DISTRIBUTED SYSTEMS

Shivaram Venkataraman
CS 537, Spring 2023
ADMINISTRIVIA

Project 6 grades → tomm AM
Project 7 → next week
Project 8 – Extra credit (4%) →

Midterm 3 conflicts →

AEFIS feedback → useful
AGENDA / LEARNING OUTCOMES

What are some basic building blocks for systems that span across machines?
RECAP
SSD OPERATIONS

Read a page: Retrieve contents of entire page (e.g., 4 KB)
- Cost: 25—75 microseconds
- Independent of page number, prior request offsets

Erase a block: Resets each page in the block to all 1s
- Cost: 1.5 to 4.5 milliseconds
- Much more expensive than reading!
- Allows each page to be written

Program (i.e., write) a page: Change selected 1s to 0s
- Cost is 200 to 1400 microseconds
- Faster than erasing a block, but slower than reading a page
FTL: DIRECT MAPPING

Cons?
- Write amplification
- No wear-leveling

Flash Pages
- Erase block frequently

Logical pages
- User or the FS seed

Physical pages
- Write amplification
- No wear-leveling

number of physical writes for 1 logical write
FTL: LOG-BASED MAPPING

Idea: Treat the physical blocks like a log

<table>
<thead>
<tr>
<th>Table:</th>
<th>100 → 0</th>
<th>101 → 1</th>
<th>2000 → 02</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Block:</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page:</td>
<td>00</td>
<td>01</td>
<td>02</td>
</tr>
<tr>
<td>Content:</td>
<td>a1</td>
<td>a2</td>
<td>c1</td>
</tr>
<tr>
<td>State:</td>
<td>V</td>
<td>E</td>
<td>E</td>
</tr>
</tbody>
</table>

Memory

| Flash Chip |

1. Erase block
2. Write pages in order (log) → Avoid read modify writes → Avoids wear on few pages
GARBAGE COLLECTION

Steps:

Read all pages in physical block

Write out the alive entries to the end of the log

Erase block (freeing it for later use)
# SSD VS HDD PERFORMANCE

SSDs are good at random access, while HDDs are better at sequential access.

---

<table>
<thead>
<tr>
<th>Device</th>
<th>Random Reads (MB/s)</th>
<th>Random Writes (MB/s)</th>
<th>Sequential Reads (MB/s)</th>
<th>Sequential Writes (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samsung 840 Pro SSD</td>
<td>103</td>
<td>287</td>
<td>421</td>
<td>384</td>
</tr>
<tr>
<td>Seagate 600 SSD</td>
<td>84</td>
<td>252</td>
<td>424</td>
<td>374</td>
</tr>
<tr>
<td>Intel SSD 335 SSD</td>
<td>39</td>
<td>222</td>
<td>344</td>
<td>354</td>
</tr>
<tr>
<td>Seagate Savvio 15K.3 HDD</td>
<td>2</td>
<td>2</td>
<td>223</td>
<td>223</td>
</tr>
</tbody>
</table>

---

Small requests are being realized as sequential operations.
SSD VS HDD COST

1TB ~ $150 on average
~1.5 cents / GB

10x more expensive
~15 cents / GB

The density is increasing.
Managing I/O devices is a significant part of OS!

Disk drives: storage media with specific geometry

SSDs: Pages, Blocks

Filesystems: OS provided API to access disk

Simple FS: FS layout with SB, Bitmaps, Inodes, Datablocks

FFS: Split simple FS into groups. Key idea: put inode, data close to each other

LFS: Puts data where it’s fastest to write, hope future reads cached in memory

https://www.eecs.harvard.edu/~margo/papers/usenix95-lfs/supplement/

FSCK, Journaling
DISTRIBUTED SYSTEMS
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 → Turing award winner

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?

- Increase throughput
- Connectivity
  - Share or send data
- Specialized machines → better performance
- Increase fault tolerance → Tolerate failures

SSD / CPU

more performance

google.com

× if 1 machine fails
WHY GO DISTRIBUTED?

More computing power

More storage capacity

Fault tolerance

Data sharing
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
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

very lightweight

file $\rightarrow$ socket

USA

Socket

Internet

Client

Server

write 1
write 2
write 3
read 1
read 2
read 3
100s of ms
RAW MESSAGES: UDP

Advantages

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

Disadvantages

– More difficult to write applications correctly
NOT A QUIZ?

Course feedback: https://aefis.wisc.edu
TCP: Transmission Control Protocol

Using software to build reliable logical connections over unreliable physical connections

interface

links, machines unreliable

mostly commonly used protocol
TECHNIQUE #1: ACK

Sender
[send message]
[recv ack]

Receiver
[recv message]
[send ack]

query "Wisconsin"

OK - I got the query

Ack: Sender knows message was received

What to do about message loss?
TECHNIQUE #2: TIMEOUT

**Sender**
- [send message]
- [start timer]
- … waiting for ack …
- [timer goes off]
- [send message]
- [recv ack]

**Receiver**
- [recv message]
- [send ack]
Adaptively configure timeout \( \rightarrow 640 \)

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!
Exactly once semantics → LOST ACK PROBLEM → retry

Sender
[send message] ← [recv message] ← [recv ack]
[timeout]
[send message] ← [ignore message]
[recv ack]

Receiver
[recv message] ← [send ack]
[send ack]
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 $K \leq N$, Msg K is already delivered, ignore it
- if $K = N + 1$, first time seeing this message
- if $K > N + 1$?

\[ \text{buffer it until N+1 arrives} \]
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 → Software abstraction
Remote Procedure Call

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!
```c
int main(…) {
    int x = foo("hello");
}

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

```c
int foo(char *msg) {
    …
}
```

```c
void foo_listener() {
    while(1) {
        recv, call foo
    }
}
```
```c
int main(...) {
    int x = foo("hello");
}

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

\[ \text{int / string / struct} \]
\[ \text{stream of bytes} \]

\[ \text{serialize} = 4 \text{ bytes} \]
\[ \text{serialize} = \text{char as a byte} \]

- big endian
- little endian
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]

Receiver
[recv]
[ack]
[exec call]
...
[return]
[tcp send]
RPC OVER UDP

Strategy: use function return as implicit ACK

Piggybacking technique

What if function takes a long time? then send a separate ACK
NEXT STEPS

Distributed Filesystems