15-410
“...Mooooo!...”

IPC & RPC
Nov. 14, 2018

Dave Eckhardt
Dave O'Hallaron
Synchronization

**Project 3 tactical considerations**

- Getting the shell running is important
  - We won't build a hand-load kernel for each test!
  - Test harness relies on shell to launch programs
- Getting a body of code *solid* is important
  - Better for `exec()` to work 1,000 times than `thr_fork` once
  - It is important to read the hurdle *web page* and the hurdle form early and often!
- Run tests as soon as you can
  - Scoreboard can be a source of inspiration!
- Carefully consider the P3extra overtime
  - In general, getting a really solid kernel is the best thing
    » For your grade
    » For your education!
Outline

**A Pattern Language (for client-server messaging)**
- Client view, server view, world view

**IPC – InterProcess Communication**

**RPC – Remote Procedure Call**

**Textbook**
- OSC - Sections 3.4-3.6
- OS:P+P - missing
Client View

Client → Request → Send → Response

Client → Receive → Response
Server View

Request → Receive → Server

Send → Response → Server
Reality?

Those views are correct with respect to each viewer.
The kernel's view is more complex.
Reality?

Those views are correct with respect to each viewer.

The kernel's view is more complex:

- Data transfer, obviously
- Buffering (maybe)
- Blocking
- Matching a live request against a blocked request, or else blocking
Receiver Prepares
Client Sends Request

Client → Request → Send → Server

Receive → Server
Send Matches Receive

Client → Request → Send → Request → Receive → Server
Client Posts Receive

Client → Request → Send → Request → Receive → Server

Client → Receive
Server Posts Reply
Reply Matches Receive
Reply Matches Receive

Other event sequences are possible!
Scope of “IPC”

Communicating processes on one machine

What about multiple machines?

- Virtualize single-machine IPC
- Switch to a “network” model
  - Failures happen
  - Administrative domain switch
  - ...
  - (“RPC”)
IPC parts

Naming

Synchronization/buffering

Message body issues
  - Copy vs. reference
  - Size
Naming

Message sent to **process** or to **mailbox**?

Process model

- send(P, msg)
- receive(Q, &msg) or receive(&id, &msg)

No need to set up “communication link”

- But you need to know process id's
- You get only one “link” per process pair
Naming

Mailbox model

- send(box1, msg)
- receive(box1, &msg) or receive(&box, &msg)

Where do mailbox id's come from?

“name server” approach

box = createmailbox();
register(box1, “Terry's process”);
boxT = lookup(“Terry's process”);

File system approach – great (if you have one)

box = createmailbox(“/tmp/Terry”);
Multiple Senders

**Problem**
- Receiver needs to know who sent request

**Typical solution**
- “Message” not just a byte array
- OS imposes structure
  - sender id (maybe process id and mailbox id)
  - maybe: type, priority, ...
Synchronization

Issue
- Does communication imply synchronization?

Blocking send()?
- Ok for request/response pattern
- Provides assurance of message delivery
- *Bad* for producer/consumer pattern

Non-blocking send()?
- Raises buffering issue (below)
Synchronization

Blocking receive()?
- Ok/good for “server thread”
  - Remember, de-scheduling is a kernel service
- Ok/good for request/response pattern
- Awkward for some servers
  - Abort connection when client is “too idle”

Pure-non-blocking receive?
- Ok for polling
- Polling is costly
Synchronization

Receive-with-timeout

- Wait for message
- Abort if timeout expires
- Can be good for highly-reliable or real-time systems
- What timeout value is appropriate?
  - Depends on each specific and complete system
  - Timeout values are error prone
Synchronization

**Meta-receive**
- Specify a group of mailboxes
- Wake up on first message

**Receive-scan**
- Specify list of mailboxes, timeout
- OS indicates which mailbox(es) are “ready” for what
- Unix: `select()`, `poll()`
Buffering

**Issue**
- How much space does OS provide “for free”?  
- “Kernel memory” limited!

**Options**
- No buffering  
  - implies blocking send  
- Fixed size, undefined size  
  - Send blocks *unpredictably*
A Buffering Problem

P1

send(P2, p1-my-status)
receive(P2, &p1-peer-status)
A Buffering Problem

P1
send(P2, p1-my-status)
receive(P2, &p1-peer-status)

P2
send(P1, p2-my-status)
receive(P1, &p2-peer-status)

What's the problem?
- Can you draw a picture of it?
Message Size Issue

Ok to copy *small* messages sender $\Rightarrow$ receiver

Bad to copy *1-megabyte* messages
  - (Why?)

Bad suggestion: “Chop up large messages”
  - Why?
Message Size Issue

Ok to copy *small* messages sender $\Rightarrow$ receiver

Bad to copy *1-megabyte* messages

- (Why?)

Bad suggestion: “Chop up large messages”

- Evades the issue!
“Out-of-line” Data

Message can refer to memory regions
- (page-aligned, multiple-page)
- Either “copy” or transfer ownership to receiver
- Can share the physical memory
  - Mooooo!
“Rendezvous”

Concept
- Blocking send
- Blocking receive

Great for OS
- No buffering required!

Theoretically interesting

Popular in a variety of languages
- (most of them called “Ada”)
Mach IPC – ports

Port: Mach “mailbox” object
  - One receiver
    - (one “backup” receiver)
  - Potentially many senders

Ports identify system objects
  - Each task identified/controlled by a port
  - Each *thread* identified/controlled by a port
  - Kernel exceptions delivered to “exception port”
    - “External Pager Interface” - page faults in user space!
Mach IPC – Port Rights

**Receive rights**
- “Receive end” of a port
- Held by one task, not published
  - receive rights imply ownership

**Send rights**
- “Send end” - ability to transmit message to mailbox
- Frequently published via “name server” task
- Confer no rights (beyond “denial of service”)

Mach IPC – Message Contents

Memory regions
- In-line for “small” messages (copied)
- Out-of-line for “large” messages
  - Sender may de-allocate on send
  - Otherwise, copy-on-write

“Port rights”
- Sender specifies task-local port #
- OS translates to internal port-id while queued
- Receiver observes task-local port #
Mach IPC – Operations

**send**
- block, block(n milliseconds), don't-block
- “send just one”
  - when destination full, queue 1 message in *sender thread*
  - sender notified when transfer completes

**receive**
- receive from port
- receive from *port set*
- block, block(n milliseconds), don't-block
Mach IPC – “RPC”

Common pattern: “Remote” Procedure Call
- Really: “cross-task” procedure call

Client synchronization/message flow
- Blocking send, blocking receive

Client must allow server to respond
- Transfer “send rights” in message
  - “Send-once rights” speed hack

Server message flow (N threads)
- Blocking receive, non-blocking send
Mach IPC – Naming

Port send rights are OS-managed capabilities
  - unguessable, unforgeable

How to contact a server?
  - Ask the name server task
    - *Trusted* – source of all capabilities

How to contact the name server?
  - Task creator specifies name server for new task
    - Can create custom environment for task tree
      - By convention, send rights to name server are located at a particular client port number (like stdin/stdout/stderr)
  - System boot task launches nameserver, gives out rights
IPC Summary

Naming
- Name server?
- File system?

Queueing/blocking

Copy/share/transfer

A Unix surprise
- sendmsg()/recvmsg() pass file descriptors!
RPC Overview

RPC = Remote *Procedure Call*

Concept: extend IPC across machines
  - Maybe across “administrative domains”

Marshalling

Server location

Call semantics

Request flow
RPC Model

**Approach**

\[ d = \text{computeNthDigit}(\text{CONST_PI}, 3000); \]
- Abstract away from “who computes it”
- Should “work the same” when remote Cray does the job

**Issues**

- Must specify server *somehow*
- What “digit value” is “server down”?
  - Exceptions useful in “modern” languages
Marshalling

Values must cross the network

Machine formats differ

- Integer byte order
- Floating point format
  - IEEE 754 or not
- Memory packing/alignment issues
Marshalling

Define a “network format”
- ASN.1 - “self-describing” via in-line tags
- XDR – not

“Serialize” language-level object to byte stream
- Rules typically recursive
  - Serialize a struct by serializing its fields in order
- Implementation probably should *not* be recursive
  - (Why not?)
Marshalling

**Issues**

- Some types don't translate well
  - Ada has ranged integers, e.g., 44..59
  - Not everybody really likes 64-bit ints
  - Floating point formats are religious issues
- Performance!
  - Memory speed $\cong$ network speed
- The dreaded “pointer problem”
Marshalling

```c
struct node {
    int value;
    struct node *neighbors[4];
} nodes[1024];

nnodes = sizeof(nodes)/sizeof(nodes[0]);

n = occupancy(nodes, nnodes);
bn = best_neighbor(node);
i = value(node);
```

Implications?
Marshalling

\[ n = \text{occupancy}(\text{nodes}, \; \text{nnodes}); \]
- Marshall array – ok

\[ bn = \text{best}\_\text{neighbor}(\text{node}); \]
- Marshall graph structure – not so ok

\[ i = \text{value}(\text{node}); \]
- *Avoiding* marshalling graph – not obvious
  - “Node fault”??
Server Location

Which machine?
- Multiple AFS cells on the planet
- Each has multiple file servers

Approaches
- Special hostnames: www.cmu.edu
- Machine lists
  - AFS CellSrvDB /usr/vice/etc/CellSrvDB
- DNS SRV records (RFC 2782)
Server Location

Which port?

- Must distinguish services on one machine
  - Single machine can be AFS volume, vldb, pt server
- Fixed port assignment
  - AFS: fileserver UDP 7000, volume location 7003
  - /etc/services or www.iana.org/assignments/port-numbers
  - RFC 2468 www.rfc-editor.org/rfc/rfc2468.txt
- Dynamic port assignment
  - Contact “courier” / “matchmaker” service via RPC
  - ...on a fixed port assignment!
Call Semantics

Typically, caller blocks
- Matches procedure call semantics

Blocking can be expensive
- By a factor of \textit{a million} over real procedure call

“Asynchronous RPC”
- Transmit request, do other work, check for reply
- Not really “PC” any more
- More like programming language “futures”
Fun Call Semantics

Batch RPC
- Send *list* of procedure calls
- Later calls can use results of earlier calls

Issues
- Abort batch if one call fails?
  - Yet another programming language?
- Typically wrecks “procedure call” abstraction
  - Your code must make N calls before 1st answer
Fun Call Semantics

Batch RPC Examples

- NFS v4, RFC 3010
- Bloch, A Practical Approach to Replication of Abstract Data Objects
Sad Call semantics

Network failure
- Retransmit request
  - How long?

Server reboot
- Does client deal with RPC session restart?
- Did the call “happen” or not?
  - Retransmitting “remove foo.c” all day long may not be safe!
Client Flow

Client code calls *stub* routine
- “Regular code” which encapsulates the magic

Stub routine
- Locates communication channel
  - If not established: costly location/set-up/authentication
- Marshals information
  - Procedure #, parameters
- Sends message, awaits reply
- Unmarshals reply, returns to user code
Server Flow

Thread pool runs **skeleton code**

**Skeleton code**
- Waits for request from a client
- Locates client state
  - Authentication/encryption context
- Unmarshals parameters
- Calls “real code”
- Marshals reply
- Sends reply
RPC Deployment

Define interface
  - Get it right, you'll live with it for a while!
  - AFS & NFS RPC layers ~15 years old

“Stub generator”
  - Special-purpose compiler
  - Turns “interface spec” into stubs & skeleton

Link stub code with client & server

Run a server!
Java RMI

Remote Method Invocations

Serialization: programmer/language cooperation

- *Dangerously* subtle!
  - Bloch, *Effective Java*

RMI > RPC

- Remote methods ≈ remote procedures
- *Parameters* can be (differently) remote
  - Client on A can call method of class implemented on B passing object located on C
    » (slowly)
RPC Summary

RPC is lots of fun
So much fun that lots of things don't do it
  - SMTP
  - HTTP

RPC = IPC
  + server location, marshalling, network failure, delays
  - special copy tricks, speed

Remote Objects? Effective Java, Bitter Java