Lecture 11: Remote Procedure Calls

July 11, 2019
Instructor: Jack Kolb
https://cs162.eecs.berkeley.edu
Logistics

• Homework 1 Due Friday
• Project/HW “Party” Friday 3-5PM
• Midterm Exam: 7/18, 5-7pm
  • No Lecture that Day
• Review Session on 7/17 during Lecture Slot!
Recall: Networking Definitions

- **Network**: Physical connection that allows two computers to communicate

- **Frame/Packet/Segment**: Unit of data transfer, sequence of bits carried over the network
  - Network carries packets from one CPU to another
  - Destination gets interrupt when frame arrives
  - Name depends on which layer (later)
Recall: Networking Challenge

• Many different applications
  • Email, Web, Online Games, etc.

• Many different network styles and technologies
  • Wireless, Wired, Optical, etc.

• How do we organize all of this complexity?
Recall: The Internet *Hourglass*

The Hourglass Model

Applications
Transport
Data Link
Physical

The Hourglass Model

SMTP, HTTP, DNS, NTP

TCP, UDP

IP

Ethernet, SONET, 802.11

Copper, Fiber, Radio
Recall: End-to-End Principal

• Seen as a guiding principle of the Internet
• Some types of network functionality can only be correctly implemented end-to-end
  • Reliability, security, etc.
• Implementing complex functionality in the network:
  • Doesn’t necessarily reduce complexity on end hosts
  • Does increase network complexity
  • Imposes a cost on all applications, even if they don’t need the functionality
Recall: Internet Protocol

• Routing – an IP packet goes anywhere
  • Just need the destination IP address
• Fragmentation – split big messages into smaller pieces
• Multiple protocols running on top

• Unreliable Delivery
• Out-of-order/Duplicate Delivery
Recall: Using Acknowledgements

- Checksum: Detect garbled packets
- Receiver sends a packet to acknowledge when a packet received and ungarbled
  - No acknowledgement? **Resend** after timeout
- What if acknowledgement dropped?
  - Packet is resent (wasteful), second chance to acknowledge
Window-Based Acknowledgements

• Send up to $N$ packets without ack
• Both source and destination need to store $N$ packets
• Each packet has sequence number

![Diagram showing window-based acknowledgements with $N=5$.]
Recall: TCP Windows and Seq. Numbers

![Sequence Numbers Diagram]

- **Sender**
  - Sent
  - Not yet sent
  - Sent not acked

- **Receiver**
  - Received
  - Buffered
  - Not yet received
  - Given to app
Recall: Congestion Avoidance

• Solution: Adjust **Window Size**

• AIMD: Additive Increase, Multiplicative Decrease
  • When packet dropped (no ack), cut window size in half
  • If no timeouts, slowly increase window size by 1 for each acknowledgement received
Review: TCP Windowing

• Assume we have a window size of 4
• What sequence numbers are inside of the "sliding window" after we receive packets with sequence numbers in the following order?

   0, 1, 2, 4, 6, 5, 3

• When does data get delivered to the user process?
Remote Procedure Call (RPC)

• Idea: Make communication look like an ordinary function call
• Wrapper library like for system calls
  • Called *stubs*
• Also wrappers at the receiving end
  • Read messages from socket, dispatch to actual function
• Look like local function calls
RPC Information Flow

Client (caller) → Client Stub → Packet Handler → Server Stub → Server (callee)

Machine A

Machine B

Network
RPC Pseudocode

Client
#include <myprotocol_stubs.h> // Generated by tool
RPCContext ctx = myprotocol_ConnectToServer(hostname, port);
...
result = myprotocol_mkdir(ctx, "/directory/name");
...

Server
#include <myprotocol_stubs.h>
main()
    myprotocol_SetupRPCServer(port);
} int real_myprotocol_mkdir(RPCContext ctx, char *name) {
    ...
}
RPC Details

• Setup: Need to specify remote machine somehow
  • Example: Host/Port

• Need to **marshall** arguments over the network
  • Need a concrete representation
  • Sometimes also called **serialization**
  • Done by stub

• Typically code generated from a **file specifying the protocol**
  • Called an **Interface Definition Language (IDL)**
  • Generates stubs, including marshalling/unmarshalling code
Interface Definition Language

Pseudocode
protocol myprotocol {
    1: int32 mkdir(string name);
    2: int32 rmdir(string name);
}
Marshalling Example: mkdir("/directory/name")
returns 0
Client Sends: \001/directory/name\0
Server Sends: \0\0\0\0\0
Interface Definition Language

• Compiled by a tool (e.g., gRPC) to generate stubs in various source languages (C, C++, Java, Python, …)

• Any client/server stubs can interact if they both adhere to the same protocol
  • Must be able to unmarshall what the other side marshalled

• Implementation language doesn't matter if we send right bits "over the wire"
  • And this is not specific to RPC
Marshalling Challenges

• Marshalling with different machine architectures
  • Remember little endian vs. big endian?
  • Need to choose a consistent format
  • Option 1: Keep in native format and annotate data
  • Option 2: Pick a standard and translate to it if necessary
    • Recall host to network byte ordering in socket code

• Marshalling data structures with pointers
  • Need to trace pointers
  • What if we have something like a linked list? Gets tricky
RPC: Really like a function call?

- What if something fails?
  
  ```c
  result = myprotocol_mkdir(ctx, "/directory/name");
  ```

- What should `result` be?

- Do we really know if the server made the directory on its side?
  - Maybe error occurred with server's file system?
  - Unrelated problem caused server to crash?

- If client doesn't hear back from server: did server crash or is it just taking a long time?
RPC: Really like a function call?

- What if we're doing remote file IO?
  
  ```c
  remoteFile *rf = remoteio_open(ctx, "oski.txt");
  remoteio_puts(ctx, rf, "Bear\n");
  remoteio_close(ctx, rf);
  ```

- What if the *client* fails before it closes?
- Will the file be left open forever?

- Remember: local case is easy, OS cleans up after terminated processes
Welcome to Distributed Systems

• Things get complicated when we have multiple machines in the picture!
  • Each can fail independently
  • Each has its own view of the world
    • Server: Client hasn't closed `oski.txt`, may still want to write
    • Client after crash: I need to open `oski.txt` again

• We'll study these and many other problems later!
Interlude: HTTP

• Application protocol for The Web
  • Retrieve a specific object, upload data, etc.

• Runs on top of TCP (sockets)

• Like any protocol, stipulates:
  • **Syntax**: Content sent over socket connection
  • **Semantics**: Meaning of a message
    • Valid replies and actions taken upon message receipt

• Arguably a form of RPC
HTTP "RPC"

GET /search.html

200 OK
< HTML Content>
HTTP "RPC"

POST /users/ocolsi/photos
<image content>

201 Created
HTTP Messages

• Text-based: We just send character strings over our TCP socket connection
• To make a request, browser might write something like the following on a socket:

GET /hello.html HTTP/1.0
Host: 128.32.4.8:8000
Accept: text/html
User-Agent: Chrome/45.0.2454.93
Accept-Language: en-US,en;q=0.8
HTTP Messages

• Text-based: We just send strings over our TCP socket connection
• We then **read** the following response from the web server:

```
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 128

<html>
<body>
  <h1>Hello World</h1>
  <p>Hello, World!</p>
</body>
</html>
```
HTTP and State

• Remember this situation?
  ```c
  remoteFile *rf = remoteio_open(ctx, "oski.txt");
  remoteio_puts(ctx, rf, "Bear\n");
  remoteio_close(ctx, rf);
  ```

• Client fails: does file stay open forever?
• Server *maintains state* between requests from client
HTTP and State

• HTTP avoids this issue – *stateless protocol*
• Each request is self-contained
  • Treated independently of all other requests
  • Even previous requests from same client!

• So how do we get a session?
  • Client stores a unique ID locally – a *cookie*
  • Client adds this to each request so server can customize its response
RPC Locally

- Doesn't need to be between different machines
- Could just be different address spaces (processes)

- Gives location transparency
  - Move service implementation to wherever is convenient
  - Client runs same old RPC code

- Much faster implementations available locally
  - (Local) Inter-process communication
  - We'll see several techniques later on
Microkernels

- Split OS into **separate processes**
  - Example: File System, Network Driver are external processes
- Pass messages among these components (e.g., via RPC) instead of system calls
Microkernels

- Microkernel itself provides only essential services
  - Communication
  - Address space management
  - Thread scheduling
  - Almost-direct access to hardware devices (for driver processes)
Why Microkernels?

Pros
• Failure Isolation
• Easier to update/replace parts
• Easier to distribute – build one OS that encompasses multiple machines

Cons
• More communication overhead and context switching
• Harder to implement?
Flashback: What is an OS?

- **Always:**
  - Memory Management
  - I/O Management
  - CPU Scheduling
  - Communications
  - Multitasking/multiprogramming

- **Maybe:**
  - File System?
  - Multimedia Support?
  - User Interface?
  - Web Browser?

*Not provided in a strict microkernel*
Influence of Microkernels

• Many operating systems provide some services externally, similar to a microkernel
  • OS X and Linux: Windowing (graphics and UI)

• Some currently monolithic OSs started as microkernels
  • Windows family originally had microkernel design
  • OS X: Hybrid of Mach microkernel and FreeBSD monolithic kernel
Break
Domain Name System (DNS)

• Another distributed system
• **Purpose:** Convert a human readable name (**www.google.com**) to an IP Address (**169.229.15.7**)

• Why?
  • Humans don't want to remember IP addresses
  • But IP routes traffic based on IP addresses

• Other benefits
  • Service can change hosts (IP Address) but keep name
  • Fancy things like sending Google users to different hosts for load balancing
Domain Name System (DNS)
DNS

• A hierarchical system for naming
• Names are divided into labels, right to left:
  • www.eecs.berkeley.edu

• Each domain owned by a particular organization
  • Top level handled by ICANN
  • Subsequent levels owned by organizations

• Resolution by repeated queries
  • One server for each domain: <root>, edu, berkeley.edu, eecs.berkeley.edu
DNS – Root Server

• How do we know where to start?

• Hardcoded list of root servers and backups (updated rarely)

• Or use your ISP's server
  • Makes repeated queries on your behalf
  • Called a recursive resolver
Where are we going next?

- We've established the motivation for all remaining topics in the course!

1. **File Systems** – How do we get convenient file and directory semantics on top of raw storage hardware?

2. **Memory Management & Paging** – We saw that "base and bound" has problems. How do we manage address spaces more effectively?
Where are we going next?

• We've established the motivation for all remaining topics in the course!

3. **Distributed Systems** – When working with multiple machines, how do we coordinate among them and deal with failures?

4. **Security** – When using sockets, how do we protect against eavesdropping and know we can trust the other side?
Summary

• Remote Procedure Call: Invoke a procedure on a remote machine
  • Provides same interface as normal function call
  • Automatic packing and unpacking of user arguments

• Microkernels: Run system services outside of kernel

• HTTP: Application Layer Protocol
  • Like RPC, but stateless

• Domain Name Service: Map names to IP addresses
  • Hierarchical organization among many servers