PERSISTENCE: LFS, DISTRIBUTED SYSTEMS

Shivaram Venkataraman
CS 537, Spring 2020
Project 5: Due on Thursday!
AEFIS feedback
P4a grades → Pizza
Optional project

→ No Mip days

Poll is promising
Due May 1st
~ 60-70% response rates
30 students!

1. I read this very closely ⇒ changes
2. Department takes this input

Thank you!
How to design a filesystem that performs better for small writes?

What are some basic building blocks for systems that span across machines?
RECAP
### FS STRUCTS

<table>
<thead>
<tr>
<th>S</th>
<th>IB</th>
<th>DB</th>
<th>I</th>
<th>I</th>
<th>I</th>
<th>I</th>
<th>I</th>
<th>I</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>48</td>
<td></td>
<td></td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>56</td>
<td></td>
<td></td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

- **Devices → Disks → RAID**
- **Filesystems**
- **LFS**
- **LFS**

**Layout of data on disk**

Data and metadata layout on disk.
LOG STRUCTURED FILE SYSTEM

Motivation:
- Growing gap between sequential and random I/O performance
- RAID-5 especially bad with small random writes

Idea: use disk purely sequentially

Design for writes to use disk sequentially – how?
WRITES

flush rings to disk
- full segment is flushed at once
- coalesce imap updates
- 30s you could have multiple imap parts in a segment

`imap inum -> Disk location j -> A4`

```
imap [N1 : N2]
where N1, N2 < N
```

`Append data blocks to segment` → large contiguous block that we’ll write to disk

Inodes are also written to the log
1. Read the Checkpoint region
2. Read all imap parts, cache in mem
3. To read a file:
   1. Lookup inode location in imap
   2. Read inode
   3. Read the file block

open / root directory
Inode num = 2
imap [2] = addr
When moving data blocks, copy new inode to point to it
When move inode, update imap to point to it

old inode is garbage after we append a block

60% 10% 95% 35% 95%

compact 2 segments to one

Copy live data from data segments write it out to a new segment

not leave any holes inside segments
(N, T) = SegmentSummary[A];

inode = Read(imap[N]);

if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
CRASH RECOVERY

What data needs to be recovered after a crash?
- Need imap (lost in volatile memory)

Better approach?
- Occasionally save to checkpoint region the pointers to imap pieces

How often to checkpoint?
- Checkpoint often: random I/O
- Checkpoint rarely: lose more data, recovery takes longer
- Example: checkpoint every 30 secs
CRASH RECOVERY

disk:

memory:

ptrs to imap pieces

checkpoint

imap [ ]

inodes

disk blocks

imap parts

3

inodes
q, y

imap ftw

my

active flocks

'segment at nose

read sequentially

after the tail

then you get segments not in CTFT

tail after last checkpoint

after last checkpoint

I active segment at most.
CHECKPOINT SUMMARY

Checkpoint occasionally (e.g., every 30s)

Upon recovery:
- read checkpoint to find most imap pointers and segment tail
- find rest of imap pointers by reading past tail

What if crash during checkpoint?
CHECKPOINT STRATEGY

Have two checkpoint regions
Only overwrite one checkpoint at a time
Use checksum/timestamps to identify newest checkpoint
Managing I/O devices is a significant part of OS!
Disk drives: storage media with specific geometry
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
QUIZ 29

block 100: ["." 0, "." 0, ("foo" 1)] // a data block
block 101: [size=1, ptr=100, type=d] // an inode
block 102: [size=0, ptr=-, type=r] // an inode
block 103: [imap: 0->101, 1->102] // a piece of the imap

create ("/foo")

garbage

block 104: [SOME DATA] // a data block
block 105: [SOME DATA] // a data block
block 106: [size=2, ptr=104, ptr=105, type=r] // an inode
block 107: [imap: 0->101, 1->106] // a piece of the imap

write ("/foo", buffer, 1024)
DISTRIBUTED SYSTEMS
HOW DOES GOOGLE SEARCH WORK?
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

widely used today
WHY GO DISTRIBUTED?

More computing power

More storage capacity

Fault tolerance

Data sharing

 locality → machine close → you can communicate

Disk failures → Power outage

1 machine × 10

4 core 2.5 GHz, 1 TB

40 cores! , 10 TB!

google.com

google.in
NEW CHALLENGES

System failure: need to worry about partial failure

Communication failure: links unreliable
- bit errors
- packet loss
- node/link failure

Why are network sockets less reliable than pipes?
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
Advantages
- Lightweight
- Some applications make better reliability decisions themselves (e.g., video conferencing programs)

Disadvantages
- More difficult to write applications correctly
TCP: Transmission Control Protocol

Using software to build reliable logical connections over unreliable physical connections
Technique #1: **ACK**

**Sender**
- [send message]
- [recv ack]

**Receiver**
- [recv message]
- [send ack]

---

**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]

message dropped
retry after 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

Goal: Exactly once delivery of a message

Sender
[send message]

[recv ack]

Receiver
[recv message]

[send ack]

Dropped in the middle

Duplicate the message or
discard the message received

[ignore message]

[send ack]

retry

[timeout]

[send message]
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 $K \leq N$, Msg K is already delivered, ignore it
- if $K = N + 1$, first time seeing this message
- if $K > N + 1$ ?
TCP: Transmission Control Protocol

Ack, Timeout, reliable delivery

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

Reliable protocol on top of an unreliable network
NOT A QUIZ?

Course feedback: https://aefis.wisc.edu
COMMUNICATIONS OVERVIEW

Raw messages: UDP

Reliable messages: TCP

Remote procedure call: RPC
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!
int main(…) {
    int x = foo("hello");
}

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

Machine A

Machine B

int foo(char *msg) {
    ...
}

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

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

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)
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
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]
RPC OVER UDP

Strategy: use function return as implicit ACK

Piggybacking technique

What if function takes a long time?
then send a separate ACK
Distributed Filesystems
P5 is due on Thursday!