Advanced Topics: Distributed Systems Intro

CS 537: Introduction to Operating Systems

Louis Oliphant

University of Wisconsin - Madison

Fall 2024

Administrivia

- Project 6 - due Dec 6th
- Final Exam: 12/19 10:05-12:05
  - Humanities room 3650
  - Microbial Sciences room 1220
  - McBurney Accommodations, CS room 1325

Review Multiprocessor Scheduling

- Multiprocessor Architecture
  - cache and cache coherency
- Multiprocessor Complications
  - cache affinity
  - synchronization
- SQMS and MQMS
- CFS Scheduler
- ULE Scheduler

Quiz 22 SSDs

https://tinyurl.com/cs537-fa24-quiz22

Distributed Systems

Building Distributed Systems That Work When Components Fail

- System objectives of performance, security, **communication**
- Unreliable Communication Layers (**UDP**)
  - **checksum**
- Reliable Communication Layers (**TCP**)
  - acknowledgement, timeout/retry
  - sequence counter
- Communication Abstractions
  - Distributed Shared Memory (DSM)
  - Remote Procedure Call (**RPC**)
    - Stub Generator
    - Run-Time Library
    - Other Issues: fragmentation/reassembly, byte ordering, synchronicity

WHAT IS A DISTRIBUTED SYSTEM?

A distributed system is one where a machine I've never heard of can cause my program to fail.
— *Leslie Lamport*

Definition: More than one machine working together to solve a problem

Examples:

– client/server: web server and web client
– cluster: page rank computation

WHY GO DISTRIBUTED?

More computing power

More storage capacity

Fault tolerance

Data sharing

NEW CHALLENGES

System failure: need to worry about partial failure

Communication failure: links unreliable

- bit errors
- packet loss
- node/link failure

COMMUNICATION OVERVIEW

Raw messages: UDP

Reliable messages: TCP

Remote procedure call: RPC

RAW MESSAGES: UDP

UDP : User Datagram Protocol

API:

- reads and writes over socket file descriptors
- messages sent from/to ports to target a process on machine

Provide minimal reliability features:

- messages may be lost
- messages may be reordered
- messages may be duplicated
- only protection: checksums to ensure data not corrupted

RAW MESSAGES: UDP

Advantages

- Lightweight
- Some applications make better reliability decisions themselves (e.g., video conferencing programs)

Disadvantages

- More difficult to write applications correctly

RELIABLE MESSAGES: LAYERING STRATEGY

TCP: Transmission Control Protocol

Using software to build

reliable logical connections over unreliable physical connections

TECHNIQUE #1: ACK

Sender	Receiver
[send message]
	[recv message]
	[send ack]
[recv ack]

Ack: Sender knows message was received
What to do about message loss?

TECHNIQUE #2: TIMEOUT

Sender		Receiver
[send message]	→ ✕	
[start timer]		
… waiting for ack …		
[timer goes off]		
[send message]	→	[recv message]
	←	[send ack]
[recv ack]		

TIMEOUT

How long to wait?

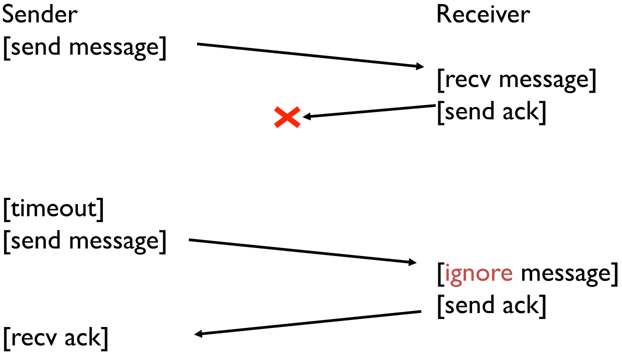
Too long?

- System feels unresponsive

Too short?

- Messages needlessly re-sent
- Messages may have been dropped due to overloaded server. Resending makes overload worse!

LOST ACK PROBLEM



SEQUENCE NUMBERS

Sequence numbers

- senders gives each message an increasing unique seq number
- receiver knows it has seen all messages before N

Suppose message K is received.

- if K <= N, Msg K is already delivered, ignore it
- if K = N + 1, first time seeing this message
- if K > N + 1 ?

TCP

TCP: Transmission Control Protocol

Most popular protocol based on seq nums
Buffers messages so arrive in order
Timeouts are adaptive

COMMUNICATIONS OVERVIEW

Raw messages: UDP

Reliable messages: TCP

Remote procedure call: RPC

RPC

Remote **P**rocedure **C**all

What could be easier than calling a function?

Approach: create wrappers so calling a function on another machine feels just like calling a local function!

RPC

Machine A

```
int main(…) {
	int x = foo(”hello”);
}

int foo(char *msg) {
	send msg to B
	recv msg from B
}
```

Machine B

```
int foo(char *msg) {
	…
}

void foo_listener() {
	while(1) {
		recv, call foo
	}
}
```

RPC

Machine A

```
int main(…) {
	int x = foo(”hello”);
}
```

client wrapper

```
int foo(char *msg) {
	send msg to B
	recv msg from B
}
```

Machine B

```
int foo(char *msg) {
	…
}
```

server wrapper

```
void foo_listener() {
	while(1) {
		recv, call foo
	}
}
```

RPC TOOLS

RPC packages help with two components

(1) Runtime library

- Thread pool
- Socket listeners call functions on server

(2) Stub generation

- Create wrappers automatically
- Many tools available (rpcgen, thrift, protobufs)

WRAPPER GENERATION

Wrappers must do conversions:

- client arguments to message
- message to server arguments
- convert server return value to message
- convert message to client return value

Need uniform endianness (wrappers do this)

Conversion is called marshaling/unmarshaling, or serializing/deserializing

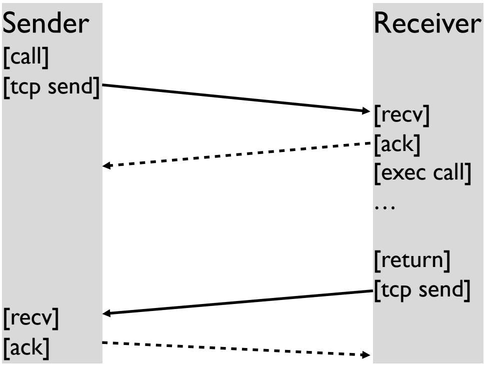
WRAPPER GENERATION: POINTERS

Why are pointers problematic?

Address passed from client not valid on server

Solutions? Smart RPC package: follow pointers and copy data

RPC OVER TCP?


Sender
[call]
[tcp send]
[recv]
[ack]
Receiver
[recv]
[ack]
[exec call]
…
[return]
[tcp send]
[recv]
[ack]

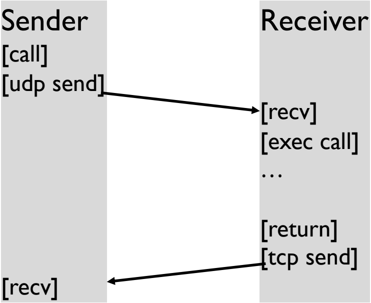
RPC OVER UDP

Strategy: use function return as implicit ACK

Piggybacking technique

What if function takes a long time?

then send a separate ACK



Other Issues

- **Long-running calls**, client periodically asks server for results
- **Data Organization** – e.g. Big-Endian vs. Little Endian
  - Sun's XDR (eXternal Data Representation) formatting standard
  - Google's gRDP uses HTTP/2
- Some systems provide both **synchronous** (i.e. wait for result) and **asynchronous** (i.e. return immediately with some type of callback)

Summary

- UDP for unreliable communication
- TCP for reliable communication
- RPC often builds on top of UDP layer, handles communication failures itself
  - has a stub generator and run-time library
  - handles issues like fragmentation and byte ordering
  - Typically synchronous calls (wait for completion)
- RPC packages include:
  - Sun's RPC system
  - Google's gRPC
  - Apache Thrift
  - JSON-RPC