Hello!

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

Program (Write)

Page → 4KB

Erase

Block - 256KB

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

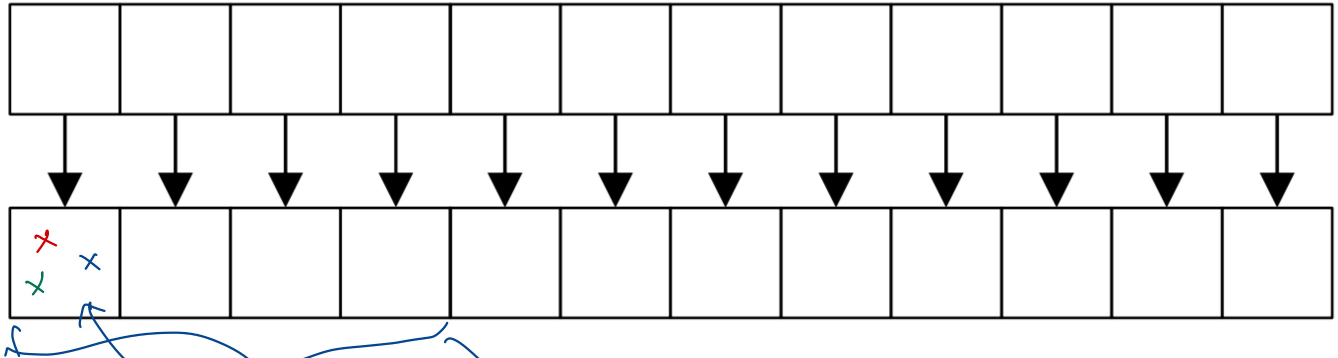
Program (i.e., write) a page: Change selected 1s to 0s

- Cost is 200 to1400 microseconds
- Faster than erasing a block, but slower than reading a page

FTL: DIRECT MAPPING

user or the FS sees

Logical pages



Physical pages

Flash Pages

wear out !

Erase block frequently

Cons?

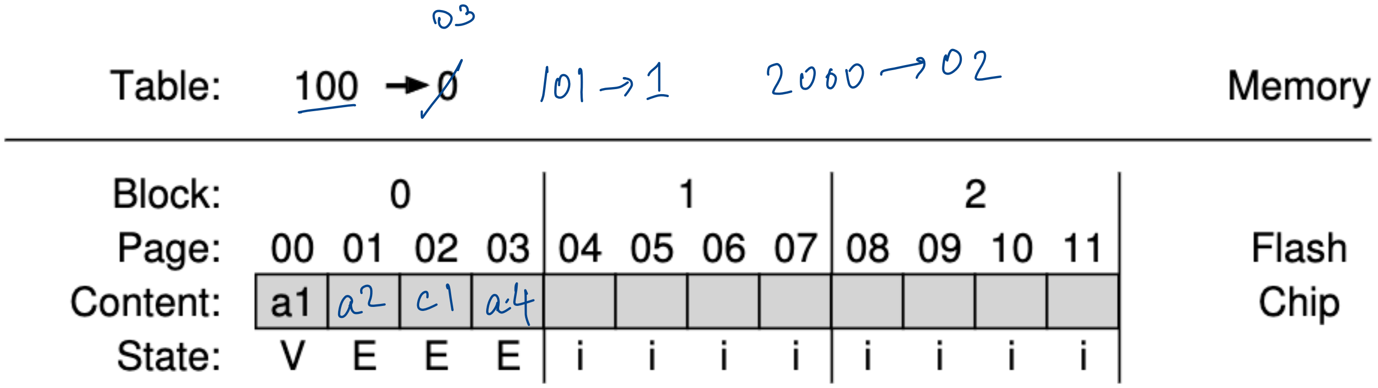
Write amplification

number of physical writes for 1 logical write

No wear-leveling

FTL: LOG-BASED MAPPING

Idea: Treat the physical blocks like a log



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)

Table: 100 → 4 101 → 5 2000 → 2 2001 → 3 — Memory

Block:	0				1				2				Flash Chip
Page:	00	01	02	03	04	05	06	07	08	09	10	11	
Content:	a1	a2	b1	b2	c1	c2							
State:	V	V	V	V	V	V	E	E	i	i	i	i	

block

Table: 100 → 4 101 → 5 2000 → 6 2001 → 7 — Memory

Block:	0				1				2				Flash Chip
Page:	00	01	02	03	04	05	06	07	08	09	10	11	
Content:					c1	c2	b1	b2					
State:	E	E	E	E	V	V	V	V	i	i	i	i	

↳ live pages end of the log

SSD VS HDD PERFORMANCE

small requests

Actual being realized as sequential operations

SLC vs. MLC

FTL layer

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

SSDs are good at random access

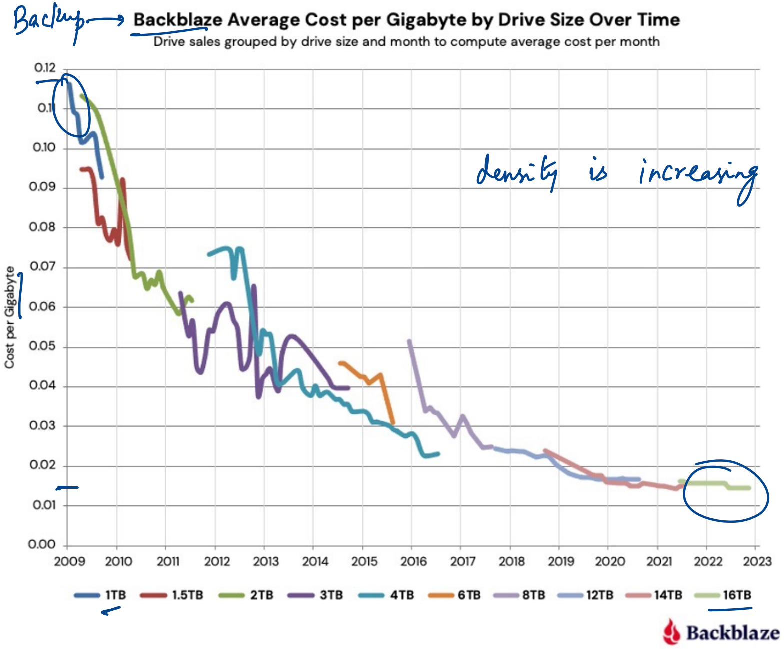
large granularity

much lower

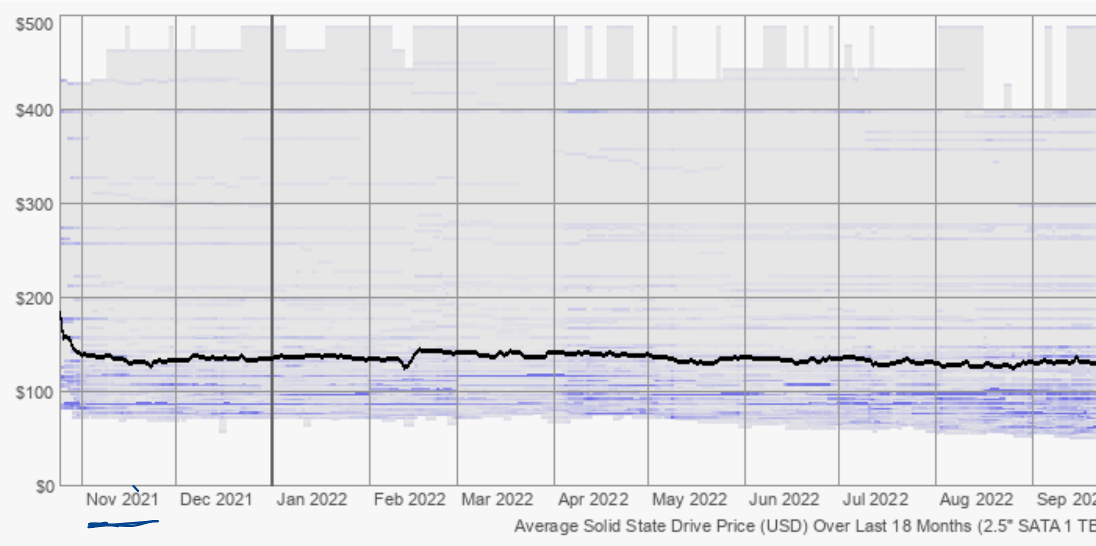
random write

SSD VS HDD COST

Backup →



density is increasing



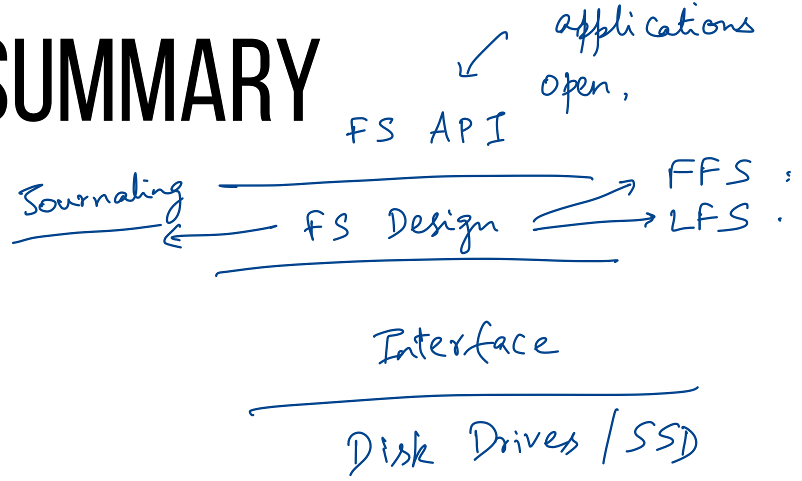
= ~1.5 cents / GB

1TB ~ $150 on average

~15 cents / GB

10x more expensive

PERSISTENCE SUMMARY



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

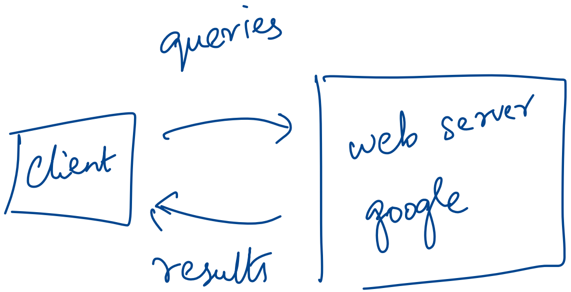
WHAT IS A DISTRIBUTED SYSTEM?

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

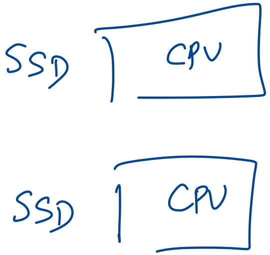
Examples:

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



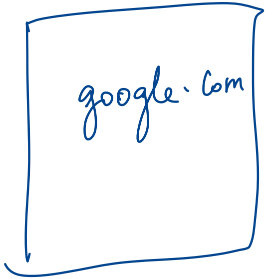
WHY GO DISTRIBUTED?

- Increase throughput



more performance

- Connectivity → share or send data
- Specialized machines → better performance
- Increase fault tolerance
  ↳ Tolerate failures



× if 1 machine fails

WHY GO DISTRIBUTED?

More computing power

More storage capacity

Fault tolerance

Data sharing

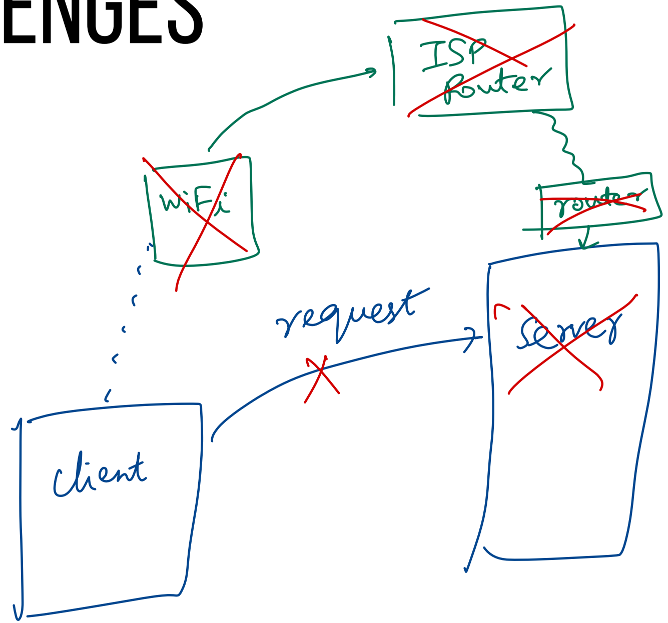
NEW CHALLENGES

System failure: need to worry about partial failure

Communication failure: links unreliable

- bit errors
- packet loss
- node/link failure

just this request was lost



COMMUNICATION OVERVIEW

Raw messages: UDP

Reliable messages: TCP

Remote procedure call: RPC

Acronyms

640 Networking

RAW MESSAGES: UDP

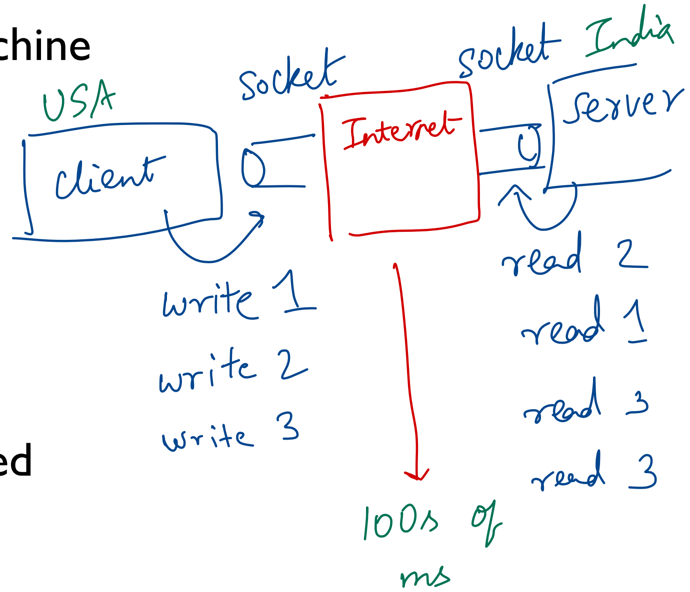
very lightweight

file ⟷ socket

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

| data | ck sum |

RAW MESSAGES: UDP

Advantages

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

Disadvantages

- More difficult to write applications correctly

Server $\xrightarrow{\text{video bytes}}$ client

reliably → retry same frame

skip frames → responsiveness

NOT A QUIZ?

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

RELIABLE MESSAGES: LAYERING STRATEGY

TCP: Transmission Control Protocol → mostly commonly used protocol

Using software to build

reliable logical connections over unreliable physical connections

interface

links, machines unreliable

TECHNIQUE #1: ACK → Acknowledgement

Sender

[send message]

query "Wisconsin"

Receiver

[recv message]

[send ack]

OK - I got the query

[recv ack]

↓

message was successful!

Ack: Sender knows message was received

What to do about message loss?

TECHNIQUE #2: TIMEOUT

send("wisconsin")

TCP

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

30s

retry →

same message

send returns

TIMEOUT

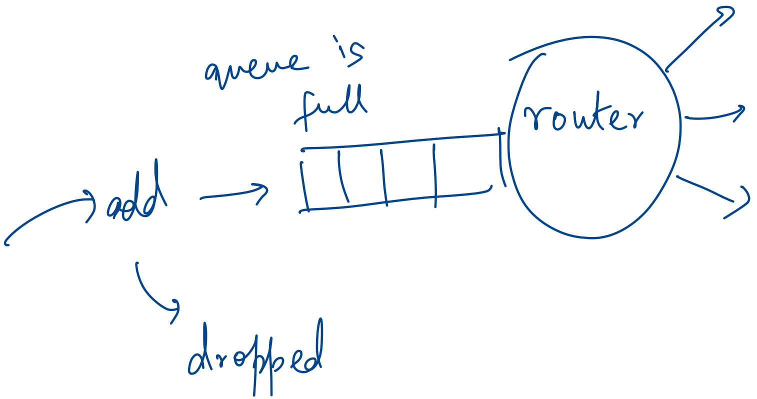
Adaptively configure timeout

↳ 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 → ~~duplicated~~

dropped → retry

LOST ACK PROBLEM

Sender

[send message]

Receiver

[recv message]

[send ack]

dropped

[timeout]

[send message]

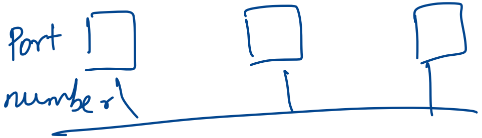
messages need an ID

Can be used to ignore messages already delivered

[ignore message]

[send ack]

[recv ack]

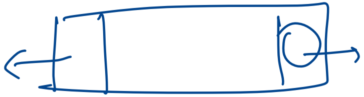


SEQUENCE NUMBERS

avoid duplication

prevent reordering

Packets = 64 bytes



Sequence numbers

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

N → last message delivered = Recv

2052 =

msg 1
seq_no : 2052

msg 2
seq_no : 2053

K
incoming message

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 ?

buffer it until N+1 arrives

deliver & increment K

TCP

TCP: Transmission Control Protocol

Most popular protocol based on seq nums

Buffers messages so arrive in order

Timeouts are adaptive

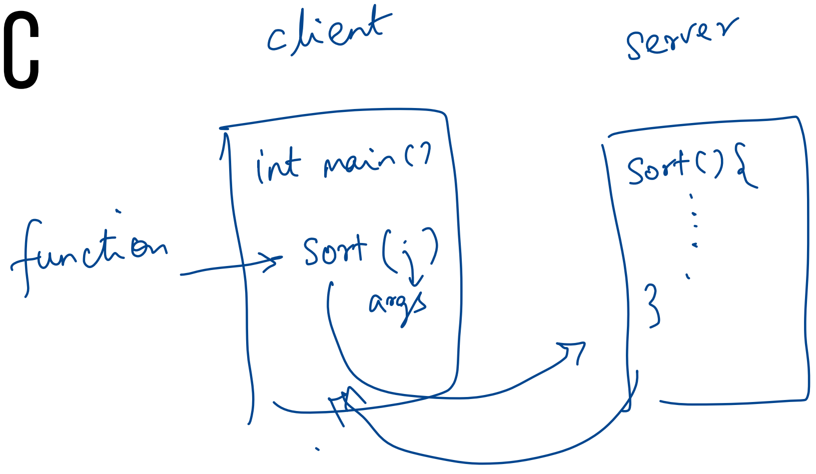
COMMUNICATIONS OVERVIEW

Raw messages: UDP

Reliable messages: TCP

Remote procedure call: RPC → Software abstraction

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 library → provides abstraction

RPC

Machine A

```
int main(...) {
        int x = foo("hello");
}

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

Clients' address space

same arguments

send args as msg

send result as a msg

Machine B

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

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

do computation result

RPC

client

Machine A

```
int main(...) {
      int x = foo("hello");
}
```

Developers!

users

Machine B

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

client wrapper

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

server wrapper

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

part of RPC library

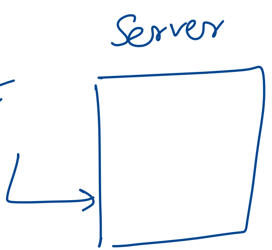
value : int → 4 bytes

ptr : int*

↳ (* int) : int → serialize

RPC TOOLS

1 thread per request



RPC packages help with two components

(1) Runtime library

- Thread pool
  so that you can handle multiple requests at same time
- Socket listeners call functions on server
  serializing / de-serializing arguments / results

(2) Stub generation

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

WRAPPER GENERATION

int / string / struct

serializing → stream of bytes

serialize = 4 bytes

serialize = char as a byte

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

big endian

little endian

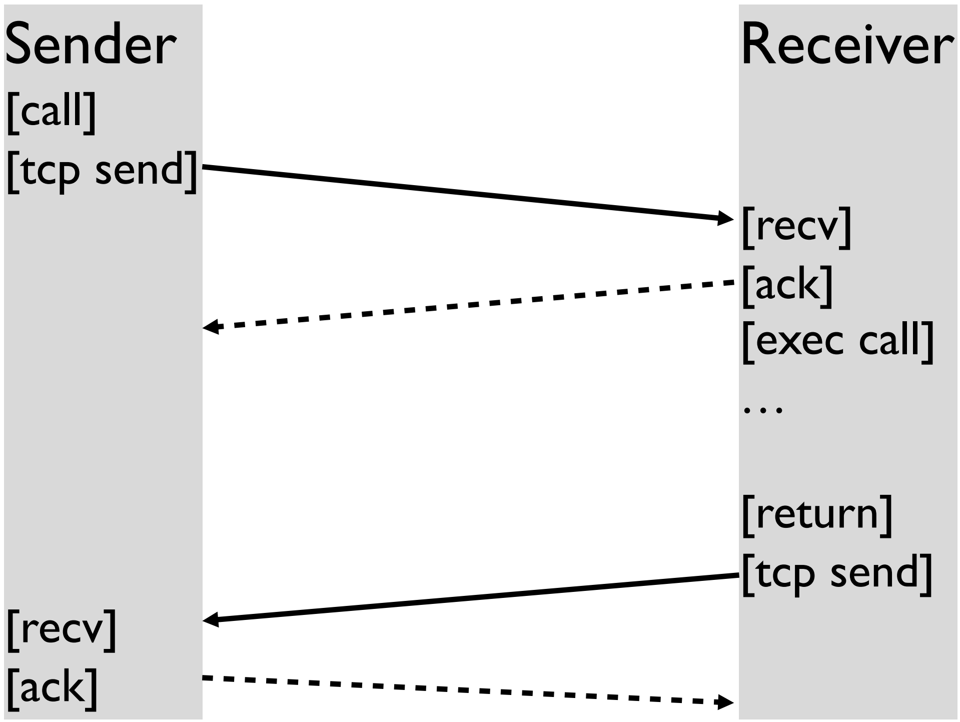
WRAPPER GENERATION: POINTERS

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]

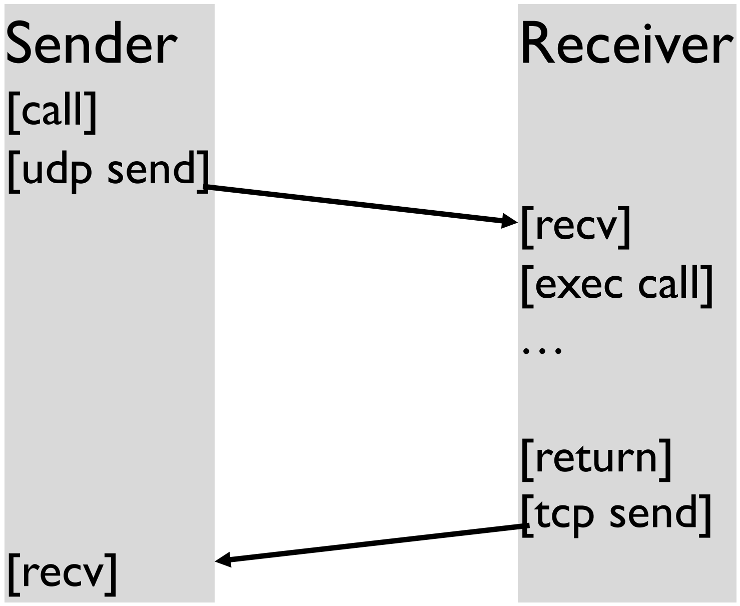
RPC OVER UDP

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