N2: Intro to networked services

Review

- Internet is a peripheral to my computer (PIO, DMA, etc.)
- Routing – distance vector
- Naming – DNS, zones
- Sharing – congestion control

Outline
Finish congestion control
- danger !
- principles

Message APIs/abstractions
- send/receiv
- rpc

Challenges
Performance: LogP
Intro: Distributed File Systems

Using messaging to build services

Send/Receive

How do you program a distributed application?
Need to synchronize multiple threads, but they are on multiple machines (no test&set)

Atomic send/receive – doesn't require shared memory for synchronizing cooperating threads

Note that send and receive are atomic
never get portion of a message (all or nothing)
two receivers can't get same message

Q: How do you know you got the whole message?
Q: How do you know no errors in message?

Mailbox – temporary holding area for messages (ports)

Looks like producer/consumer queue

-- Two “threads”: CPU and NIC; some amount of locking, signaling needed (a few extra details – don’t want to try to grab spin lock when interrupts are off; may not be OK to block IO device; …)

Receive(buffer, mbox)
→ Wait until mbox has message in it, then copy message into buffer, and return

when packet arrives, OS puts message into mbox, wakes up one of the writers

Send(buffer, mbox)
When can Send return?

- when receive gets message?
- when message is safely buffered on destination node?
- Right away, if message is buffered on source node?

Message styles

1-way – messages flow in one direcction (UNIX pipes, TCP)
2-way – request-response (remote procedure call)

1-way communication

```
Producer:
     int msg1[1000];
     while(1){
          prepare message; // add coke to mach.
          Send(msg1, mbox);
     }

Consumer
     int msg2[1000];

     while(1){
          receive(msg2, mbox);
          process message; // drink coke
     }
```

no need for producer/consumer to keep track of space in mailbox – handled by send/receive

2-way communication

What about 2-way communication? Request/response – e.g. "read a file" stored on a remote machine

Also called – client-server

Client = requestor

server = responder

Server provides "service" to client

request/response:

```
client:
    char response[1000];

    send("read rutabaga", mbox1);
    receive(response, mbox2);

server:
    char command[1000], answer[1000];

    receive(command, mbox1);
    decode command;
    read file into answer;
    send(answer, mbox2);
```

Remote procedure call

Call a procedure on a remote machine

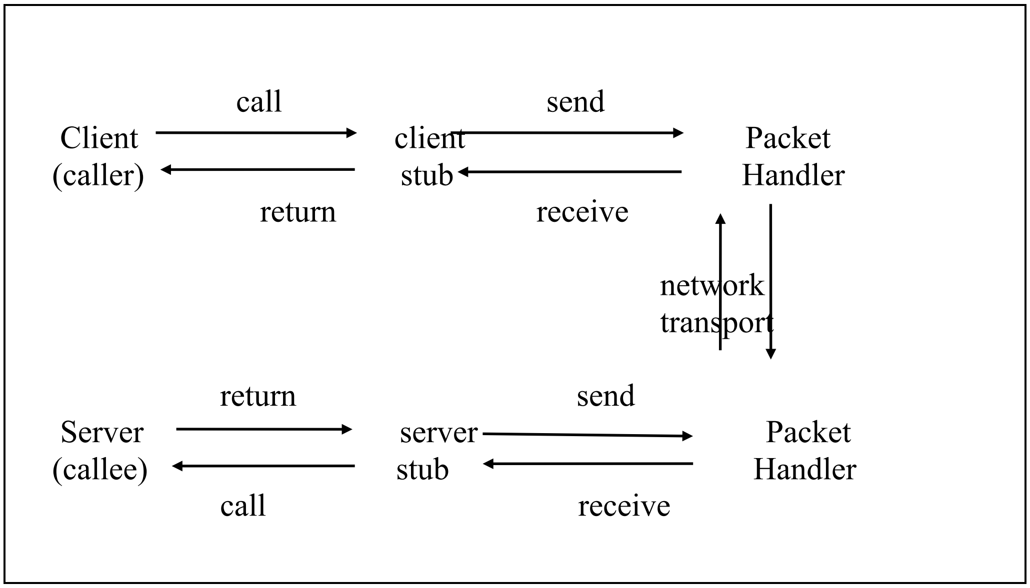
```
client
    remoteFileSys->Read("rutabaga");
```

translated into call on server:

```
fileSys->Read("rutabaga");
```

Implementat on top of request-response message passing

“stub” provides glue




```
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
```

Comparison between RPC and procedure call

What’s equivalent

Parameters – request message

Result – reply message
Name of procedure – passed in request message
return address – mbox2

Implementation issues

Stub generator – implements stubs automatically
for this, only need procedure signature – types of arguments, return value
generate code on client to pack message, send it off, on server to unpack message, call procedure

How does client know which mbox to send to? Binding
static – fixed at compile time (e.g. C)
dynamic – fixed at runtime (e.g. Lisp, RPC)

In most RPC systems, dynamic binding via name service.
Name service provides dynamic translation of service → mbox

Why runtime binding?
Access control – check who is permitted to access service
fail-over – if server fails, use another

Problems with RPC

Problem solved?

RPC provides location transparency – except

Failures -- message loss, machine crash
Performance
Consistency/replication
Security

- All hard problems.
- Fundamental limits (e.g., you can't atomically update an object replicated at multiple machines)
- Diffcult trade-offs among goals -- e.g., consistency v. availability CAP

Failures

~~Different failure modes in distributed system than on single machine~~

~~Several kinds of failure~~
~~(1) communication interruption~~

- ~~lost message~~
- ~~lost reply~~
- ~~cut wire~~
- ~~…~~

~~Simple solution:~~
~~Request/acknowledge protocol~~
~~Common case:~~
~~1) Sender sends message (msg, msgId) and sets timer~~
~~2) Receiver receives message and sends (ack, msgId)~~
~~3) Sender receives (ack, msgId) and clears timer~~

~~If timer goes off, goto (1)~~

~~How does this work? Local procedure call guarantes *exactly once*~~ semantics. ~~What does retransmission guarantee?~~

- ~~What if msg 1 lost?~~
- ~~What if ack lost?~~

~~Guarantees *at least once*~~ semantics ~~***assuming no machines crash or otherwise discontinue protocol***~~

- ~~Receiver guaranteed to recv message at least once~~
- ~~Receiver may recv message multiple times. Receiver MAY use sequence number to filter repeated transmissions so that each is acted upon just once (but what if receiver crashes and loses seq number info?)~~

~~in general -- request may be executed 0, 1, 2, or more times.~~

~~(2) Machine fails~~
~~Several variations:~~

- ~~user level bug causes address space to crash~~
- ~~machine failure, kernel bug causes all AS on same machine to fail~~
- ~~power outage causes all machines to fail~~

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

~~Example: simple send/ack protocol above -- Difficult to deal with machine crashes~~

- ~~If sender crashes (or if sender gives up because it has tried 100 times in a row) what is the post condition?~~
  - ~~Receiver may or may not have received message~~
- ~~If receiver crashes, filtering repeated messages to act on them exactly once is tricky → carefully design protocol to either (a) tolerate *at least once* semantics or (b) detect/avoid replication even across sender/receiver failures~~

~~Tricky – processing a message can have arbitrary side effects. Want *exactly once* semantics or protocol may have strange behaviors~~

~~Tomorrow: strategies for dealing with machine failures in distributed protocols~~
- ~~Ad-hoc strategies (file systems)~~
- ~~Two-phase commit~~
- ~~Persistent message queues~~