DISTRIBUTED SYSTEMS COMMUNICATION

Last class we discussed about the core challenges of building distributed systems (incremental scalability is hard, at scale failures are inevitable, constant attacks, etc.). We've said that the core approach of building distributed systems to address these challenges is to construct layers upon layers of systems and services that raise the level of abstraction increasingly through well-defined APIs. Google's storage stack, as well as Spark and other systems, are perfect examples of that layered design for distributed systems.

In this lecture, we look at one of these abstractions: RPC, which enables communication. To cooperate, machines need to first communicate. How should they do that? RPC is the predominant way of communicating in a DS. Interestingly enough, RPC – the simplest possible DS abstraction – reflects most of the challenges in distributed systems, showing you how fundamental those things are.

I. Remote Procedure Calls (RPCs)

Overall goal: To simplify the programmer's job when building distributed systems.

- many complexities when building a distributed program
- some are fundamental and must be dealt with by programmer, while others are mechanical and can be solved by good language, runtime support
- pre-RPC, all of the gory details of socket-level communication were exposed to the programmer
- post-RPC, a remote interaction looks (nearly) identical to a local procedure call, with the system hiding (nearly) all of the differences

Example of (poorly written!) socket communication code:

```
struct foomsg {
  u_int32_t len;
}
send_foo(int outsock, char* contents) {
  int msglen = sizeof(struct foomsg) + strlen(contents);
  char* buf = malloc(msglen);
  struct foomsg* fm = (struct foomsg*)buf;
  fm->len = htonl(strlen(contents));
  memcpy(buf + sizeof(struct foomsg), contents, strlen(contents));
  write(outsock, buf, msglen);
}
```

Any issues with it?
Many…
- lots of ugly boiler plate
- bug prone
- portability details are easy to miss
- hard to understand and hard to maintain
- hard to evolve protocol
- vulnerability prone

…

RPC: RPC tries to make net communication look just like a **local procedure call** (LPC):

```
Client-side code:
   z = fn(x, y)
```

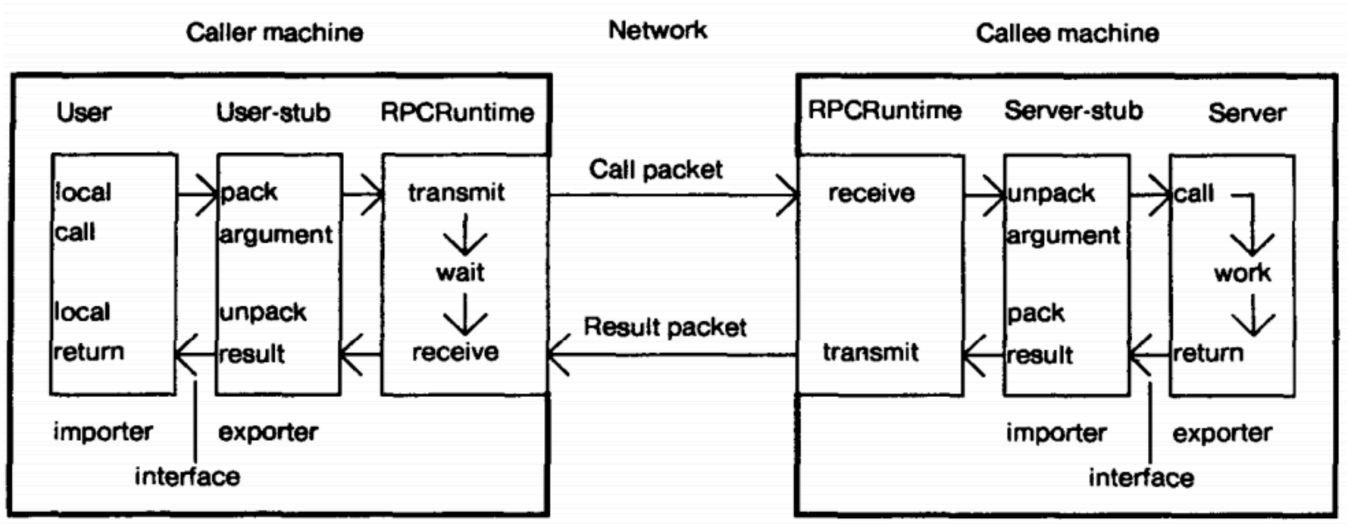
```
Server-side code:
   fn(x, y) {
      // compute result z
      return z;
   }
```

Why the LPC abstraction? Because, it's simple, elegant, and even novice programmers know how to use function calls! Other models exist, but they are (arguably) harder to think about.

RPC message diagram:

Client Server
request---> process
<---response

RPC software structure:



(figure taken from RPC paper by Birrell, et.al.)

In implementations, stubs are generated automatically by RPC frameworks (libraries), which also provide the RPCRuntime. The programmer only writes definitions for their data structures and protocols in an "interface definition language" (IDL). We'll see two examples of RPC libraries later.

Good things about RPC abstraction:

- Hides gory network/marshaling details that one would have to implement if doing, e.g., network-level communication, byte orders (big endian vs. little endian)
- Supports evolution of the communicating components independently. (Many RPC frameworks provide explicit support for evolution, e.g., protocol version numbers, rules of when to remove/rename arguments to procedure calls, etc.)
- Allows for efficient packaging
- Authentication support
- Location independence (particularly if combined with a directory service)

Problems the RPC abstraction (i.e., where distribution of RPC peeks through the LPC illusion):

- **Latency**: LPC is fast, RPC can be really expensive (esp. if it goes over WAN, but not only)

- **Pointer transfers**: local address space isn't shared with remote process. The programmer/RPC library has to make a decision of whether to follow pointers and serialize in depth, or to exclude that information. Typically, IDLs avoid this problem by requiring one to specify one's messages/data structures to avoid pointer confusion.
- **Failures**: they are more fundamental than in the single-process/LPC case.
    - If I issue an LPC, then I can be in one of three situations: (1) the LPC returns, therefore I know the procedure has been executed; or (2) the LPC doesn't return, in which case I know the LPC hasn't finished executing (it may still be executing, or the process that hosts both the caller and the callee might have died). So, with LPC, the caller and callee can't have a "split mind." With RPC, they can.
    - If I issue an RPC and get a return, I know the procedure has been executed remotely. But if I don't get a return, what do I know? Has the function completed on the remote side and I just don't know about it because, e.g., the response has been lost over the network? Has it not happened, because, e.g., my request has been lost over the network or the remote machine is dead? Or maybe the machine/network is slow – should I wait longer for the response? How long should I wait for a response? I can't tell the difference between these situations, and I don't know how to act. In certain scenarios, that's not a big problem, but in others it is.

  For example, imagine the distributed system is composed of an ATM machine and its bank back-end system. A person requests a withdrawal of cash from the ATM machine. The ATM machine needs to check that the person has sufficient funds in, and withdraw the sum from, his bank account before giving out the money. The ATM requests the money transfer and then waits, and waits, and waits. No response. Should it give out the money or should it not, if it doesn't get a response from the bank? No simple answer. Need some timeout (person can't wait forever), but what should that be? Say ATM refuses to give out the money upon a timeout. But what if the bank already deducted the money from the account, how should we deal with that? The ATM needs to keep trying to talk with the bank to tell it that it hasn't actually finished the transaction, so the bank can put back the money in the person's account. But what if the ATM itself fails at some point? Is the person's money lost forever? I would hate interacting with such an ATM.

  Alternatively, say ATM gives out the money upon a timeout. This of course can open the bank to fraud: a person might quickly go to another ATM and extract the same money again.

Herein lies the biggest difficulty in distributed systems compared to single-process systems: with single machines, you largely have shared fate (i.e., either all fails or nothing does); with distributed systems, machines/processes **can fail independently**, so you can have some machines that are up while others are down, and you can also have all machines up but unable to communicate due to network failures and partitions. This leads to coordination challenges, which in turn lead to inconsistencies between the decisions these machines make. So that goal of DS of providing a "coherent service" is difficult to achieve.

II. Semantics

The preceding ATM example leads directly to a question of semantics. What should be the meaning (semantic) of an RPC invocation? Network failures imply timeouts and retransmissions. They imply that clients will end up retransmitting requests, and the server will see duplicates. How should the server behave when it sees duplicates? Below are some potential answers, and different RPC frameworks will implement different semantics.

At-least-once: the RPC call, once issued by the client, is executed eventually at least once, but possibly multiple times. This semantic is the easiest to implement but raises issues. To implement, the RPC library on the client keeps issuing the RPC multiple times until it gets a response from the server. Assuming that failures (of network, server) are temporary, the RPC will be executed at least once. This is fine for *idempotent requests* (e.g., this email in a user's inbox has been read), but might be problematic for other types of requests (e.g., ATM example, purchasing an item).

At-most-once: the RPC call, once issued by a client, gets executed zero or one time. This semantic is next easiest to implement, but has complications if you try to do it sensibly. To implement, the server has to remember requests that it has already seen and processed, to detect duplicates to squelch. Client/servers need to embed unique IDs (XIDs) in messages. Something like follows:

```
Server maintains s[xid] (state), r[xid] (return value)
 if s[xid] == DONE:
       return r[xid]
 x = handler(args)
 s[xid] = DONE
 r[xid] = x
 return x
```

One complication: what happens if server crashes/restarts? Server will lose the s,r tables. Solution: store on disk [slow, expensive].

Another complication: even if store on disk, what happens if server crashes just before or after call to handler()?

- after restart, server can't tell if handler() was called
- server must reply with "CRASH" when client resends request

Solution: server has "server nonce" that identifies restart generation

- client obtains nonce during bind with server
- client transmits nonce in every RPC request
- if nonce doesn't match server's state, return CRASH

Exactly once: this is the ideal (it resembles the LPC model most closely and it's easiest to understand), but it's surprisingly hard to implement. This is when the RPC, once issued by the client, is invoked exactly once by the server. Why hard to build?

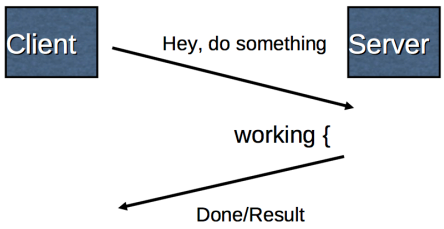
- server failure at an inopportune time can stymie -- client blocks forever
- need to implement handler() as a transaction that includes s[xid], r[xid] commit on server

III. Example Libraries

Some sample code using various RPC libraries can be found at https://columbia.github.io/ds1-class/lectures/02-rpc-handout.pdf.

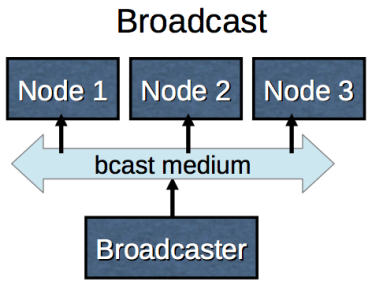
IV. Other Communication Models

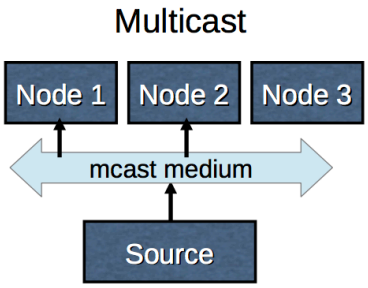
RPC is just one communication model in distributed systems. It works well when the communication pattern looks like this:

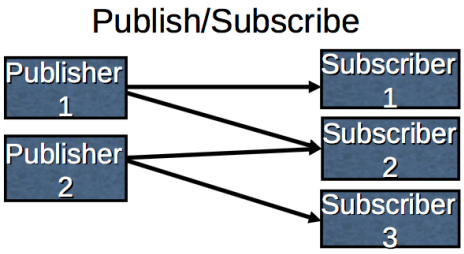


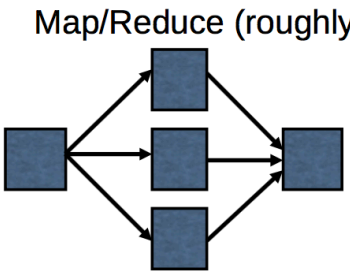
- point-to-point, request/process/response to a server or service.
- lots of communication in distributed systems look like that, but not all.

Here are some other popular communication patterns and the corresponding communication abstractions that support them. We won't study them in depth in this course.









Acknowledgements

The preceding notes were adapted from the University of Washington distributed systems course, Steve Gribble's edition:
http://courses.cs.washington.edu/courses/csep552/13sp/lectures/1/rpc.txt