**CMPSCI 677 Distributed Operating Systems** **Spring 2003**

# Homework 2 Solutions: March 12

*Lecturer: Prof. Prashant Shenoy* *TA: Gary Holness*

## 2.1 Question AST 2.6

This will not work. The notion of the *first* byte of the first word is precisely the problem. Because big-endian and little-endian machines store the first byte differently, even though the first byte sent over the wire is the first byte received, the receiving host does not know the endian-ness of the sending host. This could also be solved through clever protocols.

## 2.2 Question AST 2.7

This approach is the same as a traditional RPC (i.e. the call semantics are the same). There are differences in how the underlying mechanism works. In particular, using 2 asynchronous RPCs gives us the ACKs. This in effect makes it more reliable and increases protocol overhead. The approach using two one-way RPCs is the same if we assume reliable communication is available. As this is not the case, the client can wait for a reply from a server that never received the clients request. These call semantics are not the same as a traditional RPC.

## 2.3 Question AST 2.10

(Technically, RMI already implements exception handling). Anyway, when the client's call arrives at the server, the server executes a local method which raises an exception. This exception can be caught by the skeleton and information about it can be marshalled and sent back across the wire to the client stub. The stub, once receiving this information, unpacks it, constructs the appropriate exception and throws it.

## 2.4 Question AST 2.15

The client first send a request by issuing an asynchronous RPC call to the server. The client, then loops (or polls) calling the asynchronous receive to check if the server's response is available. Once the client has received the server's response, it is free to proceed.

## 2.5 Question AST 2.17

(partly from answer from Peter Amstutz) Persistent asynchronous communication systems (such as email) are generally use synchronous communication such as store and forward as a means of moving data. These

transfers could be implemented using RPCs. Persistent asyncrhonous communicatoin is more reliable compared to transient communications because information can be stored in case the target server is unavailable. The RPC mechanism can be used to move information, so yes it makes sense.

## 2.6 Question AST 2.22

(partly from answer of Ting Yang) As the number of clients requesting service increases, the server will become very busy scheduling and processing these requests, thus the response time of the clients cannot be garanteed. The more clients connecting, the longer response time will be as only after receiveing the server's response can a client proceed. The server is a process that runs on a host across the network. Hosts have finite resources (processor, memory and file descriptor). Eventually, a host will run out of available file descriptors for the socket connections needed to accept incoming client requests. Even in a multi-threaded approach, there is a finite upper bound on the number of threads a server (or any process) can create. A solution is to set a fixed upper bound such that the server will service a maximum number of clients (without running into machine limits). Aditionally, a pool of threads for handling requests can be pre-allocated for handling client requests. Think of an M/M/k queueing model where k is the number of threads.

## 2.7 Question AST 3.1

Let $p_{hit} = \frac{2}{3}$ be the probability of a cache hit and $p_{miss} = \frac{1}{3}$ be the probability of a chache miss. Let $T_{proc} = 15ms$ be the time to get request, dispatch and process. Let $T_{sleep} = 75ms$ be the time a thread sleeps blocking for disk I/O. In the single threaded case, we get a cache hit, the cost of processing is $C_{hit} = T_{proc}$. If we get a cache miss, the cost of processing is $C_{miss} = T_{sleep} + T_{proc}$. Since we only have a single thread, a request takes time $T_{total} = p_{hit}C_{hit} + p_{miss}C_{miss}$ so $T_{total} = \frac{2}{3}15ms + \frac{1}{3}(15ms + 75ms) = 40ms$. This means $1000\frac{ms}{sec} \times \frac{1}{40\frac{ms}{request}} = 25\frac{request}{sec}$.

In the multi-threaded case, note that when a thread sleeps, another thread can execute. So, the time that each thread is actually doing work is $15ms$. Thus, the server can perform $1000\frac{ms}{sec} \times \frac{1}{15\frac{ms}{request}} = 66.66\frac{request}{sec}$.

## 2.8 Question AST 3.2

Because the physical host on which the server process runs has finite bounds on resources such as memory, thread discriptors, stack space, etc, yes it makes sense to limit the number of threads in a server. Without such limits, a server process could keep allocating threads until it reaches machine limits (which would be very bad). In the case of kernel level threads, this is made worse because a large number of threads vying for the scheduler will degrade system performance.

## 2.9 Question AST 3.5

A lightweight process (LWP) is a mechanism by which we may have one-to-many, many-to-many, or one-to-one binding between user level and kernel level threads. By having only a single LWP only a single kernel thread can be associated with user-level threads. If a user-level thread executes a blocking call, the kernel sees the process in which the LWP resides as being blocked. By allowing more than one LWP in a process, if the kernel sees a single LWP as blocked, it can context switch to another LWP thus giving other threads

in the process a chance to run. By limiting processes to have only a single LWP, it effectively is no different from running a normal process.

## 2.10 Extra Problem 1: Leaky Bucket Policer

We would have two bucket poliecers, $P_1$ and $P_2$ with parameters $b_1$, $b_2$, $r_1$, and $r_2$. We would have $b_1 = b$, $r_1 = r$, $b_2 = 0$, and $r_2 = p$. Since we want to limit the peak rate, we do not want to allow any rate beyond $p$, thus $P_2$ has $b_2$ set to 0.

## 2.11 Extra Problem 2: Packet flow

The packet flow conforms to a leaky bucket specification (r,b) and arrives at the leaky bucket at rate less than $rt + b$ in every interval of length $t$ for all t. If the placket flow is bursty and we get a burst of size $b$ at a point when the bucket is not full, then packets will have to wait.