PERSISTENCE: LFS, DISTRIBUTED SYSTEMS

Shivaram Venkataraman

CS 537, Spring 2020

ADMINISTRIVIA

Project 5: Due on Thursday!

AEFIS feedback

P4a grades

Optional project

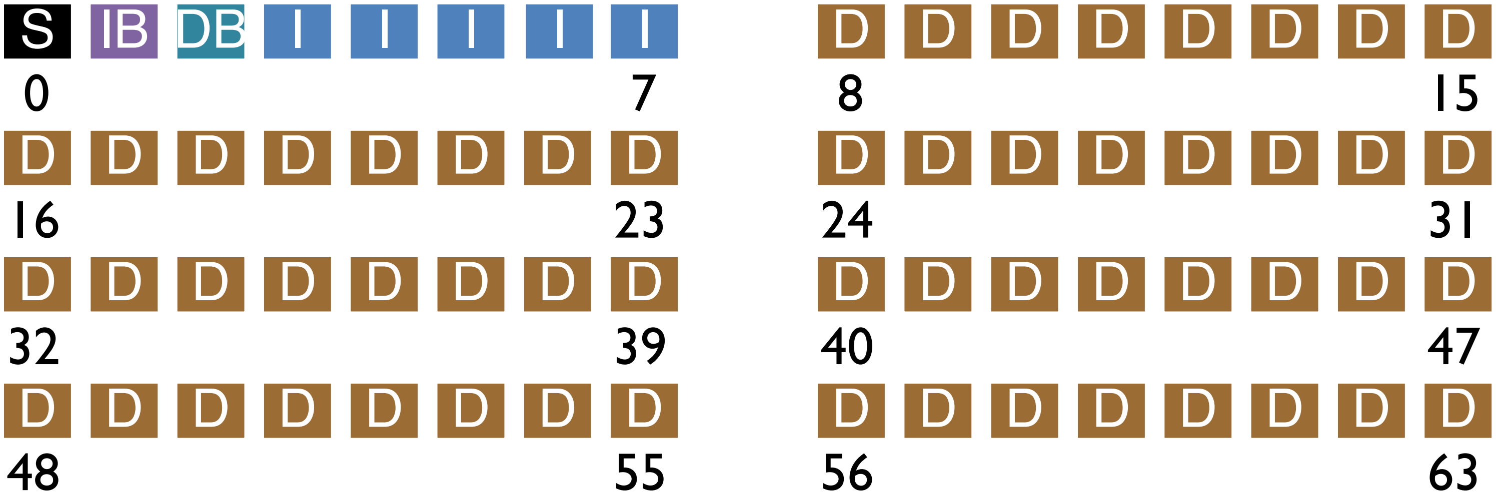
AGENDA / LEARNING OUTCOMES

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



LOG STRUCTURED FILE SYSTEM

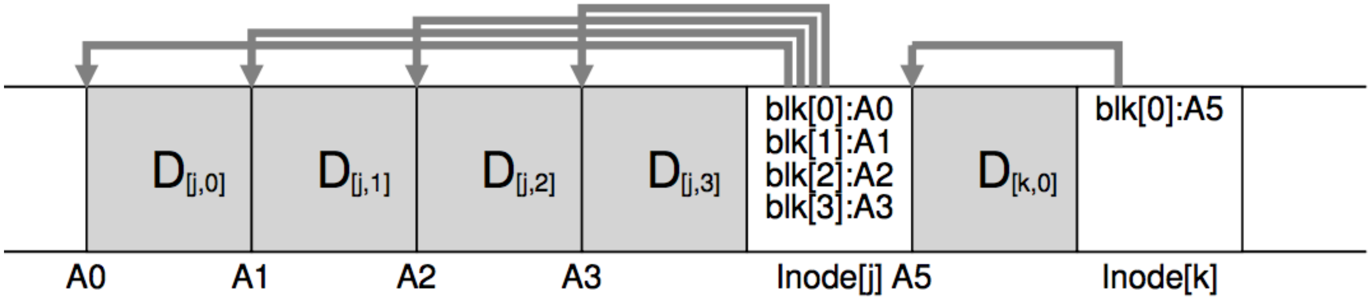
Motivation:

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

Idea: use disk purely sequentially

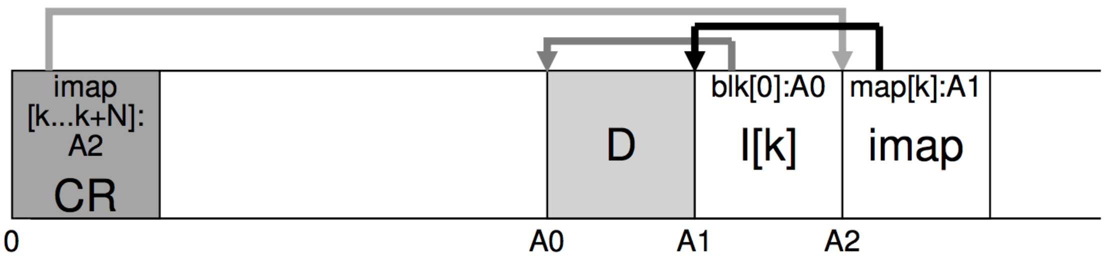
Design for writes to use disk sequentially – how?

WRITES



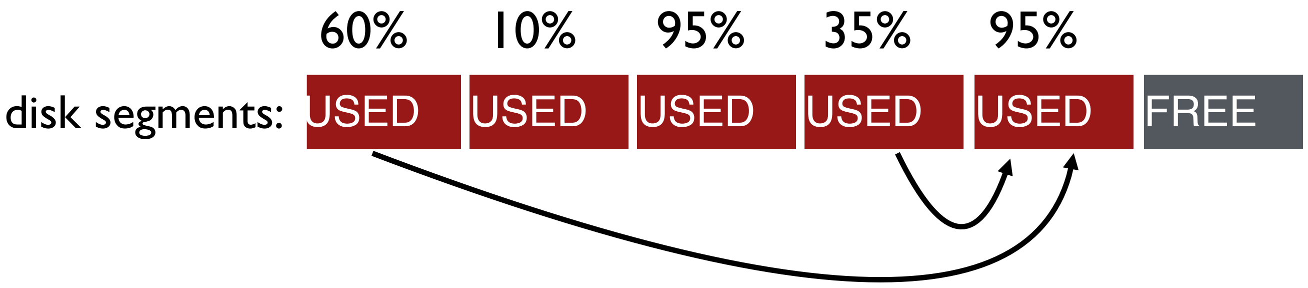
$D_{[j,0]}$
$D_{[j,1]}$
$D_{[j,2]}$
$D_{[j,3]}$
blk[0]:A0
blk[1]:A1
blk[2]:A2
blk[3]:A3
$D_{[k,0]}$
blk[0]:A5
A0
A1
A2
A3
Inode[j] A5
Inode[k]

READING IN LFS



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

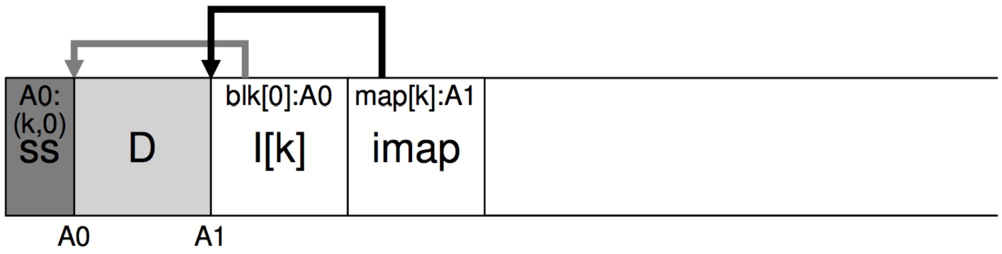
GARBAGE COLLECTION



compact 2 segments to one

When moving data blocks, copy new inode to point to it
When move inode, update imap to point to it

SEGMENT SUMMARY




```
(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)

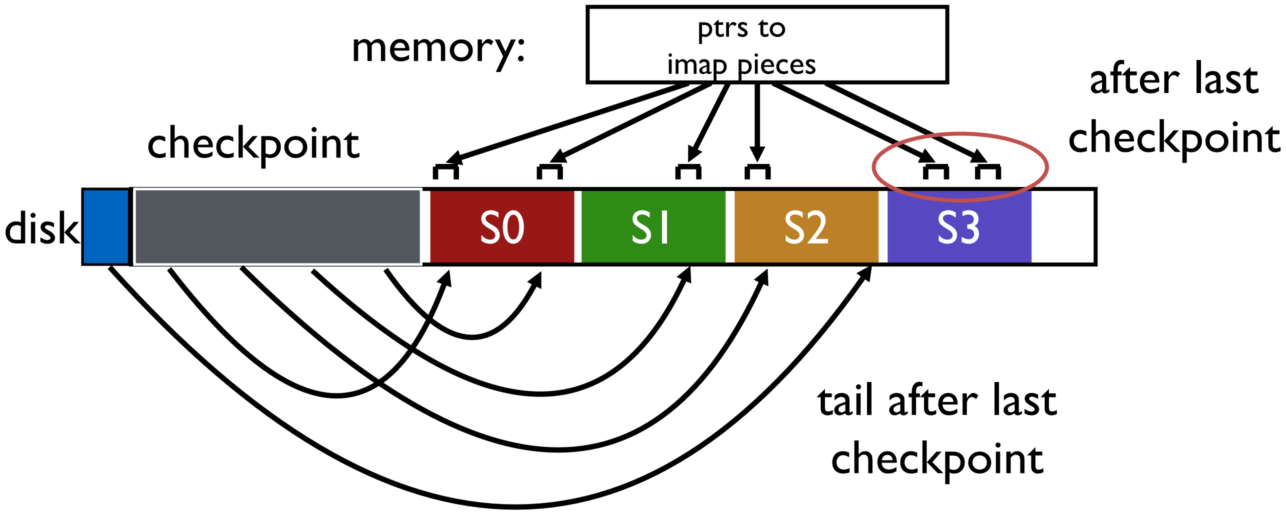
Better approach?

- 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



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

PERSISTENCE SUMMARY

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

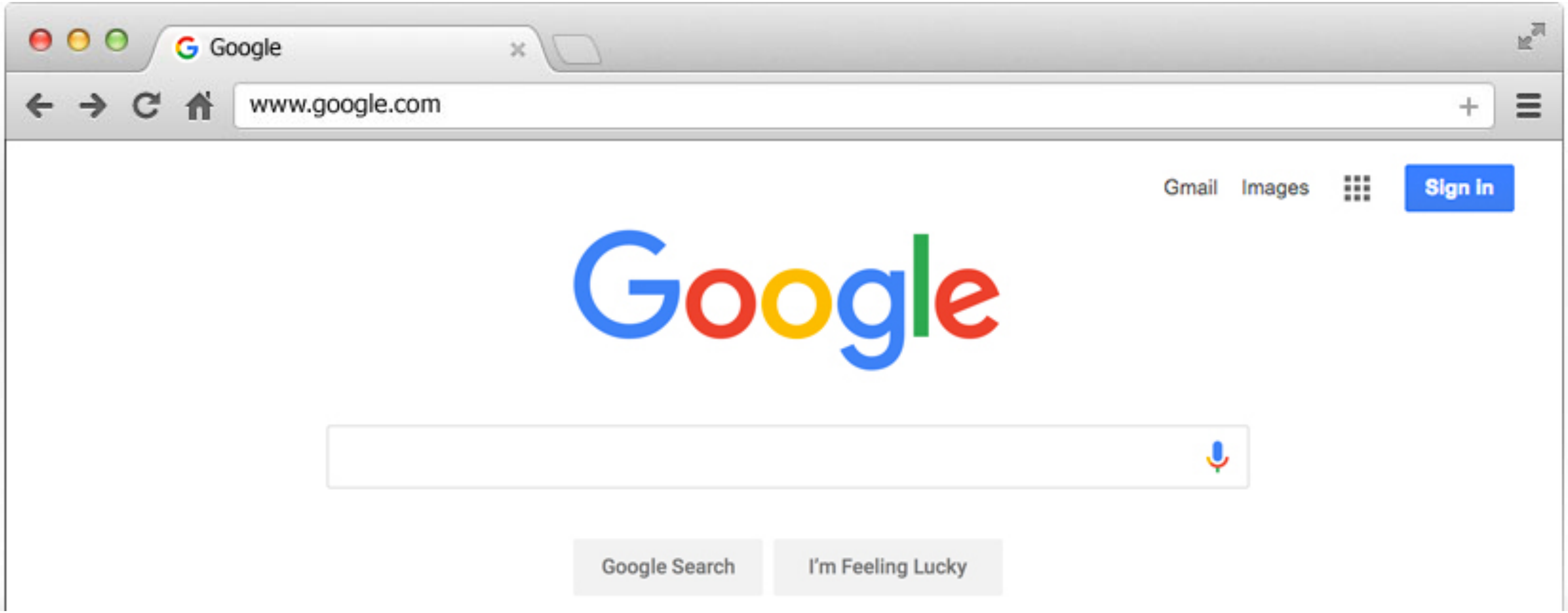
https://tinyurl.com/cs537-sp20-quiz29

```
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
```

```
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
```

DISTRIBUTED SYSTEMS

HOW DOES GOOGLE SEARCH WORK?



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

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

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

[send message]

[recv ack]

Receiver

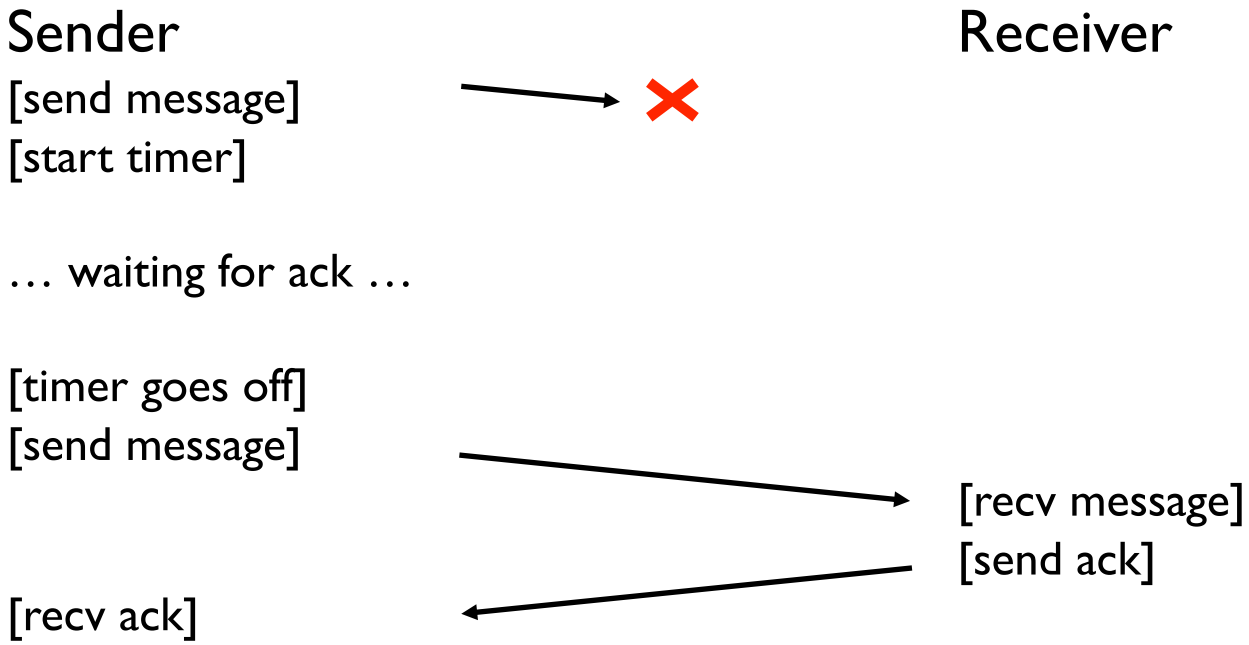
[recv message]

[send ack]

Ack: Sender knows message was received

What to do about message loss?

TECHNIQUE #2: TIMEOUT



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

Sender

[send message]

Receiver

[recv message]

[send ack]

[timeout]

[send message]

[ignore message]

[send ack]

[recv ack]

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

NOT A QUIZ?

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

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”);
}

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

client wrapper

Machine B

```
int foo(char *msg) {
        …
}

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

server wrapper

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

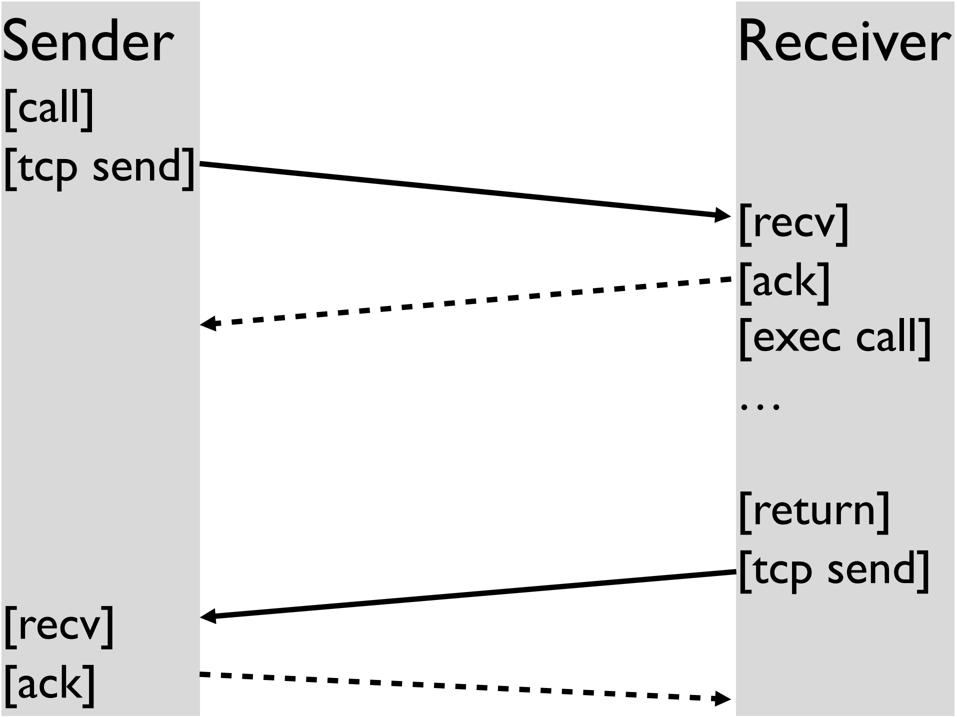
WRAPPER GENERATION: POINTERS

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]

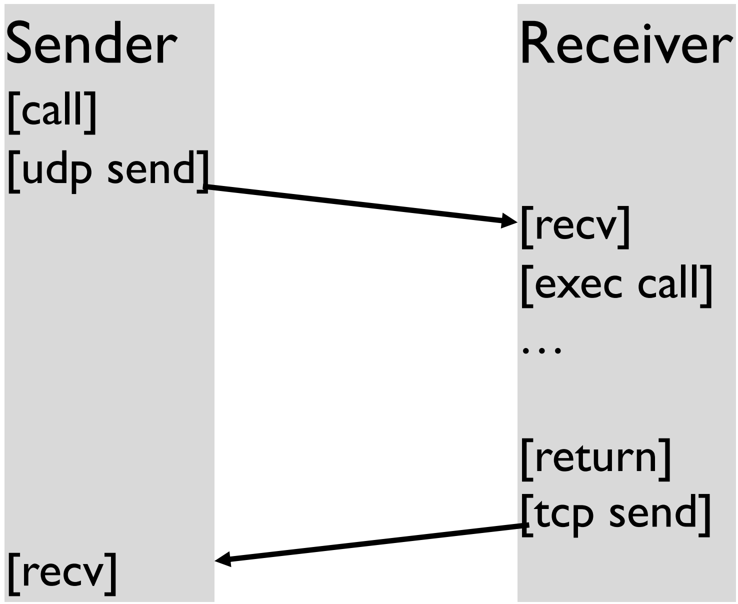
RPC OVER UDP

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

P5 is due on Thursday!