The University of Queensland
School of Information Technology and Electrical Engineering

Semester Two, 2009

COMS3200/COMS7201 – Tutorial 2 - Solutions

Question 1

In a large computer network, there are three identical servers for a particular operation. How can the RPC model be adapted to handle this situation?

Multiple servers might register under the same name. Whenever a client requests the service, the name server (the process responsible for *binding* names to server processes) will either choose one server at random, or more intelligent management can be built which fairly distributes the load among the servers, e.g. sends the request to the server with the lightest processing load.

Question 2

Parameter passing in RPC is easiest if all the computers (and operating systems) are identical. Imagine, however, that RPC is being used among machines whose binary representations for integers, floating point numbers, and characters are all different. Suggest two strategies for handling this problem and discuss their strengths and weaknesses.

One way is to apply one-to-one mapping, i.e., each machine uses its own format and identifies the format in the messages (e.g. the receiver-makes-right approach). The conversion is carried out only if it is necessary, however it incurs *n(n-1)* conversions in the network which is truly heterogeneous - that is, there are *n* different communicating machines.

An alternative method is to adopt a network standard (canonical) form (e.g. ASN.1 or XDR) and have all parameters converted to the network standard form before being marshalled. This approach has a disadvantage that if two identical machines are communicating, the conversion is still carried out.

Question 3

An RPC might take a long time to complete. If it takes too long the client may time out and retransmit the request, leading to a variety of complicated situations. How can you reduce this problem?

Retransmission of a request may lead to

- Duplicate execution of the request by the server - which is a problem if the operation isn't idempotent
- Orphan procedure calls - i.e. the server continues performing an operation when the client has given up on it and is not expecting a reply
- Server Overloading - due to orphaned procedure calls - the server ends up performing the request multiple times - possibly making it less likely that one will complete before the client times-out again.

Possible ways to overcome this problem

- If an operation is going to take a long time the server may *notify* the client The server could contact the client *requesting* additional time and not continue until confirmation is received from the client.
- The server could recognise a retransmitted request and rather than just start performing the operation again, it could return the result of the already executing request (when it finishes) as the reply for the retransmitted request.

Question 4

Consider blocking and non-blocking send and receive message passing primitives. Which might you use to construct an RPC system with
(a) at most once semantics, and

(b) at least once semantics?
How? (You should assume the existence of timeout primitives also.)
What type of control information would be in the various messages passed?

The primitives will be much the same for (a) and (b) - it's just a matter of how you use them. At-most-once semantics means that the client knows that if it makes an RPC, the operation will occur at most once (i.e. 0 or 1 times). At-least-once semantics means that the operation will occur one or more times. In both cases the client (actually the stub) could send the request using either a blocking or non-blocking send. If a blocking-send is used, the client-stub will receive an acknowledgement (by function call return) that the request has been put in the outgoing queue (but not necessarily received by the server). For either type of send, the client-stub then does a blocking receive (with timeout) to await the reply from the server.

The server will almost always use a blocking receive to await incoming requests. Once a request is received it will process it then send the reply back to the client. Either a blocking or non-blocking send could be used for this process - and once the send is completed, a blocking receive is executed again.

The difference between the at-most-once and at-least-once semantics for the RPC come in how the client stub deals with a timeout (i.e. no reply received). (Normal operation will be that the client-stub receives the reply from the server and passes it back to the client.) When an error occurs, the client-stub will timeout instead of receiving a reply. For at-most-once semantics, the stub will pass an exception back to the client code - the RPC request has been executed 0 or 1 times and it is up to the client itself to deal with any necessary recovery.

With at-least-once semantics, if the client-stub receives a timeout, it will again send the request to the server and block waiting for a reply (or a timeout). The client-stub will do this until it gets a reply from the server and will then pass this back to the client. (The stub has guaranteed that the execution has happened at least once.)

Often RPC messages contain control information such as

- transaction identifier - this may prevent a server executing the same request more than once
- server identifier - indicates which server the request is being made to
- procedure identifier - indicates the procedure that is being called
- version number - which version of the procedure
- formatting info - if a standard format isn't being used to transmit the data (arguments or reply) then the message may specify which format is being used.

Question 5

(PD5.40) Suppose we were to implement remote file system mounting using an unreliable RPC protocol that offers zero-or-more semantics. If a message reply is received this improves to at-least-once semantics. We define read() to return the specified Nth block, rather than the next block in sequence; this way reading once is the same as reading twice and at-least-once semantics is thus the same as exactly once.
(a) For what other file system operations is there no difference between at-least-once and exactly-once semantics? Consider open(), create(), write(), seek(), opendir(), readdir(), mkdir(), delete() or unlink(), and rmdir().

No difference between at-least-once and exactly-once semantics means that the operation is idempotent - i.e. it doesn't matter how many times the operation is performed, the result is the same.

open() and opendir() - are idempotent, provided we're not trying to lock the file/directory for exclusive access
write() - is idempotent if we're writing a particular block (e.g. N) rather than the "next block"
seek() - is idempotent - provided we're seeking to an absolute position in the file, rather than a position relative to the current position
readdir() - is idempotent, provided it's defined as per read() above (i.e. read Nth block)

(b) For the remaining operations, which can have their semantics altered to achieve the equivalence of at-least-once and exactly-once? What file system operations are irreconcilable with at-least-once-semantics?

create() is idempotent if it means “create if nonexistant, or open if it exists already”
mkdir() is idempotent if the semantics are “create the given directory if it does not exist, otherwise do nothing”
delete() (or unlink()) and rmdir() can be made idempotent if the meaning is “delete if the object is there, otherwise ignore”

Operations fundamentally incompatible with at-least-once semantics include exclusive open()/ create() - or any other form of file locking.

(c) Suppose the semantics of the rmdir() system call are now that the given directory is removed if it exists, and nothing is done otherwise. How could you write a program to delete directories that distinguishes between these two cases?

The directory-removing program would first check if the directory exists. If it does not, it would report its absence. If it does exist, it invokes the system call rmdir().

Question 6

The figure below shows the communication which takes place. There are three types of processes, each with a single communication endpoint, so we need to decide on communication primitive for each endpoint for each process.

A. Let’s assume that we only use send and receive primitives

Looking at each process in turn:

ATM

The ATM sends a request to a bank and waits for the reply before taking any action. The sending primitive can be either a blocking or non-blocking send and the receiving primitive should be a blocking receive (i.e. there is nothing to do whilst waiting for the message).

Bank

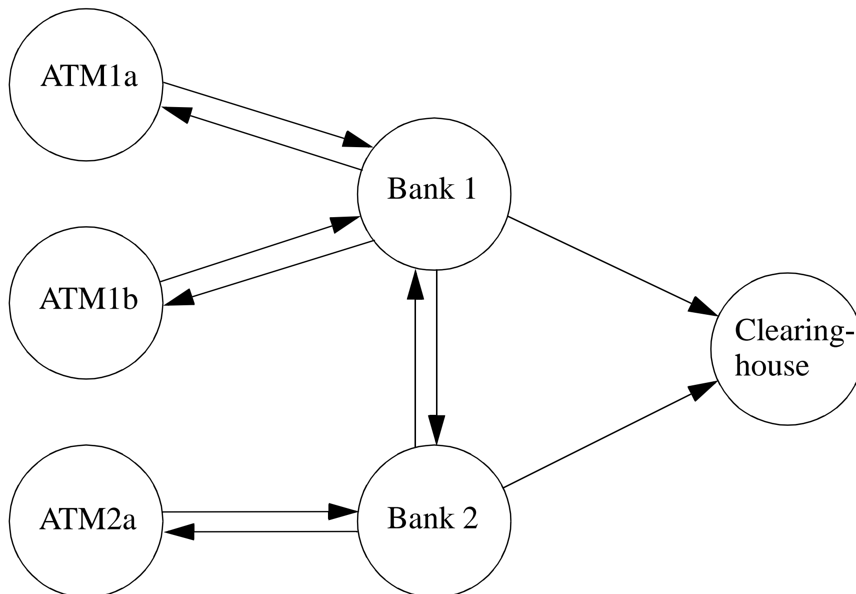
The question isn’t clear on whether the bank process has processing to do other than responding to incoming messages. If other processing is required then only non-blocking primitives should be used (for both sending and receiving). If no other processing is required, i.e., the bank just deals with incoming messages (from its ATMs or other banks) then a blocking receive should be used. Because the bank communicates with several other processes, it is possible that a message may arrive at any time. To enable quicker response, non-blocking sends should be used (i.e. the bank process may then deal with an incoming message possibly before an outgoing message has been transmitted by the operating system).

Clearinghouse

The clearinghouse doesn’t send any messages, so we only need to consider the receive primitive. If we assume that the clearinghouse only performs processing in response to messages from the banks then a blocking receive is appropriate. If additional processing is required then a non-blocking receive would be more appropriate.

B. Let’s now assume that we have not only send and receive but also RPC.

You should notice that ATM behaves like a typical client, i.e. sends a request and waits for a reply. It should therefore use RPC Call. There is no change for Bank and Clearinghouse.



Something to keep in mind ...
Because the banks have only a single communication endpoint, the stream of messages received from that endpoint will contain both messages from ATMs and messages from other banks. It will be necessary to design the message formats so that the bank can determine the type of message it has received.

Question 7

(PD7.7) Give the big-endian and little-endian representations for the the integers (a) 101 (b) 10,120 (c) 16,909,060. Assume a 32-bit representation.

Assume the left hand byte is transmitted/stored first. Big-endian is most significant byte first. Little-endian is least significant byte first. Note that we don't change the order of the bits within each byte, just the order of the bytes themselves.

(a) 101

Big-endian:	00000000	00000000	00000000	01100101
Little-endian:	01100101	00000000	00000000	00000000

(b) 10,120

Big-endian:	00000000	00000000	00100111	10001000
Little-endian:	10001000	00100111	00000000	00000000

(c) 16,909,060

Big-endian:	00000001	00000010	00000011	00000100
Little-endian:	00000100	00000011	00000010	00000001