JavaGroups

JGroups

Overview

JavaGroups is toolkit for reliable group communication.

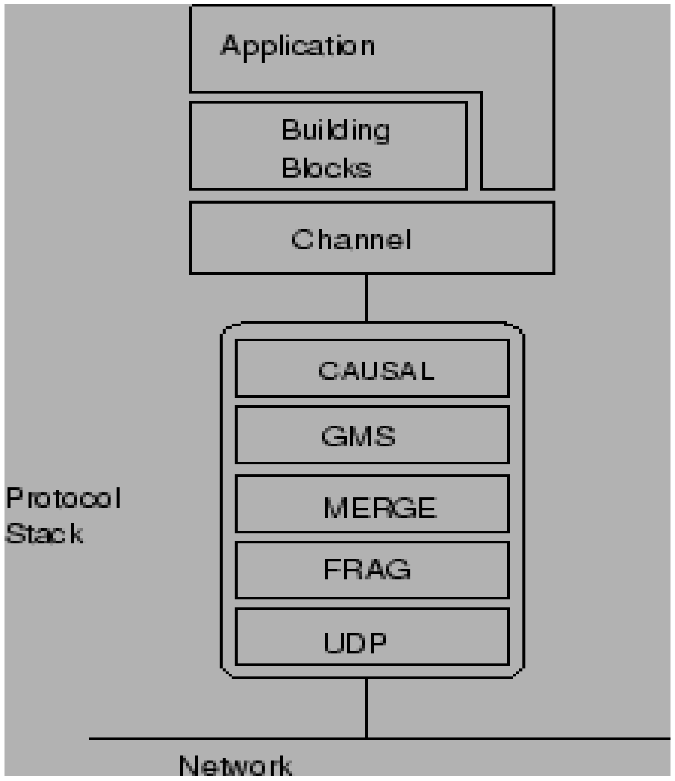
Processes can join a group, send messages to all members or single members and receive messages from members in the group.

The system keeps track of the members in every group, and notifies group members when a new member joins, or an existing member leaves or crashes.

A group is identified by its name. Groups do not have to be created explicitly; when a process joins a non-existing group, that group will be created automatically.

Member processes of a group can be located on the same host, within the same LAN, or across a WAN. A member can be part of multiple groups.

Architecture of JavaGroups



It consists of 3 parts:

(1) the Channel API used by application programmers to build reliable group communication applications,

(2) the building blocks, which are layered on top of the channel and provide a higher abstraction level and

(3) the protocol stack, which implements the properties specified for a given channel

Channel Activity

•A channel is connected to a protocol stack.

•Whenever the application sends a message, the channel passes it on to the protocol stack, which passes it to the topmost protocol.

•The protocol processes the message and the passes it on to the protocol below it.

•Thus the message is handed from protocol to protocol until the bottom protocol puts it on the network.

•The same happens in the reverse direction: the bottom (transport) protocol listens for messages on the network. When a message is received it will be handed up the protocol stack until it reaches the channel.

•The channel stores the message in a queue until the application consumes it.

Channel

To join a group and send messages, a process has to create a *channel* and connect to it using the group name (all channels with the same name form a group).

The channel is the handle to the group. While connected, a member may send and receive messages to/from all other group members.

The client leaves a group by disconnecting from the channel.
A channel can be reused: clients can connect to it again after having disconnected. However, a channel allows only 1 client to be connected at a time.

If multiple groups are to be joined, multiple channels can be created and connected to.

A client signals that it no longer wants to use a channel by closing it. After this operation, the channel cannot be used any longer.

Channel

Each channel has a unique address.

Channels always know who the other members are in the same group: a list of member addresses can be retrieved from any channel. This list is called a *view*.

A process can select an address from this list and send a unicast message to it (also to itself), or it may send a multicast message to all members of the current view.

Whenever a process joins or leaves a group, or when a crashed process has been detected, a new *view* is sent to all remaining group members.

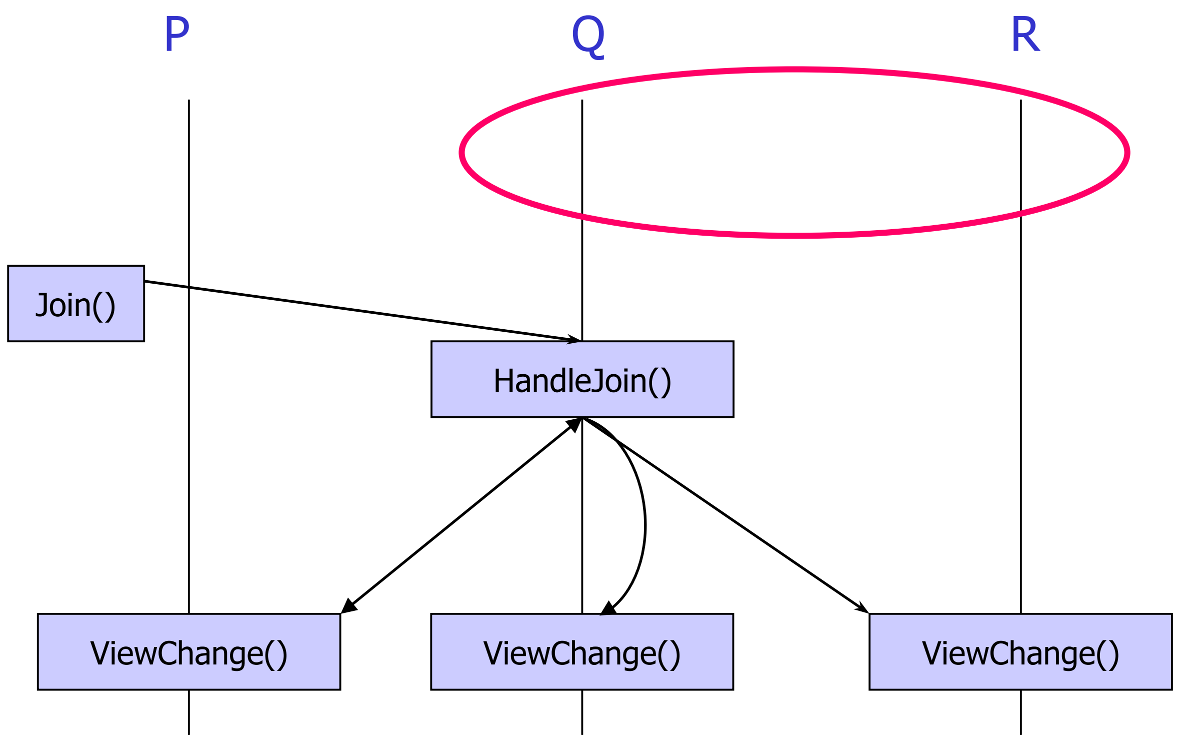
When a member process is suspected of having crashed, a *suspicion message* is received by all non-faulty members.

Thus, channels receive:

regular messages,
view messages and
suspicion messages.

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

Group Communication



Coordinator

Channel

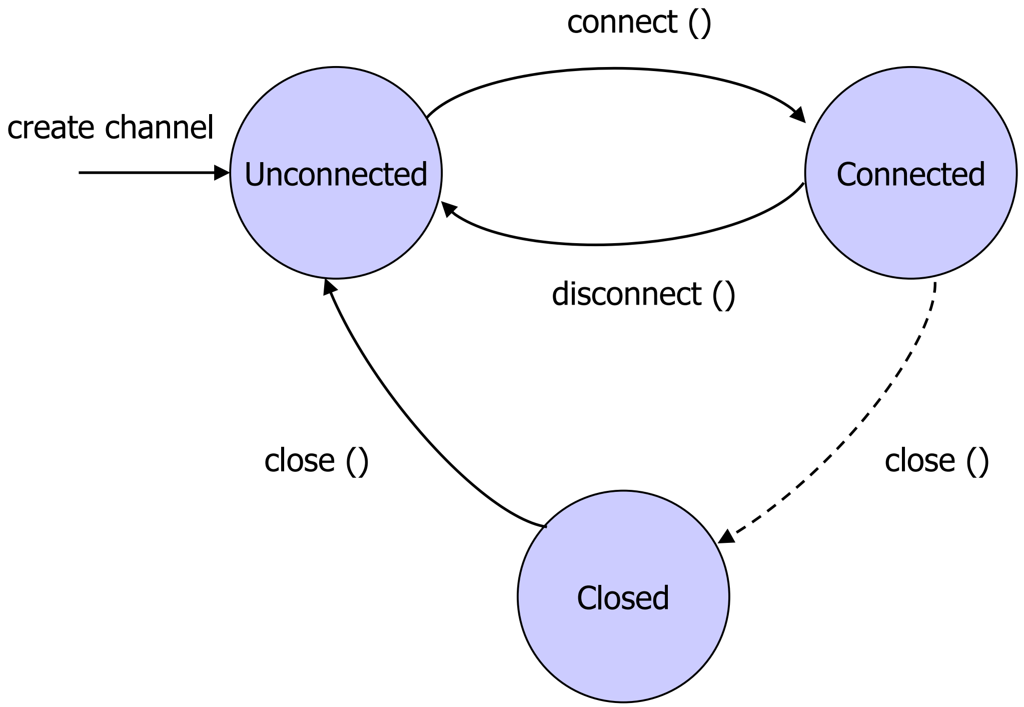
Channels provide asynchronous message sending/reception, somewhat similar to UDP. A message sent is essentially put on the network and the send() method will return immediately.

Conceptual *requests*, or *responses* to previous requests, are received in undefined order, and the application has to take care of matching responses with requests.

Also, an application has to actively *retrieve* messages from a channel (pull-style); it is not notified when a message has been received.

Note that pull-style message reception often needs another thread of execution, or some form of event-loop, in which a channel is periodically polled for messages.

Channel States



Creating Channel

The public constructor of JChannel looks as follows:

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

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

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

Connecting to a channel

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

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

The method returns as soon as the group has been joined successfully.

If the channel is in the closed state, an exception will be thrown. If there are no other members, i.e. no other client has connected to a group with this name, then a new group is created and the member joined.

The first member of a group becomes its *coordinator*. A coordinator is in charge of multicasting new views whenever the membership changes

Getting the local address and the group name

Method getLocalAddress() returns the local address of the channel.

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

Method getChannelName() returns the name of the group in which the channel is a member:

```
public String getChannelName();
```

Getting 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.

Sending a message

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

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

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

Example 1

Here's an example of sending a (multicast) message to all members of a group:

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

Example 2

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

```
 Address receiver;
 Message msg;
 Hashtable data;

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

Receiving a message

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

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

Example 3

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
```

Peeking at a message

Instead of removing the next available message from the channel, peek() just returns a reference to the next message, but does not remove it.

This is useful when one has to check the type of the next message, e.g. whether it is a regular message, or a view change. The signature of this method is not shown here, it is the same as for receive().

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.

Closing a channel

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

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

Example Summary

```
String props="UDP:PING:FD:STABLE:NAKACK:UNICAST:" +
    "FRAG:FLUSH:GMS:VIEW_ENFORCER:STATE_TRANSFER:QUEUE";
Message send_msg;
Object recv_msg;
Channel channel=new JChannel(props);

channel.connect("MyGroup");

send_msg=new Message(null, null, "Hello world");

channel.send(send_msg);

recv_msg=channel.receive(0);
System.out.println("Received " + recv_msg);

channel.disconnect();

channel.close();
```

Interfaces

Interface Transport looks as follows:

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

The MessageListener interface below provides a method to do so:

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

Interfaces

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

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

}
```

Interfaces

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

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

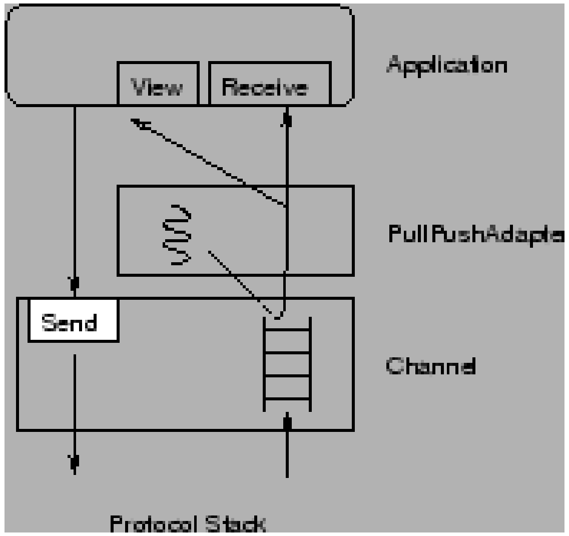
Building Blocks

Building blocks are layered on top of channels. Most of them do not even need a channel, all they need is a class that implements interface Transport (channels do).

- PullPushAdapter
- MessageDispatcher
- RpcDispatecher
- DistributedHashtable
- ReplicatedHashtable
- DistributedTree
- NotificationBus

PullPush Adapter

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



MessageDispatcher

Channels are simple patterns to *asynchronously* send a receive messages. However, a significant number of communication patterns in group communication require *synchronous communication*.

For example, a sender would like to send a message to the group and wait for all responses.

Or another application would like to send a message to the group and wait only until the majority of the receivers have sent a response, or until a timeout occurred.

MessageDispatcher

MessageDispatcher offers a combination of the above pattern with other patterns.

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

The handle() method is called any time a request is received.

The two methods to send requests are:

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

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

MessageDispatcher

The mode parameter defines whether the message will be sent synchronously or asynchronously.

GET_FIRST
Returns the first response received.
GET_ALL
Waits for all responses (minus the ones from suspected members)
GET_MAJORITY
Waits for a majority of all responses (relative to the group size)
GET_ABS_MAJORITY
Waits for the majority (absolute, computed once)
GET_N
Wait for n responses (may block if n > group size)
GET_NONE
Wait for no responses, return immediately (non-blocking). This make the call asynchronous.

MessageDispatcher

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

RpcDispatcher

This class is derived from MessageDispatcher. It allows a programmer to invoke remote methods in all (or single) group members and optionally wait for the return value(s).

```
public RspList callRemoteMethods(Vector dests, String method_name, int mode, long timeout);

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

public Object callRemoteMethod(Address dest, String method_name, int mode, long timeout);

public Object callRemoteMethod(Address dest, String method_name, Object arg1, int mode, long timeout);
```

DistributedHashtable

A DistributedHashtable is derived from java.util.Hashtable and allows to create several instances of hashtables in different processes.

All of these instances have exactly the same state at all times. When creating such an instance, a group name determines which group of hashtables will be joined.

The new instance will then query the state from existing members and update itself before starting to service requests. If there are no existing members, it will simply start with an empty state.

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

DistributedTree

Similar to DistributedHashtable this class also provides replication of a data structure across multiple processes.

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

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

DistributedTree

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

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

NotificationBus

This class provides notification sending and handling capability. Also, it allows an application programmer to maintain a local cache which is replicated by all instances.

Interface NotificationBus.Consumer:

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

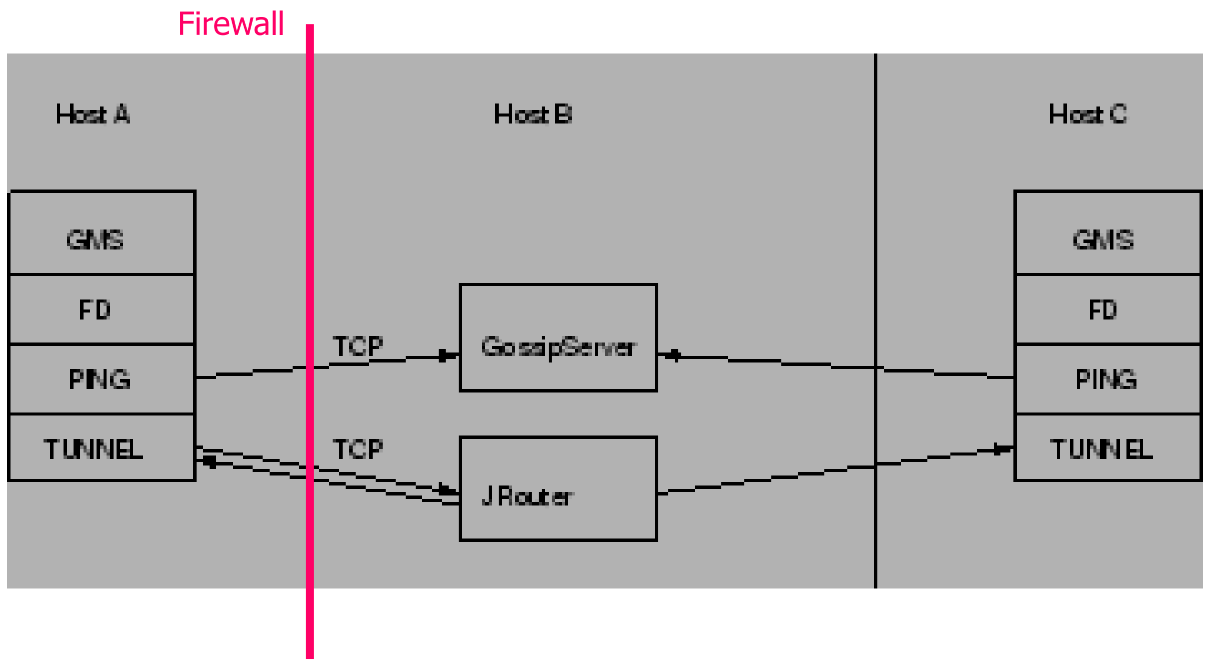
Notification consumers can subscribe to receive notifications by calling setConsumer() and implementing interface NotificationBus.Consumer.

NotificationBus

The most important methods of NotificationBus are:

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

Tunneling a firewall



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

   start org.javagroups.stack.GossipServer -port 12002

1. Start Router:

   java org.javagroups.stack.Router -port 12001

1. Configure the TUNNEL protocol layer.
2. Create a channel

Tunneling a firewall

First, the GossipServer and Router processes are created on host B. Note that these should be outside the firewall, and all channels in the same group should use the same GossipServer and Router processes.

When a channel on host A is created, its TCPGOSSIP protocol will register its address with the GossipServer and retrieve the initial membership (assume this is C). Now, a TCP connection with Router is established by A; this will persist until A crashes or voluntarily leaves the group.

When A multicasts a message to the group, Router 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.

The End