Charlie Su, Gott Phusit, Mark Yang
Masoud Saeida Ardekani
CS 490-DS0
13 March 2015

RELIABLE BROADCAST AND FIFO RELIABLE BROADCAST

Components of the Chat Client

Our Chat Client is divided into multiple parts. We have **ChatClient**, which acts as the container that pretty much holds everything together and acts as the interface to the user. We have **ChannelInterface** which deals with the lower level socket programming part and is in charge of delivering and fishing for messages through the channel. We have **ChatClientMessage** which bundles the user's message with information necessary for the broadcast algorithms. We have **ClientObject**, which acts as a miniature representation of other clients, containing the other clients information (e.g. port number, name) without the components that make the Chat Client actually work. Finally, we have the **Broadcaster** classes, including **BEBroadcaster**, **ReliableBroadcaster**, and **FIFOReliableBroadcaster**, which broadcasts the messages and also processes messages received before delivering them to the actual client.

The **ChatClient** implements **Runnable** and **BroadcastReceiver**. **ChatClient** has a separate thread for sending heartbeats to the server, which is integral to our perfect failure detector. The main thread is in charge of facilitating user input. Commands are marked with "\", such as "\list", which sends a request to the server for all currently online clients. If the user enters an invalid command, the client notifies the user of that.

The **ChannelInterface** implements **Runnable**, as it creates a separate thread for accepting new connections and receiving messages. The methods' names are pretty self descriptive (e.g. initClient means initializing with a new Client). Within the **ChannelInterface**, we have the part which implements the timer for the experiments. The timer starts right after the channel fishes a message successfully, and ends when it's done with the callback (the time it took for the message to be actually delivered to the client). If there are no commands, and if the user has already initiated a conversation with someone else, it will just send what the user puts in normally. We made the **ChatClient** pretty robust (though most of the work was put into the **ChannelInterface)** that makes it so that if any other client that it's connected to disconnects or if the server disconnects, it will be able to continue to operate normally.

The **ChatClientMessage** implements **Message** and **Serializable**. It acts as the **Message** we send between clients (and because we're sending it, we have to make it implement **Serializable**). Besides the fields/methods that adhere to the **Message** interface, we have added a new field called *type*, which is used as an identification for *how* the message is to be sent.

ID	Type
0	Best Effort Broadcast, non-broadcast messages*
1	Reliable Broadcast
2	FIFO Reliable Broadcast

**We use the same ID for Best Effort Broadcast and non-broadcast messages since both of them can be delivered directly to the client without any additional processing.*

Finally, the **ClientObject** extends **Process**, since in our initial assignment, we used the **ClientObject** as a representation of each Client, making **ClientObject** subclass of **Process** very fitting.

Broadcasting

To broadcast, we use **BEBroadcaster's** *BEBroadcast()* at the core for every broadcast. *BEBroadcast()* itself just calls the **ChannelClient**'s *whisper()* method, which grabs the **ChatClient**'s hashmap of all users, iterates through them, and sends the message through the outputstream.

Because of the different needs of different broadcast algorithms, each broadcaster does something before calling *BEBroadcast()*.

If BEBroadcast directly, type is set to 0. Nothing is appended to the message.

If RBroadcast, type is set to 1. Nothing is appended to the message.

If FIFOBroadcast, type is set to 2. The message # is also appended to the message.

Receiving

To receive, our channel constantly polls from the input stream in search of new messages. Once a new message is in the stream, it is fished out and sent to the FIFOBroadcast Object.

The FIFOBroadcast Object then calls RBroadcast, which calls BEBroadcast. At BEBroadcast, it checks for message type. If it is type 0, then it calls client.deliver() which delivers the message directly to the Chat Client. If the type is greater than 0, it returns the message to RBroadcast. RBroadcast repeats the checking process. If it is type 1, then it calls client.deliver(), otherwise it returns the message to FIFOBroadcast. Finally at FIFOBroadcast, if it is type 2, then it calls client.deliver(). This is to guarantee all the guarantees that are guaranteed as guarantees by the Broadcast algorithms. For example, FIFOBroadcast object won't be able to continue executing until the RBroadcast object is done delivering. Also, the "client.deliver()" stops the execution of the upper execution layer by returning null, making sure no extra work is done upon the successful client delivering.

ReliableBroadcast

After some discussion, we have decided to take the Lazy Algorithm approach. An e-mail exchange between Masoud and Gott reveals that we can pretend that we're in a synchronous system and use our heartbeats as a perfect failure detector. Because of the advantages of the lazy algorithm (less network congestion for example), we have decided to implement it the lazy way. However, we noticed that this approach also works in asynchronous systems. Even if the failure detector we have isn't perfect, it still has **strong completeness**, which means that we can ensure **agreement**. Even if it doesn't have strong accuracy, the extra RBroadcasts will not be RBDelivered because of the system we have set up. In effect, the more accurate our failure detector is, the more it approaches the Lazy Algorithm approach; the worst case (where every process is suspected) becomes equivalent to the Eager Algorithm approach.

FIFOBroadcast

We maintain a pending and a deliver set. No thread is used inside the FIFOReliableBroadcaster. When the message FIFOdelivers, the broadcaster continues to check for the next message in the pending set until the next message can't be found. This way, the **FIFO order** from each sender is maintained. Inside the FIFOReliableBroadcaster

class there's also an instance of ReliableBroadcaster inside to make sure the other properties are ensured.

The Makefile

There are two Makefiles in our directory; one in the parent directory and one that goes along with the source files. The one that resides along with the source files is there to compile all of the files together. The reason we are using a Makefile is that it is very convenient so instead of having to javac all of the files every time you need to compile, you can just simply type make and it will compile all of the files automatically. Also another issue that may arise if a Makefile is not used when trying to compile our files is that a lot of the files require to be compiled in a specific order and also compiled with other files otherwise problems with scoping and undefined objects will occur. The other Makefile in the parent directory allows for easy access for people to call make without having to delve into the folders in order to compile the files. The Makefiles we have implemented just make it really easy to compile everything in one place.

Experiment 1 (1 Client/Machine)

FIFO

Throughput: 0.109 0.0965 0.079 0.086 0.05 0.043 0.445 0.0865 0.089 0.0805

RB

Throughput: 0.046 .049 .0445 .049 0.1005 0.0945 0.0955 0.0835 0.091 0.0589

Note: the distinct throughput measurements are probably due to the fact that some of the clients are in the same network with the server.

Experiment 2 (2 Clients/Machine)

FIFO

0.0615 0.07 0.0645 0.0725 0.078 0.07 0.081 0.082 0.071 0.085

RB

.0755 .077 .0695 .0735 .073 .0735 .0715 .0755 .078 .085

Miscellaneous Notes/Observations

1] Printing is noticeably slower than the broadcasting, which makes some sense since IO operations are supposed to be slower. It made us doubt the correctness at first since the other clients received more messages than the crashed sender could receive until we could figure out this explanation.