Vladimir Urazov
CS 4210

Homework 3

1. One assumption I make here is that the URPC request takes up one time cycle. That is, the time cycle is consumed by the client making the URPC request, and the server starts handling the request on the next cycle. Similarly, when the server is done with the request, the client thread resumes execution on the next cycle. Whereas donating the processor and returning it takes five cycles. Also, I assume that for the client threads, the scheduling is FCFS.

Time	P1	P2	P3	P4
0	T1	T2	FS-idle	GS-idle
1	T1	T2	FS-idle	GS-idle
2	T1-URPC-FS	T2	FS-idle	GS-idle
3	T3	T2	FS-T1	GS-idle
4	T3	T2	FS-T1	GS-idle
5	T3	T2-URPC-GS	FS-T1	GS-idle
6	T3	Idle	FS-T1	GS-T2
7	T3	Idle	FS-T1	GS-T2
8	T3	Idle	FS-T1	GS-T2
9	T3	Idle	FS-T1	GS-T2
10	T3	Idle	FS-T1	GS-T2
11	T3	T2-URPC-FS	FS-T1	GS-idle
12	T3	Donate to FS-T2	FS-T1	GS-idle
13	T3	Donate to FS-T2	FS-idle	GS-idle
14	T3	Donate to FS-T2	FS-idle	GS-idle
15	T3	Donate to FS-T2	FS-idle	GS-idle
16	T3	Donate to FS-T2	FS-idle	GS-idle
17	T3	FS-T2	FS-idle	GS-idle
18	T3	FS-T2	FS-idle	GS-idle
19	T3	FS-T2	FS-idle	GS-idle
20	T3	FS-T2	FS-idle	GS-idle
21	T3	FS-T2	FS-idle	GS-idle
22	T3	FS-T2	FS-idle	GS-idle
23	T3	FS-T2	FS-idle	GS-idle
24	T3	FS-T2	FS-idle	GS-idle
25	T3	FS-T2	FS-idle	GS-idle
26	T3	FS-T2	FS-idle	GS-idle
27	T3	Return to T1	FS-idle	GS-idle
28	T3	Return to T1	FS-idle	GS-idle
29	T3	Return to T1	FS-idle	GS-idle
30	T3	Return to T1	FS-idle	GS-idle
31	T3	Return to T1	FS-idle	GS-idle
32	T3	T1	FS-idle	GS-idle
33	T3	T1	FS-idle	GS-idle

2. In the design and implementation of the RPC system in the Birrell and Nelson paper, the authors make several assumptions about the way RPC calls are made, which makes the implementation simpler, and also more efficient.

   For instance, the server assumes the client will not issue a new call until the results of an existing call are received, therefore once the server returns the results of a call to the client, if it receives a new call, that is sufficient acknowledgement of the receipt of the results by the client, and the server will not expect an acknowledgement packet. In a system where the calls are asynchronous, this is not going to be the case, as the client (an activity, in the authors' terms) may issue a new call before the results of a previous one are received.

   For the purposes of detecting duplicate calls, each call has a call ID number associated with it. In the paper's implementation, these numbers are required to be unique and increasing from one call to the next. For each activity contacting the server, the server keeps only the ID of the latest call it received from that activity, and if it receives another one with the ID less than or equal to the one it already has, it knows the call is a duplicate and discards it. This is fine when there is only one outstanding call per activity, but with asynchronous calls that might not be the case, and can result in a mix-up. The paper indicates that for performance reasons it omits some of the checks of the lower-level transport protocols, so if we are using something like UDP which does not guarantee that the order of the packets on the network is preserved, it is entirely possible that the call packets will be mixed up and one with a larger ID number might arrive before the one with the smaller number. Therefore, to support asynchronous calls, the server would need to keep all the ID numbers received to reliably discard duplicate calls, and not only the largest one received so far.

   In addition to the call ID, each packet contains the source and destination process IDs. This helps the stub to dispatch the packet more efficiently to the target process. In the original implementation, when the calling activity initiates a new call, it just uses as its destination the ID of the process that handled the previous call from this activity. However, this is not actually going to work for asynchronous calls, as a new call may be initiated while the server process is still handling the previous call from the activity and is unavailable for another request. Hence, this optimization does not apply any more.

   Another concern is what happens between the time the call packet is sent out and when the result is received on the client. In the original implementation, the RPC stub just blocked and waited for the result to arrive. With asynchronous calls, this should not be the case. The stub must return pretty much immediately. Then, there is need for another thread to wait for the receipt of the result. Also, one would need to maintain a buffer of results in the stub which can be polled to collect the function call results. (Alternatively, one could implement it with some sort of call-back mechanism, where a user-specified handler is called when the results are received.) Therefore, the client stub would have to export an interface not only to make a remote procedure call, but also to poll for the results to see if they have arrived yet. This would require exposing the ID number of the call to the client, so that when the poll function is called, the stub knows which invocation of a particular function we are

talking about. Of course, this also means that the client stub must maintain some of its own threads to handle the calls while the client thread that actually made the RPC is working on something else.

A sub-problem of what goes on in the client while waiting for the results of a procedure call is the periodic probing of the server. In the original implementation, while waiting for the results of a call, the client stub periodically sends probe packets to the server expecting acknowledgements, to ensure that the server didn't crash while handling the call. With asynchronous calls, depending on how the threads that wait for the result are implemented, one might need to have a separate thread that concerns itself with polling the server (say, keeping a priority queue of pending requests, sorted by the time the next probe needs to be sent out to the server).

One more problem that needs to be solved by the client for asynchronous calls is error handling. In the paper's implementation, upon receiving notification of an error, the client stub just raises an exception and, depending on what the client does with the exception, sends a notification to the server to either unwind its call stack terminating the call or to continue execution. One problem we face with asynchronous calls is we can't just raise an exception, because the caller client code isn't any more in the same call stack as the code that receives the result from the server. Therefore, we need to incorporate exception handling in the polling mechanism in the client stub. But then another issue is, for normal execution, the stub will just send a result acknowledgement upon receipt of the result back to the server, however, it can't do so upon receipt of the exception, as it doesn't know what the client wants to do with it. So, the server is left waiting for the response for the client telling it how to handle the exception, but what if the client never actually polls the results of the function (possibly due to a crash), and never responds. Then the server needs to have a mechanism similar to the waiting for the result on the client side, to wait for the response on the handling of the exception from the client.