CSE 6117 Distributed Computing Assignment 2

Omar Abid
Student ID: 211295573
CSE Login: omarabid

01-20-2016

Introduction

In this exercise we look at two models which have multiple processes each of which communicate using some form of a message passing system. We assume that the channels are asynchronous and that no process failures occur. Furthermore, each process has a unique ID and knows the processes ID of the message received as well as which one it needs to send to.

Model Properties

Model A and Model B properties are described as such.

1. Model A
   - 8-bit message
   - messages may be reordered in the channels
     - may require an explicit message ordering implementation
   - each message guaranteed to be delivered eventually
2. Model B
   - messages can be dropped (communication failure)
   - at least one copy will be delivered eventually
   - each channel is FIFO (aside from dropped messages)
     - Implicit message ordering
   - no limit on the length of messages

1 Is Model A at least as strong as Model B? Show your answer is correct.

In order to show this, we have to show that Model A simulates B, that is, all the primitive step of B can be simulated in A.

Primitive Steps that we must simulate are consistent for both Models, that is a send and receive step.

Simulating FIFO can be done with some function send(m,c_i). That is, for each message sent from process ID k to process ID i, we send a time stamp counter given by c_i. This will be used to simulate the FIFO system of model B. Because of the nature of the scheme, a new message will not be sent until the previous one has been fully received and an acknowledgement sent (we will discuss this further).

Simulate any length n message with only an 8-bit channel by breaking the message into $\lceil \frac{n}{6} \rceil$ separate messages followed by a *begin block* and an *end block* each consisting of 8-bits. Below shows an example of a message being passed with 15 bits.

begin block : [0 1 6-bit message] middle block : [1 0 6-bit message] middle block: [1 0 3-bit message 3 bit dummy 0's] end block: [1 1 (0 0 0 0 1 1)].
Here we see that the end block is specified with a [1 1], and the remainder (0 0 0 0 1 1) specifies where the last message ends in binary. Hence we see that only the first 3 bits of the last middle block before the end block contains our message. Using this scheme we can send any length n message across our channel using an 8-bit channel.

Sending a Message is done by simulating the FIFO and simulating any length n message and sending it across the channel. Since each message sent across the channel is guarenteed to be delivered eventually, we do not have to worry about any dropped messages. However, because the counter only tells which 'block' of the message is currently being sent, we can only send one message between each pair of processes at a time. Hence after a message is sent, we wait for an acknowledgement to be received from the receiver before sending a new message (since we need to now reset the counter before sending the next message).

Receiving a Message from a given process follows similar logic. Note that we know where the message is coming from, and we also know the counter of the message currently received. We take each 8-bit message and look at it's counter ID for this process and compare to our current counter ID, if it is the same, we append the 6-bit message (taking away the 2-bit which indicates which block: begin, middle, end it is) to an array. If the rec(m,c_i) is not the same as this process's counter ID, then it will offset the message to the correct array index like so:.

Let's assume that the first and third messages have been received thus far.
(1 0 1 0 1 1) (6-bit empty message: 0 0 0 0 0 0) (0 0 1 1 0 1)
.

We also assume that this is an extendable array and that if the message is longer, we can extend it to fit the new contents. Finally, if there's an end block received, we can then deduce the total size of the message (in bits) and we can allocate space.
Once the entire message has been received, the process resets it's counter ID for the given process $myCounterID_i$ (to a value say 0), then sends an acknowledgement to the process that the entire message has been received so that it can begin sending a new message.

Assumption First In First Out occurs for each channel, not between channels. Also, since this is an asynchronous system, if we still have not received an acknowledgement from the receiver process, we can move onto sending a message to a different process and come back to the process next time (using a loop) to check if an acknowledgement has been received. As stated earlier, we know that each message sent across the channel is guaranteed to be delivered eventually, hence in this way, we do not have to be hung up waiting for one process to respond before sending messages to another process.

Correctness We have showed that each primitive operation of Model B can be simulated in Model A. Hence Model A is at least as strong as Model B.

2 Is Model B at least as strong as Model A? Show your answer is correct.

In order to show this, we have to show that Model B simulates A, that is, all the primitive step of B can be simulated in A.

Primitive Steps that we must simulate are consistent for both Models, that is a send and receive step.

Sending an 8-bit message This step is trivial, since in model B we have no limit on the length of messages that can be sent, we simply choose to send an 8-bit message.

Messages may be reordered In Model B there is a FIFO system, so the order of messages is implicitly defined. In Model A we encountered a problem of messages being reordered and devised a scheme to deal with this. Here, on the receiver side we know the order of the messages sent and hence we can deal with them accordingly. Hence we do not have to worry about messages being reordered.

Receiving a message Is also trivial, when we receive a message from a specific channel, we know which process has sent it, since Model B is a FIFO system, we know the order in which they were sent. Each message received is an 8-bit message, conforming to the specifications of model A. An acklowedgement will also be sent to the sender with the counter ID indicating which message has been received. This will again be resent every $\triangle t$ so as to deal with dropped messages.

Guaranteeing a message delivery Here we note that in Model B if one process P sends the *same* message to another process Q infinitely many times, at least one copy of it will eventually be delivered to process Q. In order to guarantee message delivery, we first send a message across a specific channel (using the 8-bit message scheme). Since this model can drop messages and we want to ensure delivery for simulating Model A, we wait for the receiver to send an acknowledgement. Since the acknowledgement message may also fail, each process can send queries to another process after a certain time interval to see if there's a message waiting to be received. This prevents the two generals problem from occurring since any two processes which would like to send/receive will wait for an acknowledgement (if it is the sender) while the receiver will periodically send a query (which doubles as an acknowledgement). Further to ensure that a message is not lost between each query and send command, the sender and receiver will have a counter c_i for the corresponding process i. This will tell the receiver 'i am sending message 1'. And the receiver will reply 'i have received message 1' before sending the next message. Since each process can be both a sender and a receiver, we can have separate variables for keeping track of each of these.

Asynchronous Again, since this is an asynchronous system and it is unpredictable when the messages will be received, we do not want one process waiting for another process to respond before attempting to send (or receive) a message from another process. Hence, for each process, we loop through all of the other processes (excluding itself). First we check if there's any incoming message, if there is we increment our counter. Then we send an acknowledgement ack(c) with the counter of the message received, we do this regardless of whether the message was received or not (to compensate for dropped messages). Then we move onto sending a message to this process (if there is any to be sent), but before we do this, we ensure we have received an acknowledgement from the process that the last message has been received (if this is the first message, send it anyway). We then increment our counter for every message sent. We continue this for each process that we would like to send/receive messages from.

Correctness We have showed that each primitive operation of Model A can be simulated in Model B. Hence Model B is at least as strong as Model A.