Solution 5
Multicast Chat

1 Overview

We give here the complete solution for Exercise 5.

2 Getting, Compiling and Running the Application

2.1 Compilation

The complete source code is available for download at the exercise page of the course web site. The archive contains an ant (*http://ant.apache.org*) build file that can be used to compile the sources and generate the Java documentation. To do this, simply type in the *Ex5* directory[1]:

```
ant
```

2.2 Starting the Client

To start the client, type the following command in a console in the *Ex5* directory:

java -classpath ./build/classes/ ch.epfl.lpd.ids.client.ChatClient ⟨multicastAddress⟩ ⟨port⟩

3 Architecture

Figure 1 summarizes the overall architecture of the multicast chat application.

4 Code Layout

There is no server implementation, the entire chat application is hosted by the chat clients. What follows is a quick tour of the packages within the application (the fully qualified package name of course starts with `ch.epfl.lpd.ids`):

`client` Contains the `ChatClient` application, along with the interfaces `IChatClient` and `IChatClientProperties`.

`exceptions` Contains the exception classes `MessageTooBigException` and `MulticastInitializationException`.

`fd` Containes the specification and the implementation of the failure detection mechanism.

`gui` Contains the classes making the graphical interface.

`membership` Contains the classes for disseminating heartbeat (i.e., "I am alive") messages.

`network` Contains the classes implementing the low-level diffusion mechanism. The Multicast-Socket is encapsulated here.

`rmulticast` These classes implement the reliable multicast abstraction on top of the multicast layer.

[1]Please use the latest version of ant.

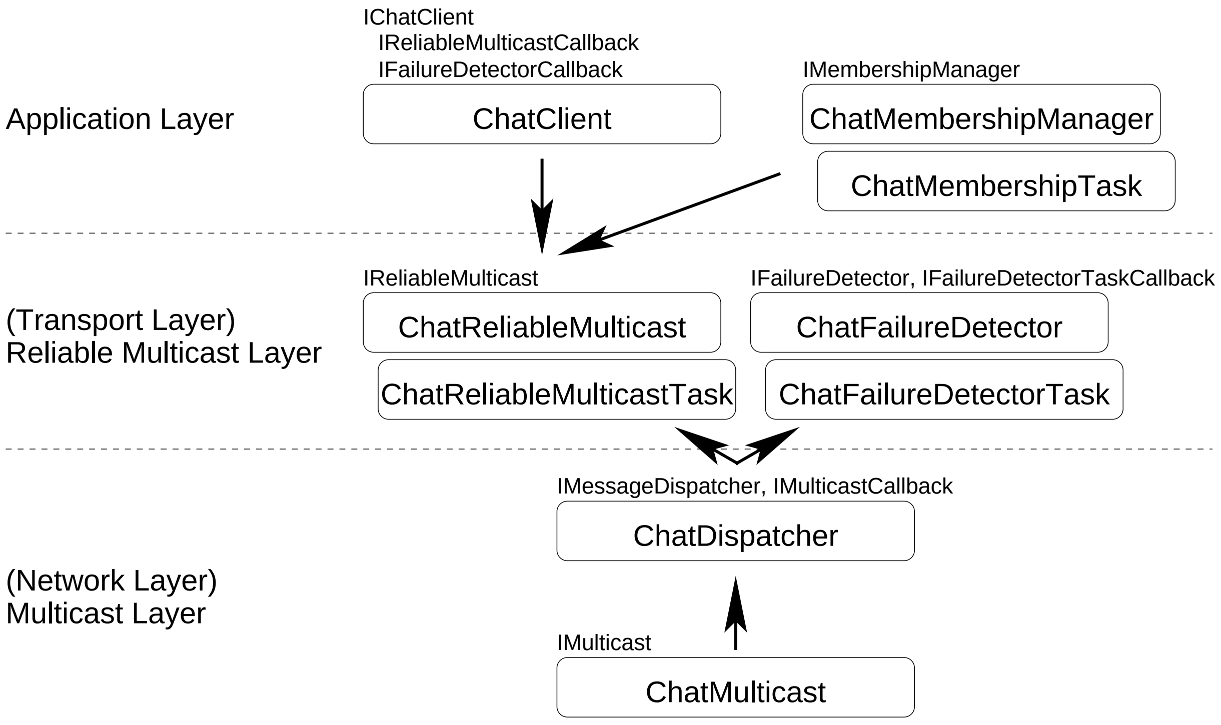

Figure 1: Overall architecture of the multicast chat application

`serialization` The specificaiton and the implementation of the message are described in this package.

`utils` Constants and utility methods are gathered altogether in this package.

We provide a top-to-bottom approach of the mechanismes implementation by the application in the following sections. We start with the `ChatClient`.

The application heavily uses the callback mechanism, for which handler component register so as to receive events (messsages, failure detection, and so on).

4.1 ChatClient Class

The role of the client is first to `initialize` the communication layer, and then start each component up when the user `connects`, before accepting messages from the user. A message is sent through a call to `sendMessage` from the graphical interface.

The `initialization` method creates, in order: a `ChatMulticast` instance and initializes with the multicast address and port number specified on the command line, a `ChatDispatcher`, a `ChatMembershipManager`, a `ChatFailureDetector` and a `ChatReliableMulticast` instance.

The `ChatClient` instance serves of message dispatcher for

- the reliable multicast layer and is registered as the `Reliable MulticastCallback` handler. The `ChatClient` implements the `IReliableMulticastCallback` interface, which declares the single method `rdeliver(IMessage)`. This method is called upon reception of a message.
- the failure detector layer, and is registered as the `FailureDetectorCallback` handler. The `ChatClient` implements the `IFailureDetectorCallback` interface, which declares the single method `newUsers`. This method is called by the failure detector module when a user connects or disconnects to or from the chatroom, so that the client can refresh its display.

In this idea, the `ChatClient` registers as the listener for the failure detector mechanism, and for the reliable multicast mechanism.

The sending of a message is done, inside the `sendMessage` method, by invoking the method `rsend` of the `ChatReliableMulticast` instance.

4.2 ChatReliableMulticast class

The class implements the reliable multicast abstraction. Messages that are sent but not yet acknowledged are internally buffered as `InternalMessage`'s.

Upon receiving a message, the message is delivered to the register callback, unless the message is an acknowledgement. In this case, the sender of the acknowledgement is removed from the list of clients which must yet acknowledge the message. Once the message has been acknowledged by all clients, it is not sent anymore.

The class `ChatReliableMulticastTask` is in charge of retransmitting the messages yet to be acknowledged by some client. Each message is associated a counter of the number of clients which have not acknowledged that specific message yet. The message is retransmitted as long as the counter does not reach zero.

The sending of messages is performed through the `ChatMulticast` class.

4.3 ChatMulticast class

The `ChatMulticast` class encpasulates the multicast socket, that is opened by specifiying the multicast address and port number given by the user on the command line.

A `MessageToBigException` is thrown by the `send` method, if the message size is bigger than the maximum message size allowed.

4.4 ChatDispatcher class

This class implements a general message handler and message dispatcher, through which all the messages received converged and are then dispatched to all registered handlers.

4.5 ChatFailureDetector class

The failure detector abstraction monitors the message received from the clients and decides that a client has crashed when no message is received from this client after a given timeout period.

The `handle` method receives hearbeat message, and updates the timestamp at which the last message from every known client was received.

The `ChatFailureDetectorTask` is reponsible for detecting the crash of clients from which no message has been received with the last interval. In this case, it invokes the method `changed` within the failure detector module, which in turns calls the registered call back method, in our case the `ChatClient` application itself.

4.6 ChatMembershipManager class

The membership service sends heartbeat messages to every other client within a regular interval. The `ChatMembershipTask` serves for implementing the timer, and executes the actual sending of the heartbeat message.

5 Questions

1. *Is it possible, according to the specification, that two correct chat clients receive two distinct messages in a different order? Why?*

   It is indeed possible that two correct chat clients receive two distinct messages in a different order, i.e., two chat clients c_1 and c_2 receive two distinct messages m_1 and m_2 in the order respectively $m_1 \rightarrow m_2$ and $m_2 \rightarrow m_1$. This is possible since the IP multicast protocol used within the application does not ensure any form of total ordered delivery of IP datagrams. The reliable multicast layer added by the chat application on top of the IP multicast layer does not implement a total order primitive, but only a reliable delivery primitive. For instance, consider a scenario where a chat client that sends two messages m_1 and m_2, in this order. Because of the architecture of the IP protocol, m_1 and m_2 do not necessarily follow the same path in terms of hops in the network. As a result, two correct clients c_1 and c_2 might very well receive m_1 and m_2 in a different order, m_1 being delayed for c_2 but not for c_1.

2. *How can we deal with message size greater than 64kB?*
   The IP layer does not support datagram of more than 64kB, since the field for encoding the size of the datagram is 16bits long (and since it would be much impracticable to have packets of more than 64kB on the network! In practice, IP fragments rarely exceed 576 bytes!). For supporting messages of size greater than 64kB, we would need to implement a algorithm for fragmenting the messages into several IP fragment, each of size at most 64kB. Note that IP is designed with this purpose in mind, and all the necessary fields (length of fragment, offset within datagram, etc.) for doing the fragmentation and the reassembly are available in the IP header.

3. *What hypothesis should be done on the system if we want a message to be delivered to all the effectively correct chat clients?*
   In the current specification of the chat application, a client c_1 that detects another client c_2 as crashed because c_1 does not receive a heartbeat from c_2, considers c_2 as effectively crashed. In reality however, the heartbeat message sent by c_2 to c_1 in this case might well get lost in the network, or the timeout value chosen by c_1 before deciding c_2 is crashed might be too small. The consequence is that a chat client c_2 that is correct is now suspected to have crashed by another client c_1. In this case, c_1 does not send any future message to c_2 anymore, even though c_2 is a correct chat client. The simplest way of solving this problem is that any chat client systematically "forwards" any message that this client receives, to any other chat client, and we do not need any form of failure detector. With stronger synchrony assumptions on the network (i.e., if the network is totally synchronous), it is possible to come up with a solution that reduces the number of messages exchanged.