Homework #4 – A Totally Ordered Multicast Protocol Using Java

Note:
1. Submission must be made in electronic format to the assistant.
2. You need to work in groups of two, choose your partner.

Requirements

You are required to write Java programs that implement a sequencer-based multicasting service and a small client application to exercise it. You will use Java RMI and multicast sockets.

The main classes in your implementation will be:

TestSequencer.java
Each instance (you will create several) will allow the user to enter strings and multicast them to a group of instances of TestSequencer.

Group.java
TestSequencer uses an instance of Group for group communication services. Group in turn uses a MulticastSocket to receive incoming messages, and uses RMI to the sequencer for sending messages and other, sequencer-specific operations.

Sequencer.java, SequencerImpl.java
The interface to and implementation of a sequencer

History.java
This is implementation of the sequencer's history for storing a certain number of multicast messages. The sequencer keeps the history since some of the past messages may not have reached some group members. History does not need to have a fixed capacity, but obsolete items should be removed whenever its size grows beyond a threshold, say 1024 entries, or the sequencer may use a special synchronization phase with all the group members.

You will be given:

1. The RMI interface to an object which will implement the sequencer (Sequencer.java) and two trivial classes associated with the Sequencer definition: SequencerJoinInfo.java, SequencerException.java
2. The outline code for Group.java
3. Some hints on marshalling messages into datagram packets
4. A reference implementation for the application (class files only).

You will need to

1. Write SequencerImpl.java and History.java from scratch.

Choosing a multicast address and time-to-live
SequencerImpl should take as an argument the IP multicast address that the corresponding Groups are to use. To avoid collisions with numbers chosen by others, incorporate a random number into the multicast IP address that you use -- e.g. 234.day.month.rand, where day and month are chosen from a team member's birthday, and rand is a random number between 1 and 254. Don't set your sockets' time-to-live (TTL) to more than 1. That way, your multicast packets will not be transmitted beyond the local Mbone router.

Organization

This project is suitable for implementation by groups of two students. The following division of labor might be used:

Team member 1: SequencerImpl.java and any associated helper classes apart from History Team member 2: Group.java Together: TestSequencer.java and History.java

Teams hand in all code and a written report of between four and six pages, explaining the design and its rationale.

Teams are required to demonstrate their implementation. The demonstration should include as many as possible of the following features: simple message sending, stress-testing, recovery from simulated multicast datagram loss, heartbeat messages and history truncation.

Reference implementation

A reference implementation (class files only) is here.

Code

Sequencer.java (complete) -- Interface

package sequencer;
import java.rmi.*;
import java.net.*;
import java.io.*;
public interface Sequencer extends Remote {
    // join -- request for "sender" to join sequencer's multicasting service;
    // returns an object specifying the multicast address and the first sequence number to expect
    public SequencerJoinInfo join(String sender) throws RemoteException, SequencerException;

    // send -- "sender" supplies the msg to be sent, its identifier,
    // and the sequence number of the last received message
    public void send(String sender, byte[] msg, long msgID, long lastSequenceReceived) throws RemoteException;

    // leave -- tell sequencer that "sender" will no longer need its services
    public void leave(String sender) throws RemoteException;
}
// getMissing -- ask sequencer for the message whose sequence number is "sequence"
public byte[] getMissing(String sender, long sequence)
    throws RemoteException, SequencerException;

// heartbeat -- we have received messages up to number "lastSequenceReceived"
public void heartbeat(String sender, long lastSequenceReceived)
    throws RemoteException;

SequencerJoinInfo.java (complete) – Helper class

package sequencer;

import java.io.*;
import java.net.*;

public class SequencerJoinInfo implements Serializable
{
    public InetAddress addr;
    public long sequence;
    public SequencerJoinInfo(InetAddress addr, long sequence)
    {
        this.addr = addr;
        this.sequence = sequence;
    }
}

SequencerException.java (complete) – Helper class

package sequencer;

import java.io.*;
public class SequencerException extends Exception implements Serializable
{
    public SequencerException(String s)
    {
        super(s);
    }
}

Group.java (outline only)

package sequencer;

import java.net.*;
import java.util.*;
import java.io.*;
import java.rmi.*;
public class Group implements Runnable
{
    public Group(String host, MsgHandler handler, String senderName)
    throws GroupException
    {
        // contact Sequencer on "host" to join group,
        // create MulticastSocket and thread to listen on it,
        // perform other initialisations
    }
    public void send(byte[] msg) throws GroupException
    {
        // send the given message to all instances of Group using the same sequencer
    }
    public void leave()
    {
        // leave group
    }
    public void run()
    {
        // repeatedly: listen to MulticastSocket created in constructor, and on receipt
        // of a datagram call "handle" on the instance
        // of Group.MsgHandler which was supplied to the constructor
    }
    public interface MsgHandler
    {
        public void handle(int count, byte[] msg);
    }
    public class GroupException extends Exception
    {
        public GroupException(String s)
        {
            super(s);
        }
    }
    public class HeartBeater extends Thread
    {
        // This thread sends heartbeat messages when required
    }
}

**Marshalling data into a datagram**

ByteArrayOutputStream bstream = new ByteArrayOutputStream(MAX_MSG_LENGTH);
DataOutputStream dstream = new DataOutputStream(bstream);
dstream.writeLong(aLong); // marshals a Long into the byte array underlying bstream

After marshalling, the data can be obtained from bstream for inclusion in a DatagramPacket:
byte[] theData = bstream.toByteArray();

(For unmarshalling there are corresponding classes ByteArrayInputStream and DataInputStream.)

**Theoretic Background**

This section describes the reliable multicast protocol developed by Kaashoek and Tanenbaum for group communication. This is a totally ordered multicast protocol that can be configured to provide a range of degrees of reliability. It employs a sequencer to order multicasts totally, and it uses employs multicast to minimize the number of messages transmitted when multicasting over a local area network. The main achievement of this protocol is that in the normal case where no messages are lost, a multicast requires only two messages: (1) From the message originator to the sequencer (point-to-point) (2) From the sequencer to the group members. The group members send messages only if they notice they miss a message from the sequence number of the last message they have received. If all multicast messages are funneled through a single member of a process group, that member can assign message identifiers from a sequence counter. The funneling process is called the *sequencer*. Figure 1 illustrates the role of the sequencer in the transmission of a multicast message. The originator of a multicast message sends it to the sequencer, which adds a sequence number and then relays it to the other members using a single broadcast message. The sequence numbers are used to ensure that multicast messages are delivered in the same order to all members.

**Figure 1. Multicast with a sequencer**

![](image)

*Note:* the dashed arrows indicate (infrequent) requests for missing messages.
The components that take part in the protocol are the members of a process group and the communication kernels that reside at each computer hosting a group member. The kernels implement multicast; the sequencer for sending and the rest for receiving the multicast messages. At any one time, just one of the kernels acts as the sequencer. Several members of a group may run on the same computer. It is a straightforward matter to distribute a multicast message arriving at a computer to all the local group members. We therefore concentrate on multicasting a message to a set of destinations, whereby ‘destination’ is meant computer.

In the simplest version of the protocol, the sequencer keeps the following information:

- A list of destinations for the multicast messages (we assume that there is only one group and therefore one such set of destinations, but there could be more);
- A sequence number that is incremented by one for every new multicast message. The sequence numbers are used to ensure that request messages are delivered in the same order to all the multicast destinations;
- A history buffer, which holds a list of messages already sent to the destinations, together with their sequence numbers.

The originator of a multicast attaches a unique identifier to the message and transmits it point-to-point to the sequencer. The sequencer increments the sequence number and appends it to the message, which it stores in the history buffer. Then it transmits the message to all of the destinations. Hardware broadcast or multicast is used by the sequencer; assuming it is available, allowing the same network packet to be used to reach the multicast destinations and to acknowledge the originator’s message. Otherwise, point-to-point messages would have to be used.

The originator times out and retransmits the message as necessary until it receives its own message from the sequencer in multicast. The sequencer checks for repeated messages against its history buffer, and merely sends an acknowledgement to the originator if the message is found there. By contrast, the communication between the sequencer and the destinations uses a negative acknowledgment scheme in which the destinations request retransmission of lost messages from the sequencer. Lost messages are detected when a message arrives with a higher sequence number than is expected by the recipient. This is an example of a protocol designed to perform best under the normal conditions for local area networks – that is the loss of messages is infrequent. On the other hand, if a message does not arrive at a member that hasn’t multicast messages for a while, then it won’t find out that the message is missing until it sends a ‘heartbeat’ message to the sequencer, as discussed below. Unless precautions are taken, the number of messages stored at the sequencer will grow indefinitely. The protocol takes the following steps to ensure that the history buffer capacity is not exceeded:

1. The highest sequence number received by the originator is piggybacked on all the multicast messages. This enables the sequencer to record the highest message identifier seen by each member. Messages are removed from the history buffer after all members of the group have acknowledged them.
2. To ensure that there are regular acknowledgments even when members are not originating multicasts, each member is expected to send periodic ‘heartbeat’ acknowledgments of the highest sequence number received.
3. If the space occupied in the history buffer exceeds a pre-defined limit, the sequencer enters a synchronization phase during which no multicasts are done and the sequencer
requests and ensures that all the members fetch any outstanding messages. After that the sequencer deletes the history buffer’s contents and resumes its normal role.

Consider the case of a single group member multicasting repeatedly, with no other communication by group members. This is a worst-case scenario from the point of view of deleting messages from the history buffer. If the history buffer can hold $h$ messages and if there are $N$ group members, then after $h$ messages are sent each of $N - 1$ members will have sent a heartbeat message, as mentioned above, in order that the history buffer is not filled. Assuming hardware multicast or broadcast is used, the number of packets required for $h$ messages is $2h + N - 1$, and the number of packets per message is therefore $2 + (N-1)/h$. This number is liable to be close to 2, for a reasonably sized history buffer.

One potential problem with this protocol is that the sequencer may become a bottleneck when the number of group members becomes large and multicasting is frequent. The results of a theoretical analysis are quoted by Kaashoek and Tanenbaum [1991], who suggest that this would be a problem at around 400 nodes for the most multicast-intensive applications they tried, although they do not specify what this is.

The most serious deficiency of this protocol as described is that its behavior is unsatisfactory for many applications under conditions of failure.