Lecture 3, RPC continued

RPC semantics
RPC with multiple clients, multiple RPCs/client
NFS
Distributed objects

First any questions about last time? Everyone have a project team?

1) Quick review: Remote procedure call vs. procedure call

What's equivalent?

Parameters -- request message

Result -- reply message

Name of procedure -- passed in request message

return address? Source address/port in the request packet

2) Implementing RPC

at least once, easy – just retry
exactly once – two phase commit
at most once -- ?

A bit of background on the Internet: the worst possible network, except for all others. Packets can be dropped, delivered very very late, duplicated in transit, etc. Non-transitive connectivity is possible too.

An example: packet sent before a crash, delivered after the machine reboots

A problem! if packet was received and processed (ignore resend) vs. never received (do operation)

Internet has two main protocols:

UDP is for simple message exchange, so the client simply sends a message, telling the OS to fill the packet with the right server address, port. On the server side, it replies, telling the OS to fill in the client address, port

Our interface is UDP-like, just a bit simpler. The reason for not using TCP will become obvious later, as we'll need to use messages for implementing certain protocols.

Note although we didn't ask you to do it for this assignment, you will need to be able to do RPC's from multiple clients to the same server. That is, the server will need to keep different conversations distinct. [More on that in a sec.]

TCP sets up a bidirectional byte stream between two nodes. For TCP, it actually contacts the server when you connect. This allows state to be set up on the remote side. Hint: for at most once, you need state on the remote side to make your RPC work!

(That is to say, even though we're using UDP – you probably need a message exchange to start things off, to be able to deal with failures correctly.)

[skip]
With TCP the client does:

Socket = socket(TCP)
Connect(socket, server address, port #) – exchanges start messages
Send data (socket) – do the RPC
Read data (socket) – wait for the reply

On the server side:
Socket = socket(TCP)
Listen(socket) – marks this as a server
Fd = accept an incoming connection(socket)

Read data (fd)
Do operation
Write reply on fd
[end skip]

3) What if there are multiple clients?

RPC described so far:

Client: calls local procedure

Server: loop

```
  Wait for incoming request
  Execute request
```

But if there are multiple clients, server will stall waiting for the client
Perhaps ok if single CPU, and requests are short

What if request goes to the file system/disk? (e.g., fetch page)

One approach (Apache): use threads and TCP

[a thread is an asynchronous procedure call]

```
  Wait for incoming connection (fd = socket accept)
  Fork thread to handle requests from that client (fd)
```

```
Each server thread has loop
   Wait for incoming request (read fd)
   Execute request
   Reply (write fd)
```

Alternate approach: events

```
Server: loop
   Get next event
   Process event (send request to file system, get completion notification from file system, get command from user)
```

OK, let's make this a bit more interesting

Google instant: Type command, get the search result, but you can keep typing, and it updates as it goes.

Does it do an RPC for every character? (sort of: unlike a procedure call, it doesn't wait for the reply. You can keep typing – needs to match up the replies to what you are typing.)

Two equivalent ways of thinking about this, events vs. threads:

```
Loop
Wait for event to arrive
If event command from user,
    Build message
    Send msg
Else if event is reply from server
        Unpack reply
        Display result
End
```

Project uses events, so that we don't need to require 451

Alternate view: threads

```
Loop
  Wait for user to type character
   Fork thread to do RPC based on characters typed so far
```

(actually 2 RPCs – one to get the suggested word completion, another to get the search results)

Another thread manages the screen

```
Loop
   Redraw screen based on results from latest RPC
```

4) Case study: NFS

File server, many clients using the file server

Transparent to application: it does file read/write, file system does rest
Implication: failures are transparent to application!

Design ideas:
a) portability: client/server can be different types of machines
=> RPC format is machine-independent (XDR) and file system independent (can be implemented on server side by various types of file systems)

Q: how does the web achieve this? Contents of web requests are (usually) in text format

b) Stateless protocol
Server can crash, restart, and system continues to work

- ⇨ each RPC contains complete information to do operation

(example, no file seek; rather ReadAt)
All operations must reach disk at server before the server replies
(why?)

c) (Most) operations are idempotent
- ⇨ client resends RPC until completion

Q: what happens when the client crashes?
[client can lose work, but that's the same as before!]

Q: is delete name from directory an idempotent operation?
[what about two clients: one does delete, the other does create?]

d) file handles are unique for all time (file # (aka inode), generation #, filesystem ID)

otherwise, couldn't reuse a deleted file # -- a replay message might refer to the old deleted file #, or the new file stored in its place

d) Mount as a binding step

Which server do we send RPCs to? Does client have permission to access files on the server?

Issues (forward references to later in the course):

a) original security model assumes you can trust the client OS

b) weakly cache coherent (next week)

c) timestamps (next week): how does Makefile work?

5) Distributed Objects

How do we extend RPC to an object-oriented system? Why is RPC not sufficient?

Can only pass copies of data structures to the client or server
for example, can you pass a C++ object?
Pass a pointer and dereference it remotely?

programmer must design a scheme for naming remote objects

for name design, we need to maintain mapping from name to object
  locks: lockid_t
  extents: extendid_t
  files/directories: inums

Typical goals of distributed object systems:
  transparent RPC for object methods
  allow passing of object references as arguments
  allow migration of objects (object not just located on origin server)
  distributed GC, needed for remote refs

Situations in which one client might pass remote object ref to another?

Forward an email attachment on a thin link
  lots of modules: shopping cart, item db, checkout, front end

first a simple call/return
```
  o = ???;
  o.fn("hello");
```
  which server to send to?
  what object on server?
  what about "hello"?
  what does RPC message contain?
  how does server find the real object?
  where does server-side dispatch fn come from?

what does a stub object look like?
  type?
  contents?
  where did it come from?

is there anything special about the server-side "real" object?

is "hello" sent as a remote object ref?

how about passing an object as an argument?
```
  o1 = ???;
  o2 = ???;
  o1.fn(o2);
```
  what must o2 look like in the RPC message?
    server host, object ID
  what if o1's server already knows about o2?

  how do we know if two objects are the same?
  must have a table mapping object ID to ptr to o2
 what if o1's server does not know about o2?
  where does it get stub type, implementation?
  can stub stuff be generated purely by client?

there are probably type IDs, so client can re-use stub code
 an object ID must contain type ID, or an RPC to fetch it
 clients and servers must have tables mapping type ID to stub code

when can a server free an object?
 only when no client has a live reference
 server must somehow learn when new client gets a reference
 and when client local ref count drops to zero
 so clients must send RPCs to server to note first/last knowledge
 what if C1 passes to C2, C1 sends de-ref RPC before C2 sends ref?

what if a client crashes?
 will server ever be able to free the object?