Comp 734 - Assignment 4: Extendible Fault-Tolerant Broadcast

Date Assigned: Tue November 3, 2015

Part 1 Completion Date: Tue November 5, 2015

Part 2 Completion Date: Tue November 12, 2015

Part 3 Completion Date: Tue Nov 24, 2015

The assignment is divided into three parts to make it easier. In the first part, you will simply convert your previous assignment to GIPC session port using GIPC P2P communication among asymmetric processes. This part should be done in the Thursday, Nov 5 class.

In the second part, you will convert the P2P version to a symmetric one in which the relayer can be changed dynamically.

In the final part, you will convert part 2 to a fault tolerant part.
See package sessionport.rpc.duplex.direct.counter_example for a template for your project. You will have to pull the new version of GIPC and change the name of your initPortLauncherSupport to have the correct spelling.

Part 1: GIPC InputPort → GIPC Session Port with Atomic Broadcast

In the previous assignment, you used GIPC duplex RPC port to connect your clients to a relayer. Also you did not implement atomic broadcast. In this part, convert your input port to a session port. This means, both your client and server launcher will implement the overriding method:

```
   protected PortKind getPortKind() {
                 return PortKind.SESSION_PORT;
   }
```

When your port kind is session port, your port is of type DuplexRPCSessionPort rather than the vanilla InputPort.

A session port can be configured to do P2P or relayed communication through a session server. As you already have a relayer, choose the P2P option by implementing the overriding method:

```
   protected SessionChoice getSessionChoice() {
                 return SessionChoice.P2P;
  }
```

For each client, you need to override the method:

Moved (insertion) [1]

protected String getClientId()

to return a String denoting a port number. For example, for Alice you can return “9092” and for Bob return “9093”.

The client constructor should also, after invoking super, invoke the following method:
DirectedRPCProxyGenerator.setDoShortCircuitLocalCallsToRemotes(false);
This makes sure that a call to the local remote object (a relayer in this example) goes through the various distribution layers.

A session port needs a session server to keep track of who is registered. GIPC comes with a session server with a main. However, it uses the default serializer. To use your serializer, subclass ASessionServerLauncher, and as before override the method:

```
        protected void initPortLaucherSupports()
```

method to register your own serializer as in the previous assignment. Put the @Override annotation before the method to make sure you are indeed overriding it. Make sure you call super.intPortLuancherSupports() as the first line in the method.

Moved up [1]: For each client, you need to override the method:¶
¶
protected String getClientId()¶
¶
to return a String denoting a port number. For example, for Alice you can return "9092" and for Bob return "9093".¶

Write your own main in this extended class, which is like the main of the inherited class, except it creates an instance of your new class rather than ASessionServerLauncher. The constructor of the inherited (and new class) take a server name and server id as arguments. Your client launchers should refer to them now as the names of the server and client. Use the same GIPC constants as the super class (SessionServerLuancher.SESSION_SERVER_ID and SessionServerLauncher.SESSION_SERVER_NAME).

You should launch the server before you launch the client processes.

As in the previous assignment, your relayer will register the remote object for relaying messages. The clients will need to connect to it. We will not require your server to start before other processes. Each member of the session can register a connect listener by overriding the method:

```
protected  ConnectionListener getConnectionListener (InputPort
anInputPort) ;
```

The returned listener is registered with the port. As you have created a session port, anInputPort will actually be a subclass of InputPort – DuplexRPCSessionPort. This port type is needed to create the proxy.

The method has the following signature.

```
public synchronized void connected(String remoteEnd, ConnectionType
aConnectionType);
```

When a member joins, the connection type is: ConnectionType.MEMBER_TO_SESSION .
The remoteEnd is the name of the joiner. You can get a proxy to some object registered by a peer by invoking:

```
            DirectedRPCProxyGenerator.generateRPCProxy(sessionPort,
remoteEnd, nameOrClassOfRegisteredObject, null);
```

When your port kind is session port, your port is of type DuplexRPCSessionPort rather than the vanilla InputPort. The variable mainPort in the laumcher holds a reference to the session port and you should cast it to DuplexRPCSessionPort when passing it to the proxy generation method.

This should be sufficient to do part 1.

The method:

```
            public synchronized void disconnected(String remoteEndName,
                  boolean explicitDsconnection, String systemMessage,
                  ConnectionType aConnectionType)
```

is called when a session member leaves voluntarily or via failure. This method will be needed in the last part.

Part 2: Symmetric Dynamically Changeable Client+Relayer Processes

In part 2, make this more symmetric, getting rid of your special relayer process. Each client process now can act as a relayer. Choose some algorithm to elect the current relayer (for e.g. lexically the first one). It should be possible to run the algorithm locally. You cannot use the order in which the connect method is called as the local user always joins first. Make the elector a separate object that is selected by a factory, so that you can change it without affecting other parts of the program. Also make sure it is independent of the application, that is, knows nothing about the specific calls to join or leave a custom relayer or how to relay messages through it. Your application can know about the elector. Later, this elector will become the application-independent fault tolerance manager.

You will have to change your out coupler to always send to the current relayer object rather than through a fixed object. This means it must know which proxy to use based on the current elected relayer. It can get this indirectly information from the relayer connector, which communicates with the relayer elector and can keep track the proxies, or it can communicate directly with the elector and keep track of the proxies.

The relayer elector needs to elect a new relayer each time a new member is added or removed from the session. These events are communicated by GIPC to the connect listener. This means the connect listener must inform the elector about these events.

When a new relayer is elected, the client must leave the old relayer and join the new one, using application-specific join and leave calls. The relayer elector should not do this application-specific task. Instead, the connect listener, on discovering a session change, should inform the elector about it, make it run its election algorithm, and if a new relayer is elected, make the application-specific calls.

Part 3: Fault Tolerant Transparent Broadcast

The goal of this part is to support 1-fault at a time transparent tolerance for relayer failures. Relayer failure occurs when the current relayer dies, possibly in the middle of relaying a series of messages. Tolerance for this failure means that its tasks are taken over by some other process, while maintaining communication consistency. Transparency means that except for launcher or configuration code, the communicating application (relayer server and relayer clients) is unaware of the fault and fault recovery. This means that fault tolerance is managed by external objects. These fault tolerant objects can make the following assumptions: the application broadcast messages, and that all broadcast messages are sent reliably and atomically through the relayer. Communication consistency means that the reliable atomic broadcast constraints are still maintained in the presence of faults with the caveat that a message generated by the site of the failing relayer may not be delivered to anyone. To ensure this constency, the fault tolerance manager must be able to know which messages are sent to the relayer and which are sent by the relayer. This knowledge is captured by an application-dependent configuration object. Thus, as in RMI, some application-specific effort must be expended to support fault tolerance. Transparency will substantially increase your implementation effort. You can assume that all application messages are sent through GIPC. You can also assume that all session members join before the first message is sent.

You must both choose an appropriate software architecture and an algorithm. Let us consider first the software architecture. You need a way to intercept messages sent and received through GIPC, and decide between messages sent to and by the relayer. You already know how to intercept messages in GIPC to interpose your own serializer (and deserializer). Replace your serializer and deserializer with a new object that both serializes/desrializes messages and forwards messages to a fault tolerance manager object. Naturally, you should keep the forwarding and (de)serializing code in separate classes. Ideally these classes should have a delegation relationship – I took the easier and less pure approach of inheritance.

As mentioned above, your forwarders will need to distinguish between messages sent to and by the relayer. To support separation of concerns, assume a filter object, which contains methods that make this distinction. This filter object should be created through a factory so your forwarder is isolated from the application object. Not all interceptable messages are generated by application code – there are also calls (such as join and leave) made by the underlying GIPC session port layer to the GIPC session manager. These are not intercepted – they are processed as they would be in the underlying GIPC system. A forwarder may process a serialized/deserialized message by (a) forwarding it (to the next stage in the pipe line) without any processing (through the fault tolerance manager) , (b) not forwarding it, instead storing it in the fault tolerance manager, or (c) forwarding a series of stored messages. A forwarded sent message can wrap a sent message and be a new kind of message communicated between FT managers. In both of these cases, the message must be interpreted by the receiving FT manager.

As your application makes method calls, the filter object needs to know something about how method calls are represented in GIPC. Each method call is marshalled into an instance of the type SerializableCall. This type has several methods to determine which method is called. Of particular relevance is the toHeader() method that represents the method header as a String,. To make this object application independent, let it define methods to register headers for remote method calls that are invoked in and by the relayer. Your launcher/configure can call these registration methods.

Two alternative architectures are possible to connect the forwarders, fault tolerance manager, and filter object. The forwarder knows about only the manager, which knows about the filter or the forwarder knows about both kinds of objects, and the manager does not know about the filter.

This leaves us with the most complex task here, the fault tolerance manager. Create this object also through a factory. It is closely related to the relayer elector of the previous part. You can make its class a subclass of the class of the former. As before, we want the fault tolerance manager to be a generic component.

Here are some aspects of the implementation based on the algorithm I implemented.

A fault tolerance manager at each site (not just the site of the current relayer) generates sequence numbers to enable recovery – when a relayer dies, the rest of the FT processes must go into a synchronization phase to ensure consistency. Ensuring consistency is complicated by the fact that dead relayer may have sent some but not all messages. In particular, both the sender of a message and the new relayer may not have received a message partially broadcast by the dead relayer.

As a relayer broadcasts sequentially, at quiescence (when all sites have received pending messages), the sites can be off by only one message, assuming that the machine of the dead relayer has not also died, or it is not responsible for retransmitting old unreceived messages, or the connection close message is sent after all preceding messages have been received. If this is the case, the new relayer site needs to send at most one message to each of the remaining sites.

As the FT manager at this site itself may not have this message, this message must be sent to it by the FT manager of some existing site. Rather than it asking all of the other sites, which results in an extra message and increases latency, each of the (FT manager at) other sites can simply send the latest message they have sent or received from the last relayer. The reason for storing the last received message is that the latest message might be the one sent by the dead relayer. The FT manager of the new relayer must find the max of these messages and resend it if there is at least one site whose most recent message is not this max message. As the FT manager itself is not responsible for the relaying, it must forward the max message to the local application-specific relayer, which will of course send the message to all sites, even those that had received it as it is fault tolerance unaware. The FT manager can suppress the delivery of the messages to sites that have received it, or it can send the message to all sites, which can filter out duplicate messages. This means that the FT manager at each site keeps track of the next sequence number it is expecting. This, in turn, means that sequence numbers are sent by the fault tolerance manager of the current relayer , the master FT manager, to the FT managers of all other sites, the slave FT managers, which keep track of the next expected sequence number.

While the various sites can be at off by at most one in the messages they receive, each site could have several in transit messages destined for the old relayer that never were never received. Hence these sites must keep track of all sent messages that have not been echoed back. This means when a message is echoed back by a master, it must be wrapped with information to identify the original sender. This in turn means the FT manager of the original sender must wrap the message appropriately before sending it to the master.

Breaking the assumption that message transmission occurs in rounds and message delivery is off by at most one, requires some extensions. One extension is that the new relayer starts the synchronization phase by sending a message to everyone about the latest message it has. No synchronization message is sent by a site until the start synchronization message is received. Each of the remaining sites site then sends messages it has generated or received from the dead relayer that are greater than this sequence number. This means each local site must buffer its messages. Another extension is to make each site that sends a message to also send with the message the sequence number of the last message it received. Now a site has to buffer only those messages that have never been received by any site. A variation of this approach would not require a start synchronization message. A site would send all buffered messages to the new relayer. If site A knows that the new relayer has received all messages from A, then A can send an empty buffer.

It can be proven that in an asynchronous system (in which messages do not go in rounds) it is not possible to have a fault tolerant distributed consensus. So we assume no fault tolerance during the synchronization phase.

As the relayer is fault-tolerance unaware, the FT manager must keep track of the relationship between the broadcast sent by a client to the relayer and the relay messages generated by the application-specific relayer. On each broadcast received from a slave, it increments the sequence number. This sequence number is wrapped in each of the N relay message caused by the broadcast and generated by the application-specific relayer – where N is the size of the session. One way to do so is for the master FT manager to delay delivery of a new broadcast message to the relayer until the relayer has sent the appropriate number of relay messages to the session numbers.

During synchronization, the application might still be generating new messages. These need to be buffered somewhere. We can buffer them either at the sending site or the site of the new relayer. As these messages must be sequenced after the unechoed sent messages, buffering them at the sending site seems to be the only alternative.

The synchronization phase starts when a fault is detected. It consists of each site (including the new relayer site) making an application-specific join call (via the relayer elector) in the relayer of the new relayer site, and sending to the FT manager of the site of the new relayer their most recent message. The FT manager asks the local relayer to send a message, if necessary, and then sends a finish synchronizing message to the sites of all other FT managers, including itself.

Based on this discussion. A slave/master fault manager must process:

1. Sent-broadcast: A message sent by a relayer-client to a relayer.
2. Received-broadcast: A message received by a relayer from a relayer-client.
3. Sent-relay: A message sent by a relayer to a relayer-client.
4. Received-relay: A message received by a relayer-client from a relayer.
5. Received-synchronizing message: A message received by the master during the synchronization phase.

6. Received-finished synchronizing message: A message indicating the end of the synchronization phase received by the slave.

A slave FT handles:

1. A sent-broadcast by wrapping it and performing book keeping before forwarding it (giving it to the next pipe line stage).
2. A received relay by unwrapping and performing book keeping and forwarding it.
3. A received-finished synchronizing message as mentioned above.

A master FT handles:

1. A sent-broadcast by unwrapping it, performing bookkeeping and forwarding it.
2. A sent-relay by wrapping it, performing bookkeeping, and forwarding it.
3. A received synchronizing message by processing it as mentioned above.

You can implement two different classes for the slave and master FT manager or you can create a single class that does both functions. The site of the current relayer behaves both as a master and a slave. The other sites do only slave functions.

Submission (to be edited)

(a) Create a YouTube video demonstrating part 3 .

(b) Submit your code on Sakai.

.