HW4 – Reliable Inter-Process Communication

Estimated time: 20-28 hours per person

Objectives

- Gain experience with creating reliable communication from underlying unreliable communications
- Become more familiar with unit testing techniques
- Become more familiar with using infrastructure in the cloud

Overview

During this assignment, you will implement and test the remainder of your communication subsystem and complete about 50-60% of your application-layer components. By the end of HW4, your communication subsystem must support reliable communications for all protocols that require it, which should be most of them. This means that a conversation will make a "reasonable effort" to achieve successful completion in the presence of lost messages, duplicate messages, late message delivery, and out of order messages. You may need to refine your protocol definitions to describe how conversations will deal with each of these situations.

You will also extend your unit test cases to verify the correctness of the new (and previous) functionality.

As with previous assignments, your execution environment will be a virtual machine in a cloud environment.

Ideas for Implementing Reliable Communications

Reliability in conversations have to be implemented in several ways.

Reliability for initiators of a send-receive sequence

First, any protocol that requires a process (P_1) to send a message (M_1) to one or more process and expects a response (M_2) from at least one of those process, needs to implement reliability for a send-receive sequence. To do this, P_1 must able to determine if M_1 is lost, if M_2 is lost, or if the receiving process(es) failed. Unfortunately, within the context of a single conversation, there's no sure way to know that any one of these things occurred. So, P_1 needs to make a "guess" when it suspects one of these failures. This is typically done with a timeout. Specifically, P_1 estimates an upper bound on the time that should lapse between the sending of M_1 and the receipt of M_2. If more than that amount goes by with no M_2 coming in, then P_1 "guesses" that one of the above failures occurred.

However, it doesn't know which one. If M_1 or M_2 were lost, the conversation still may be able to succeed by resending M_1. If the receiving process(es) failed, then successful completion may not be

possible. Recovery from process failure typically goes beyond the scope of a single conversation because it requires other conversations to restart the failed process or to allow another process to take over the failed processes responsibilities. Because P_1 doesn't know which failure really occurred, it optimistically re-sends M_1.

Below is a basic algorithm for achieving this type of "send-receive" reliable within the context of a conversation.

```
Let M1 be the outgoing message
Let RMT be a set of types of valid response messages (M2)
Let T be the upper bound on expected time between
            sending of M1 and receiving of M2
Let R be the number sends the process will do before
            giving up and terminating the conversation with
            a failure status

SendsRemaining = R
While (SendsRemaining > 0 && a valid response hasn't coming in yet)
{
      Send M1 to its destination(s)
      SendsRemaining--
      Wait for an incoming message to this conversation for up to T
      If got a message then
            If the type of message is in RMT, then
                  Got a response
            Else
                  Ignore message
}
If got a response
      Process the response (do whatever the conversation is
      supposed to do next)
Else
      Terminate with a failure status
```

This algorithm works well if each conversation is encapsulated in its own object and especially if it is running on its own thread. However, it can be varied to match other approaches for handling conversations or communication subsystem designs.

Also, note that this algorithm is easily generalized using the Template Method design pattern, so each specific type of conversation only has to implement the parts that differ, like the setup of RMT, T, and R; the processing of the response; and the terminate with a failure status. If you are not familiar with the Template Method design pattern, I strongly recommend that you look it up and study it.

Reliability for message receivers

A second reliability issue arises when a message receiver (P_2) might receive a duplicate of a message (M_1) from a sender (P_1) in the context of the same conversation. This can happen if P_1 guesses that there was a failure and resends M_1. P_2 can either simply reprocess M_1 and send a new response, resend

its previous response (M_2), or ignore the duplicate. The choice for which approach takes depends on the conversation's protocol and its purpose in context of the whole system.

A third reliability issue exists within the context of a conversation, when a receiver (P_2) needs to receive and process more than one message and their order is important. In such situations, if P_2 receives messages out of order, then it has to done some work to put them in correct order before processing. To address, P_2 needs to place any incoming messages that are out of order in a "holdback queue" until the required preceding messages coming in.

Instructions

1. Refine / extend your protocol specifications, architectural design, and functional requirements, as needed
2. Refine / extend your design for reliable communications
3. Look for opportunities to generalize and reuse the core logic for achieve reliable communications. (Localize design decisions; don't create duplicate code.)
4. Implement all the necessary changes
5. Test your communication subsystem thoroughly
6. Implement and test (using ad hoc methods) about 50-60% of your application logic
7. Run in the system (at least the parts are complete) in a cloud environment

Submission Instructions

Zip up your design documents and entire implementation workspace into an archive file called CS5200_hw4_<*fullname*>.zip, where fullname is your first and last names. Then, submit the zip file to the Canvas system

Grading Criteria

Criteria	Max Points
A well-thought out, appropriate, and documented architectural design, plus updated functional requirements and protocol specifications	30
A good initial implementation of a reusable communication subsystem	45
Thorough executable unit tests for key components in the communication subsystem	45
Implementation of about 50-60% of the application's functionality, with ad hoc system testing in a cloud environment	30
TOTAL MAX POINTS	150