LECTURE

Dist Sys.
Intro

Distributed Systems

⇒ what are they?

⇒ why use them?

⇒ new problems / issues?   new: communication / networking

primary: how to handle **failures** → both a problem
of parts of system   an opportunity

client/server
cluster

Communication

⇒ basics: all comm is
unreliable
(bit ① errors, packet loss ③, node/link failure ②)
do 1,2 ⇒ then ask, what if no 1,2
⇒ buffer mgmt ⇒ (must drop packet)

⇒ commonly available

UDP/IP: unreliable datagrams

TCP/IP: reliable byte stream

how to make reliable?

send   client   server
(but keep
message   send
around)   ① "ack" each
                              →   message
(ok to   ack   →give
discard)   msg to
                            APP)

what if you don't receive an ack?

② **timeout**
send   (lost on network)
→ X
t/o {
send again (retry)
→
← ack

side
question:
how long should
timeout wait?
be?

but what if this happens?

send
→
X ← ack
T/o
send
→ ?

solution? ③ **sequence #**
counter:1   counter:∅ 1
Send (1)
→
ack(1)
←
Send (2)   counter: 2
X ← ack
Send (2)
→   (but don't
ack(2)   pass message
←   up to
application)

Communication Paradigms

⇒ Raw messages   pros/cons?

⇒ OS abstractions
→ File System (global)
→ VM system
(go into example)

⇒ PL abstraction
function call / method call
(Remote Procedure Call)
(or RMI)

Remote Procedure Call

client: looks like normal PC
e.g. func open("file", O_RDONLY);
server: just implements routines
e.g. int func open (char *file, int mode) {
}

but through some magic...
client calls **func()**;

(just
like
normal PC)

client
waits
for
reply

msg gets
sent   → server
executes
func
returns
return
code

clients
gets
re   ← message
reply

components of RPC package:

client   server
client stub   server stub   → stub compiler
rtlib   rtlib   → runtime library

stub compiler: what does it do?

```
int func1(int arg) {

    return 0;
}
```
compiler generates **stub code** → client
→ server
client/server **link** w/ this code
client side:
func1(10);
↳ int func1(int arg) {
pack func ID→msg;
pack arg →msg;
details → (send msg (server, msg)
handled by   wait for reply (&r;r msg);
runtime   unpack re ← r msg;
return re;
}
server side: similar

Stub compiler: harder case

```
int func_2 (char *buffer, int size) {
    //eg. like write(fd, buffer, size)


}
```
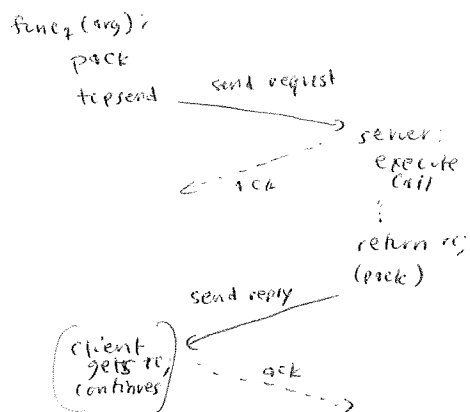
why hard?

Runtime Library

→ naming / binding
    (connecting client w/ right server)

→ managing communication
    option 1: use TCP (or similar)

```
func_2 (arg);
    pack
    tcp send  ──send request──→  server:
                                    execute
              ←⋯ ack               call
                                    :
                                    return rc
                                    (pack)
              ──send reply──
( client  )←
( gets rc )  ←⋯ ack ⋯→
( continues)
```

⇒ (seems wasteful: why?)

Problem?

option 2: use UDP (or similar)
    now, must guarantee semantics
    [if no crash ⇒ exactly once
      if server
        crashes ⇒ @ most once]

    ⇒ use timeout/retry + sequence #

problem: what if remote PC just
    takes a long time?

problem: what if args don't fit
    into single packet?
        (fragment / reassemble)

technique:
(piggybacking)
(i) [use reply as ack of
      request]

3 cases:
⇒ → request lost
   → reply lost
   → server crash