DISTRIBUTED SYSTEMS

Distributed Systems – Assignment 2

Submitted: October 20, 2014
By: Devan Shah 100428864
Submitted to: Weina ma & Ying Zu

1. (a) Outline a scheme to solve the problem of dropped messages in IP multicast. Use message retransmissions. You should assume that (1) there may be multiple senders, (2) only a small fraction of messages are dropped, (3) recipients may not necessarily send a message within any particular time limit, (4) messages not dropped arrive in sender order. [5]

   (b) Does your solution in part (a) differ from the definition of reliable multicast? If so, explain how. [5]

   a) To solve the problem of dropped messages that use message retransmissions with IP multicast we have to consider how each of the individual assumptions would be solved, this way we can come up with a complete solution that would help solve the problem of dropped messages.

      The first assumption states that there may be multiple senders of messages, either at the same time or after each other. To accommodate for this the client (sender) needs to attach a sequence number to each of the messages that are sent to the server (receiver). This way the server knows what to expect, but for the server to know exactly what the sequence number is it needs to be able to determine what the next sequence number is going to be, therefore the server needs to keep track of the sequence number, possible save it. The server saving the sequence number that is received from the sender's message will allow for a way for the server to check the sequence numbers on each message that is retrieved from the server.

      The second assumption of only a small fraction of messages are dropped can be resolved with the use of negative acknowledge, which refers to the server (receiver) requesting the server to send the missing messages. Using the sequence number implementation the server can determine if a message is missed or not based on the sequence number that is received from the next message that it receives. When a missing message is detected by the server, the server can send a message back to the client mentioning that message with sequence number <sequence number> was missed please send it again. In this case the sender has to keep track of all the messages that are send to the server so that if there is a request to send a missing message again the server is able to retrieve the message again from client history and send it back to the server as a unicast message. This will help make sure that all the dropped messages are accounted for and are all received by the server.

      The third assumption of recipients may not necessarily send a message within any particular time limit, can be resolved with the use of a time out limit on the client for sending messages. With the use of acknowledges an acknowledge timeout is required or else the client will be waiting for a reply from the sever for ever and holding up the message queue. This issues can be resolved by implementing a time out on the client where, the client sends a message and then waits a certain period and then discards the message.

      Finally for the fourth assumption of messages not dropped arrive in sender order, will be handled by the sequence numbers that are attached to the sender's message.

   b) Yes the solution in part a does differ from the definition a reliable multicast which is classified as any message that is transmitted has to be received by all members of the group or none of the members. Therefore in the solution above where it is

mentioned that the server will ask for retransmit of the message if the server has not received the message, this will mean that the client has to retransmit the message this can mean that some members receive the message and some do not. There is also the occurrence of any crashes on the client side where the message gets lost like a connection issues. This can be resolved by store the message information in a file and restore from there when restarted.

2. Provide an application example for each of the following RPC exchange handshakes and provide a reasoning of why that particular handshake is suitable for the application you listed. [10]

	Messages sent by		
Handshake	***Client***	***Server***	***Client***
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

The *request* (R) protocol

The *request* (R) protocol refers to sending a single request message from the client to the server. The client continues on processing information as normal. An example of an application that uses this method for transfer would be a power button on a TV remote. The reason that this is a *request* (R) protocol is because the power button does not wait for a response from the TV when the button is clicked.

The *request-reply* (RR) protocol

The *request-reply* (RR) protocol refers to having the client send a request message and the server will reply with a message, this is technically considered an acknowledgement. An example of an application that uses this method would be a web browser retrieving a webpage using a URL. The reason that this is a (RR) is because the web browser requests for a web page using the URL and the web server is responsible for replying with the web page that was requested.

The *request-reply-acknowledge reply* (RRA) protocol

The *request-reply-acknowledge reply* (RRA) protocol refers to exchanging three messages which are request-reply-acknowledge reply. An example of an application that uses this method would be DropBox which performs synchronization between client and server of the data files. The reason that Dropbox uses *request-reply-acknowledge reply* (RRA) protocol is because the Dropbox client would request for an update of all the files, the server would respond with the new data (data is downloaded), at this point the Dropbox client would send a checksum of the entire Dropbox file archive to the server and at this point the sever knows that all the files have ben synced (confirmation by the checksum).

3. Show how to use Java reflection to construct the client proxy class for the *Election* interface. Give the details of the implementation of one of the methods in this class, which should call the method *doOperation* with the following signature: [10]

   byte [] doOperation (RemoteObjectRef o, Method m, byte[] arguments);

   Hint: an instance variable of the proxy class should hold a remote object reference

   The Election interface provides two remote methods:
   vote: with two parameters through which the client supplies the name of a candidate (a string) and the 'voter's number' (an integer used to ensure each user votes once only). The voter's numbers are allocated sparsely from the range of integers to make them hard to guess. *result*: with two parameters through which the server supplies the client with the name of a candidate and the number of votes for that candidate.

   Following is the algorithm needed to design implementation of one of the methods:
   - Use java reflection to construct the method objects
     - private static Method voteMethod Election.class.getMethod ("vote", new Class [] { java.lang.String.class, int.class });
     - private static Method resultMethod resultMethod = Election.class.getMethod ("result", new Class [] { }) ;
   - Use the RemoteObjectRef type object to create the Remote interface with all the host and port information
   - Build the byte array that contains the voter number and the casted vote
   - Use the doOperation to invoke the functions that
     - doOperation (RemoteObjectRef ref, resultMethod, byte [] args)
     - Note the args include voter number and caster voter name
   - This will perform the operation of getting the results
   - The same can be done for the vote function.

   Following is the pseudo code for implementing the results function with the use of java reflection:

```
public class ElectionClientProxy
{
  // Define the variable that will contain the remote invocation information
  RemoteObjectRef ref ;
  private static Method voteMethod;
  private static Method resultMethod;
  static
  {
    try {
                    voteMethod = Election.class.getMethod ( "vote", new Class[]
{java.lang.String.class,int.class} ) ) ;
                    resultMethod = Election.class.getMethod ( "result", new Class[] {} )
) ;
                } catch(NoSuchMethodException e) { System.out.println (
"NoSuchMethodException: " + e.getMessage() ) }
  }

  public synchronized Vector<Object> result () throws RemoteException
```

```
  {
    try
    {
      doOperation ( ref, voteMethod, {} );
    }
  }
}
```

4. The *Hello* RMI example has been given to you. Extend the programs to support RMI callback feature. Modify the clients so that they provide a *notify* remote method. Modify the server so that it provides two additional remote methods, *registerForCallback* and *unregisterForCallback*. Each new client registers with the server by remote invocation of *registerForCallbacks*, and upon this event, the server makes callbacks to *notify* method of all the current clients, informing them of the current number of clients. The server must use a data structure to maintain the list of clients. The client should also let user specify the length of time to remain registered, after this time expires, the clients unregisters with the server.
   You should provide screen shots of execution of your programs to show that the RMI callback feature has been implemented. You should also show the code you have added to implement the RMI callback. [20]

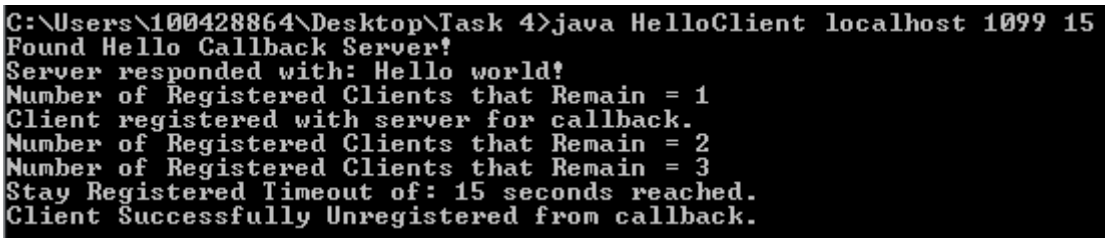

Figure 1 Client one registering for callback for 15 seconds

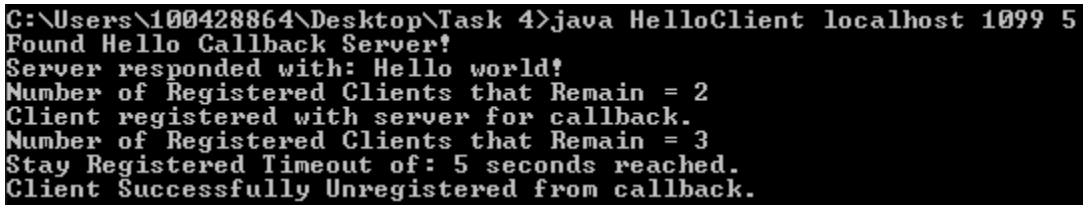

Figure 2Client two registering for callback for 15 seconds

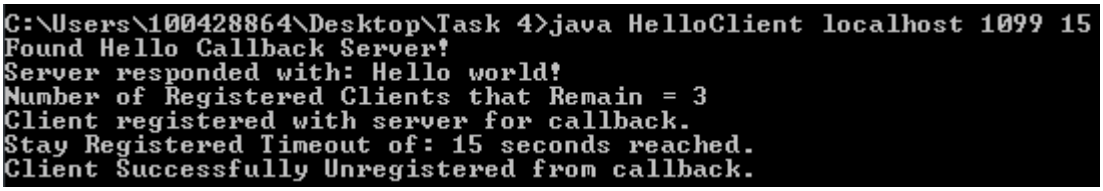

Figure 3Client three registering for callback for 15 seconds

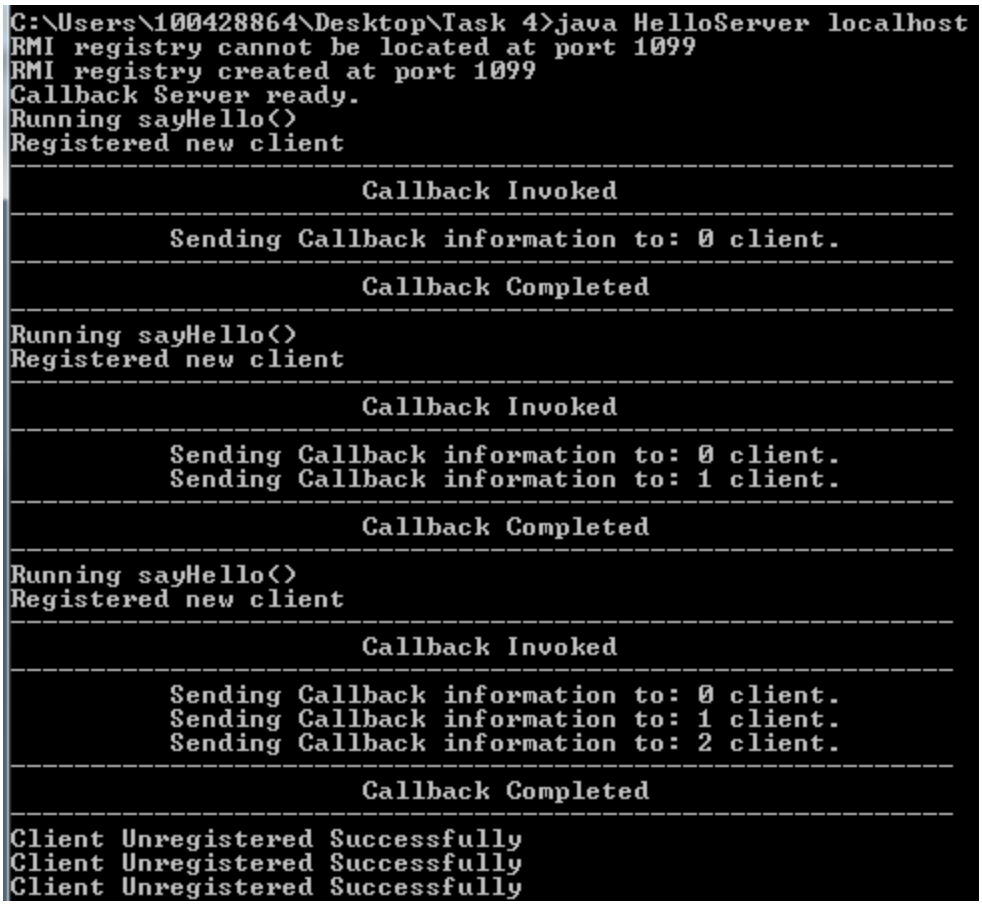

Figure 4Hello Server registering the clients and unregistering after 15 second timeout reached

Above is the output of the execution of the program, I have included all the source codes for the implementation of the RMI callback. The source files can be found under "Distributed Systems - Assignment 2 - Question 4/src" in the attached zip file.

5. The text book presents a Use Case for tuple space communications based on the JavaSpaces software application. You can read this on pages 271-274. An example is presented of a Fire Alarm and its propagation to interested parties. For the FireAlarm example create a Class diagram and appropriate sequence diagram for the scenario of raising an alarm. [10]

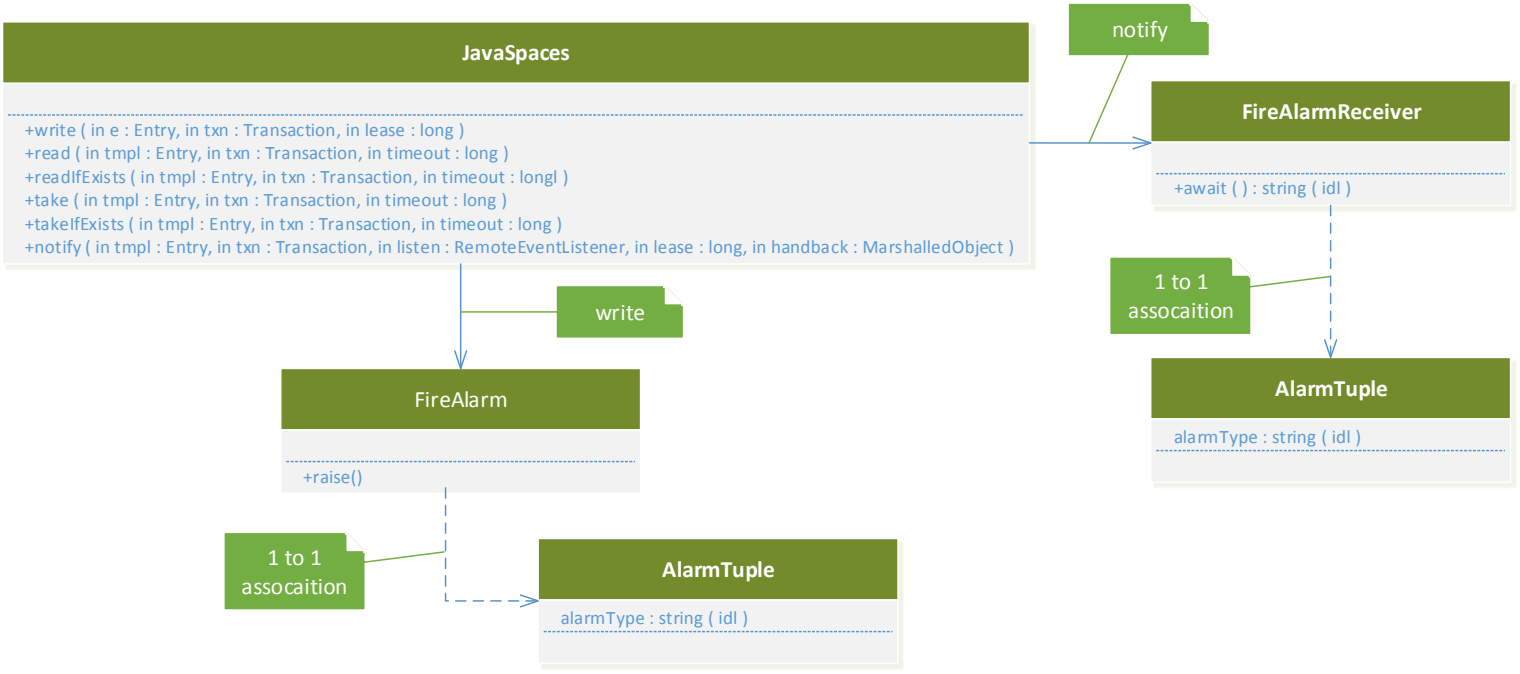

Figure 5 Class diagram for the tuple space communication.

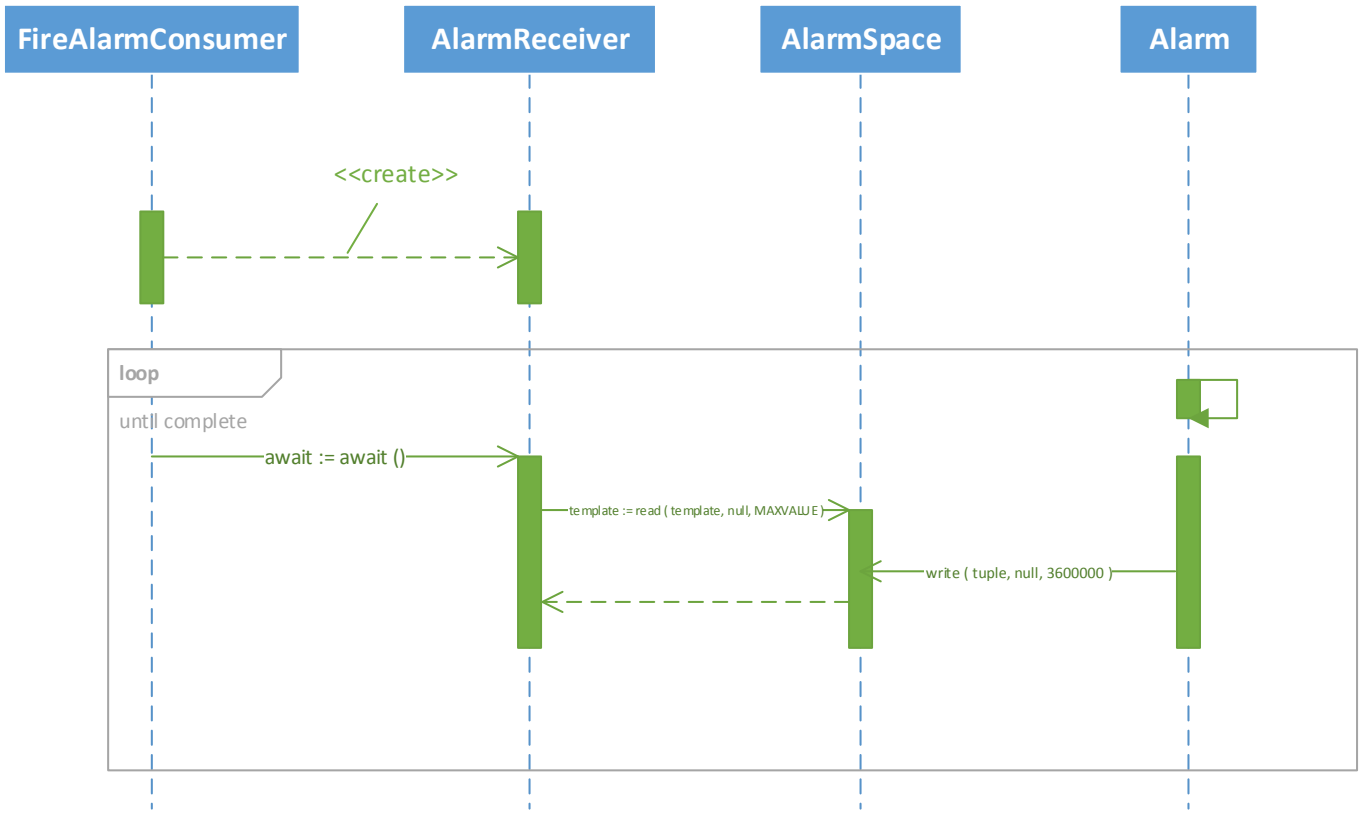

Figure 6 Sequence diagram for the raising an alarm scenario.