2. Writing a simple application

The goal of this chapter is to write a simple text-based chat application (SimpleChat), with the following features:

- All instances of SimpleChat find each other and form a cluster.
- There is no need to run a central chat server to which instances have to connect. Therefore, there is no single point of failure.
- A chat message is sent to all instances of the cluster.
- An instance gets a notification callback when another instance leaves (or crashes) and when other instances join.
- (Optional) We maintain a common cluster-wide shared state, e.g. the chat history. New instances acquire that history from existing instances.

2.1. JGroups overview

JGroups uses a JChannel as the main API to connect to a cluster, send and receive messages, and to register listeners that are called when things (such as member joins) happen.

What is sent around are Messages, which contain a byte buffer (the payload), plus the sender’s and receiver’s address. Addresses are subclasses of org.jgroups.Address, and usually contain an IP address plus a port.

The list of instances in a cluster is called a View, and every instance contains exactly the same View. The list of the addresses of all instances can get retrieved by calling View.getMembers().

Instances can only send or receive messages after they’ve joined a cluster.

When an instance wants to leave the cluster, methods JChannel.disconnect() or JChannel.close() can be called. The latter actually calls disconnect() if the channel is still connected before closing the channel.

2.2. Creating a channel and joining a cluster

To join a cluster, we’ll use a JChannel. An instance of JChannel is created with a configuration (e.g. an XML file) which defines the properties of the channel. To actually connect to the cluster, the connect(String clustername) method is used. All channel instances which call connect() with the same argument will join the same cluster. So, let’s

To join a cluster, we'll use a JChannel. An instance of JChannel is created with a configuration (e.g. an XML file) which defines the properties of the channel. To actually connect to the cluster, the connect(String clustername) method is used. All channel instances which call connect() with the same argument will join the same cluster. So, let's actually create a JChannel and connect to a cluster called "ChatCluster":

```
import org.jgroups.JChannel;

public class SimpleChat {
    JChannel channel;
    String user_name=System.getProperty("user.name", "n/a");

    private void start() throws Exception {
        channel=new JChannel(); // use the default config, udp.xml
        channel.connect("ChatCluster");
    }

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

First we create a channel using the empty constructor. This configures the channel with the default properties. Alternatively, we could pass an XML file to configure the channel, e.g. new JChannel("/home/bela/udp.xml").

The connect() method joins cluster "ChatCluster". Note that we don't need to explicitly create a cluster beforehand; connect() creates the cluster if it is the first instance. All instances which join the same cluster will be in the same cluster, for example if we have

- ch1 joining "cluster-one"
- ch2 joining "cluster-two"
- ch3 joining "cluster-two"
- ch4 joining "cluster-one"
- ch5 joining "cluster-three"

, then we will have 3 clusters: "cluster-one" with instances ch1 and ch4, "cluster-two" with ch2 and ch3, and "cluster-three" with only ch5.

2.3. The main event loop and sending chat messages

We now run an event loop, which reads input from stdin (*a message*) and sends it to all ch2 and ch3, and "cluster-three" with only ch5.

2.3. The main event loop and sending chat messages

We now run an event loop, which reads input from stdin (*a message*) and sends it to all instances currently in the cluster. When "exit" or "quit" are entered, we fall out of the loop and close the channel.

```
private void start() throws Exception {
    channel=new JChannel();
    channel.connect("ChatCluster");
    eventLoop();
    channel.close();
}

private void eventLoop() {
    BufferedReader in=new BufferedReader(new InputStreamReader(System.in));
    while(true) {
        try {
            System.out.print("> "); System.out.flush();
            String line=in.readLine().toLowerCase();
            if(line.startsWith("quit") || line.startsWith("exit"))
                break;
            line="[" + user_name + "] " + line;
            Message msg=new Message(null, null, line);
            channel.send(msg);
        }
        catch(Exception e) {
        }
    }
}
```

We added the call to eventLoop() and the closing of the channel to the start() method, and we provided an implementation of eventLoop.

The event loop blocks until a new line is ready (from standard input), then sends a message to the cluster. This is done by creating a new Message and calling Channel.send() with it as argument.

The first argument of the Message constructor is the destination address. A null destination address sends the message to everyone in the cluster (a non-null address of an instance would send the message to only one instance).

The second argument is our own address. This is null as well, as the stack will insert the correct address anyway.

instance would send the message to only one instance).

The second argument is our own address. This is null as well, as the stack will insert the correct address anyway.

The third argument is the line that we read from stdin, this uses Java serialization to create a byte[] buffer and set the message’s payload. Note that we could also serialize the object ourselves (which is actually the recommended way !) and use the Message contructor which takes a byte[] buffer as third argument.

The application is now fully functional, except that we don’t yet receive messages or view notifications. This is done in the next section below.

2.4. Receiving messages and view change notifications

Let’s now register as a Receiver to receive message and view changes. To this end, we could implement org.jgroups.Receiver, however, I chose to extend ReceiverAdapter which has default implementations, and only override callbacks (receive() and viewChange()) we’re interested in. We now need to extend ReceiverAdapter:

```
public class SimpleChat extends ReceiverAdapter {
```

, set the receiver in `start()`:

```
private void start() throws Exception {
    channel=new JChannel();
    channel.setReceiver(this);
    channel.connect("ChatCluster");
    eventLoop();
    channel.close();
}
```

, and implement `receive()` and `viewAccepted()`:

```
public void viewAccepted(View new_view) {
    System.out.println("** view: " + new_view);
}

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

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

The viewAccepted() callback is called whenever a new instance joins the cluster, or an existing instance leaves (crashes included). Its toString() method prints out the view ID (an increasing ID) and a list of the current instances in the cluster

In receive(), we get a Message as argument. We simply get its buffer as an object (again using Java serialization) and print it to stdout. We also print the sender's address (Message.getSrc()).

Note that we could also get the byte[] buffer (the payload) by calling Message.getBuffer() and then de-serializing it ourselves, e.g. String line=new String(msg.getBuffer()).

2.5. Trying out the SimpleChat application

Now that the demo chat application is fully functional, let's try it out. Start an instance of SimpleChat:

```
[linux]/home/bela$ java SimpleChat

-------------------------------------------------------------------
GMS: address=linux-48776, cluster=ChatCluster, physical address=192.168.1.5:42442
-------------------------------------------------------------------
** view: [linux-48776|0] [linux-48776]
>
```

The name of this instance is linux-48776 and the physical address is 192.168.1.5:42442 (IP address:port). A name is generated by JGroups (using the hostname and a random short) if the user doesn't set it. The name stays with an instance for its lifetime, and maps to an underlying UUID. The UUID then maps to a physical address.

We started the first instance, let's start the second instance:

```
[linux]/home/bela$ java SimpleChat

-------------------------------------------------------------------
GMS: address=linux-37238, cluster=ChatCluster, physical address=192.168.1.5:40710
-------------------------------------------------------------------
** view: [linux-48776|1] [linux-48776, linux-37238]
>
```

```
-------------------------------------------------------------------
GMS: address=linux-37238, cluster=ChatCluster, physical address=192.168.1.5:40710
-------------------------------------------------------------------
** view: [linux-48776|1] [linux-48776, linux-37238]
>
```

The cluster list is now [linux-48776, linux-37238], showing the first and second instance that joined the cluster. Note that the first instance (linux-48776) also received the same view, so both instances have the exact same view with the same ordering of its instances in the list. The instances are listed in order of joining the cluster, with the oldest instance as first element.

Sending messages is now as simple as typing a message after the prompt and pressing return. The message will be sent to the cluster and therefore it will be received by both instances, including the sender.

When "exit" or "quit" is entered, then the instance will leave the cluster. This means, a new view will be installed immediately.

To simulate a crash, simply kill an instance (e.g. via CTRL-C, or from the process manager). The other surviving instance will receive a new view, with only 1 instance (itself) and excluding the crashed instance.