Lecture 7: RPC Implementation Issues

7.0 Main Points

Stub generators
Binding
Object-oriented RPC
RPC on TCP
General's paradox
Transactional RPC

7.1 Stub Generators

```
Client stub:
  build message
  send message
  wait for response
  unpack reply
  return result

Server stub:
Create N threads to wait for work to do
  loop:
    wait for command
    decode and unpack request parameters
    call procedure
    build reply message with results
    send reply
```

Possible to generate stubs automatically!

Need signature of every procedure in the interface -- types of arguments (including lengths of arrays!), return value.

What if parameter is a pointer to a complex data structure (e.g., a tree?)

call by value -- make a copy of the data structure, put it in the message
marshalling

call by opaque reference -- pass the pointer!
can't be dereferenced on destination, but can be used in another call

```
Add(queue, rutabaga);     // remote queue
rutabaga = Remove(queue);
//remote queue doesn't care what's in
```

```
// rutabaga
```

7.2 Binding

With normal procedure call, resolve all call targets at link time.

What about RPC -- how does client know which destination port to send to?
static -- fixed at compile time (C)
dynamic -- fixed at runtime (Lisp, RPC)

Most RPC systems use dynamic binding via name service.

Name service translates service name -> destination IP address, port #
Why runtime binding?
Access control -- check who is permitted to access service
Fail-over -- if server fails, use another
Dynamic load balancing among server replicas -- use fastest replica

7.3 Object-oriented RPC

How do RPC's work in an object-oriented environment?

```
queue.Add(rutabaga);
```

What if queue is remote?

One solution:
Every object has a local **stub** object , to translate local object invocations into RPCs to remote object

If object is passed by reference, translate
pointer to local stub object -> pointer to stub object on the remote machine

Another solution:
Every pointer is globally valid (*requires long pointers -- 64 or 128 bits*)

Ex: pointer = machine # | address on that machine

If dereference pointer local to current machine, mask off machine#.

If dereference pointer on remote machine, invoke via RPC

Fundamental tradeoff:
function-shipping (move code to data) -- RPC
data shipping (move data to code) -- migration, caching and replication

What would it take to allow objects to migrate?

Tradeoff depends on size of object, amount of computation, CPU and memory demand on each machine, network bandwidth and load, ...

7.4 RPC Transparency -- Benefits and Limitations

Benefits: make procedure calls in the same way whether caller and callee are
in same address space
in different address space on same machine
on different machines

Location transparent -- service can be local or remote
For example in the X windowing system
Another example: Desktop publishing
Need editor, spreadsheet, document formatter, database, etc.

Now bundled, but might be composed of separable pieces.

Principle behind Microsoft OLE -- allow you to mix and match applications, which cooperate via defined RPC interface.

Problems with RPC
RPC provides location transparency, except:

1. Different failure modes in distributed system than on single machine.

Several kinds of failures:
-- user-level bug causes address space to crash

-- machine failure, kernel bug causes all AS on same machine to fail

-- earthquake causes all machines to fail

Before: whole system would crash. Now: one machine can crash, while others stay up. If file server goes down, what do the other machines do?

2. Cost of a procedure call << same-machine RPC << network RPC

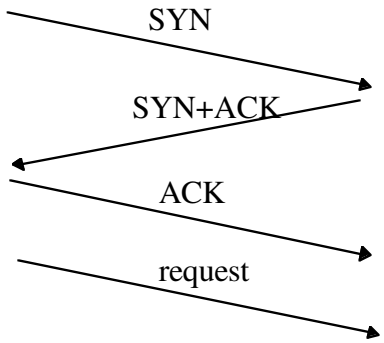
Means programmers must be aware that RPC is cheap, but it's not free.

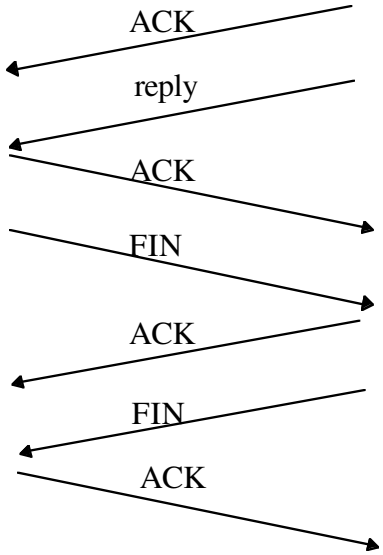
Caching can help performance, but that can make failure handling more complex.

These two issues are what make programming a distributed system hard.

7.5 RPC on TCP

How do RPC's map onto TCP?





How do we reduce the number of messages?

1. Delayed acknowledgment (part of most TCP implementations)

Receiver waits for 200 milliseconds, until either another packet arrives, or data to be sent. Allows single reply+ACK message.

2. UDP -- optimally, RPC only needs two messages, one for request, one for reply (assuming both fit in a single packet). Reply serves as ack.

Most local area RPC systems use UDP, with the RPC system responsible for retransmissions, timeouts, etc. Sequence #'s, buffers, etc. set up during binding.

Most wide area RPC systems use TCP, because of need for congestion control in using wide area resources.

3. Persistent HTTP (proposed standard)

Don't close connection after every web request; keep it open in case need to fetch another part of the web page.

4. T/TCP -- "transactional" TCP (proposed standard)

For repeated connections between same pair of hosts:

a. Use three way handshake to initialize sequence #'s

b. After init, each request = SYN, data, FIN
Each reply = SYN, reply, FIN

c. if client crashes, restarts using three way handshake

if server crashes, client won't know, will still send SYN, data, FIN.
Server replies with SYN+ACK, signalling client to resend data in separate message.

7.6 Failure Models

How many times is an RPC done?

Exactly once?

How can client tell difference between

server crashes before request arrives?
vs.
server crashes after ack, but before reply is sent?
vs.
server crashes after reply, but reply is dropped?

At most once?

Client sends off request; if it gets a reply, knows RPC was done
If no reply, can retry.
But if server crash, can't know if request was done or not.

At least once?

Client keeps retrying until server replies. If server crashes, RPC might be done twice.
Requires interface to be designed with idempotent operations.

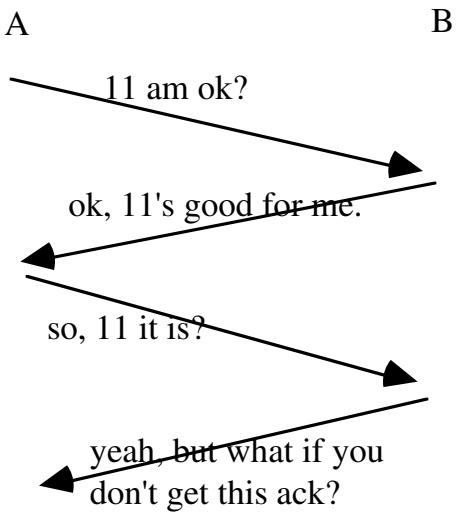
7.7 General's Paradox

Can I use messages and retries over an unreliable network to synchronize two machines so that they are guaranteed to do some operation at the same time?

Remarkably: no, even if all messages get through.

General's paradox: two generals, on separate mountains. Can only communicate via messengers; the messengers can be captured.

Need to coordinate the attack; if they attack at different times, then they all die. If they attack at the same time, they win.



.....

Even if all messages are delivered, can't coordinate! Can't simultaneously get two generals or two machines to agree to do something at the same time.

No solution to this -- one of the few things in CS that's just impossible.

7.8 Transactional RPC -- "one phase" commit

Since I can't solve the General's Paradox, let me solve a related problem.

Abstraction: distributed transaction -- two machines agree to do something, or not do it, atomically (but not necessarily at exactly the same time).

Use log on each machine to keep track of whether RPC has happened.
A writes to its disk: "about to make request to B"

A->B: sends request

B gets request, writes to disk: "got request from A; about to send reply"

B->A: sends reply

A writes to disk: "got reply from B"

7.9 Two phase commit

First phase, ask if each can commit -- for instance, transfer of funds from one bank to another.

A writes, "begin transaction" to log
A -> B: OK to transfer funds to me?

Not enough cash:

B-> A: transaction aborted

A writes "abort" to log

Enough cash:

1:
B: Write new X account balance to log

2:
B->A: OK, I can commit

Second phase, A can decide for both, whether they will commit.

3:
A: Write new Y account balance to log

4:
Write commit to log

5:
Send message to B that commit occurred

6:
B: Write "got commit" to log