to it using the group name (all channels with the same name form a 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 what other members are in the same group: a list of channel 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 received, which is installed locally in the channel. 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 actively removes the next one. When no message is currently available, a client is blocked until the next available message has been received.

A channel can be implemented over a number of alternatives for group transport. Therefore, a channel is an abstract class, and concrete implementations are derived from it, e.g. a channel implementation using its own protocol stack, or others using existing group transports. Currently, JavaGroups offers 3 implementations of channels: JChannel, which is the default group transport, EnsChannel, which is based on Ensemble [CS 97], a reliable group communication toolkit written in ML, and IbusChannel, which uses iBus [ibu98] as transport. Applications only deal with the abstract channel class, and the actual implementation can be chosen at startup time.

1.1 Building Blocks

Channels are simple and primitive. They offer the bare functionality of group communication, and have on purpose been modeled after BSD sockets. The reason is that an application can make use of just this small subset of JavaGroups, 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 will use ca. 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 peeked for messages.
JavaGroups offers building blocks that solve these problems. Building blocks make use channels, and are an additional layer between a channel and the application. The latter would now communicate with a building block, instead of directly with the channel. Building blocks are intended to save the application programmer from having to write tedious, recurring code.

1.2 The JChannel Protocol Stack

As discussed above, JavaGroups provides 3 channel implementations: an Ensemble-based channel, an iBus-based channel, and its own channel based on a Java protocol stack, called JChannel. JChannel is a protocol stack containing a number of protocol layers in a bidirectional list. All messages sent and received over the channel have to pass 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 receiving side.
The composition of the protocol stack, i.e. its layers, is determined by the creator of the channel: a property string 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.
If a new protocol is to be created for JavaGroups using the JChannel protocol stack, or if an existing protocol is to be modified, then knowledge about its implementation is necessary. Implementing a simple protocol is not difficult, however, one has to know the environment in which the protocol will be embedded, e.g. what events it will receive, when it is supposed to be started/stopped etc. The implementation of the JChannel protocol stack is discussed in chapter 4.

Chapter 2

Installation and Configuration

2.1 Installing and compiling the source distribution

JavaGroups is provided in `.tar.gz` and `.zip` form. In the first case, use `gzip` and `tar` to unzip and untar it, in the second case, `unzip` will do. A directory `JavaGroups` with several subdirectories will be created. Add this directory to your `CLASSPATH`, e.g.:

```
export CLASSPATH=$CLASSPATH:/home/mydir
```

(JavaGroups would be in `/home/mydir/JavaGroups`). Note that it is probably advisable to add a period (`.`) to your `CLASSPATH` so that class files in the current directory are found correctly (e.g. for the demo programs).
All the class files are already available in the source distribution, so you only need to recompile if you have modified some files. To recompile, type `make all` in the `JavaGroups` directory. If you also want to create the iBus adapter (class `IbusChannel`), type `make ibus`. (You need to have the iBus classes if you want to do this). To create the demo applications, type `make demos`.
Note that the above description applies to a UNIX like environment, where tools like `make` are available. In a different environment, e.g. a project builder, source files may have to be added to a project list, depending on the builder.

2.2 Choosing the channel implementation

2.2.1 JChannel

There are no special steps to be taken when using JavaGroups over the default (JChannel) transport.

2.2.2 Ensemble

To use JavaGroups on top of Ensemble, Ensemble has to be installed first. Refer to Ensemble's installation instructions for details, it can be downloaded from `http://www.cs.cornell.edu/Info/Projects/Ensemble/index.html`
The following steps have to be taken to use JavaGroups over Ensemble:

- Set the environment variables required for Ensemble (`ENS_PORT`, `ENS_GOSSIP_HOSTS` and `ENS_GOSSIP_PORT` etc).

- Put the `outboard` executable somewhere in your `PATH`. JavaGroups will spawn this executable in a separate process and therefore needs to find it.
- Start a `gossip` daemon *before* using JavaGroups in case Ensemble is not enabled to use IP multicast (see the Ensemble documentation on how to enable/disable Ensemble for IP multicast).

2.2.3 iBus

- Download iBus from `http://www.softwired-inc.com/ibus>SoftWired`.
- Put the `ibus_xx.jar` file in your `CLASSPATH`:

```
export CLASSPATH=$CLASSPATH:/mydirectory/ibus\_xx.jar
```

2.3 Testing your Setup

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

```
javap JavaGroups.Channel
```

You should see the following output if the class is found (may vary slightly):

```
Compiled from Channel.java
public abstract synchronized class JavaGroups.Channel
       extends java.lang.Object implements JavaGroups.Transport
/* ACC_SUPER bit set */
{
public static final int BLOCK;
public static final int VIEW;
public static final int SUSPECT;
public static final int LOCAL;
protected JavaGroups.Channel();
protected JavaGroups.Channel(java.lang.Object);
public abstract void Connect(java.lang.Object);
public abstract void Disconnect();
public abstract void Close();
public abstract void Send(JavaGroups.Message);
public abstract java.lang.Object Receive(long);
public abstract java.lang.Object Peek(long);
public abstract JavaGroups.View GetView();
public abstract java.lang.Object GetLocalAddress();
public abstract java.lang.Object GetGroupAddress();
public abstract void SetOpt(int, java.lang.Object);
public abstract java.lang.Object GetOpt(int);
public abstract void BlockOk();
}
```

Javap lists the variables and methods of class `Channel`. If this does not work, the JavaGroups classes have not been added correctly to your `CLASSPATH`. Go back to the installation instructions to correct this problem.

2.4 Running a Demo Program

Change to the `./Tests` directory.

2.4.1 Testing for Normal Use

Run the `Draw` program:

```
java Draw
```

A drawing window should appear. Start a second instance. Both instances should change their membership (shown in the title bar) to 2. When drawing the mouse 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. See the documentation on gossip 3.8 to use a mechanism where instances query a server to find each other, and on the UDP protocol layer 7.2.14 to learn how to disable IP multicast and use point to point messages instead.
Note that `CLASSPATH` needs to contain a period (`.:`) to find the local `Draw` class ! If this is not the case, the JVM will complain that it cannot find class `Draw` !

2.4.2 Testing for Use over Ensemble

The `Draw` program can be run over Ensemble using the `-type` option when starting it:

```
java Draw -type ens
```

JavaGroups tries to start the `outboard` executable and waits some seconds until outboard is running. Then JavaGroups connects to outboard via a socket. There are several problems that might occur when trying to run JavaGroups over Ensemble:

- The `outboard` executable is not found. It is probably not in your `PATH`.
- The number of seconds to wait is too short to start up outboard before connecting to it. This might be the case when the machine is heavily loaded. Either decrease the load or increase the timeout value. In the latter case, look for `Thread.sleep` in file `JavaGroups/Ensemble/HotEnsemble.java` and increase its timeout.
- The instances don't 'find' each other. The problem might be that IP multicast is not enabled for Ensemble and the `gossip` daemon is not running. Start gossip before running the program:

  ```
  gossip&
  ```

2.4.3 Testing for Use over iBus

Make sure that you have compiled the iBus adapter (class `IbusChannel`). Also ensure that you have the iBus JAR file in your `CLASSPATH`:

```
javap iBus.Stack
```

You should see the contents of this class printed. Run the `Draw` program using the `-type` option:

```
java Draw -type ibus
```

The behavior of the samples should be the same as before.

Chapter 3

Using JavaGroups

This chapter explains the classes available in JavaGroups that will be used by applications to build reliable group communication applications. The focus is on creating and using channels. Also, the main building blocks will be discussed. For users who want to configure their own protocol stack, an overview of the protocol layers will be given (the specification of all available protocols is given in the appendix). Finally, some problems with regard to firewalls and IP multicasting are discussed.
Information in this document may not be up-to-date, but the nature of the classes in the JavaGroups toolkit described here is the same. For the most up-to-date information refer to the Javadoc-generated documentation in directory `JavaGroups/doc`.

3.1 Utility classes

These classes are used frequently, therefore their use is discussed before channels.

3.1.1 Version

The current version of JavaGroups can be determined by invoking

```
java JavaGroups.Version
```

3.1.2 Addresses

Each member of a group has a unique address. In JChannel, it is the IP address of the machine on which the stack is running and the port on which the stack is receiving incoming messages; it is represented by class `JavaGroups.JavaStack.Address`. Instances of this class are only used within the JChannel protocol stack; *users of a channel see addresses (of any kind) only as objects (opaque).* Since an address uniquely identifies a channel, and therefore a group member, it can be used to send messages to that group member.

3.1.3 Messages and Headers

Data is sent between members in the form of messages (`JavaGroups.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 fig. 3.1.
A message is similar to an IP packet and consists of the payload (a byte buffer), the addresses of the sender and receiver, and a number of headers. Any message put on the network can be routed to its destination (receiver address), and replies can be returned to the sender's address.

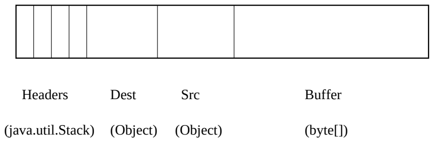

Figure 3.1: Structure of a message

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 would be returned to some other member.
The destination address (receiver) can be an object, denoting the address of a member, determined e.g. from a message received previously, or it may be `null`, which means to send the message 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, new String("Hello").getBytes());
```

When traveling down a protocol stack, layers may want to add headers with information pertaining to that layer to a message. Headers (class `JavaGroups.Header`) are wrappers around atomic or more complex data types. Messages have methods to add, remove and peek at headers of a message. Although any type of data types can be tucked into a header, protocols will typically want to create their separate header types as *classes*, and add them to a message, e.g. as in the example of the fragmentation layer:

```
class FragHeader implements Serializable {
    public long    msg_id=0;
    public int     frag_id=0;
    public int     num_frags=0;

    public FragHeader(long msg_id, int frag_id, int num_frags) {
        this.msg_id=msg_id;
        this.frag_id=frag_id;
        this.num_frags=num_frags;
    }

    public String toString() {
        return "[FRAG: msg_id=" + msg_id + ", frag_id=" + frag_id +
               ", num_frags=" + num_frags + "]";
    }
}
```

The object to be tucked into a header has to be serializable, because headers are part of messages, and messages will need to be serialized to be sent over the network. Since we might want to debug protocol layers later on, it is good practice to provide a `toString` method, which prints the type and contents of a header.
When a fragmentation layer encounters a Message (traveling down the stack) which size is too big, it would split it into several smaller messages (fragments), and add a header to each fragment which contains a) the original message ID, b) the fragment's ID and the total number of fragments to be expected:

```
Message    frag, big_msg;
FragHeader hdr=new FragHeader(big_msg.GetId(), next_frag_id, tot_frags);
```

```
frag.AddHeader(hdr);
// pass down the stack
```

In case the message is not too big, it would just be passed on down to the next layer, without adding any header.
The fragmentation layer on the receiver's side would check whether the top header is of type `FragHeader`, and if this is the case, remove it and process it (e.g. retrieve all fragments, reassemble them into the original big message, and finally pass the message up the stack:

```
FragHeader hdr;
Object     obj=msg.PeekHeader();  // does not remove header
if(obj != null && obj instanceof FragHeader) {
    hdr=(FragHeader)msg.RemoveHeader();
    // process header
}
```

3.1.4 Queue

Queues (`JavaGroups.Queue`) are frequently used when data needs to be passed on from one thread to another (producer-consumer relation). Objects are added to a queue by producers in FIFO order, and consumers can peek at and remove objects from a queue. When no object is available, a consumer will be blocked until an object is added. Queues can be closed, which has the effect that trying to add or remove objects will result in an exception to be thrown. Closing a queue will immediately remove all elements. A more graceful version of closing exists, in which an end-marker is inserted at the7 tail of the queue. When it is encountered, the queue will be closed. This essentially allows all existing objects at the time of the close to be removed (flushed) before actually closing the queue.

3.1.5 View

A View (`JavaGroups.View`) is a local representation of the current membership of a group. Only one view is installed in a channel at a time. Views contain the address of its creator, an ID and a list of member addresses. Views are generated by a previously elected coordinator and are installed in each channel belonging to the same group whenever a member joins or leaves. Each view is uniquely identified by the creator/ID pair, and the ID is incremented for each new view. 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 if 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.

3.1.6 SlidingWindow

To guarantee lossless delivery, a unique (increasing) sequence number (ID) is attached to every message. The combination of the message's sender address and its ID identifies every message uniquely. To overcome message loss, sender and receiver agree on some start ID, e.g. TCP connection establishment involves the sender and receiver exchanging their start sequence numbers in the SYN/ACK packets. Now the IDs of messages received from a sender S always have to monotonically increase: if message 56 was received from S, the next message has to be 57. If a gap is detected, the receiver sends a retransmission request to the sender of the message. Upon receiving a retransmission request, the sender resends the message with the requested ID. Being able to resend old messages requires the senders to store recently sent messages for some time, usually until it is know that the receiver(s) has/have received all messages below a certain ID. In this case the messages below this ID may be deleted. When

a receiver only sends retransmission requests when a gap is detected, the mechanism is called *negative acknowledgment NAK*. NAK is used in systems where message loss is infrequent, so NAKs would only rarely have to be used to retransmit lost messages.
When communication links are noisy and loss rate high, ACK schemes are preferred: a sender has to receive an acknowledgment from each receiver of a message. If ACKs are not received, the message is resent until an ACK has been received.
The idea of *sliding windows* ([Tan89, ch 4.4] and [Bir96, 83–88]) is to keep track of the acknowledgements for each ID. However, a scheme in which a sender send a single message (e.g. to multiple receivers in a group) and then waits for all ACKs is to slow: a sender should be able to send a number of messages and a separate thread should receive ACKs, and resend messages with ACKs missing. The senders and receivers each maintain a window of messages for which no ACKs have been received: a window is essentially a sequence of message IDs, starting with a low water mark and bounded by a high water mark. Whenever an ACK is received, the low and high water marks are advanced by 1, this allows 1 more ACK to be received, therefore *sliding* the window 1 to the right. When the window is full, an ACK is either discarded, or some kind of flow control is used to throttle the sender until there is more space available.
Sliding windows usually start out with a given size, however, more sophisticated protocols will dynamically adapt the window size, trying to find an agreed-upon size between sender and receiver.
The characteristics of sliding windows used at the sender and receiver usually involve (but do not have to !) a) error correction (by retransmission), b) flow control and c) message ordering by sender (FIFO). The latter property can easily be incorporated in a sliding window protocol, but sometimes, it is preferred to be implemented as a separate protocol for easier maintenance / replaceability.
Class `JavaGroups.SlidingWindow` is based on the concept of sliding windows, however, it is based on NAKs, not ACKs. A receiver adds messages to the table, where they are inserted according to their sequence number (ID). A counter keeps track of the low water mark, i.e. the lowest sequence number. Method `Remove` tries to remove a message, which succeeds as long as there are messages whose sequence number is equal to the low water mark (incrementing the latter). E.g with sequence numbers `23 24 26 27`, `Remove` would successfully remove the first 2 messages in turn before returning `null`. Only when the message with sequence number 25 has been added will `Remove` succeed to remove messages 25, 26 and 27 (advancing the low water mark to 28).
A separate retransmission thread is started in each sliding window object. Its task is to send NAKs to senders from which messages have not been received with increasing sequence numbers, e.g. in the above example (before receiving message 25), the retransmission thread might have sent a NAK for message 25 to its sender. To do so, it invokes method `Retransmit` in an object (implementing interface `RetransmitCommand`) which registered previously with the sliding window table. The retransmit command should send a retransmit request to the sender, causing the sender to resend the requested message. When a retransmit request to a sender has failed twice, the request will be multicast to all group members. When a group member receives a retransmission request of which it wasn't the sender, and it still has the message, it will retransmit the message to the requester (changing the message's sender field to that of the (apparently failed) original sender[1].
A `SlidingWindow` is configured with the sender from which messages are to be received (there is usually 1 sliding window per sender) and a retransmit command.
When a message has been received, it can be added to the sliding window table corresponding to the sender's address using method `Add`. If the message's sequence number is smaller than the table's low water mark, it is discarded. Otherwise, it is added in order, creating empty

[1]This work only if the request was a multicast request, so all members received (and stored) a copy of the request.

entries if necessary. For example, if messages 3, 6 and 7 are added to the table in succession, then the first entry is for 3, then 2 empty entries for 4 and 5 are added, and finally entries 6 and 7 are added. If the low water mark was at 3 before the messages were added, subsequent calls to `Remove` would remove message 3 and return null for each subsequent call, until messages 4 and 5 are added.
The retransmission thread continually iterates through all entries and checks the age of each entry. If it exceeds a certain user-defined value, a retransmission request is sent using the retransmission object registered with the sliding window table in the constructor.
This implementation of `SlidingWindow` does not maintain a window size; it can rather store as many messages as are received. This is clearly a disadvantage and needs to be changed in a future release. However, using a flow control protocol layer would also correct the problem.

3.1.7 ReusableThread

Regular threads are usually created to execute a task and then die. Reusable threads can be assigned any number of tasks after they are done, so by reusing them, thread creation and destruction time (albeit small) is reduced. When a `ReusableThread` is created, it awaits its first task. A task has to be an object implementing interface `Runnable`. When a task is assigned (using method `AssignTask`), the task will be started (calling its `run` method) and the reusable thread waits until it is completed. When done, the reusable thread is ready to process another task, and so on, until it is killed. When a task is processed, no other task can be assigned, until the current task is finished. Calling `AssignTask` in this case results in an error message. An example of using reusable threads is shown below:

```
ReusableThread t=new ReusableThread();
t.Start();
MyThread m1=new MyThread(1);
MyThread m2=new MyThread(2);

t.AssignTask(m1);
t.WaitUntilDone();  // passive wait
t.AssignTask(m2);
t.WaitUntilDone();
t.Stop();
```

Two threads (m1 and m2) are created (but *not* started) and assigned to the reusable thread in turn. The caller can use `WaitUntilDone` to wait for completion of a task.

3.1.8 ThreadPool

A thread pool (`JavaGroups.ThreadPool`) maintains a number of reusable threads. A new thread can be created using method `GetThread`. It will be created, added to the pool and returned as result of this call. A thread pool can be given a limit on the number of threads to create. When `GetThread` is called, the existing threads are checked for completion. If an existing thread is found that has completed its task, then it will be returned. Otherwise, if the pool limit has not yet been reached, a new reusable thread will be created, added to the pool and returned. If the limit has been reached, `null` will be returned.

3.1.9 Scheduler

The scheduler (`JavaGroups.Scheduler`) maintains a queue of tasks. It continually processes the first task of the queue (using a reusaable thread and a thread pool) and removes it when done. Tasks are by default added to the end of the queue (`Add`). However, there is a possibility to process a task with high priority, that means, before all others. Using `AddPrio`,

processing of the current task is suspended, and the new task inserted at the head of the queue and processed immediately. When done, the suspended task is resumed. High priority tasks may preempt previously added high priority tasks, and – when done – will be resumed in the order in which they were suspended.
The scheduler is used by the `RequestCorrelator` to solve blocking of circular requests causing deadlocks (see 5.2 for details).

3.1.10 Util

This class contains some (mainly static) helper methods that can be used by all classes.

ObjectToByteBuffer, ObjectFromByteBuffer

The first method takes an object as argument and serializes it into a byte buffer (the object has to be serializable). The byte array is then returned. This method is often used to for example serialize method calls (see 3.4.2) or other 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 reconstructed.

FragmentBuffer, DefragmentBuffer

These 2 methods fragment a byte array into smaller byte arrays (given a fragmentation size), and re-assemble a number of smaller byte arrays into a larger one. The first method takes a byte array and a fragmentation size as parameters, and returns a vector of byte arrays that correspond to the fragmented byte buffers. If the fragmentation size is greater than the original buffer's size, the original buffer is returned in as the only element in the vector. The second method takes a vector of fragments (byte arrays) and concatenates them into a larger byte array, which is returned. These methods are used in the fragmentation layer (FRAG, cf. 7.2.4).

3.2 Interfaces

3.2.1 Transport

Interface `Transport` (`JavaGroups.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 3.4) 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 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 (JavaGroups.MessageListener) below provides a method to do so:

```
public interface MessageListener {
    public void Receive(Message msg);
}
```

Method Receive will be called when a message is received.

3.2.3 MembershipListener

The MembershipListener interface (JavaGroups.MembershipListener) 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();
}
```

3.3 Channels

Channels are the means by which member processes in a group can communicate with each other: they are used to send and receive messages and views. A channel represents a single member process; at any point only a single process can be connected to a channel. When a client connects to a channel, it indicates the group address (usually 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 group membership protocol (GMS, see 3.5.3 and 7.2.5) takes care that channels with the same group name find each other: whenever a client connects to a channel given group name G, then the GMS tries to find existing channels with the same name, and joins them, resulting in a new view being installed (which contains the new member).
A state transition diagram for the major states a channel can assume are shown in fig. 3.2.

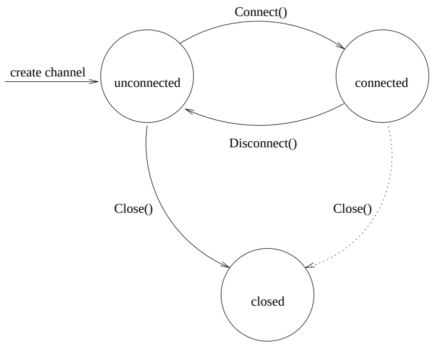

Figure 3.2: Channel states

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 receives 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 below.

3.3.1 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. `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` (`JavaGroups.JChannel`).
The public constructor of `JChannel` looks as follows:

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

It creates an instance of `JChannel`. The argument defines the composition of the protocol stack (number and type of layers, parameters for each layer, and their order). If it 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. (Refer to 4 for more information).

3.3.2 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(Object group_address) throws ChannelClosed;
```

To preserve genericity, the group address is typed as an object, However, channel implementations would typically require more specific group addresses. In the case of `JChannel`, the group address must be a string, naming the group to be joined. All channels that are connected to the same group (same name) form a group. Messages multicast on any channel in the group will be received by all members (including the one who sent it[2]).
The method returns as soon as other members have been found, and the group has been joined. If the channel is in the closed state (see fig. 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 (conceptual) group is created and the member joined. The first member of a group becomes its *coordinator*, i.e. the member who receives join and leave requests. A coordinator is in charge of multicasting new views whenever the membership changes.

3.3.3 Getting the local and group addresses

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 Object GetLocalAddress();
```

Method `GetGroupAddress` returns the address of the group in which the channel is a member:

[2]Local delivery can be turned on/off using `SetOpt`.

```
public Object GetGroupAddress();
```

Again, the result in undefined if the channel is in the unconnected or closed state. An implementation is free to return any value. In the case of `JChannel`, the group address is a string (the name of the group), but channel implementations are free to choose a more sophisticated structure to define groups.

3.3.4 Sending data to member(s)

Once the channel is connected, messages can be sent:

```
public void Send(Message msg) throws ChannelNotConnected, ChannelClosed;
```

The only argument is the message to be sent. Its 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 message is sent asynchronously, i.e. as soon as it is put on the network (usually this means added to the outgoing buffer of the network interface), the method will return. Like UDP, there is no acknowledgment.
If the channel is not connected, or was closed, an exception will be throw upon attempting to send a message.

3.3.5 Receiving data

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 this method is called, the next available message from the top of the 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 `Timeout` 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 3.3.11), the following 4 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` creates 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).

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 *until a new view has been received.* It is used to synchronize messages between views, so that all messages are received in the view in which they are sent. When the application has stopped sending messages, it needs to acknowledge this message with a `Channel.BlockOk` method. Messages of this type are not usually enabled on a channel (see 3.3.11), therefore an application will not receive blocks at all. In case they *are* enabled, `BlockOk` has been invoked, and the application sends a message before having received the next view, the view in which the message will be delivered is not determined: it may be the next view, or any the message may even be discarded ! When blocks are not enabled, the view in which a message will be delivered, is not determined (but no message will be discarded !)

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.

SetStateEvent Received as response to a `GetState(s)` method call. The argument contains the state of a single member (`Object`) 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.

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.

3.3.6 Peeking at data

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`.

3.3.7 Getting the group's state

A newly joined member may wish to retrieve the state of the group before starting work. This is done calling either `GetState` of `GetStates`. The first method returns the state of one member (in most cases, of the oldest member, the coordinator) whereas the latter returns the states of all members. This method 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[3].
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(s)` returned true, then a valid state (non-null) will be returned, otherwise a null state will be returned.
Note that state transfer is currently only available with JChannel !
The following code fragment shows how a group member participates in state transfers:

```
channel=new JChannel("UDP:PING:FD:GMS:STATE_TRANSFER:QUEUE");
channel.SetOpt(Channel.GET_STATE_EVENTS, new Boolean(true));
channel.Connect("TestChannel");
boolean rc=channel.GetState(5000);

...

Object ret=channel.Receive(0);
if(ret instanceof Message)
    ;
else if(ret instanceof GetStateEvent) {
    // copy=CopyState(state)
    channel.ReturnState(copy);
}
else if(ret instanceof SetStateEvent) {
    SetStateEvent e=(SetStateEvent)ret;
    // set state from ret.GetArg();
}
```

A JChannel has to be created whose stack includes the `STATE_TRANSFER` protocol. Option `GET_STATE_EVENTS` should be enabled, as the channel might probably want to return its current state if asked. 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.
If state transfer events are enabled, then `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 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.
The details of state transfer as implemented in JavaGroups are discussed in section 7.2.11.

3.3.8 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.

[3] A member will *never* retrieve the state from itself !

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.

3.3.9 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.

3.3.10 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.

3.3.11 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.VIEW The argument must be a boolean value `Boolean`. If true, views will be received (default is true). If false, no views will be received.

Channel.SUSPECT The argument is also a boolean object. If set to true, suspicion messages will be received, otherwise not (default is false).

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).

GET_STATE_EVENTS Reception of `GetStateEvents`. If this option is turned off, then the group's state may be retrieved, but this member will not respond to requests fro its state (client role).

The equivalent method to get options is `GetOption`:

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

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

3.4 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 `Transportable` (channels do). This enables them to work on any type of group transport that obeys this interface.

3.4.1 PullPushAdapter

This class is a converter (or adapter, as used in [GHJV95]) between the pull-style of actively having to receive messages and the push-style where clients register a callback method which is invoked whenever a message has been received. It allows clients of a channel to be notified when messages have been received instead of having to actively poll the channel for new messages. This eliminates any need for the clients to allocate a separate thread for receiving messages.
A `PullPushAdapter` (`JavaGroups.PullPushAdapter`) is always created on top of a class that implements interface `Transportable` (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 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 `Transportable` instance (e.g. a channel)[4], 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 `Transportable`, 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 fig. 3.3.

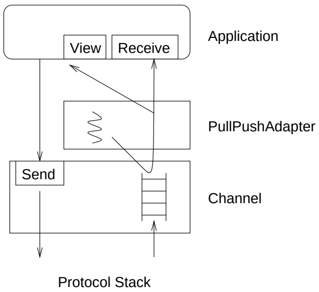

Figure 3.3: Class `PullPushAdapter`

As is shown, the thread constantly pulls messages from the channel and forwards them to the registered client. 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]Note that the channel currently has to be connected, otherwise an error message is issued.

Example

```
import JavaGroups.*;

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

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

    public void Start() throws Exception {
        channel=new JChannel();
        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) {
            System.err.println(e);
        }
    }
}
```

Contrary to using channels (using pull-style message reception), no separate thread has to be allocated to receive messages. Instead, a `PullPushAdapter` is layered on top of the channel and a reference to the client object added. This causes the client's `Receive` method to be called whenever a message has been received by the `PullPushAdapter`. 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.

3.4.2 MethodCall

This class (`JavaGroups.MethodCall`) represents a Java method plus its arguments that can be invoked against a target object. Its main differences from `java.lang.reflect.Method` are that it a) contains its parameter, so that `Invoke` does not need any arguments, and b) it allows to customize the method lookup algorithm. A method call is serializable which means that it can be sent over the network to some remote peer, and be invoked there (it takes its arguments with it). A message may contain a serialized method call object in its byte

buffer[5].
A `MethodCall` instance is created given the method name (string) and 0 or more arguments, depending on the number of formal parameters. For convenience, a number of constructors are provided, which take 1 to 3 arguments. If additional arguments are needed, the overloaded methods `AddArg` can be used to add more arguments.
Method `Invoke` is used to actually call the method against a target object. Its 2 parameters are the target object, which has to contain the method, and a method lookup instance which allows to override the default method lookup. If the second argument is null, the default method lookup algorithm will be used. `Invoke` essentially looks up the method to call (an instance of `java.lang.reflect.Method`) using the class of the target object and the number and types of the arguments of the method call object.
The second parameter to `Invoke` is a method lookup class which is supposed to return a `java.lang.reflect.Method` object, given the target's class, the method name and a number of arguments.
Method calls are an important concept in JavaGroups: they provide synchronous (or, if desired, asynchronous) method invocations on top of an inherently asynchronous transport (e.g. channels). Extending this concept to groups, it enables a client to make synchronous *group method invocations* on all members of a group, and retrieve the results as objects (see 3.4.4).

3.4.3 RequestCorrelator

Typical protocols can be divided into protocols that perform some work on a message, and then pass it up or down the stack, and protocols that interact with their peers in different protocol stacks[6]. The latter often need to send requests to their peer(s) and receive responses. However, in distributed peer computing (as is the case in groups), any member may be the *originator and receiver* of a request. That means that – besides sending requests and receiving the corresponding responses – every member must be enabled to process requests sent by other members and send responses. The main functions of a peer protocol are therefore

Send requests Send a request (message) to a single or all group members and receive 0 or more responses. If the request is asynchronous, no responses will be received. It should be possible to specify how many responses are to be received, e.g. all, majority, first.

Receive responses Receive a response to a request. If the request was sent to the whole group, receive a number of responses. No response will be received if the request was sent asynchronously. If a member has crashed before sending a response in the synchronous case, its response should be marked as 'not received'. Note that correlation between requests and responses is important, i.e., if a sender sends out requests r1, r2 and r3, incoming responses have to be matched against their originating requests.

Receive request Receive a request sent by a group member, process it and send a response (see below). Since local messages are enabled on a channel by default, a sender will receive its own request and has to process it like any other group member.

Send response Send a response to a request. Usually invoked when done processing a request.

The above functionality has been incorporated in class `RequestCorrelator` (`JavaGroups.RequestCorrelator`). This class can be used in an application on top of a channel, or to implement a peer protocol. The only requirement is that clients of the

[5] See 3.1.10 for two helper methods to serialize/unserialize objects to/from byte arrays.
[6] See 4.3 for a detailed discussion of this division.

class provide a `Transport` instance for sending requests/responses and call `Receive` or `Suspect` respectively, whenever a request/response is received or a member suspected.
The structure of a request correlator is shown in fig. 3.4.

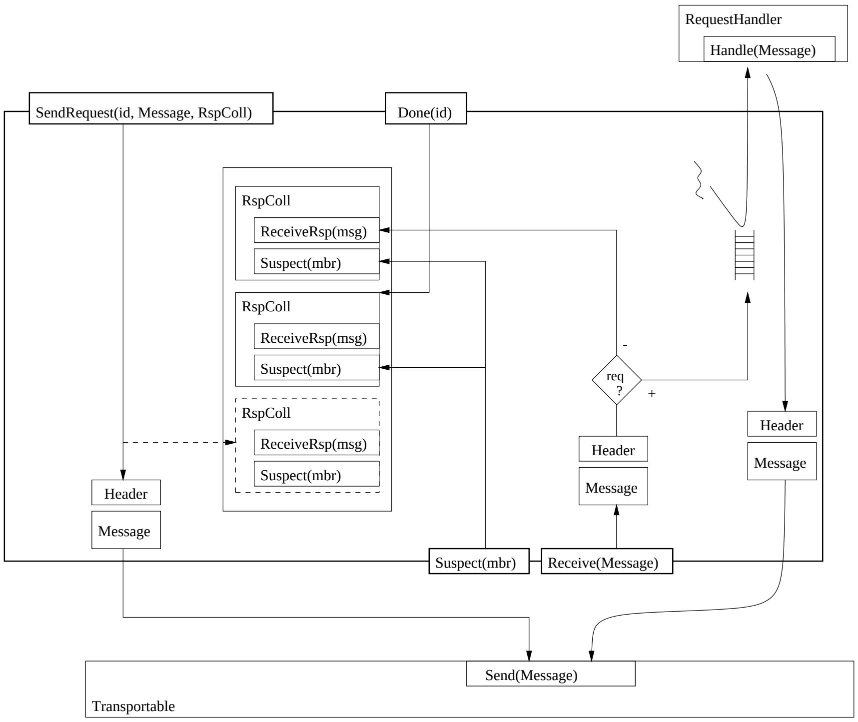

Figure 3.4: Request correlator

Essentially, a `RequestCorrelator` has 1 object that is responsible for handling all requests (`RequestHandler`), and 0 or more *response handlers* (`RspCollector`) which will receive the response(s) for a specific request. Each request is identified by a unique ID (for example the current time in milliseconds) and added to a request table. When a response comes in, its ID is matched against the requests, the correct request is found and the response directed to it. Having multiple requests pending at any time (compared to just a single one) gives clients of request correlator more freedom: the decision whether to send a single or multiple requests at a time is theirs.
The `RequestCorrelator` has methods to a) send a request (synchronously or asynchronously, to a single member or to all group members), b) submit a suspicion message, c) make it receive a message and d) remove a request (calling method `Done` with the request ID). It allows to register a request handler which will handle all requests directed towards this request correlator.
The following actions take place when a group request is sent:

Sending a request

The caller creates a new instance of `RspCollector` and invokes `SendRequest` with a unique ID that it wants to assign to the request, a message and the response collector. All the messages received for that specific request will be directed towards the response collector. If it is null, this means that the request is sent one-way, and no responses are to be returned. If it is not null, it is added to a response collector table.
`SendRequest` adds a specific header to the message and uses the underlying transport's `Send` method to send the message to its destination(s). The header contains

1. The ID assigned to the request by the caller
2. The type. A type can only be a request or a response.
3. Whether a response is expected or not. This is determined by `SendRequest`: if a non-null response collector is given as argument, this field will be true, otherwise false. The receiver of a request can thus determine whether to return a response.
4. A name that is associated with this specific instance of request correlator. Every request correlator has a unique name. When removing the header from a message, it is checked whether the header is of type `RequestCorrelatorHeader` and whether the current request correlator's name matches the one in the header. Only if both conditions are true, the message is processed. Otherwise, it is rejected (`Receive` returns false).

The message's destination field determines whether a unicast or multicast message will be sent, i.e. whether the message will be sent to a single member (destination field contains a member address) or to all group members (destination field is null).

Receiving a request

The user of a request correlator has to take care to receive regular and suspicion messages, and to 'feed' them to the request correlator. When a regular message is received, method `Receive` is called. It peeks at the header and checks for 2 things: a) whether the header is of the correct type (`RequestCorrelatorHeader`) and, if yes, b) whether the name field in the header matches the request correlator's name. If either condition is false, `Receive` will return false. A user will thus typically try to feed an incoming message to the request correlator, and, if `Receive` returns false, pass it on to some other layer, e.g. the layer above the current one (when used in a protocol).
If the condition is true, the header type is checked. If it is a request, the message will be added to the request queue (without yet removing the header), where a separate thread will handle it (see 3.4.3). If the header type is a response, the header will be removed from the message and the message added to the response collector that matches the header's ID (calling `ReceiveResponse`, see 3.4.3).
When the client feeds a suspicion message to the request correlator, the `Suspect` method callbacks of all request collectors will be called in turn. This allows building blocks on top of request correlators to return gracefully. For example, a group request building block (see 3.4.4) might block indefinitely when trying to send a request to 3 group members and waiting for all responses. If one of the group members crashes, no response from this member would ever be received. Receiving a suspicion notification, however, allows the group request to terminate the group call (if it received responses from the 2 other members)[7].

Handling a request

When a request is received, it is handled by a separate thread. This thread is only created if a request handler (`RequestHandler`) was given at the time of the request correlator's creation (parameter of the constructor of RequestCorrelator). Otherwise, if requests are received, they will just be discarded if no request handler was installed.
The thread removes the message header (which was not removed by `RequestCorrelator.Receive` before !) and determines whether the message is one-way or not, i.e. whether it needs to send a response. It then invokes the request handler's `Handle` method, which processes the request by interpreting the message's buffer (e.g. extracting an object from it) and returns an object. If a response need to be returned to the sender of the message, then the thread creates a response message, inserts the object

[7]If no suspicion service is available, timeouts can be used instead.

into the response's buffer, and adds another header to the message. This header would indicate that the message is in fact a response, and would contain the request's ID. Finally the message is sent using the underlying transport's `Send` method.

Receiving a response

When `RspCollector.ReceiveResponse` is called, the client of request correlator (the same who created the `RspCollector` object) decides what to do. In the case of `GroupRequest`, it might wait until all the responses have been received (minus the responses from suspected members), until the first response has been received, or it might wait for a majority of all responses to arrive. Note that no response will be received if the response collector argument to `SendRequest` was null; no response collector entry will ever be created in this case, and the message will be sent one-way.
When all the responses have been received, method `RequestCorrelator.Done` should be called to remove the response collector entry from the table. If this was not done, response collector entries would accumulate in the table and take up space. `Done` *has* to be called by the request correlator client, because only it knows when 'all' responses have been received: for example 'all' could mean a majority, and in other cases it could mean 'responses from all members'.

3.4.4 GroupRequest

Although `RequestCorrelator` can be used by itself, `GroupRequest` (`JavaGroups.Algorithms.GroupRequest`) is a building block making use of request correlator, that simplifies sending messages to all members of a group and collecting responses. Its most important public methods are:

```
public class GroupRequest {
    public GroupRequest(Message m, RequestCorrelator corr, Vector members,
                        int rsp_mode);
    public GroupRequest(Message m, RequestCorrelator corr, Vector members,
                        int rsp_mode, long timeout, int expected_mbrs);

    public synchronized boolean Execute();
    public void                 Reset(Vector members);
    public RspList              GetResults();
    public Vector               GetSuspects();
}
```

The 2 constructors need a request correlator, the message to be sent, a response mode, an initial membership, a timeout and the number of expected members.
The initial membership is a vector of member addresses from which responses are expected. If the message's destination is not null, meaning that the message is to be sent to a single destination (unicast), then the membership might only contain a single address, namely that of the destination from which a response is expected. If no response is expected, the membership vector can be empty.
The timeout can be used in combination with, or as replacement for, a suspicion service. If a message is sent to members P, Q and R, and responses are expected from all members, then, if R crashes, a suspicion service will feed this information to the underlying request correlator, which in turn will feed it to the group request object sitting on top of the correlator. A group request that has already received responses from P and Q will block until it receives the response from R, or until a suspicion message about R is received. If no suspicion service is available, a timeout specifies the maximum amount of time a group request should wait for all responses, preventing blocking. Timeouts and a suspicion service can also be combined. A timeout smaller than or equal to 0 means wait indefinitely.

The number of expected members is the number of responses expected. It is only used if the response mode is `GET_N` (see below).
The response mode is one of

`GroupRequest.GET_FIRST` `Execute` returns as soon as the first response has been received (or a timeout has elapsed, or a suspicion message was received).

`GroupRequest.GET_ALL` `Execute` returns when responses from all members have been received (or a timeout has occurred, or a suspicion message was received).

`GroupRequest.GET_MAJORITY` `Execute` returns when the majority of the members have responded (or a timeout occurred, or a suspicion message was received). The majority is computed using the membership given in the constructor. If a suspicion message is received, the majority is computed anew (dynamically), without the suspected member.

`GroupRequest.GET_ABS_MAJORITY` `Execute` returns when the majority of the members have responded. Contrary to the above mode, the majority is *not* re-computed when a member crashes.

`GroupRequest.GET_N` `Execute` returns when n responses have been received. If n is greater than the number of members in the membership, then the call will block forever (unless a timeout is specified).

`GroupRequest.GET_NONE` `Execute` returns immediately. The message is sent asynchronously, no responses are expected, and none will be generated at the receivers' end.

Method `Execute` returns true if the group request succeeded, and false in case of an error. `Reset` allows to reuse the same group request object, by just adjusting the membership, and resending the same message (note that there are other reset methods). `GetSuspects` returns a vector of suspected members (empty if none were suspected), and `GetResults` returns a response list `RspList` (`JavaGroups.RspList`) which can be queried for the response from each member, whether a member did not send a response, whether it crashed, and – if a response was received – its value.
Note that the group request relies on a loss-less message delivery: no response may ever be lost unless the sender crashed. This property must be guaranteed by the underlying transport, by for example adding a retransmission layer to the protocol stack. If this is not the case, a group request may block forever, e.g. if a response was generated by a receiver, but was lost, and the receiver did not crash (unless a timeout was specified).

3.4.5 MessageDispatcher

This class is used on top of a channel and allows one to broadcast requests (in the form of messages) to all or single group members and to optionally wait for the response(s). It is the application-level equivalent of `MessageProtocol` (see 4.3.1 for a detailed discussion). Applications need to implement the `RequestHandler` interface by implementing method `Handle` which is called whenever a request has been received. The return value will be sent back to the sender of the request as response.
`MessageDispatcher` uses both a `RequestCorrelator` and `GroupRequest` to implement its functionality.

3.4.6 RpcDispatcher

This class is derived from `MessageDispatcher` and is the application-level equivalent of `RpcProtocol` (see 4.3.2 for more details.) It allows one to invoke remote methods in all (or single) group members and optionally wait for the return value(s). An application would typically create a channel and layer the RpcDispatcher builing block on top of it, which allows it to dispatch remote method invocations and at the same time be invoked by other members. The code below shows an example:

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


    public int Print(int number) {
        System.out.println("Print(" + number + ")");
        return number * 2;
    }


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

        for(int i=0; i < 10; i++) {
            Util.Sleep(1000);
            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 `RpcDipatcher` defines method `Print` which will be invoked subsequently. The entry point `Start` method creates a channel and an `RpcDispatcher` which is layered on top of the channel. Method `CallRemoteMethods` 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.
As can be seen, the `RpcDispatcher` builidng 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.

3.4.7 DistributedHashtable

A `DistributedHashtable` (`JavaGroups.DistributedHashtable`) is derived from `java.util.Hashtable` and allows to create several instances of hashtables at different

locations. 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, i.e. implement interface `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 notified when such an event occurs. Notification is always a local process; 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).
Distributed hashtables allow members in a group to share common state across process and machine boundaries.

3.5 Customizing the JChannel protocol stack

This section applies only to the JChannel channel implementation, not to the Ensemble or iBus channel implementations.
When creating a `JChannel`, the properties of the underlying protocol stack can be specified as argument. A null argument means use the default composition of layers in the protocol stack (which may change between releases of JavaGroups).
A possible property specification may instruct JavaGroups to create an unreliable, UDP-based channel, another one may specify a loss-less, FIFO channel, and yet a third one may create a loss-less, FIFO, virtually synchronous, total order channel.
This section discusses how to create a JChannel with a customized protocol stack and the syntax of property strings will be described. It does not describe the protocols in detail; see 7.2 for details. Also, as JavaGroups is work in progress, new protocols may be added, or existing ones removed/renamed. Refer to the javadoc documentation for a complete reference.

3.5.1 Properties

A property string consists of a number of properties separated by colons:

```
"<prop1>(arg1=val1):<prop2>(arg1=val1;arg2=val2):<prop3>:<propn>"
```

Each property relates directly to a protocol layer, which is implemented as a Java class. When a protocol stack is to be created based on the above property string, the first property 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, as the string is parsed from left to right. Each property has to be the name of a Java class that resides in directory `./JavaGroups/JavaStack/Protocols`[8]. Note that only the base name has to be given, not the fully specified class name (`UDP` instead of `JavaGroups.JavaStack.Protocols.UDP`).
Each layer may have 0 or more arguments, which are specified as a list of name/value pairs in parentheses directly after the property. In the example above, the first protocol layer has

[8]The location of the directory may change in th future.

1 argument, the second 2, the third none. When a layer is created, these properties (if there are any) will be set in a layer by invoking the layer's `SetProperties` method[9].
As an example the property string below instructs JavaGroups to create a JChannel with protocols `UDP`, `PING`, `FD` and `GMS`:

```
"UDP(mcast_addr=228.10.9.8;mcast_port=5678):PING:FD(trace=true):GMS"
```

The `UDP` protocol layer is at the bottom of the stack, and it should use mcast address `228.10.9.8.` and port `5678` rather than the default IP multicast address and port. The only other argument instructs `FD` to output debug information while executing. Property `UDP` refers to a class `JavaGroups.JavaStack.Protocols.UDP`, which is subsequently loaded and an instance of which is created as protocol layer. If any of these classes are not found, an exception will be thrown and the construction of the stack will be aborted.

3.5.2 Dependency checking

Some protocol layers require the presence of other layers below or above them, e.g. GMS requires that a PING layer be somewhere below it to handle initial membership discovery. Whenever GMS needs to discover the initial membership, it delegates this to the PING layer by passing a `FIND_INITIAL_MBRS` event down the stack. PING would handle this event and send a `FIND_INITIAL_MBRS_OK` event back to GMS.
To be more precise: GMS does not actually require the presence of a layer called PING below it, but it requires a layer *that catches the* `FIND_INITIAL_MBRS` *event and returns the initial membership*. So not the type of the layer is important, but which events it catches (provides), and which ones it requires from others. Thus, any layer can provide events (*services*) to any other layer.
Class `Protocol` (and therefore every layer) contains 4 methods that describe which events are required and which events are provided by the layer. Thus, when a protocol stack is assembled, dependency checks ensure that a correct stack is created. Each method returns a list of event types that are required/provided to/by the layers above/below:

RequiredUpServices Returns a list of events (defined in class `Event`) that are required to be handled by one of the layers above this one.

RequiredDownServices Returns a list of events that are required to be handled (i.e., provided) by one of the layers below this one.

ProvidedUpServices Returns a list of events (traveling down the stack) that are handled by this layer (and sent back up the stack). Essentially the list of services provided to the layers above this one.

ProvidedDownServices Returns a list of events that are handled by this layer (received traveling up the stack). Essentially the services provided to the lower layers.

The default for all 4 methods is to return null: no services required, no services provided, neither up or down. A protocol implementor may choose to override any method to enforce the presence of an other protocol upon which the current layer's implementation depends, or to specify the services provided by the current protocol.
A dependeny check performed by the `Configurator` when constructing the protocol stack ensures that all requirements are met. If this is not the case, the construction will fail and the protocol stack will not be created.

[9] If in doubt what properties are accepted by a layer, this method can always be consulted in the source code.

3.5.3 Available Protocols

This section gives a brief overview of some of the most important protocol layers available in JavaGroups. As layers may be modified, added or removed, this information may not be accurate. Refer to the javadoc documentation for up-to-date information. Section 7.2 describes each protocol in detail.

UDP

UDP is meant to be at the bottom of a stack. It sends messages received from its upper layer using unreliable UDP and IP multicast, and passes up messages received from the network up to the next higher layer. 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.
UDP adds a header with the group name to each outgoing message and strips incoming messages off their header. If a header's group name does not match the channel's group name, it will be discarded (many channels can use the same IP multicast address, therefore the group name has to be used as discriminator).

PING

The function of this layer is to establish an initial membership when a channel is created. It does so by either broadcasting[10] a 'ping' message to a well-known IP address/port, or by contacting a gossip server (see 3.7.2). Every member that receives this message (in the PING layer) responds with its own address and the address of the coordinator. Based on the replies, the newly created channel then tries to contact the coordinator to join the group, using a unicast message. If no members reply (after some timeout), the new channel will be the first member in the group. If this layer is not present, then the group membership layer (GMS) will unable to establish contact with existing members, which will result in a number of group members not 'finding' each other (singleton groups).

FD

The failure detection layer (FD) functions as a suspicion service; it periodically pings its neighbor to check liveness. If it does not receive a response within a timeout period, it will multicast a suspicion message to all members. The coordinator will then typically remove that member from its local view and multicast a new view to all members.

GMS

The group membership service (GMS) layer maintains group membership for a certain group, joining new members, excluding left or crashed members, and installing new views. It is the main layer in JavaGroups that implements *virtual synchrony* [Bir96].

3.6 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

[10]The terms *multicasting* and *broadcasting* are used synonymously in the user's guide.

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 directory `./Demos`) shows how to use two different channels.

3.7 Disabling IP multicast

The UDP layer of the JChannel protocol stack by default uses IP multicast to broadcast messages to all group members. However, in some environments, using IP multicast may not be possible:

- IP multicasting is disabled between subnets in a local network. IP multicasting is enabled only within the same subnets, bridges drop IP multicast packets.
- Members of a group span multiple geographically separate networks, some routers between the networks drop IP multicast packets.
- Some members are behind a firewall (see section 3.8).

In these cases, instead of sending an IP multicast message to n group members, n unicast messages can be sent. Although not as efficient (and using more bandwidth), it is sometimes the only possibility to reach group members.
IP multicasting is used in 2 situations:

1. To discover the initial membership of a group (when a channel is created). This is done in the PING layer.
2. To broadcast messages to all group members. This is done in the UDP layer.

To disable the use of IP multicast, UDP has to be instructed to use unicast instead of multicast messages to send messages to the group (see 3.7.1) and a *gossip daemon* (see 3.7.2) has to be used to find the initial membership of a group. Note that it is possible to mix unicast and multicast: to use the gossip daemon to find the initial members, but then to use regular IP multicast to send messages, or to use IP multicast to find initial members and then use unicast messages to send messages to all members. However, in most cases, either IP multicast or unicast will be used for both types of communication.
To find out whether IP multicasting is enabled at all, the demo program `JavaGroups/tests/Mcast.java` can be used: it has to be started on each host, and messages sent by one Mcast process should be received by all others.

3.7.1 Disabling IP multicast

By setting the "ip_mcast" argument of property UDP to false, IP multicasting is disabled, e.g.:

```
"UDP(ip_mcast=false):..."
```

Whenever a message is received from the layer above it and IP multicasting is disabled, UDP sends the message as unicast message to each member of the current membership (which UDP cashes whenever receiving a VIEW event, cf. 4.1.2).

3.7.2 Using the gossip daemon

The gossip daemon is a process with which newly created JChannels, which use gossiping, register. A registration request containing the group name and the channel's address is sent to the gossip daemon periodically. Therefore, the daemon can be queried for all member addresses of a certain group. When a registration for a certain member is not renewed within a certain period of time, it becomes stale and will be removed by gossip. This tries to account for crashed members. When a member leaves the group, it will unregister with the gossip daemon.
The gossip daemon is implemented in `JavaGroups.JavaStack.GossipServer`. It is started given a port as follows (the port argument is optional, default port is 12001, host on which it is started is janet):

```
GossipServer -port 7879
```

The PING layer uses class `JavaGroups.JavaStack.GossipClient` to periodically register with the gossip daemon. It creates an instance of the client and starts it when the group is joined and stops it when the group is left. The gossip client periodically registers with the gossip daemon as long as it is member of the group and unregisters when it leaves the group. The PING layer has to know the address and port of the gossip daemon, which is done using arguments to property PING (using the above example of gossip daemon):

```
"...:PING(gossip_host=janet.cs.cornell.edu;gossip_port=7879):..."
```

If PING is parameterized with the gossip daemon's address and port, then gossiping is automatically enabled, otherwise it is disabled. If only 1 parameter is given, gossiping will be *disabled* !

3.7.3 Troubleshooting

Members don't find each other

When JChannel members don't seem to find each other, then multiple solutions exist: the first is to check whether IP multicast is enabled. If this is the case, and the problem persists, try to increase the `ip_ttl` argument of the UDP layer (default is 31). This may be too small for WAN settings. Try to run the `JavaGroups/tests/Mcast.java` demo program to see whether the problem persists. If this program works, and JavaGroups doesn't, then the protocol stack is probably setup wrongly, e.g.UDP arguments `mcast_addr` and `mcast_port` are not the same in all group members.
If IP multicast is disabled, try to use unicast, as described above. Make sure that a gossip daemon has been started before starting any channels that use gossiping.
To see whether messages are received at all, insert a TRACE protocol layer just above the UDP layer, e.g.:

```
"UDP:TRACE:PING(gossip_host=janet.cs.cornell.edu;gossip_port=7879):..."
```

This layer prints all messages it receives from the network or from the layer above it. See section 4 for a discussion on how to write one's own layer, e.g. a modified TRACE layer which only prints messages received from the network, and does not print messages passed from the layer above.

3.8 Firewalls

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 2 TCP ports are enabled for outgoing ports; one for the gossip daemon and one for the JRouter.

3.8.1 Using TUNNEL, JRouter and Gossip to tunnel firewalls

JavaGroups tunneling solution works as follows. A channel inside a firewall has to use protocol layer TUNNEL instead of UDP as bottommost layer in the stack. This layer uses the gossip daemon (outside the firewall) to register itself (periodically) and to retrieve the initial membership for its group. This is done using TCP. It also establishes a TCP connection to a *JRouter daemon* (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. JRouter 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 JRouter 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.
PING has to be given the address and port of the gossip daemon (see 3.7.2 above) and TUNNEL has to be configured with the address and port of the JRouter daemon, e.g.:

```
"TUNNEL(router_host=janet;router_port=12001):
 PING(gossip_host=janet;gossip_port=12002):..."
```

This setup assumes a JRouter daemon is running on host `janet` at port `12001` and a gossip daemon is running on the same host at port `12002`.
Any time a message has to sent, TUNNEL forwards the message to JRouter, which distributes it to its destination: if the message's destination field is null (send to all group members), then JRouter 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 JRouter. If the destination is a valid member address, then that member's TCP connection is looked up, and the message sent forwarded to it[11].
To tunnel a firewall using JavaGroups, the following steps have to be taken:

1. Check that 2 TCP ports (e.g. 12001 and 12002) are enabled in the firewall for outgoing traffic.

[11] To do so, JRouter has to maintain a table between groups, member addresses and TCP connections.

2. Start the gossip daemon:

```
GossipServer -port 12001
```

3. Start JRouter:

```
JRouter -port 12002
```

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

The general setup is shown in fig. 3.5.

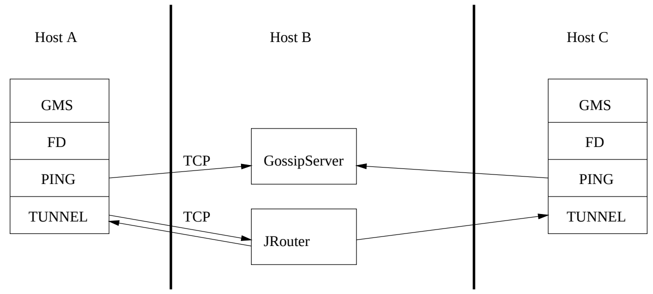

Figure 3.5: Tunneling a firewall

First, the gossip and JRouter daemons are created on host B. Note that these should be outside the firewall, and all channels in the same group should use the same gossip and JRouter daemons. When a channel on host A is created, its PING layer will register its address with the gossip daemon and retrieve the initial membership (assume this is C). Now, a TCP connection with JRouter is established by A; this will persist until A crashes or voluntarily leaves the group. When A multicasts a message to the group, JRouter 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 JavaGroups: the HTTP server would be located on host B and the gossip and JRouter daemons 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 gossip and JRouter daemons constitute a single point of failure (if host B crashes), 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 JRouter), 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.
Therefore, the possibility exists for users of JavaGroups to create their own tunneling mechanisms, modifying for example the TUNNEL layer and the gossip and JRouter daemons[12].

[12] The alternatives outlined below are currently (March 1999) not implemented. However, the modifications proposed should not be difficult to implement.

3.8.2 Using multiple JRouters

The first, second and third problem can be solved by using a *replicated gossip daemon* and multiple instances of JRouter.

A replicated gossip daemon can be implemented using building blocks `DistributedHashtable` (see 3.4.7): all gossip daemons would join a common (administrative) group, and every registration to one gossip daemon would be propagated to all gossip daemons. A lookup, however, could be performed on the local gossip daemon (since no distributed state is modified).

Multiple JRouters can be used when IP multicast is enabled outside the firewall, and only a few channels are actually located behind firewalls: every channel behind a firewall would connect to a JRouter (outside the firewall) via TCP (as in the current solution), but the JRouters would use IP multicast to forward messages between themselves instead of TCP. This is shown in fig. 3.6.

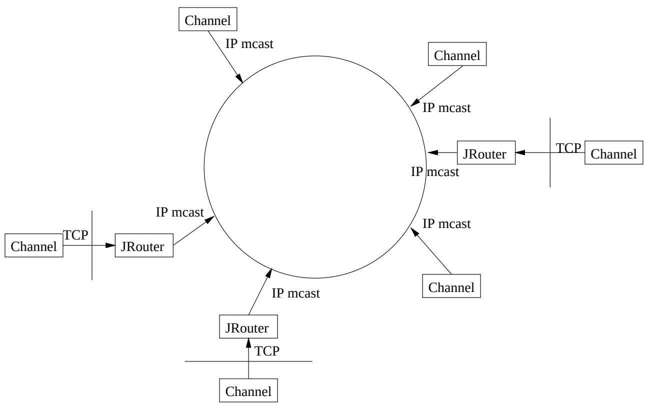

Figure 3.6: Modified JRouter using IP multicasting

Every channel that is not behind a firewall would use IP multicasting to broadcast and receive messages to/from other group members: their protocol stack would contain UDP as bottommost layer. Every channel *behind* a firewall would use a proxy JRouter (outside the firewall), behaving like a regular channel. The JRouter would listen on the group's multicast address and port for incoming IP multicast traffic, and forward it to the channel behind the firewall via the TCP connection established by the latter. The channel behind the firewall would use the TUNNEL protocol as its bottommost layer.

3.8.3 Merging JRouter and gossip

Another problem listed above is that most firewalls will typically only allow outgoing TCP connections on port 80 (HTTP). In most cases, it will not be possible to 'open' additional TCP ports (unless the JavaGroups user is the system administrator). Therefore, the gossip and JRouter daemons would have to be merged and listen on port 80. Merging the two daemons should be rather straight forward, as they contain overlapping functionality anyway (it is on the author's todo list). A merged daemon would be accessed by both the PING (registration, initial member lookup) and TUNNEL (message routing) protocol layers.

The TUNNEL and PING protocols could contact the merged daemon directly via a TCP connection to port 80 which would persist until the channel leaves the group. However, in the case of Java applets, an HTTP server would also have to listen on the same port, as applets are only allowed to connect back to the host from which they were downloaded[13],

[13]This is changed in JDK 2.0 with the improved security model.

leading to port conflicts. In this case, the merged daemon would have to be modified to act as a *web server proxy*: whenever it receives a request at port 80t, and the request is not from the TUNNEL or PING protocols, it would pass the request on to an HTTP server running on a different port (e.g. 8000).

Chapter 4

Implementing Protocols Using JChannel

This chapter discusses how to write protocols for JavaGroups. It describes the architecture of the JChannel protocol stack (creation, setup and event flow) and the various types of protocols that can be written subclassing different protocol 'templates'. It is intended for people who want to write their own protocols or modify an existing one.
The beauty of protocol layers is that they are independent from each other, therefore they can easily be replaced by other versions. They're also (usually) fairly small, focusing on just one functionality, which makes them easy to understand. If a protocol becomes too large, incorporating multiple functionalities, it should be broken into a number of smaller ones.
Protocol layers can be added to JavaGroups (currently by placing them in directory `JavaGroups/JavaStack/Protocols`), without modifying JavaGroups' core code. Note, however, that the protocol code will be loaded into and running with JavaGroups' core code in the same JVM, therefore it may affect the whole JChannel[1].
Since each protocol has the same interface and is completely independent from other protocols, it can be stacked on top or below every other protocol. However, this wouldn't make sense in most cases semantically: a certain order has to be observed. Therefore, although protocols are *syntactically independent*, they are not *semantically independent*: for example the GMS protocol requires that a PING (or PING-like) protocol be present somewhere below it for the discovery of the initial membership. More precisely: the GMS protocol at some point will send a `FIND_INITIAL_MBRS` event down the stack (see 4.1.2), which it expects to be handled by some other protocol. That protocol is required to perform the task of finding the initial members and passing a `FIND_INITIAL_MBRS_OK` event up the stack, containing the initial membership. The GMS layer will wait for the response, or time out and become a singleton member in the group. Therefore, although GMS does not syntactically depend on PING (it does not call a method of PING), it does have a dependence on some layer below it that answers to `FIND_INITIAL_MBRS` events. Yet this kind of dependence is less binding than a method-call binding: any layer that sees fit may answer to the request for initial membership, and GMS does not even know which layer it will be.

[1] However, exceptions is the JChannel protocol stack, caused by a malfunctioning protocol, are caught at the JChannel level, and can therefore not cause havoc to the channel itself. Yet, malfunctioning protocols may affect messages (adding wrong headers, modifying message contents, or dropping messages), therefore the reliability of a protocol stack is the sum of the robustness and reliability of all its layers.

4.1 The JChannel protocol stack

4.1.1 Utility classes

The following utility classes are used by the JChannel protocol stack.

Address

Addresses (`JavaGroups.JavaStack.Address`) are used to represent group member locations, they are mainly used in the destination and source fields of messages (see 3.1.3). An address at the `Channel` level is simply an opaque `Object`, however, channel implementations will typically use subclasses to represent implementation-specific addresses. The `JChannel` implementation uses the combination of an IP address and a port for its addresses. An `Address` contains a method for comparing them to other `Addresses`; this will be used to establish a deterministic ordering between sets of addresses (used by `Membership` below).
The JChannel implementation knows about `Addresses`: the UDP layer for example creates a local address when started (in the form of an `Address`). However, when passed up to the application, it would be passed as an `Object`. JChannel protocol layers would cast the `Object` to an `Address`, but addresses would not be allowed to be interpreted as any subclass when crossing the channel boundary: applications treat addresses as opaque *handles* which can be used to send messages. This allows to port JavaGroups onto different group transports, with different address schemes, without affecting applications.

Membership

The `JavaGroups.JavaStack.Membership` class maintains a vector of member addresses. Like vectors, addresses can be added (only if not yet present) and removed, the membership can be set/merged from/with another vector, or sorted (based on `Address.Compare`). Every view of a member (cf. 3.1.5) maintains a membership: it is crucial for certain algorithms that the order of the membership is the same in all views, which is guaranteed by `Membership`. This simplifies for example coordinator election; the first member is assumed to have stayed the longest in a group and can therefore be chosen as coordinator. As all members have the same views, every member will make the same decision, and therefore, complicated election algorithms can be avoided.

ViewId

Virtual synchrony [Bir96] guarantees that all non-faulty members of a group will receive the same sequence of views: if a member P receives V1, V2 and V3, then all members of that group will also receive V1, V2 and V3, in that order (unless they are removed from the group). Also, all members will receive the same *set* of messages between views[2].
`ViewIds` (`JavaGroups.JavaStack.ViewId`) are used to differentiate between views, they are attached to messages generated by the GMS layer and removed by the corresponding GMS layer at the receiver's side. A `ViewId` contains a logical timestamp (Lamport time) and its creator (`Address`). These 2 fields are represented in the `View` class (cf. 3.1.5) as fields `creator` and `id`.
The main occurrence of `ViewIds` is in the GMS layer, where they are used to enforce virtual synchrony properties, e.g. every message has to be delivered in the view in which it was sent (otherwise the message is discarded), views must be received in increasing order of their logical timestamp (views with timestamps smaller than the local view are discarded) etc.

[2]Virtual synchrony does not guarantee any ordering properties for the messages between views, this would have to be handled by a separate layer (e.g. TOTAL).

4.1.2 Events and messages

The difference between events and messages is that events are used for intra-stack communication, whereas messages are used for inter-stack communication. That means that in a JChannel protocol stack, only *events* are passed between layers, whereas *messages* are exchanged between stack, i.e. put on the network and received from the network by the bottommost layer. Events are used to pass information relevant to a single stack between layers. Messages are *wrapped* in events by the bottommost layer and passed up the stack. When `JChannel.Receive()` is called by an application, if the event's content on the JChannel queue is a message, then it is 'unwrapped' and returned. When a message travels down the stack (wrapped in an event), the bottommost layer will unwrap the message and put it on the network.
An event has a type (integer) and an argument (object). Message events have for example type `Event.MSG` and as argument the message. The type of an event can be determined calling method `Event.GetType`.
The number and types of events used in JavaGroups is fixed and determined in file `Event.java`. However, in some cases, developers of new protocol layers might find it useful to add new event types. There are two possibilities: either a new event is added to `Event.java`, modifiying the source code, or the predefined event `USER_DEFINED` is used. The former solution requires feedback into the main distribution, otherwise a number of incompatible distributions might exist. Also, the same event type numbers might be chosen by different developers, requiring disambiguation when merging those modifications back into the original distribution. The latter allows developers to use their own event types, embedded in the `USER_DEFINED` event type. The user-specific event would be contained in the argument of the user-defined event. This has the advantage that no source code needs to be changed. The types currently available for events are listed in table 4.1[3].

Although the events and messages are distinctively different, the term 'message' is used synonymously with 'event' in the following sections, where the flow of events/messages is more important than the difference between them.

4.1.3 Architecture

The architecture of the protocol stack is shown in fig. 4.1.
A JChannel is connected to a `JavaGroups.JavaStack.ProtocolStack` object which manages the protocol stack. The latter uses a configurator (`JavaGroups.JavaStack.Configurator`, see 4.1.5) to create and configure a protocol stack, and essentially functions as an *adapter* between the topmost protocol and the JChannel. Whenever `ProtocolStack` receives a message from above, the message is passed on to the topmost protocol layer, and when it receives a message, it is passed up to the JChannel. When a JChannel is created, it will create an instance of `ProtocolStack` and use it to manage the stack on its behalf. When an application connects to the channel, the protocol stack will be started, and when it disconnects the stack will be stopped[4]. When the channel is closed, the stack will be destroyed, releasing its resources. To do this, `ProtocolStack` again uses the help of `Configurator`.
Protocol layers are connected through bidirectional FIFO queues. Each protocol has 2 queues: one for storing incoming messages (traveling up the stack) and one for outgoing messages (traveling downwards). Each queue is handled by a separate thread: its only function is to remove 1 message from the queue (it will block until a message is added to the queue) and pass it up/down to the next layer by invoking `Up` or `Down` respectively. The next layer will then store it in its incoming or outgoing queue. If a message traveling down is received by

[3]File `JavaGroups/Event.java` will always be the most current source of information concerning events.
[4]For initialization and de-initialization see 4.1.6.

Event Type	Argument	Description
START	–	Sent up the stack. The protocol stack is started. Generates a response from bottommost layer. The layer may start its work now
START_OK	–	Response to START, generated from bottommost layer (sent up)
STOP	–	Stop working, UDP layer responds with a STOP_OK
STOP_OK	–	Generated by bottommost layer in response to STOP
MSG	Message	Message traveling up or down the stack
CONNECT	Group name	Connect the channel to the given group. The GMS layer will start a JOIN protocol in response
CONNECT_OK	–	Response generated by GMS layer when JOIN protocol has completed. Sent up the stack.
DISCONNECT	Address	Leave the group. GMS will start LEAVE protocol and send up LEAVE_OK when done
DISCONNECT_OK	–	Generated by GMS layer, sent up the stack
VIEW_CHANGE	View	Generated by GMS, sent up and down the stack when a new view was received (member joined, left group)
GET_LOCAL_ADDRESS	–	Return the local address, response should be generated by bottommost layer. Deprecated.
SET_LOCAL_ADDRESS	Address	Generated by bottommost layer, either in response to GET_LOCAL_ADDRESS or when started (after sending START up the stack)
FLUSH	–	Not yet used
FLUSH_OK	–	Not yet used
SUSPECT	Address	Generated by failure detection layer (e.g. FD). Sent up the stack, handled by the GMS layer. Argument is the suspected member
FIND_INITIAL_MBRS	–	Determine the initial membership. Generated by GMS layer and sent down. Handled by PING layer, which also generates response and sends it up the stack
FIND_INITIAL_MBRS_OK	Vector	Response to FIND_INITIAL_MBRS. Generated by PING, argument is a Vector of PingRsps. Sent up the stack, caught by GMS
MERGE	Vector	Generated by MERGE layer when it detects other members with the same group name, but not in the local view. Sent up the stack. Handled by GMS layer
TMP_VIEW	View	Install a temporary view (not an official one). Use by GMS and other layers to make UDP send out a message to the correct membership. Will in most cases be followed by VIEW_CHANGE
BECOME_SERVER	–	Sent up/down the stack by the GMS layer when a client has successfully joined the group.

Table 4.1: Event types and their arguments

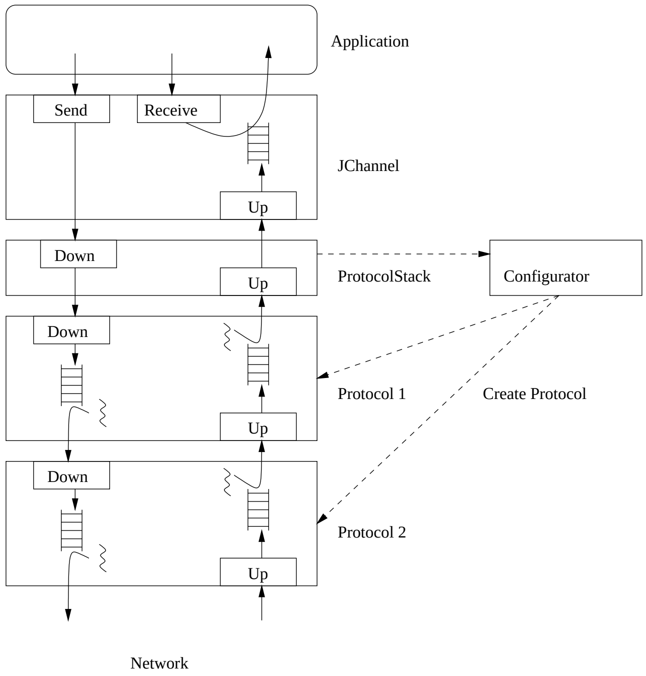

Figure 4.1: Architecture of the JChannel protocol stack

the bottommost layer, it will be put on the network. If a message traveling up is received by the topmost layer, it is passed on to the `ProtocolStack`, which forwards it to the JChannel. JChannel stores all messages received by the protocol stack in a queue, where it can be removed by an application calling `JChannel.Receive` (or by a `PullPushAdapter`, see 3.4.1). Messages sent by the JChannel are just passed on to the protocol stack, which in turn forwards it to its topmost protocol layer.

4.1.4 Class `Protocol`

All protocols inherit indirectly or directly from `JavaGroups.JavaStack.Protocol`. A `Protocol` defines the default behavior for all protocol layers, it for example sets up the incoming and outgoing queues, creates the two threads that pass messages between queues and maintains a reference to the layer above and below the current layer. All behavior can be overridden in subclasses, it is for example possible to write a protocol layer that does not use threads and queues altogether.
The innards of a `Protocol` layer are shown in fig. 4.2.
Each protocol contains the following variables:

`up_prot, down_prot` Each layer has a reference to the layer above and below it. If the down reference is null, it means that the layer is the bottommost layer in the stack (e.g. UDP). These are used to pass events up or down the stack.

`stack` A reference to the `ProtocolStack` object itself. Layers used this to obtain information global to the stack. *This field is deprecated and will be removed in a future release: events will be used instead to distribute information between protocol stack and individual layers. This breaks the dependence on* `ProtocolStack`.

`up_queue, down_queue` 2 queues for storing messages from the layer below, to be processed and possibly passed on to the layer above, and for storing messages from the layer above, that are traveling downward.

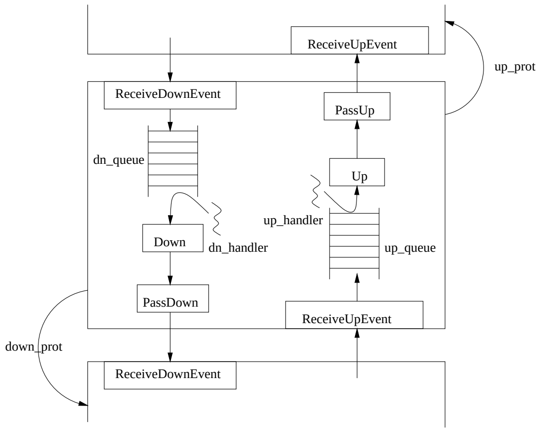

Figure 4.2: Architecture of `Protocol`

`up_handler`, `down_handler` 2 threads handling the message passing in a layer. The `up_handler` removes messages from `up_queue` and calls method `Up`. This method may be overridden by subclasses of `Protocol` to process the message[5]. The default `Up` method calls `PassUp` which in turn calls `ReceiveUpEvent` on its `up_prot` reference to the layer above it. `ReceiveUpEvent` just adds the message to the next layer's `up_queue`.

The processing for the `down_handler` is the same, except that messages travel down the stack. The 2 threads are started and stopped by methods `StartInternal` and `StopInternal`, which are called by the `Configurator` (see 4.1.5) when a stack is created and destroyed.

The 2 methods of `Protocol` that may need to be overridden in subclasses are `Up` and `Down`: layers will typically modify, reorder, or discard messages, and this is done in these 2 methods. Also important are methods `PassUp` and `PassDown` to forward a message to the layer above / below. They are typically called from with `Up` or `Down` to pass a message up/down the stack. By overriding methods `StartInternal` and `StopInternal`, subclasses can govern the creation of the 2 threads: if both methods are empty, then no threads will be created, and the `ReceiveUpEvent` / `ReceiveDownEvent` methods will have to call `Up` / `Down` directly.
The default mechanism guarantees that messages are received in FIFO order, and that at most 1 down and 1 up message is handled in a protocol layer at the same time. Therefore, a message Q passed down the stack after P will never 'pass' P, i.e. if any layer receives both P and Q, then P will always be received before Q (unless a layer discards either or bother of the two messages).
There are 2 subclasses of `Protocol`: `MessageProtocol` (cf. 4.3.1) and `RpcProtocol` (cf. 4.3.2): whereas both enable peer communication between layers of different JChannel protocol stack instances, the former uses (synchronous) message passing to do so, while the latter employs (synchronous) remote method invocation.

4.1.5 Construction of the protocol stack

When a new JChannel is created, it instantiates a `ProtocolStack` object, giving it the property string and a reference to itself as arguments, which the `ProtocolStack` object stores in its instance variables: it will use the JChannel reference to pass messages up to the JChannel, and the property string to setup and configure the protocols.
After creating the protocol stack object, JChannel calls method `Setup` on it. This triggers a whole chain of actions on the side of the stack object, at the end of which the `ProtocolStack`

[5]The overridden method could possible reorder or even discard the message altogether.

will consist of list of protocol layers, connected with their neighboring layers, with queues set up and threads waiting to receive messages for processing. The `ProtocolStack` does not do this work itself, but it calls `Configurator.SetupProtocolStack` to do so. This method parses the property string, instantiates layers represented by the Java classes expressed by the colon-separated string, sets the arguments in each layer, connects the layers (`up_prot`, `down_prot`) and returns a reference to the topmost layer. The `ProtocolStack` then sets the `up_prot` instance variable of the topmost layer to point to itself, so that it will receive all messages, which it will pass on to the JChannel. Note that messages are passed *down* by the JChannel using `ProtocolStack`'s `Down` method (see fig. 4.1).
Finally, the `ProtocolStack.Setup` method starts the protocol stack by calling `Configurator.StartProtocolStack` with a reference to the bottommost layer as argument. For initialization and de-initialization of the protocol stack see 4.1.6.

4.1.6 Initialization and de-initialization of the protocol stack

The chain of events caused by a) channel creation, b) connection, c) disconnection and d) channel close are summarized below.

Creation of JChannel

1. Create new `ProtocolStack`: `new ProtocolStack()`
2. The constructor calls `ProtocolStack.Setup()`
3. `Setup` creates all layers and sets up queues between layers, ready to process events

`JChannel.Connect()`

1. JChannel calls `ProtocolStack.Start()`
2. `Start` sends a `START` event from bottom of the stack upwards
3. UDP initializes its sockets (unicast and multicast). From now on, it is able to receive messages. Then it sends a `SET_LOCAL_ADDRESS` event up the stack. This allows various layers to cache the local address of the stack. Then it sends a `START_OK` event up the stack.
4. When `ProtocolStack.Start` receives the `START_OK`, it returns. (Note that there is also a timeout, if it expires, `ProtocolStack.Start` throws an exception)
5. JChannel sends down a `CONNECT` event, waits for `CONNECT_OK`
6. Various layers process this event and pass it down
7. GMS passes the event down. Then it establishes the initial membership. Finally it passes up a `CONNECT_OK` event.
8. `JChannel.Connect()` waits for `CONNECT_OK` event, or, it the timeout expires, returns without having received a `CONNECT_OK`.

`JChannel.Disconnect()`

1. Sends down a `DISCONNECT` event, waits for a `DISCONNECT_OK` event
2. GMS updates the membership. Then it passes down the event. Then it sends up a `DISCONNECT_OK` event.

3. JChannel.Disconnect() waits for the DISCONNECT_OK event, or, if the timer expires, throws an exception

4. JChannel.Disconnect stops the protocol stack calling ProtocolStack.Stop()

5. Stop sends a STOP event down the stack

6. When the lowest layer (UDP) receives STOP, it closes all its sockets. Then it sends up a STOP_OK event.

7. When ProtocolStack.Stop receives STOP_OK, it returns. Otherwise, if the timer expires, it returns too.

JChannel.Close()

1. Calls JChannel.Disconnect() if still connected

2. Clears the JChannel-internal messages queues (for incoming messages)

3. Calls ProtocolStack.Destroy()

4. Destroy destroys all queues between layers and finally all layers, too.

5. JChannel.Close() makes the channel unusable. Any calls to it will result in an exception. To participate in a group, a new JChannel has to be created.

4.2 Creating a sample protocol

This section discusses the implementation of the fragmentation protocol layer (FRAG, source file JavaGroups/JavaStack/Protocols/FRAG.java) as an example of a relatively simple layer which fragments large messages into smaller ones and reassembles them at the receiver's end before passing them up the stack.
FRAG essentially works as follows: whenever a message exceeding a maximum size is passed to the FRAG layer from the layer above, it is fragmented into a number of smaller messages and a header with the original message's ID, the fragment ID and the number of fragments added to each fragment. The fragments are then sent down the stack (the original message is discarded). At the receiver's side, the FRAG layer removes the header containing the original message's ID and the fragment ID and adds the fragment to the corresponding entry in a fragmentation table. When all fragments have been received, the original message is reconstructed from the fragments and passed up the stack (and the fragments are removed from the fragmentation table).
The FRAG layer extends class Protocol and overrides methods GetName, SetProperties, Up and Down.
GetName is used to identify the protocol and is used by several building blocks to differentiate between protocols: e.g. the configurator (cf. 4.1.5) checks whether duplicate protocol names occur in the stack. If this is the case, stack construction ends with an exception.
SetProperties (see 3.5.1) is used to convey parameters to the FRAG layer: current properties include a debug flag (trace) which, if set to true, causes the protocol to write diagnostic messages to stdout during its execution and frag_size which specifies the maximum size for message. If this size is exceeded, messages will be fragmented.
The fragmentation header is represented as a separate, serializable class. It was previously shown in section 3.1.3. It contains fields msg_id, the original message sequence number, frag_id the fragment ID and num_frags, the number of fragments to be expected. An example of 3 successive fragmentation headers for original message 45 seen by the receiver would be:

```
{45, 0, 3} {45, 1, 3} {45, 2, 3}
```

The 3 fragments 0, 1 and 2 would then be reassembled into the original message 45, and the latter would be passed up the stack.
Method `Down` is shown below:

```
public void Down(Event evt) {
    if(evt.GetType() == Event.MSG) {
        Message msg=(Message)evt.GetArg();
        long    size=msg.Size();
        if(size > frag_size) {
            Fragment(msg);  // Fragment and pass down
            return;
        }
        else
            msg.AddHeader(new FragHeader(msg.GetId(), 0, 0)); // Add dummy header
    }
    PassDown(evt);  // Pass on to the layer below us
}
```

When a message is passed to the FRAG layer by the layer above it (calling `Down`, FRAG checks whether the event *is* actually a message (not just a regular event). If this is the case, the message is extracted from the event and its size checked. If the size is larger than `frag_size` (set by `SetProperties` previously), the message is fragmented using method `Fragment`. Otherwise, a dummy header with a 0 for the number of fragments is created, added to the message and the message is passed down the stack. All fragmentation headers with a 0 for the fragment number are just passed up the stack by the corresponding FRAG layer on the receiver's side (no need for reassembly).
Note that FRAG *always* adds a header to every single message. This has the disadvantage that every message gets a little bit bigger, although most of the messages may never need to be fragmented ! If this pattern is known (many messages are very small, only a few will need to be fragmented), then a better mechanism can be used: fragmentation headers are only added when the message needs to be fragmented, otherwise no header is added. On the receiver's side, if no header is present, the message is passed up the stack (non-fragmented message). Otherwise, a fragmented message was encountered and has to be reassembled accordingly.
The `Fragment` method is shown below:

```
private void Fragment(Message msg) {
    ByteArrayOutputStream  out_stream;
    ObjectOutputStream     out;
    byte[]                 buffer;
    Vector                 fragments;
    FragHeader             hdr;
    Message                frag_msg;
    Object                 dest=msg.GetDest(), src=msg.GetSrc();
    long                   msg_id=msg.GetId(), rsp_id=msg.GetRspId();
    boolean                oneway=msg.IsOneway();

    try {
        out_stream=new ByteArrayOutputStream();
        out=new ObjectOutputStream(out_stream);
        out.writeObject(msg.GetHeaders());     // write headers
        out.writeObject(msg.GetBuffer());      // write buffer
```

```
            buffer=out_stream.toByteArray();
            fragments=Util.FragmentBuffer(buffer, frag_size);  // create Vector of byte[]
            for(int i=0; i < fragments.size(); i++) {
                frag_msg=new Message(dest, src, (byte[])fragments.elementAt(i));
                if(i == 0) frag_msg.SetId(msg.GetId());
                frag_msg.SetRspId(rsp_id);
                if(oneway) frag_msg.SetOneway();
                hdr=new FragHeader(msg_id, i, fragments.size());
                frag_msg.AddHeader(hdr);
                PassDown(new Event(Event.MSG, frag_msg));
            }
        }
        catch(Exception e) {
            System.err.println("FRAG.Fragment(): " + e);
        }
    }
```

It serializes the original message's headers and buffer to a byte array, and then breaks that array into several smaller arrays, using `Util.FragmentBuffer`, giving the buffer and fragmentation size as arguments The fragmented byte buffers are returned in a vector. Now a new message is created for every element in the vector. A `FragHeader` is created for every message, containing the original message's sequence number, the current fragment number and the total number of fragments to be expected for the original message's sequence number. The header is then added to the message and the latter passed down the stack. Note that the original message is discarded.

Fragmented or non-fragmented messages are handled by the receiver's FRAG layer in method `Up`:

```
    public void Up(Event evt) {
        byte[]                m;
        ByteArrayInputStream  in_stream=null;
        ObjectInputStream     in=null;
        Stack                 headers;
        byte[]                tmpbuf=null;
        int                   len;

        if(evt.GetType() == Event.MSG) {
            Message    msg=(Message)evt.GetArg();
            FragHeader hdr=(FragHeader)msg.RemoveHeader();

            if(hdr.num_frags > 0) {
                m=frag_table.Add(msg.GetSrc(), hdr.msg_id, hdr.frag_id,
                                 hdr.num_frags, msg.GetBuffer());
                if(m != null) {
                    msg.SetId(hdr.msg_id);
                    try {
                        in_stream=new ByteArrayInputStream(m);
                        in=new ObjectInputStream(in_stream);
                        headers=(Stack)in.readObject();
                        tmpbuf=(byte[])in.readObject();
                    }
                    catch(Exception e) {
                        System.err.println("FRAG.Up(): " + e);
                        return;
                    }
                    msg.SetHeaders(headers);
```

```
                msg.SetBuffer(tmpbuf);
                PassUp(new Event(Event.MSG, msg));
            }
            return;
        }
    }
    PassUp(evt); // Pass up to the layer above us
}
```

If the event is not a message event it is simply passed up the stack. Otherwise, the fragmentation header is removed from the message. If the number of fragments is 0, then the message is not fragmented and passed up. Otherwise, the fragment is added to the *fragmentation table*, which is an internal class used only by FRAG, using method `Add`. This method checks whether there is already an entry for the original message's sequence number. If not, a new one will be created. Each entry keeps track of all fragments received so far. When the last missing fragment has been received, `Add` reassembles the fragments into the original byte buffer[6], removes the entry together with all fragments and returns the byte buffer. In all other cases, `Add` returns `null`.
When a byte buffer has been returned from `Add`, the headers and message buffer of the original message are reconstructed from it, and the message is passed up the stack.
Note that the FRAG layer requires loss-less delivery, i.e. if a fragment is lost, and no retransmission is initiated (by some other layer), FRAG will not be able to reconstruct the original message. Also, FRAG does only focus on fragmentation/defragmentation, it does not guarantee FIFO ordering with respect to sender. This again follows the design that layers should only focus on one functionality, making them independent from others and relatively simple.
This example of a protocol layer was a typical example: the FRAG layer is not overly complicated, and only required overriding the `Up` and `Down` methods of `Protocol` from which it is derived. More sophisticated protocols will be discussed in the next sections.

4.3 Peer protocols

Protocols can be categorized according to their interaction with other protocols: some protocols interact primarily with their neighbor protocols `in the same stack`, and others interact with their corresponding peer instances on different protocol stacks. Therefore this type of interaction can be called *vertical interaction*:

- pass a message to its neighbor (up or down the stack) unchanged,
- add a header to the message, but not modify it,
- modify the message, add a header and pass it on (e.g. encryption, fragmentation layers),
- reorder the message (e.g. FIFO, TOTAL layers) or
- discard the message (e.g. DUPL layer)

These tasks can easily be done with protocol layers inheriting from `Protocol`.
The second types of protocols interact not primarily with other layers in the same stack, but with their equivalent peer instances *in different stacks*. Therefore this type of *peer protocol* interaction is called *horizontal interaction*. A typical peer protocol is a group membership service (GMS) layer which needs to interact with the GMSs in all member stacks to e.g. install a new view.

[6]Using `Util.DefragmentBuffer`, cf. 3.1.10.

Peer protocols frequently involve sending a request to all member peer protocols, where the request is received and processed, and possibly a response sent back, received by the originator of the request. The above case requires quite some work on behalf of the programmer: a request has to be given a unique request number, it has to be sent to all members, the receiving stack has to make sure that the correct destination layer receives it, then it will be processed, possibly a response may need to be returned, in the sender's stack it has to be sent to the correct (originating) layer, and the requester has to correlate the response(s) with the original response. This is especially complex when a sender can send multiple requests concurrently.
The two protocols discussed below try to address this issue. Their goal is to provide a minimal framework for (synchronous) peer protocol interactions. It uses classes `RequestCorrelator` and `GroupRequest` to do so (see sections 3.4.3 and 3.4.4 respectively).
The idea of peer protocols is that every peer can *send*, but also *receive* requests, i.e. every peer is a client but also a server possibly at the same time. A peer in the client role would send a request to all peers and receive the response(s) synchronously, i.e. it would block until all responses have been received. A peer in the server role would receive requests, process them and return a response. The two basic elements of 'sending request/receive response' and 'receive request/return response' have been incorporated into `MessageProtocol`. This class is derived from `Protocol` and extends it with peer interaction as described above. Any protocol that needs to interact with peer protocols may benefit from the additional methods present in this class, by inheriting from it rather than from `Protocol` directly.
Whereas `MessageProtocol` deals with *messages*, `RpcProtocol` deals with *methods*. The latter allows peers to define methods which can then be invoked remotely by other peers (instead of receiving messages as with the former class). `RpcProtocol` is derived from `MessageProtocol`. The two types of peer protocols are discussed below.

4.3.1 MessageProtocol

The architecture of `JavaGroups.JavaStack.MessageProtocol` is shown in fig. 4.3.

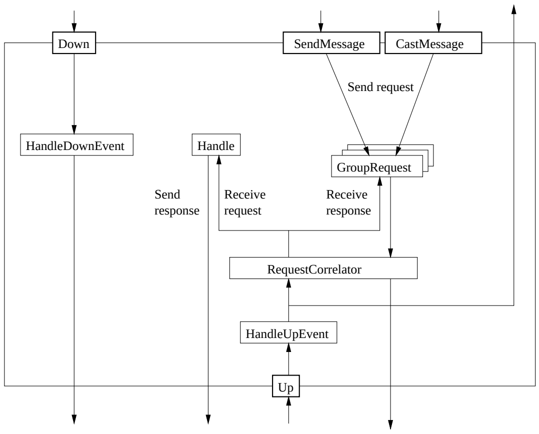

Figure 4.3: Architecture of `MessageProtocol`

Methods `Up` and `Down` are inherited from `Protocol`, but they are made *final*, i.e. they cannot be overridden in further subclasses of `MessageProtocol`. To handle events, methods `HandleDownEvent` and `HandleUpEvent` are introduced: `Up` will call `HandleUpEvent` and, if the return value is true, pass the message up the stack, or discard it. The same applies to `Down`: if `HandleDownEvent` return false, the message is discarded, otherwise it is passed down to the next layer.

Methods `SendMessage` and `CastMessage` provide synchronous group messages (discussed below): they make use of `GroupRequests` and a `RequestCorrelator` (see 3.4.4 and 3.4.3) to do so. Any time one of the two methods is called, a new group request will be created and registered with the request correlator. The group request object then uses `RequestCorrelator.SendRequest` to send a message. The request correlator will assign a unique request number to the request, add a header to is so that it will be received by the correct peer layer in the receiver's protocol stack, and passes it down the stack.
The peer layer in the receiver's stack receives the message, and calls `HandleUpEvent` to do any local processing based on the message. If false is returned, the message is discarded. Else method `Receive` of the request correlator is called. It checks the header to see whether the message was generated by the corresponding peer layer in the sender's stack and, if so, receives it. Otherwise it rejects it by returning false. In this case, the message is just passed on to the layer above the current one. Otherwise, the header is scrutinized to check whether the message is a response or request. If it is a response, the corresponding request is looked up and the message is delivered (calling method `RspCollector.ReceiveResponse`). Otherwise, method `Handle` (which is overridden by a subclass) is called. It processes the request and returns an object. If the request was not one-way, i.e. requires a response, the object is serialized and send back in a response message.
The important methods to implement a peer protocol are `CastMessage`, `SendMessage` and `Handle`:

```
public RspList CastMessage(Vector dests, Message msg, int mode, long timeout);
public Object SendMessage(Message msg, int mode, long timeout)
              throws Timeout, Suspected;
public Object Handle(Message req);
```

`CastMessage` sends message `msg` to all members of `dests` and waits for `mode` responses or a timeout (whichever comes first) and returns a response list (cf. 3.4.4). Mode can be any of the following:

FIRST Wait only for the first response and discard all others.

ALL Wait for all responses.

NONE Wait for no response. This makes the message one-way, i.e. no response will be generated at the receiver.

MAJORITY Wait for the majority of all receivers to respond.

When waiting for any responses, and a group member has been suspected, then that response is marked as suspected in the response list, without causing the caller to block indefinitely. Alternatively, a timeout can be employed: when `mode` responses are expected and not all have been received yet, when the timeout occurs, the call returns and the resulting response list contains all the responses received so far. A timeout of `0` means to wait indefinitely.
`SendMessage` is similar to `CastMessage` except that the message is sent only to a single group member. Argument `mode` should be either `NONE` or `FIRST` (although `ALL` and `MAJORITY` would also work). Instead of returning a response list, an object is returned, representing the response from a single receiver (may also be null). If the receiver is suspected, or a timeout has occurred, then the corresponding exception is thrown.
Callback `Handle` will probably be overridden by a peer protocol that processes incoming requests. It accepts the request in the form of a message. The body of the message will contain the request data, e.g. a request object. Depending on the request, `Handle` will perform some processing and optionally return a result value (or null). If the request was not one-way, a response message will be generated (with the result in the message's buffer) and sent back to the sender of the request.

4.3.2 RpcProtocol

This class is similar to `MessageProtocol`, but instead of sending and receiving *messages*, it invokes remote *methods* in its peer protocols. It is derived from `MessageProtocol` and makes use of the above mentioned methods.
Fig. 4.4 shows how the `RpcProtocol` class works conceptually.

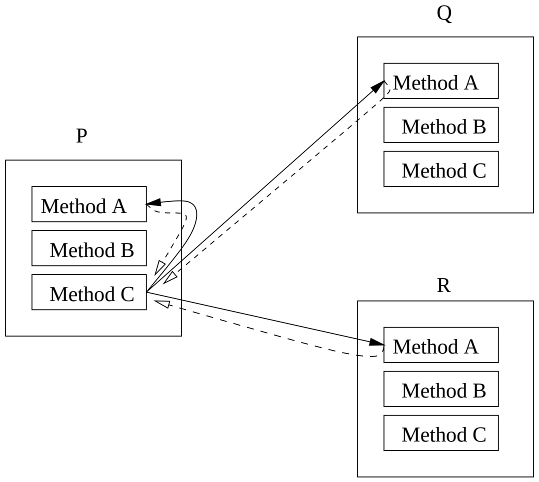

Figure 4.4: Remote method invocations between peer protocols

In the example, the group of peer protocols would consist of the 3 members P, Q and R. Each is a protocol layer in a different protocol stack (e.g. on a different machine). P now wants to call method `A` of all members in the group and collect the responses. It does so by invoking one of the `CallRemoteMethods` methods of `RpcProtocol` (discussed below). A `MethodCall` object (see 3.4.2) will be constructed, inserted in the buffer of a message and sent to all peers. Each peer receives the message, extracts the method call object, invokes it against itself and returns a result in the form of an object (or null, if the method has a void return value). The results will be received by the calling method as a response list.
Note that, since the sender is invoking methods in all members of the group, *its own* `A` *method will also be invoked*. Therefore special care has to be taken to avoid deadlocks (see 5.1).
A typical protocol based on `RpcProtocol` would define a number of methods and then use one of the `CallRemoteMethod(s)` methods to invoke them. The `CallRemoteMethod` series invokes a method in a single receiver, while the `CallRemoteMethods` does so in all group members. The former takes a destination address, the method name, a number of arguments (may be zero), a mode and a timeout. The last 2 parameters are the same as for `MessageProtocol`. If the number of parameters is too small, a `MethodCall` object may be created directly and given as parameter.
The implementation of `CallRemoteMethods` is as follows:

```
public RspList CallRemoteMethods(Vector dests, MethodCall call,
                                 int mode, long timeout) {
    byte[]   buf=null;
    Message  msg=null;

    try {
        buf=Util.ObjectToByteBuffer(method_call);
    }
    catch(Exception e) {
        System.err.println("RpcProtocol.CallRemoteMethods(): " + e);
```

```
            return null;
        }
        msg=new Message(null, null, buf);
        return CastMessage(dests, msg, mode, timeout);
    }
```

The method name and arguments of the other `CallRemoteMethods` methods are used to create a `MethodCall` object. It is then serialized into a byte buffer which is inserted into a message. `MessageProtocol.CastMessage` is then used to send the message to all destinations and receive the responses.
Method `Handle` of `MessageProtocol` is overridden in `RpcProtocol` to invoke a method in the current peer:

```
    public Object Handle(Message req) {
        Object      body=null, retval=null;
        MethodCall  method_call;

        if(req == null || req.GetBuffer() == null) {
            System.err.println("RpcProtocol.Handle(): msg or msg buffer is null !");
            return null;
        }

        try {
            body=Util.ObjectFromByteBuffer(req.GetBuffer());
        }
        catch(Exception e) {
            System.err.println("RpcProtocol.Handle(): " + e);
            return null;
        }

        if(body == null || !(body instanceof MethodCall)) {
            System.err.println("RpcProtocol.Handle(): msg not a MethodCall !");
            return null;
        }

        method_call=(MethodCall)body;
        retval=method_call.Invoke(this, method_lookup_clos);
        return retval;
    }
```

First, the message's byte buffer is extracted and converted into a `MethodCall` object. If this fails, `null` will be returned as result[7]. Then the method call is invoked against the current `RpcProtocol` instance (the one that received the method call) and the result returned.

[7] `RpcProtocol` assumes that all communication is done via method calls, therefore it will not process other objects. However, the implementation of `Handle` could be overridden to allow this.

Chapter 5

Advanced Concepts

5.1 Avoiding deadlocks using synchronous group method invocations

Synchronous group messages / method calls are a powerful tool: they raise the peer protocol abstraction level to that of normal *method invocations*. Protocols can therefore be described as a set of simple procedures (methods); these procedures can be invoked remotely, and procedure bodies contain the protocol logic. Not having to write complex code for low-level tasks such as request/response correlation makes the resulting code more clear and shorter. Looking at a protocol layer as a class with methods makes understanding it much easier. Thus, an interaction between two peer protocols boils down to a remote method invocation. However, as with all types of concurrent and distributed processing, there is the danger of deadlocks. Aggravating the problem is the requirement of ordering in group communication settings: a method invocation caused by the reception of a message may not be 'passed' by another method invocation. Therefore, unlike in modern object request brokers (ORBs), where every request is served on a separate thread (taken from a thread pool) which avoids deadlock in most cases, we cannot introduce concurrency to handle requests, since this would adversely affect ordering guaranteed by the protocol stack[1].
To illustrate the problem, consider the protocol interaction typically performed in a GMS layer when a new client joins a group, as shown in fig. 5.1.

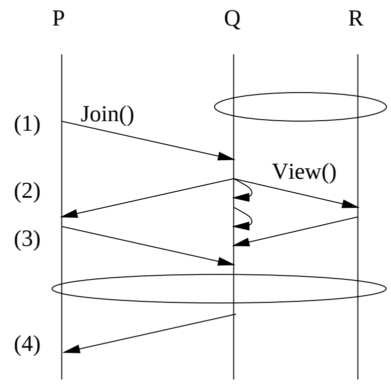

Figure 5.1: Client joining a group

[1] Note that in some cases ordering may not be strict (e.g. no FIFO ordering required). In such a case, concurrency could be introduced.

Method `Join` would be invoked by a client in the coordinator, who would join the new member and then invoke `View` on all members. In the example, P would be the new member, and Q the current coordinator (Q and R form the existing group). To join the group, P invokes a unicast synchronous `Join` in the coordinator. The method returns true if the member has been joined successfully, otherwise false, in which case the client would have to retry until it is finally successful. When a `Join` method is invoked in a coordinator (1), it adds the new member to its local view and invokes a synchronous `View` method in all members (2). When the `View` method invocation has returned (3) from all members, then the `Join` method returns (4). The 2 method invocations in the example have been chosen to be synchronous because return signals successful method invocation (a one-way method invocation does not return) and, in the case of `Join`, the return value (false or true) is needed to decide whether the join was successful.
However, choosing the 2 methods to be synchronous leads to disaster in our protocol stack, which delivers the messages (hence the method invocations) in FIFO order: when the `Join` method is invoked synchronously in the coordinator, and the coordinator multicasts a synchronous `View` method invocation to all members (including itself !), the coordinator is blocked waiting on the `View` method invocation *to itself* to return (which waits to be processed). This is shown in fig. 5.2.

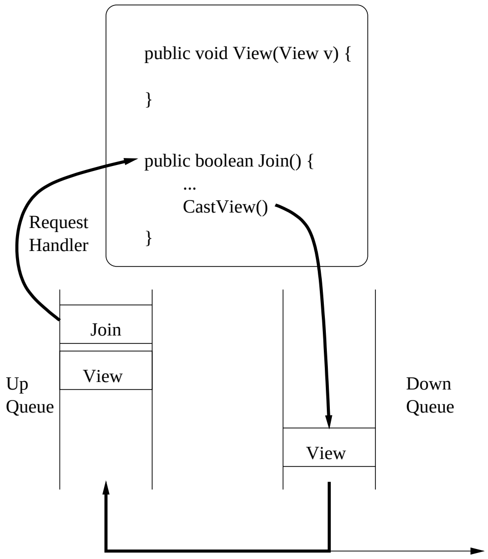

Figure 5.2: Request handler queue in a deadlock situation

As the GMS protocol is based on `RpcProtocol`, there are two queues: one for storing incoming methods and one for sending outgoing methods. When a request (method invocation) is received it is added to the up queue. A request handler thread continually retrieves methods from the up queue and invokes them synchronously, waiting for completion before handling the next method.
In the example, the queues at the coordinator are shown. A client sent a synchronous `Join` method which, as it is now at the top of the queue, is removed from the queue and processed. The `Join` method at some point invokes a synchronous `View` on all members. To do so, a message containing the `View` method call is put in the down queue to be sent down the stack. Since the view is sent to all members, it will also be received by the *sender*: it is added to the up queue, waiting for the request handler thread to remove and process it. However, the request handler is still busy processing `Join`, waiting for it to return (because it is synchronous). `Join` in turn is waiting for the `View` methods it sent to all members to return: only if the `View` methods were all executed and returned, `Join` would return. This

is a clear deadlock, caused by two synchronous method invocations. The thick lines in the example show the recursion causing the deadlock.
Note that this problem might not occur in multithreaded servers: since the `View` method would be executed on a different thread, `Join` would receive all responses and thus not block. However, as our protocol stack has to observe ordering, we can only allocate a single thread per request, ensuring FIFO order of request processing.
As can be seen, one has to be very careful when constructing method call chains containing synchronous method calls. A certain degree of recursion is always involved in distributed group communication systems, as requests sent to the group will always also be received by the sender (unless local delivery is turned off, cf. section 3.3.11).
As we can see, most deadlock problems occur when a message in the up queue blocks the request handler thread from processing other requests. The down queue is in most cases never a problem, as requests are just passed down the stack and the put on the network.
There are a number of workarounds to the above problems. First, careful design of protocols may reduce the number of synchronous method calls by simplifying the interactions between peer protocols. In the above example, the `View` method invocation may be made asynchronous, i.e. the `CastView` method would return immediately after sending out the `View` method invocations. In this case, no deadlock would occur. The purpose of having made `View` synchronous, namely to receive confirmation from each member that it received the view, can be implemented by having an acknowledgment/retransmission layer somewhere below the GMS layer: messages sent will always be received by all (non-faulty) members.
Another solution would be to have a separate thread invoke the `View` on the group: `CastView` would spawn a new thread (or reuse a thread from a thread pool) to handle the view installation and return after the thread has been created. This would cause `Join` to return without blocking, thus avoiding deadlock. However, introducing concurrency in this case leads to weaker guarantees about ordering between the view change notification and the return of the `Join` call: contrary to fig. 5.1, a joiner may return from the `Join` method before having received the view change. The client protocol would have to account for this behavior with additional code, adding more complexity.
JavaGroups contains a GMS protocol (`JavaGroups.JavaStack.Protocols.GMS`) that makes use of synchronous method invocations. However, a newer version is currently in work: `JavaGroups.JavaStack.Protocols.RpcGMS` is derived from `RpcProtocol` and uses (synchronous) method invocations to implement its functionality.
A third solution to the above problem is to enable the `RequestCorrelator` to detect deadlocks by setting a flag. This is described in the next section.

5.2 Using deadlock detection with the RequestCorrelator

The `RequestCorrelator` class allows to give higher preference to *circular* requests that would cause deadlock, thus breaking potential deadlock situations. Circular requests are chains of synchronous requests where a sender occurs more than once in the chain, e.g. a synchronous request sent from P to Q which, upon reception, sends a synchronous request to R, which in turn sends a synchronous request to P would form the call chain P $\rightarrow$ Q $\rightarrow$ R $\rightarrow$ P. When P receives the request, it cannot be processed because P *sent* the request and is waiting for Q's response, which is waiting for R's response, which in turn is waiting for P's response. The circularity causes a deadlock. This is shown in fig. 5.3[2].
Caused by the reception of another message (or an event) m1, P sends m2 to Q, which sends m3 to R, which sends m4 to P, causing a deadlock. However, we can see that, if message m1 was preempted by m4, that is if m1 was suspended temporarily and m4 was processed

[2]A similar (circular) deadlock situation is shown in fig. 5.2.

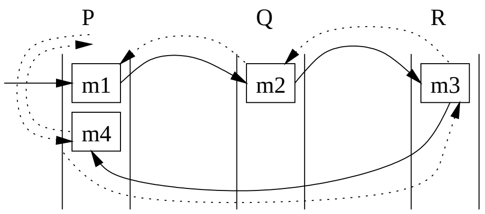

Figure 5.3: Circular deadlock situation

instead, m4 would return (shown by dotted lines), which would cause m3, m2 and m1 to return in turn, thus avoiding deadlock. There are two problems involved with this scheme: first, we have to detect when to 'prioritize' a request and second, we have to make sure that ordering guarantees given by the stack are not violated by this prioritizing scheme.

The first problem is solved by tagging synchronous requests with their *call chain* (also called *call stack*): when a synchronous request is sent, the sender's address is pushed onto a *call chain stack* in the header generated by the `RequestCorrelator`, and the header added to the message. In the example above, when m1 is received by Q, the call chain would be `P`[3]. Q would then add its own address to the call chain of m2, so that m2's call chain would be P $\rightarrow$ Q and so on. When P finally receives m4, m4's call chain would be P $\rightarrow$ Q $\rightarrow$ R $\rightarrow$ P. On every reception of a synchronous request, the receiver checks *whether its own address is already contained in the call chain.* If this is the case, the request is prioritized, that is, added to the head of the request queuee, otherwise it is just added to the tail. Thus, when P receives m4, m1 will be suspended, m4 processed and a response returned. Then the processing of m1 will resume. This scheme allows to detect direct or indirect circularity in synchronous call chains, and to avoid deadlock by selective prioritization of requests[4].

The second problem requires examination of ordering properties. Ordering properties, such as FIFO or total order, guarantee that, if m1 was sent before m4, then m1 should be delivered before m4. However, in our case, there are two solutions: the definition of 'delivery' which, in its strongest form, requires a request to *return* to complete delivery, could be weakened to one which excludes returning from the request. The choice of definition is up to the application, some applications need strong ordering, whereas for others the weaker definition is sufficient. A stronger cause can be made when considering call chains: a call chain maintains *causality* for the requests contained in it. Since P sent m1, which indirectly caused m4, m4 can without harm interrupt m1 at P and return before m1. This can be compared to a purely local execution: processing of m1 would start before processing of m4, but m4 would 'pass' m1 and return first, causing the stack to unwind. Note that causality maintained in call chains is not a *potential* one[5], but a *real* one: a call chain is only created when – during the processing of a synchronous request – another synchronous request is sent to a different member, therefore directly relating the previous request to the next one.

The `RequestCorrelator` can be created with or without deadlock detection (default is without): there is a constructor that allows to set deadlock detection, or method `SetDeadlockDetection` can be used alternatively. When deadlock detection is off, no call chains will be added to synchronous requests, therefore neither procesing cycles nor memory is wasted. When it is on, only *synchronous* requests will be tagged with call chains by the `RequestCorrelator`, and prioritization may be used for requests causing potential deadlocks.

[3]For simplicity excluding the sender of the request that caused P to send m1.

[4]Note that only *synchronous* requests are tagged with call chains, asynchronous requests will never cause the sorts of deadlocks described above, because the caller does not wait for their completion.

[5]This is different from *causal order* [Bir96, ch. 13.12.6] which records potential causality between message reception and sending, as it has no means to detect real causality.

Chapter 6

Implementing view changes

This chapter discusses in detail the interactions (events exchanged and actions taken) between protocol layers that happen when a new view needs to be installed in all members. A new view is created in the `RpcGMS` layer (1) when a new member joins the group, (2) a member leaves the group or (3) a faulty member has been detected (e.g. by a suspicion service). The latter case is the same as when a member leaves and the same actions will be taken.

According to virtual synchrony [Bir96], all messages sent in the current view have to be seen by all members before a new view can be installed. This is also called *flushing all messages out of the system* and is implemented by a flush protocol. It essentially works as follows: when a new view is to be installed, the coordinator sends a FLUSH message to the group. Each member sends its pending messages and then stops sending messages (until the new view has been received). It also returns all messages to the coordinator which are not known to be *stable*. (Stable messages are ones which are known to have been received at every member). If a member has crashed during the FLUSH protocol, a new round of the FLUSH protocol is started immediately. When the coordinator has received all responses it merges the unstable messages, removing duplicates, and re-sends them, making sure that every member receives them.[1] This is necessary to ensure atomicity, i.e. all members receive all messages. This could for example be violated if a member P wants to send a message m1 to all group members using multiple unicasts, but crashes after sending the message only to member Q. The FLUSH protocol ensures that Q returns m1, so m1 will be broadcast to all members (Q just discards the duplicate) in the current view. Finally the new view is broadcast to every member. Messages sent from now on are delivered in the new view. Messages received from the previous view are discarded.

There are basically 4 protocol layers that implement the above scheme: `RpcGMS`, `FLUSH`, `NAKACK` and `STABLE`. Each layer implements only a specific functionality and 'invokes' other functions by sending events to different layers. `RpcGMS` is the protocol that invokes the view change mechanism: it takes care of triggering the FLUSH protocol and broadcasting the new view. `FLUSH` implements the FLUSH protocol. `NAKACK` ensures that all messages within a view are ordered correctly (FIFO order): it does this using a NAK-based scheme, but can be switched to an ACK-based scheme when necessary (e.g. to reliably deliver view changes, whereas regular messages use NAKs). Finally, `STABLE` uses a gossipping mechanism to determine which messages have been received at all members, so that they can be garbage-collected. The following sections describe the 4 protocols, and the interactions taking place when a view is installed.

[1] This is for example possible by switching from a *NAK-based* to an *ACK-based* scheme.

6.1 Protocol layers that work together to implement view changes

6.1.1 RpcGMS

Whenever a new view is to be installed, the FLUSH protocol is run first and then the view is broadcast. The FLUSH protocol is started by sending down a `FLUSH` event and waiting for a `FLUSH_OK` event (or a timeout). Thus the FLUSH protocol has to be below the `RpcGMS` layer. (If a `FLUSH` layer was missing, `RpcGMS` would time out waiting for `FLUSH_OK` and proceed without running the FLUSH).
When a new member is to be joined, the FLUSH is sent to all current group members. The view, however, will be sent to all current members plus the new member. When a member is excluded from the group, the FLUSH is sent to all current members minus the one to be removed from the group. The view is also sent to all members minus the excluded one.

6.1.2 FLUSH

The task of the FLUSH layer is mainly to respond to a `FLUSH` event sent from above by coordinating the FLUSH protocol among all group members. This involves broadcasting a `FLUSH` message to all group members and collecting the responses, possibly resending the FLUSH to a smaller group if some member(s) crashed before receiving all responses. In handling the FLUSH protocol each `FLUSH` layer needs to return unstable messages. To determine unstable messages, it needs the `STABLE` layer somewhere underneath it. In case this layer is missing, the set of unstable messages returned will be empty.

6.1.3 NAKACK

The task of this layer is lossless delivery: all messages sent are tagged with (1) the view in which they were sent, (2) the sender's address and (3) a monotonically increasing sequence number. There are 3 modes the NAKACK layer can operate in: NAK, ACK and OUT_OF_BAND. By sending down a special event (`SWITCH_XXX`, the mode can be changed. However, if the mode is not NAK, then it will be switched back to NAK after sending 1 message with the different mode. So if 5 messages need to be sent using ACKs, then each message has to be preceeded with a `SWITCH_ACK` event.
The default is using NAKs: the sender tags a message with a sequence number and the receiver only requests retransmission if there is a gap between the expected and actually received message sequence number.
In the ACK scheme the sender tags a message with a sequence number and the receiver acknowledges every message by sending an ACK back to the sender. When no ACK has been received fro a certain time, the sender retransmits the message. Note that the NAK and ACK schemes use the same sequence numbers. Both modes reset their seqnos to 0 when receiving a view change.
The third mode is for out-of-band control messages (activated by SWITCH_OUT_OF_BAND): this mode does neither employ view IDs, nor does it use the same seqnos as NAK and ACK. It uses its own seqnos, unrelated to the ones used by NAK and ACK, and never resets them. In combination with the sender's address this makes every out-of-band message unique. Out-of-band messages are used for example for broadcasting `FLUSH` messages.
Since messages are tagged with the view in which they were sent (NAK and ACK mode), messages that have a a smaller view (previous view) than the one currently installed are discarded. Messages with a greater view are stored for future delivery, and message with the same views are delivered.

Modes are switched multiple times during a view change: regular messages are sent using NAKs (default mode). When there is a view change, the FLUSH message is sent using the OUT_OF_BAND mode. Then all unstable messages are re-sent using ACKs. This means we have to wait until we are sure that all unstable messages have been received reliably by every member and have thus become stable. Finally the last message sent in the old view is the view change, which is also sent using ACKs. When receiving the view change all `NAKACK` layers reset their sequence number to 0. The first regular message broadcast to the group will thus be tagged with sequence number 0.

6.1.4 STABLE

An ACK-based scheme can always determine when all members have received a certain message: it is safe to deleted a message when acks from all members have been received. This contrasts with a NAK-based scheme (the default used in the `NAKACK` layer). Here, the `NAKACK` layer cannot determine when a message has been received by all members due to a lack of acks. The problem is that messages accumulate until the new view change (when they can safely be deleted). Unfortunately, we cannot know *when exactly* view changes occur, as this is application dependent. As there may be many messages accumulating between view changes, we have to garbage collect them in order to reduce memory consumption.
The `STABLE` layer gossips with other `STABLE` layers in different stacks (of the same group) to determine *stable* messages, .i.e. message that are know to have been received by all members. It uses the scheme presented in [GHvR$^+$97] to do so. It periodically (or after n messages) sends out a gossip message to a subset of the group, including its stable messages: this is called a *gossip round*. Each gossip round maintains a vector of members that have already seen this specific gossip message. When all members have seen a round, a STABLE event is generated and sent up the stack. Other layers (such as the `NAKACK` layer) may use this event to remove stable messages from their caches.

6.2 Interaction between protocol layers implementing view changes

The view change interaction between the four layers described above is shown in fig. 6.1.

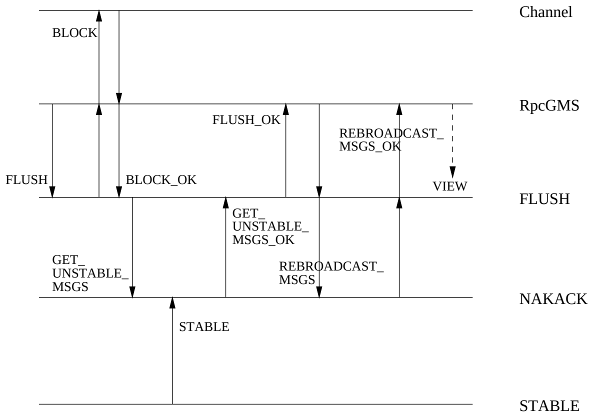

Figure 6.1: Protocol interaction for a view change

The `RpcGMS` layer triggers the view change protocol when it (a) receives a join, leave or suspect message and it (b) is the current coordinator (a designated member taking care of view changes, usually the oldest member). It first starts the FLUSH protocol by sending down a FLUSH event. If no FLUSH layer is present, then RpcGMS will time out and continue without running the FLUSH protocol (it will just broadcast the new view).

When the FLUSH layer receives the FLUSH event, it broadcasts a FLUSH message to all members (including itself). This causes each member to send pending messages and then block sending further messages (until a new view has been received). To do so, each member sends a BLOCK event up the stack. At the top level, the channel notifies the application that it is not supposed to send any more messages until it receives a view change. Messages sent nevertheless will simply be queued until the new view has been installed, and then delivered in the next view. Subsequently, a BLOCK_OK event is generated in response and sent down the stack. When the FLUSH layer receives this event, it is now ready to proceed: it has to collect all unstable messages and return them to the coordinator (which broadcast the FLUSH message to it). To collect unstable messages, it asks the NAKACK layer underneath it by sending down event GET_UNSTABLE_MSGS. The NAKACK layer periodically receives a STABLE event from underneath it (from the STABLE layer) and removes all stable messages from its 'sent messages' store. It therefore essentially returns all messages in 'sent messages'[2]. These unstable messages are returned in a GET_UNSTABLE_MSGS_OK event. When the FLUSH layer receives this event it returns the unstable messages to the coordinator. The coordinator now merges all unstable messages returned from all members (removes duplicates) and re-sends them by delegating the resending to the NAKACK layer underneath by sending down a REBROADCAST_MSGS event.

When the NAKACK layer receives the REBROADCAST_MSGS event, it does the following: for each message in the list, the message is wrapped into a new message and broadcast to the group. When such a message is received, it is unwrapped and sent up the stack. Typically most resent messages will just be discarded because they have already been seen, but in a case where for example the sender crashed just after sending the message to only one member, all other members will receive this message, resulting in uniform delivery. When all messages have been resent, the NAKACK layer returns a REBROADCAST_MSGS_OK event which is caught by the RpcGMS layer.

Only when all unstable messages have been resent, can the RpcGMS layer proceed to install the new view in all members. It does so by broadcasting a VIEW_CHANGE message using an ACK scheme (to ensure all members have received the view). When the view is received, a member is again allowed to send messages to the group. Messages that may have been sent during the block and were therefore queued will now be delivered in the new view.

[2] Note that if no STABLE layer is present underneath, *all* messages in 'sent messages' will be returned ! However, the 'sent messages' store is cleared whenever a new view is installed.

Chapter 7

Appendix

7.1 Events

7.1.1 BECOME_SERVER

Direction	Up/down
Originator	GMS layer
Argument	–
Description	Sent when client has joined a group, from now on client is a server (some protocol layers only start work when they become a server)

7.1.2 CONNECT

Direction	Down
Originator	JChannel
Argument	Group name
Description	Sent when a client connects to the channel, effectively joining a group. Caught by the GMS layer (but passed on to notify other layers), which collaborates with other GMS layers to join the client. GMS passes up `CONNECT_OK` (and `BECOME_SERVER` up/down) when done.

7.1.3 CONNECT_OK

Direction	Up
Originator	GMS
Argument	–
Description	Sent up the stack by the GMS when a group has successfully been joined.

7.1.4 DISCONNECT

Direction	Down
Originator	JChannel
Argument	Address of member to disconnect from the channel
Description	Sent when a client wants to leave the group, processed by the GMS layer which will collaborate with the other GMS peer layers to remove the member from the group. GMS passes down the event and then a `DISCONNECT_OK` up the stack to signal it is done.

7.1.5 DISCONNECT_OK

Direction	Up
Originator	GMS
Argument	–
Description	Sent up the stack by the GMS when done processing the `DISCONNECT` event.

7.1.6 FIND_INITIAL_MBRS

Direction	Down
Originator	GMS
Argument	–
Description	Sent down by the GMS to retrieve the initial membership when joining as a new member. Handled e.g. by the PING layer. A `FIND_INITIAL_MBRS_OK` event is sent back up the stack when done. It contains a vector of `PingRsps`.

7.1.7 FIND_INITIAL_MBRS_OK

Direction	Up
Originator	PING
Argument	Vector of `PingRsps`
Description	Sent by the PING layer as a response to the `PING_INITIAL_MBRS` event.

7.1.8 GET_APPL_STATE

Direction	Up
Originator	STATE_TRANSFER
Argument	Address of requester
Description	Sent up by the STATE_TRANSFER layer when the application's state should be retrieved. JChannel will return a `GetStateEvent` when `Receive` is called (if the channel was enabled to receive state transfer events). The response will be a GET_APLLSTATE_OK event sent down the stack.

7.1.9 GET_APPL_STATE_OK

Direction	Down
Originator	JChannel
Argument	Object (state)
Description	Response to GET_APPLSTATE. if the channel is not enabled for state transfer event reception, this event is generated immediately by JChannel and sent down the stack in response (with a null argument). Otherwise, this event is added to the channel's event queue. When `Receive` is called, a `GetStateEvent` is returned which has to be answered by calling `JChannel.ReturnState()`. This method then generates a GET_APPLSTATE_OK event.

7.1.10 GET_STATE

Direction	Down
Originator	JChannel
Argument	Integer (0 or 1)
Description	Send by the JChannel when method `GetState` was called. If the argument is 0, this means get the state from 1 existing member of the group. If it is 1, get the states of all members (as a Vector). The `STATE_TRANSFER` layer will catch this message and eventually return a `GET_STATE_OK` event with the state(s).

7.1.11 GET_STATE_OK

Direction	Up
Originator	STATE_TRANSFER
Argument	Object (single state) or Vector (all states)
Description	Response to a GET_STATE event. Will be caught by JChannel. The argument may be null, if no state could be retrieved.

7.1.12 MSG

Direction	Up/down
Originator	JChannel / UDP
Argument	Message
Description	Generated by JChannel when message is to be sent (down event), or by UDP layer (or generally the bottommost layer) when message was received (up event).

7.1.13 SET_LOCAL_ADDRESS

Direction	Up
Originator	UDP (or other bottommost layer)
Argument	Local address (address of channel)
Description	Sent from the bottommost channel when local address is determined. In the case of UDP, this occurs just after a `START` event has been received. Any layer that needs the local address may store it now. It is guaranteed not to be modified until a new `STOP/START` sequence has been received.

7.1.14 SUSPECT

Direction	Up
Originator	FD (or any failure detection layer)
Argument	Address of suspected member
Description	Sent up the stack when a member is suspected to have failed. Caught e.g. by the GMS layer, which may remove the faulty member and install a new view.

7.1.15 START

Direction	Up
Originator	ProtocolStack
Argument	–
Description	Generated by ProtocolStack object (attached to JChannel) when its `Start` method is called (this happens when `JChannel.Connect` is called. A `START` event is 'injected' at the bottommost layer and travels up the stack. Every layer may start doing its work.

7.1.16 START_OK

Direction	Up
Originator	UDP
Argument	–
Description	Sent by bottommost layer (UDP) as result to `START` event: first a `SET_LOCAL_ADDRESS` is sent up, then a `START_OK`. The ProtocolStack object waits until it receives this event (or times out).

7.1.17 START_QUEUEING

Direction	Up/down
Originator	(E.g) STATE_TRANSFER
Argument	–
Description	When received by the QUEUEING layer from below, it will start queueing all up events until it receives a `STOP_QUEUEING` event. When received from below, all down events will be queued.

7.1.18 STATE_RECEIVED

Direction	Up
Originator	JChannel
Argument	Object (state) or Vector (states)
Description	When JChannel receives a GET_STATE_OK event, it generates a STATE_RECEIVED event and places it on the channel's event queue. When `Receive` will be called, a corresponding `SetStateEvent` will be returned when this event is encountered.

7.1.19 STOP

Direction	Down
Originator	ProtocolStack
Argument	–
Description	Sent by the ProtocolStack object when its `Stop` method is invoked: this occurs when `JChannel.Disconnect` is called. Every layer has to stop doing its work. The bottommost protocol layer is supposed to send up a `STOP_OK` event in response. The ProtocolStack waits for this or times out.

7.1.20 STOP_OK

Direction	Up
Originator	Bottommost layer (UDP)
Argument	–
Description	Sent by the bottommost layer up the stack as a response to a `STOP` event.

7.1.21 STOP_QUEUEING

Direction	Up/down
Originator	(E.g) STATE_TRANSFER
Argument	Vector (event list)
Description	When received by the QUEUEING layer from below, it will a) pass up all events in the argument (event list) and then b) release all elements in the queue. From now on events will not be queued. Dito for events from above traveling down.

7.1.22 SWITCH_ACK

Direction	Down
Originator	(E.g) FLUSH
Argument	–
Description	Handled by NAKACK layer. Switches to ACK-based message transmission.

7.1.23 SWITCH_NAK

Direction	Down
Originator	(E.g) FLUSH
Argument	–
Description	Handled by NAKACK layer. Switches to NAK-based message transmission.

7.1.24 TMP_VIEW

Direction	Up or down
Originator	GMS
Argument	View
Description	Sent when a temporary view has to be installed. E.g. sent to the UDP layer when a new member is about to join (but hasn't been joined yet), so that when a message is received from this member, it should not be discarded. This event will almost always be followed by a `VIEW_CHANGE` event.

7.1.25 VIEW_CHANGE

Direction	Up/down
Originator	GMS
Argument	View
Description	Sent by the GMS layer when a new view has been received/is to be installed. Any layer may cache it.

7.2 JChannel Protocols

This section describes the protocols currently available (March 1999). For an accurate description refer to the online javadoc documentation. All protocol layers are located in directory `JavaGroups/JavaStack/Protocols`.

7.2.1 Templates

There are 3 templates for protocols derived from a) `Protocol`, b) `MessageProtocol` and c) `RpcProtocol`. They may be copied as initial skeleton for the type of protocol to be written. The templates are a) `EXAMPLE.java`, `MessageProtocolEXAMPLE.java` and `RpcProtocolEXAMPLE.java`. These protocols by default just pass any event they receive up/down the stack, so a new protocol can be created and integrated into a protocol stack in a matter of minutes. This protocol would not do anything useful yet, but it can be coded in steps (with testing followed every step) so that incremental changes are possible.

7.2.2 Delay

This layer can be used to delay incoming, outgoing or both types of messages. It is usually inserted directly on top of the bottommost layer (e.g. UDP). The delay for *incoming messages* (i.e. messages traveling up the stack is set using property `IN_delay`, for outgoing messages, `out_delay` is used. Both delays are the number of milliseconds a message should be delayed before passing it to the next layer. The message will be delayed a random value of milliseconds, where the range is between 0 and the delay specified in the property. As example, a delay of `100` for `out_delay` will delay all outgoing messages by a random number of milliseconds between 0 and 100.
This layer can be used to experiment with message delays, e.g. in testing other protocol layers.

7.2.3 FD

The failure detection layer (FD) periodically tries to reach its nearest neighbor to the right (see fig. 7.1.

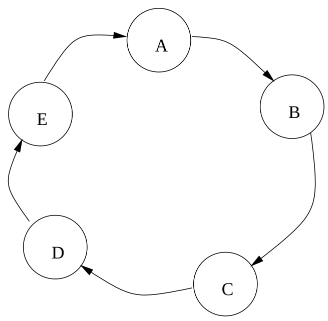

Figure 7.1: Failure detection layer

The nearest neighbor is always computed based on the local view. Since all views in all stacks have the same member ordering, every member can always determine its next neighbor to the right. When a new view is received, the neighbor is recomputed.
The FD layer periodically pings its neighbor. When no response has been received after `max_tries` (each with a timeout), a `SUSPECT` message is multicast to all members of the group. The GMS layer which is currently the *coordinator* processes the message, all others ignore it[1]. In the example, the coordinator would be `A`. It pings `B` which in turn pings `C`. The last member (`E`) 'wraps around' and pings the first (`A`).

[1]Except when the suspected member is the coordinator: in this case, the next member in rank takes over and becomes the new coordinator. Decisions based on member ordering are possible because ordering is always deterministic. This means that no distributed election algorithm needs to be run. The coordinator is commonly the *first* member.

7.2.4 FRAG

The FRAG layer was described in detail in section 4.2. It essentially breaks up bigger messages into smaller ones and reassembles them at the receiver's side.

7.2.5 GMS

The group membership service is probably the most important protocol layer, and also the most complex to implement. The description is based on the current (March 99) GMS layer, but work is underway to replace this layer with a new one (`RpcGMS`).

When a `CONNECT` event is received by the GMS layer, it tries to join the group. To do so, it first tries to retrieve the initial membership. If no other members can be found, it assumes it is the first member and sends a `VIEW_CHANGE` event up/down the stack. Otherwise, it determines the coordinator and sends a unicast message to the latter to join it to the group. The coordinator adds the new member to its local view and multicasts the new view to all GMS layers, which then in turn generate `VIEW_CHANGE` events up/down the stack.

When a member wants to leave a group, it disconnects from the channel. This causes a DISCONNECT event to be sent down where it is caught by the GMS layer. The latter sends a unicast LEAVE request to the coordinator, which in turn removes the member from its local view and multicasts the new view.

When a `SUSPECT` event is received by the GMS layer, if the layer is the current coordinator, the member will be removed from the local view and a new view multicast to all members. Otherwise, the event is just dropped. In case the suspected member is the *coordinator* itself, the next member of the group in order takes over and multicasts a new view (excluding the old coordinator).

7.2.6 MACK

The (multicast acknowledgment) MACK layer is an acknowledgment-based loss-less error correction layer. Whenever a message is sent, every receiver has to acknowledge it with an ACK message. When ACKS from all members have been received, the message was successfully sent to all members. When an ACK from a member is missing, MACK retransmit the message only to that member. When no ACK for a message has been received after n retries, that member is suspected: a `SUSPECT` event is passed up the stack. The MACK layer is best used on a noisy communication line, where packet drop is frequent. On a good communication substrate, layer MNAK should be used instead.

7.2.7 MNAK

The multicast negative acknowledgment layer (MNAK) is based on sequence numbers: all members of a group start sending with sequence number 0. Whenever a new member joins the group, all senders reset their sequence numbers to 0[2].

Every member maintains a table of senders from which it received (multicast) messages: a message from a member from which no message has been received so far triggers creation of a new entry in the table. The MNAK layer expects sequence numbers of messages from the same sender to increase monotonically: if a gap is detected, i.e. messages 4, 5, 7 and 8 were received from P, then a NAK for message 6 is sent to P. If the sender does not response after n tries, then the NAK is sent to *all* group members. If one of the group members still has the message, then it will send it on behalf of the original sender.

MNAK uses the `SlidingWindow` building block (see 3.1.6) to store messages. To purge messages that have been received by every MNAK layer, a STABLE layer would have to

[2]This is not yet implemented, therefore MNAK is not usable at the time ! It will be implemented when the `FLUSH` protocol is written.

be added to the protocol stack[3]. Currently, messages are only purged when a new view is installed, because that resets the senders' sequence numbers anyway.
Compared to the MACK layer, MNAK is more efficient, as no ACK messages have to be sent to acknowledge every single message, but NAK messages have to be sent only in the case when a message was not received.

7.2.8 PING

The PING layer is responsible for finding the initial membership of a group. It does so upon reception of a `FIND_INITIAL_MBRS` event (sent by the GMS layer). When done, a `FIN_INITIAL_MBRS_OK` event is sent up the stack, containing the members found as argument. GMS waits until it receives the initial membership, and – based on it – determines the current coordinator to which it then sends a join request. When not receiving the initial members within a certain time frame, a timeout is received, and the GMS creates a singleton group (with itself as only member).
Currently the initial membership is found either using a multicast to an IP multicast address to which all members respond, or using the gossip daemon (see 3.7.2), if IP multicast is not enabled.

7.2.9 PT2PT

The point-to-point protocol (PT2PT) is used for sending/receiving unicast messages to/from single members in a group. When a message is sent to a unicast destination (msg.dest == unicast dest instead of null (multicast dest)), if there is not yet a connection to that destination, a connection is established (similar to TCP's connection establishment) and the message sent. ACKs are used to retransmit lost messages, and all messages are delivered in FIFO order. Sliding windows are used at the sending and receiving side (with fixed window size, without window size adaptation).
The JChannel protocol layers often need to make a distinction between unicast and multicast messages; the two cases are often treated differently. For example, in some multicast layers, unicast messages would just be passed up/down the stack. Similarly, in PT2PT, multicast messages are just passed through.
PT2PT maintains a table of recently received messages for each sender. Whenever a message is sent to a member for which no table entry is available, a negotiation protocol similar to TCP's SYN is started, which exchanges the start sequence numbers that the communication partners are going to use. Error recovery through retransmission (NAKs) and FIFO ordering are provided by PT2PT.

7.2.10 QUEUE

Queues events selectively (either up or down). When a `START_QUEUEING` event is received from below, all subsequent events will be queued (in the up queue). The only events that will not be queued in this case is a `STOP_QUEUEING`: its argument is an event list that is passed up before the up queue is release. This means that all events in the up queue are passed up the queue, the up queue is emptied and queueing disabled.
The same scheme applies to down events. This layer is used by the `STATE_TRANSFER` layer.

7.2.11 STATE_TRANSFER

The `STATE_TRANSFER` layer is responsible to reply to `GET_STATE` events sent from the channel and return the state in a `GET_STATE_OK` event. It will interact with other group members to get the group state. The protocol interaction is shown in fig. 7.2.

[3]Not yet available as of March 99.

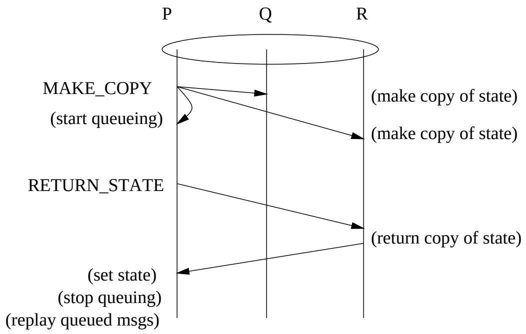

Figure 7.2: State transfer protocol interaction

A state transfer is initiated by multicasting a `MAKE_COPY` message to all members (including the initiating member). When the initiating member receives this message, it starts queueing subsequent messages. All other member, upon reception of the message, make a copy of the current state.
In the second phase, the initiator selects one member (possibly the oldest, or the coordinator) and sends a `RETURN_STATE` message to it. The receiver returns its copy of the state that was created upon reception of `MAKE_COPY`. The initiator sets its state and then replays all queued messages before accepting regular messages again. (Message will not be queued after replaying the queued messages).
The above scheme requires that messages are received by all members in the same order (total ordering). Thus, message `MAKE_COPY` is received by all members at the same logical time, allowing the initiator of a state transfer to queue all subsequent messages until reception of the state.
The events exchanged between peer `STATE_TRANSFER` layers and their JChannel instances are shown in fig. 7.3.

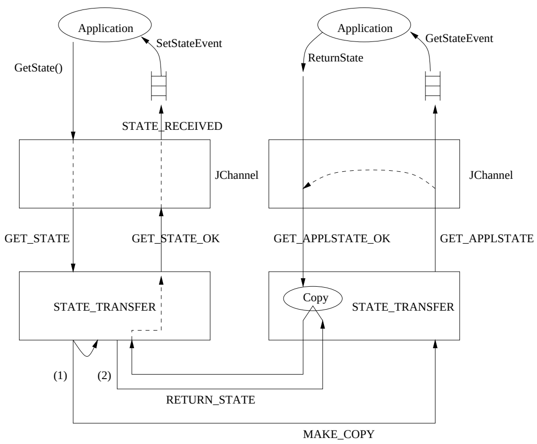

Figure 7.3: Events between `STATE_TRANSFER layers and JChannel`

The protocol on the left side represents the initiator of a state transfer, the one on the right the participant (e.g. an existing group member). The protocol is started when a `GET_STATE` event is received. It causes 2 messages to be sent to a) all members and b) one selected member.
The first message is `MAKE_COPY` (1) and it is sent to all members. Upon reception, the *initiator* reacts differently from the participants: it starts queuing all subsequent messages.

Participants try to retrieve the state from the application. This is done by sending up a GET_APPLSTATE event and waiting for a GET_APPLSTATE_OK event to be returned. If the channel does not support state transfer events (corresponding option is not set), then the GET_APPLSTATE_OK response will contain a null state. Otherwise it will be a copy of the application's current state[4]. The copy (if valid) is then stored in the STATE_TRANSFER layer for subsequent retrieval by the initiator.
The second message is a RETURN_STATE message (2) sent only to one selected member[5]. This is currently the oldest member (usually the coordinator). Note that the message is *never* sent to the initiator itself. The receiver just returns the previously saved copy of the state.
When the initiator receives the copy from the selected member, it does two things: first it sets its own state from the data received. Then, it blocks all message reception, replays the message queue (to which all messages received after MAKE_COPY were queued), disables queueing and unblocks message reception. From now on, messages received will not be queued, but passed up to the channel and on to the application where they will be applied to the application's current state. Queueing ensures that any message sent after MAKE_COPY and before receiving the state will not be lost, but properly applied to construct a valid state.
When the state was received, a GET_STATE_OK event is passed up the stack. When processed by JChannel, a STATE_RECEIVED event is created and inserted into JChanel's event queue. After this, a caller blocked on the `GetState` method will return (true or false). One of the next calls to `Receive` will return a `SetStateEvent` object containing the state, which can then be set by the application.
The implementation of STATE_TRANSFER relies on total ordering of messages multicast to all group members: it is based on the assumption that all members receive the MAKE_COPY message at the same logical time. If this was not the case, then the different participants might save different versions of the state. For example if participant Q received messages m1, MAKE_COPY and m2, and participant R received m1, m2 and MAKE_COPY then, the state saved by Q would include m1, but R's state would include m1 and m2.

7.2.12 TOTAL

The total order protocol works as follows: every member sends messages it wants to broadcast to a sequencer (usually the first member in the group). The sequencer broadcasts the message on behalf of the sender to all group members. Members order the messages FIFO, therefore establishing a total order between all messages. This protocol was implemented by Manish Sambhu (mms21@cornell.edu).

7.2.13 TUNNEL

The TUNNEL protocol can be used as the bottommost protocol in a stack, e.g. as replacement for UDP. Instead of sending and receiving datagrams, it connects to a router host (defined by the `router_host` and `router_port` options) using TCP. Messages are sent/received directly to/from the router, making TUNNEL essentially a forwarding protocol. See section 3.8 for details.

7.2.14 UDP

UDP is currently the bottommost layer available to use in a protocol stack.
When it receives a START event (upon channel connection), it creates a unicast and an IP multicast socket. The IP address plus the port number of the unicast socket form the channel's address: every message sent to either a single destination or the whole group will be

[4] Note that the data type of the state has to be serializable as it will be marshaled/unmarshaled to be sent over the network. If this is not the case, an exception will be thrown.

[5] If `Channel.GetStates()` was called, then this message will be sent to *all* members instead.

marked with this address (in the `source` field). As soon as the local address is know, a `SET_LOCAL_ADDRESS` event will be sent up the stack, followed by a `START_OK`.

When a `STOP` event is received, the sockets are closed and a `STOP_OK` event is passed up the stack as acknowledgment. When a channel is closed, local address becomes meaningless (since based on a open socket).

Messages sent down the stack will be added a UDP header containing the group name and then put on the network as datagram packets. Datagrams received from the network will be converted into messages and their UDP header removed. If the header's group name is different from the channel's group name, then the message will be dropped. Otherwise, it is passed up the stack.

UDP can either use IP multicasting or unicasting to disseminate messages to the group. If option `ip_mcast` is false, unicast is used: when a message is to be sent to all group members, it is sent n times to each member. To do this, UDP has to cache the membership when receiving `VIEW_CHANGE` events.

The IP multicast address and port can be configured using options `mcast_addr` and `mcast_port`. Although it is not a problem when different groups use the same IP multicast address and port (since messages by members belonging to a different group are discarded by UDP), it is often preferable to choose different parameters. However, all group members must have their UDP layers configured too the same IP multicast address and port (when using IP mcast).

Bibliography

[Bir96] Kenneth P. Birman. *Building Secure and Reliable Network Applications*. Manning Publications Co., 1996.

[CS 97] CS Dept Cornell University. *The Ensemble Distributed Communication System*, 1997. http://www.cs.cornell.edu/Info/Projects/Ensemble/index.html.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GHvR+97] Katherine Guo, Mark Hayden, Robbert van Renesse, Werner Vogels, and Kenneth P. Birman. GSGC: An Efficient Gossip-Style Garbage Collection Scheme for Scalable Reliable Multicast. Technical report, Department of Computer Science, Cornell University, 1997.

[ibu98] Softwired Inc. *iBus - The Java Multicast Object Bus*, 1998. http://www.softwired-inc.com/ibus.

[Tan89] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, second edition, 1989.