Remote Procedure Calls

I. Architectural Considerations

Several contributions:

- Presentation layer is very expensive, but is usually ignored (30x transport!)
- Can only handle loss/reordering if the units makes sense to the application => ADUs
- Several loops touch the data if you follow the layering: better to have fewer loops
   (integrated layer processing)
- In general, need more flexibility in protocols, and therefore must follow the
  applications needs (framing, reordering, loss handling, presentation)

II. Lightweight Remote Procedure Call

Focus of this paper: light-weight cross-domain procedure calls -- an optimization to traditional cross-machine RPC

History: RPC developed by Birrell and Nelson starting around 1983. It was found to be very useful as a
way to build distributed systems. Here we cover RPC first and then LRPC:

Traditional RPC: think of it as carrying an abstract thread to a remote machine and back. On the client
(caller) and server (callee) there are concrete local threads that the abstract thread maps onto...

Goal: transparency => it should look and work just like a local procedure call. (it turns out this is not possible
and near transparency is a bad idea, as it misleads users into thinking they have 100% transparency; Java’s RMI
has this problem in spades)

Key idea: generate local stubs automatically given the procedure interface:

- client stub receives local procedure call from client and packages call for network
  transmission (marshalling)
- server stub receives network requests and issues local procedure call to server

Steps:

Client: calls stub
CS=Client Stub: alloc buffer
      marshall args
      call kernel to send
Kernel: copy message to kernel space
RPC

xmit

--- switch to server side ---

Kernel:   receive message
copy to server address space
dispatch/schedule server
SS=Server Stub: unmarshall message
call server procedure
Server     execute call
SS         marshall result
call kernel to send
Kernel:   copy to kernal space
xmit

--- switch to client side ---

Kernel   receive message
copy to client address space
dispatch/schedule client
CS:      unmarshall result
return result

Notes:

  o there are 8 copies: 2 marshall, 2 unmarshalls, 2 copies into kernel space, and 2 copies
    out of kernel space
  o marshalling is an expensive kind of copying, since you need to understand the data
  o there are 4 context switches, 2 on the client, 2 on the server

Where do the stubs come from?

  o typically: servers register interfaces with a name server: (name, interface) pairs
  o clients looks up name and gets back the client stub (directly or via a interface
    conversion step)
  o setting up the client stub is called “binding”

Binding:

  o goal: set up and optimize the client stub
  o bind-time work should reduce the cost of RPCs, see LRPC below
  o this is an example of “bind and fast access”, as seen in active messages and later in
    exokernel

There are lots of subtle issues, some of which were hinted at above... in no particular order:

Who checks the arguments to the server?  it has to be the server stub, since the client stub can’t be trusted
RPC

Within one machine: Why copy immutable args to the server side? If not, the client could modify them, which would not only violate their immutability, but also gets around the arg checking done by the server stub. This is because we don’t trust the client or the client stub (since the client could have modified it).

Why not copy mutable args? This is a subtle point, and the paper does not make it clear; the authors may not understand the issue. The reason is because you don’t have to: the client has the right to modify the args, so the server already has to deal with this (for callers in its own address space). Not doing the copy improves performance, although with some calling conventions you will always have to do a copy anyway.

BUT: mutable args aren’t really mutable!!! If the client modifies the arg after it is passed, it HAS NO EFFECT on the copy the server is using. Thus the calling convention is no longer “pass by reference”, in fact, it is not clear that if the server modifies the mutable arg that those modifications ever get back to the server. In theory, the client stub could copy mutable arguments from the AS back to the client’s version on return. This would be “pass by value-copy”, which means copy the arg by value to the callee and copy the modified version back; but it is definitely NOT pass by reference.

Would we want pass-by-reference for RPC? Probably not, since we don’t trust the client -- pass by value-copy is much safer, since the client can’t muck the the arg while the server is using it.

Key Point: RPC and variations cannot be transparent, due to these pass-by-reference issues, and more generally, due to the lack of trust. My problem with RPC work to date is that they pretend it is transparent when it is NOT, thus misleading the programmer and causing some terrible bugs.... for example, pass-by-reference args may work differently in the RPC, IPC, and local case for the same procedure. There may be two copies of a shared data structure, in which case one set of modifications may get lost. Careful locking could avoid this, but only if the programmer realizes that there are two copies during the RPC!! (which means it can’t be transparent)

Other reasons for non-transparency: new exceptions have to be handled in the IPC/RPC case. These are fundamentally due to the potential for partial failure; in contrast if a callee in your address space fails, then you’re dead too and don’t care. The need to deal with remote failures (or those in another domain) ensures that RPC can’t be transparent. In practice, if you look at the code around an RPC (e.g. RMI), it doesn’t even look like a local procedure call, which implies it is not very transparent.

What if a client dies? LRPC thread should continue to completion in the server, so as to not screw up its locks or invariants

What if a server dies? must return a “call failed” to the client, which is one of the non-transparent exceptions that client must now handle

What if the middle server dies in the client -> A -> B calling sequence? B will detect A is gone on return, but it can pop A’s LR off the stack and return “call failed” directly to the original client (even though B’s call succeeded, A’s clearly did not)

If we pass a pointer off node, how long is it good for? We can’t free or move that storage if we may later get a remote use of it... Ninja solution: all off-node references must include a lease, which expires in a finite time, at which point the storage can be deallocated. Another solution: don’t allow references to go off node at all, make everything pass by value (or pass by value-copy).

Lessons:

- RPC is a good way to write distributed systems
RPC

- It is **not** transparent and can’t be.
- Pass-by-value-copy is a safer alternative to pass-by-reference, so we should switch to it explicitly (rather than pretending to be pass by reference).
- Make RPCs look different than local calls, so that the programmer won’t think they are the same.
- All we really want to be transparent is the marshalling and the matching of the reply to the caller.