P56
Disks and Networks

Friday 15:00-17:00 SE02

Andrew.W.Moore@cl.cam.ac.uk

A reminder please.

From here on, **do not measure performance of unoptimized code** -- too much of the time is wasted on things that should not be there, and the results are unrelated to deployed, production, optimized code. Mostly measures memory access time!

Disk timing

The Four Fundamental Resources

CPU ☜

Memory ☜

Disk/SSD

Network

Jeff Dean's 'Numbers Everyone Should Know'

L1 cache reference	**0.5 ns**	**O(1) ns**
Branch mispredict	3.0 ns	O(10) ns
L2 cache reference	**4.0 ns**	**O(10) ns**
Mutex lock/unlock	17.0 ns	O(10) ns
Main memory reference	**100.0 ns**	**O(100) ns**
Compress 1K bytes with Zippy/Snappy	2,000.0 ns	O(1) us
Read 1 MB sequentially from memory	**4,000.0 ns**	**O(10) us**
Send 2K bytes over 1 Gbps network	20,000.0 ns	O(10) us
Read 1 MB sequentially from SSD	62,000.0 ns	O(10) us
Round trip within same datacenter	500,000.0 ns	O(1) ms
Read 1 MB sequentially from spinning disk	947,000.0 ns	O(10) ms
Disk seek	3,000,000.0 ns	O(10) ms
Read 1 MB sequentially from network	10,000,000.0 ns	O(10) ms
Send packet CA->Netherlands->CA	150,000,000.0 ns	O(100) ms

Taken from a mashup of two slides stacks and some updated numbers from URL below
http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/people/jeff/stanford-295-talk.pdf
http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf
https://kousiknath.medium.com/must-know-numbers-for-every-computer-engineer-6338a12c292c
https://norvig.com/21-days.html#answers

The Four Fundamental Resources

CPU

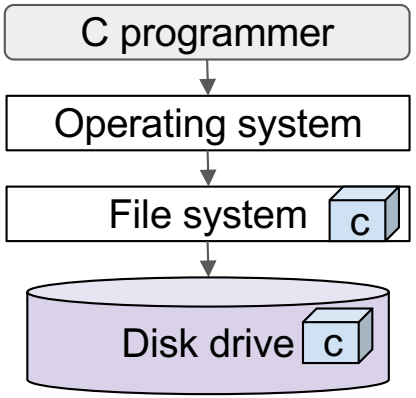
Memory

Disk/SSD ☜

Network

About hard disks

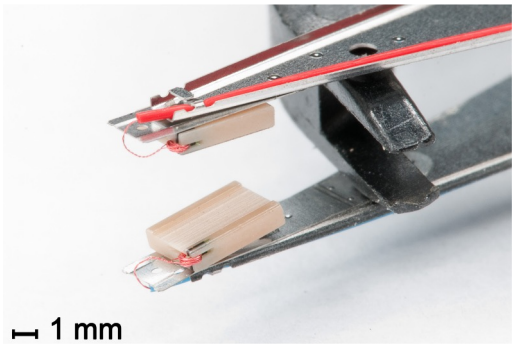
Several layers of software involved, and at least two caches

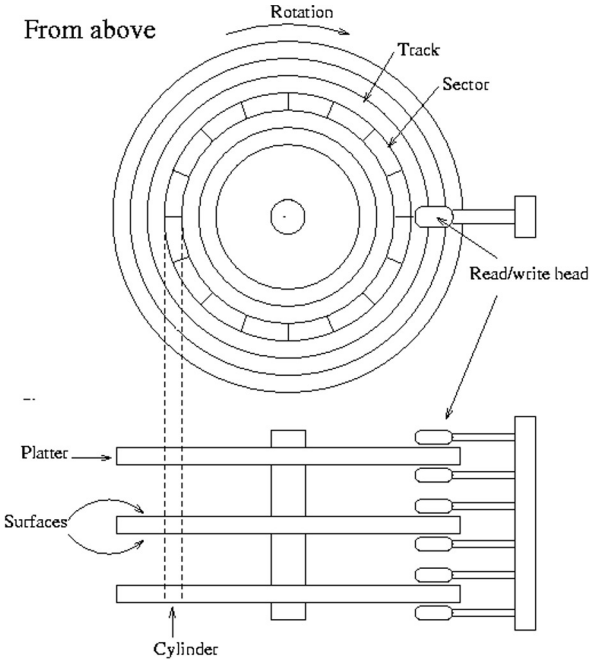


About hard disks

Read-write head per surface

Cylinders, tracks, blocks (sectors)





About hard disks

To reach a given logical block address, LBA, the drive will seek to the right track, switch to the right head, settle on the track, and then wait for the right block to rotate around under the read/write head.

Seeks run 4-15 msec, depending on distance from current head location
7200 rpm = 120 rps = 8.33 msec per revolution
5400 rpm = 90 rps = 11.11 msec per revolution

Accessing data that cross a track boundary involves switching to a different read head and settling on the next track, perhaps 1/8 to 1/4 of a revolution.

Jeff Dean's 'Numbers Everyone Should Know'

L1 cache reference	0.5 ns	O(1) ns
Branch mispredict	3.0 ns	O(10) ns
L2 cache reference	4.0 ns	O(10) ns
Mutex lock/unlock	17.0 ns	O(10) ns
Main memory reference	100.0 ns	O(100) ns
Compress 1K bytes with Zippy/Snappy	2,000.0 ns	O(1) us
Read 1 MB sequentially from memory	4,000.0 ns	O(10) us
Send 2K bytes over 1 Gbps network	20,000.0 ns	O(10) us
Read 1 MB sequentially from SSD	**62,000.0 ns**	**O(10) us**
Round trip within same datacenter	500,000.0 ns	O(1) ms
Read 1 MB sequentially from spinning disk	**947,000.0 ns**	**O(10) ms**
Disk seek	**3,000,000.0 ns**	**O(10) ms**
Read 1 MB sequentially from network	10,000,000.0 ns	O(10) ms
Send packet CA->Netherlands->CA	150,000,000.0 ns	O(100) ms

Taken from a mashup of two slides stacks and some updated numbers from URL below
http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/people/jeff/stanford-295-talk.pdf
http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf
https://kousiknath.medium.com/must-know-numbers-for-every-computer-engineer-6338a12c292c
https://norvig.com/21-days.html#answers

Disk reads

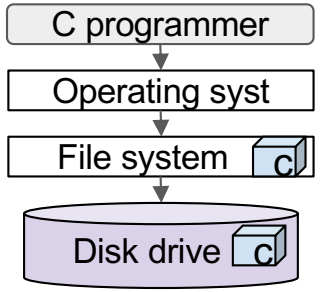
This week: Disk timing, problems to avoid

Problem: File system caches data. Write then immediate read may never touch disk

Answer: Defeat by opening O_DIRECT

Problem: File system maintains last-accessed time in directory file, doing extra disk work whenever file is accessed

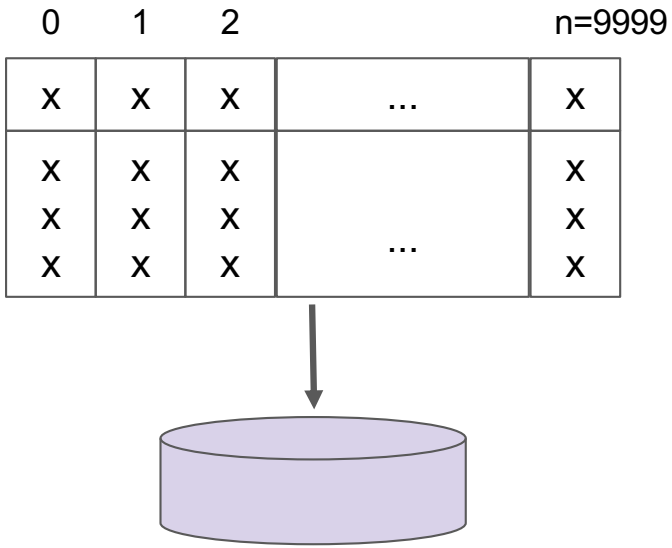
Answer: Defeat by opening O_NOATIME



Disk read timing, overall design

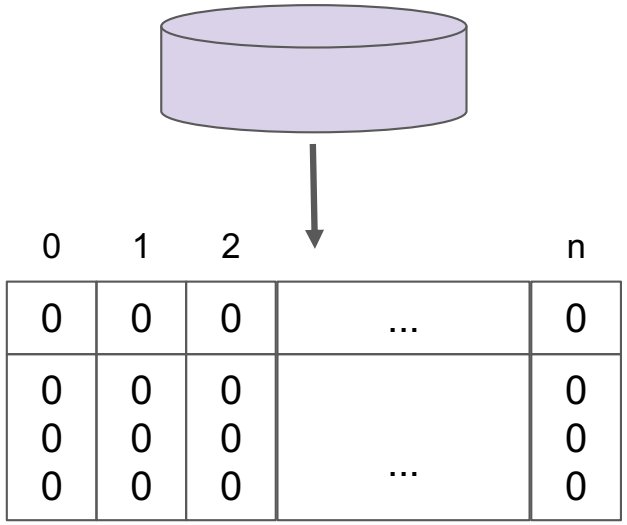
[0] Buffer in memory bigger than any expected on-disk cache

[1] Random nonzero bits in buffer, write to disk



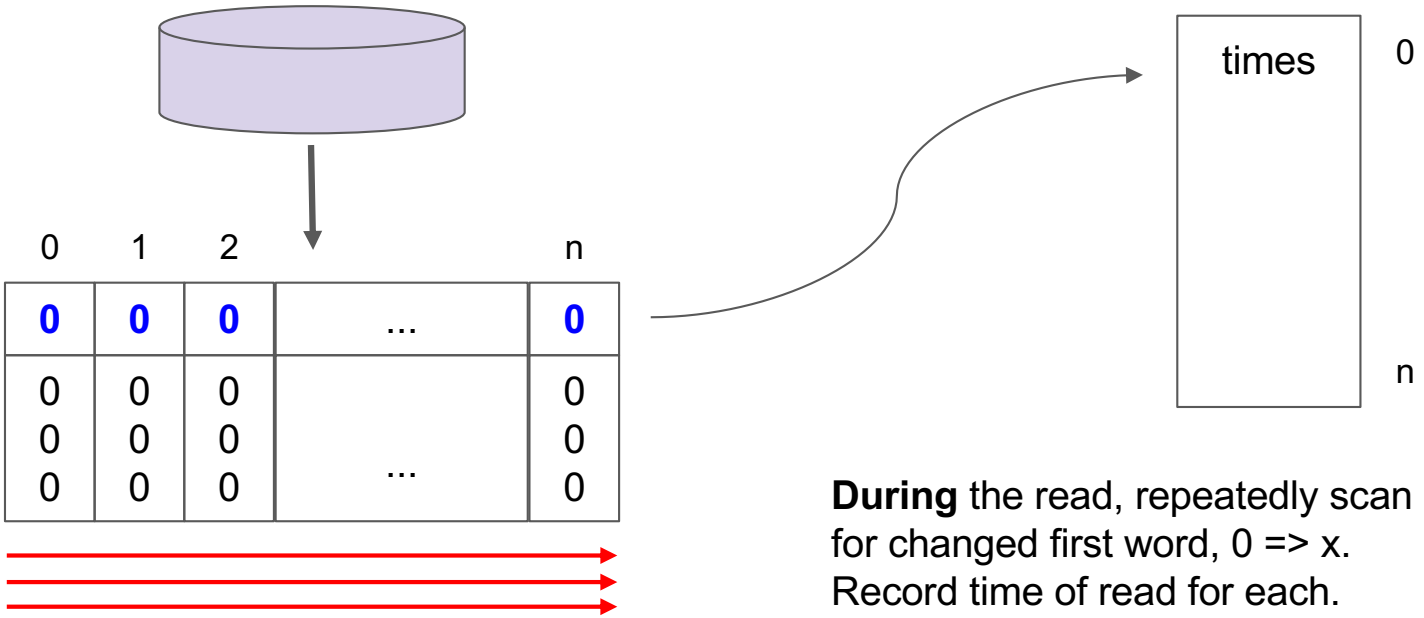
Disk read timing

[2] Zeros in buffer, **asynchronous** read from disk



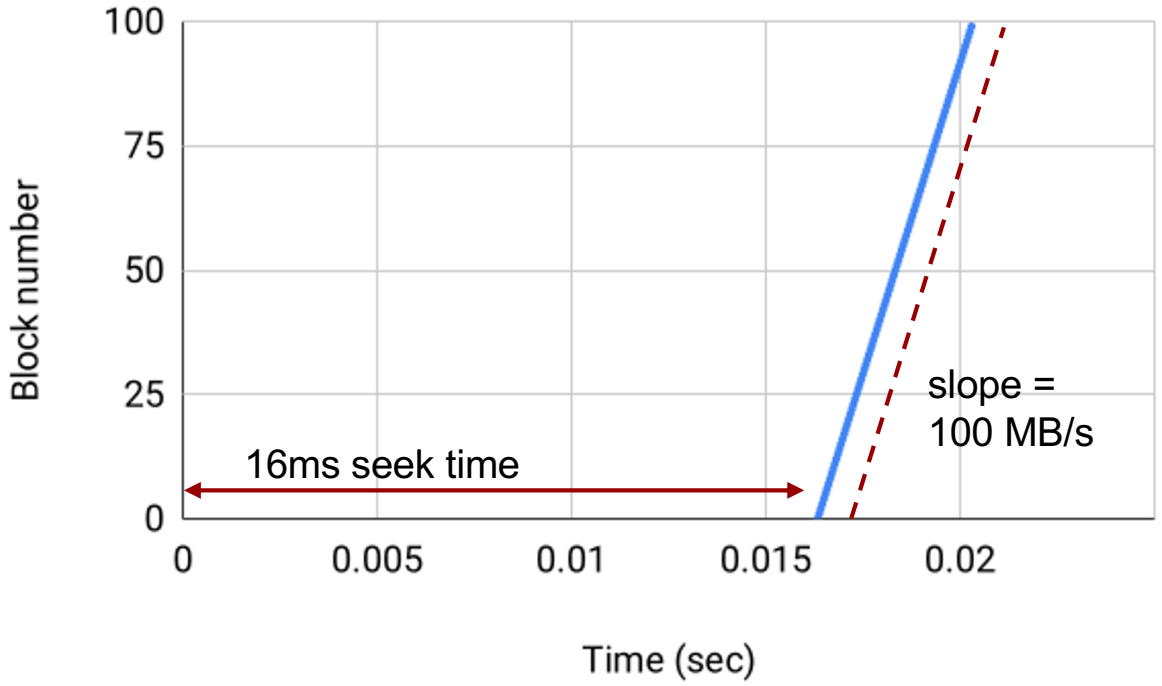
Disk read timing

[2] Zeros in buffer, **asynchronous** read from disk



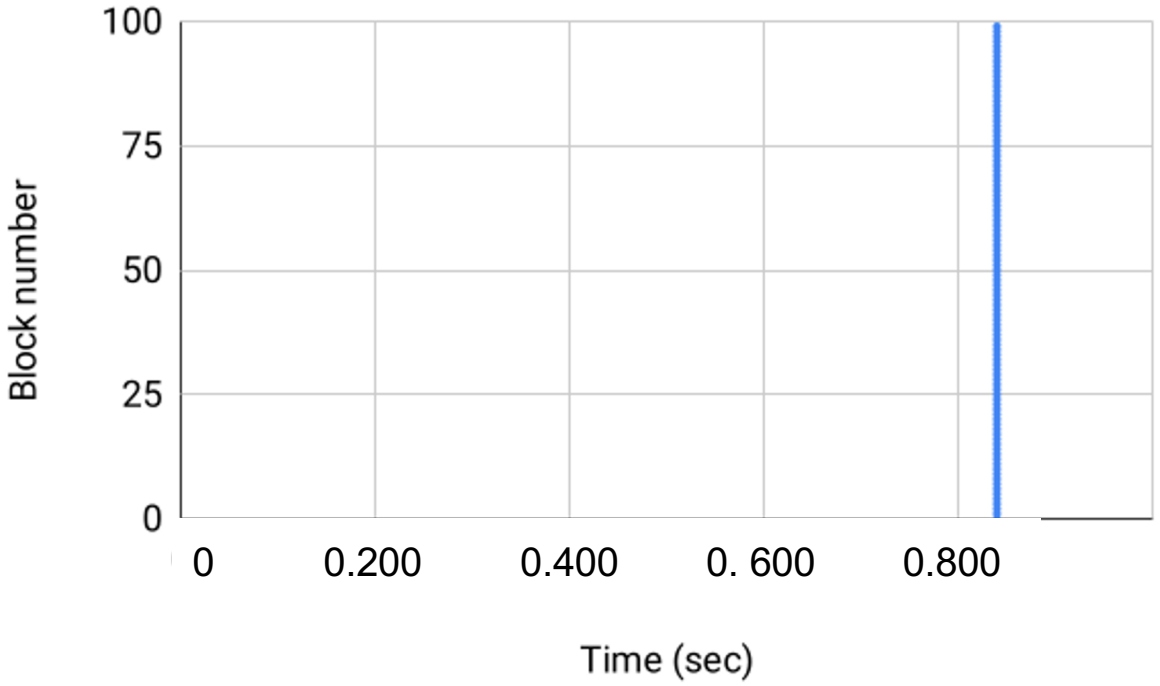
During the read, repeatedly scan for changed first word, 0 => x. Record time of read for each.

Possible disk read result -- seek, read

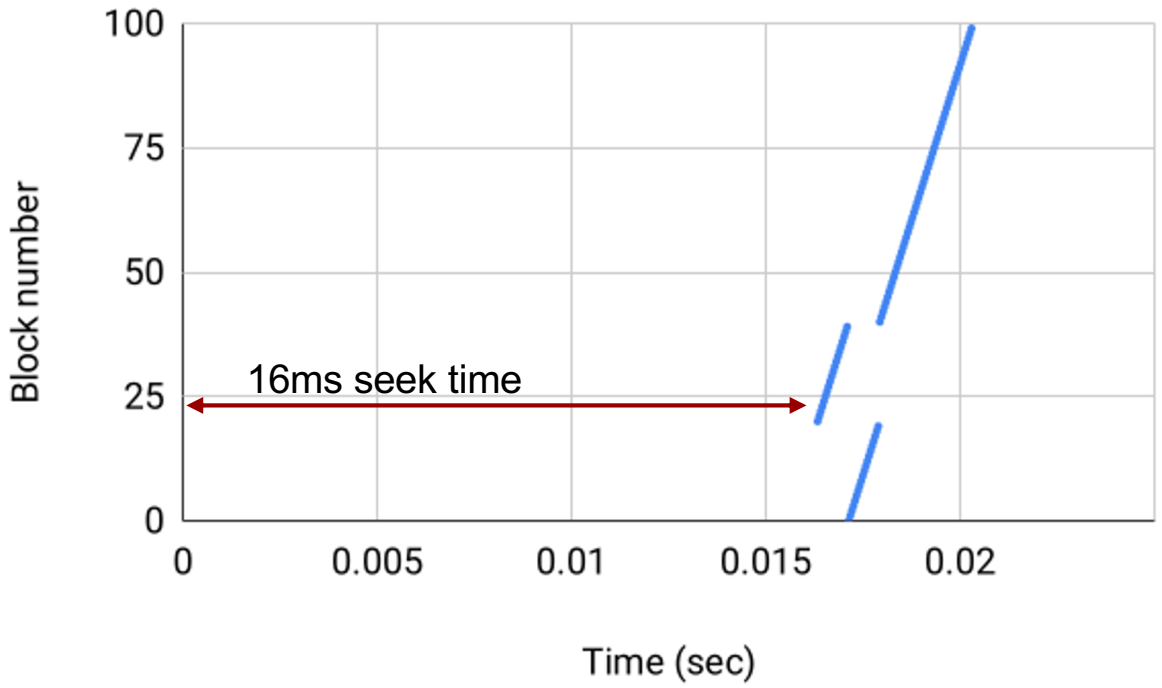


Possible disk read result

OS buffers, flips Page Table Entries



Possible disk read result -- start in middle, wrap, more



A little back-of-the-envelope

- **About how long should it take to transfer a single 4KB disk block?**

- At O(100) MB/sec, a disk transfers O(100) bytes in a microsecond, so 4KB should take roughly 40 usec.

- For our slow disks, it might be closer to 60 usec. Total time for 10,000 blocks should be about 600 msec.

- Communications limits SATAI/II/III are 1.5Gb/s 3Gb/s and 6Gb/s respectively

- USB 1.1/2.0/3.0 are 12Mb/s 480Mb/s and 5Gb/s

- Beware on the R.Pi 4 the USB ports are limited in total to 5Gb/s iirc.

A little back-of-the-envelope

How long will each iteration of our "look at the first word of each 4KB block" loop take?

Items are all 4KB apart, so will not spread out well in L1, L2, or L3 cache: just a few cache sets will be used -- likely *all cache misses*.

I/O hardware is simultaneously accessing the *same* data, so might be further delaying any imagined CPU cache behavior.

50ns/miss * 10,000 = ~500us/pass, but new disk blocks every ~60us

A little back-of-the-envelope

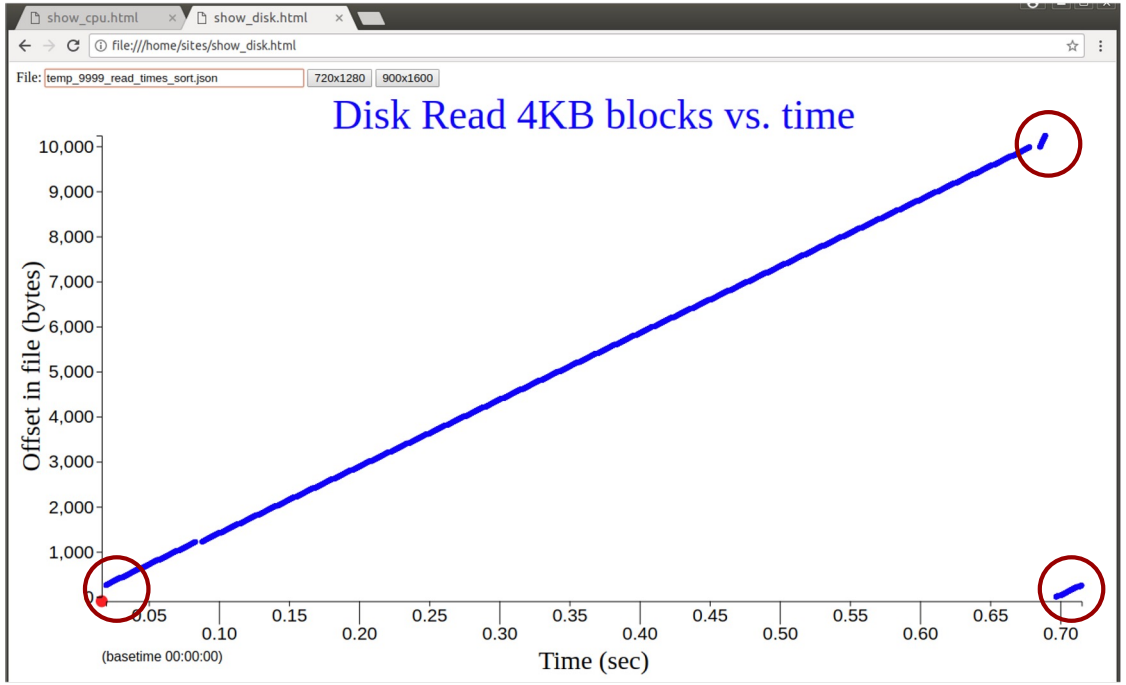
Do you see the problem?

To avoid giving crummy time resolution while also avoiding spending all the time in gettimeofday() system calls, we strike a balance and renew the time of day every 256 iterations of the 10,000 total, or roughly every 20 usec.

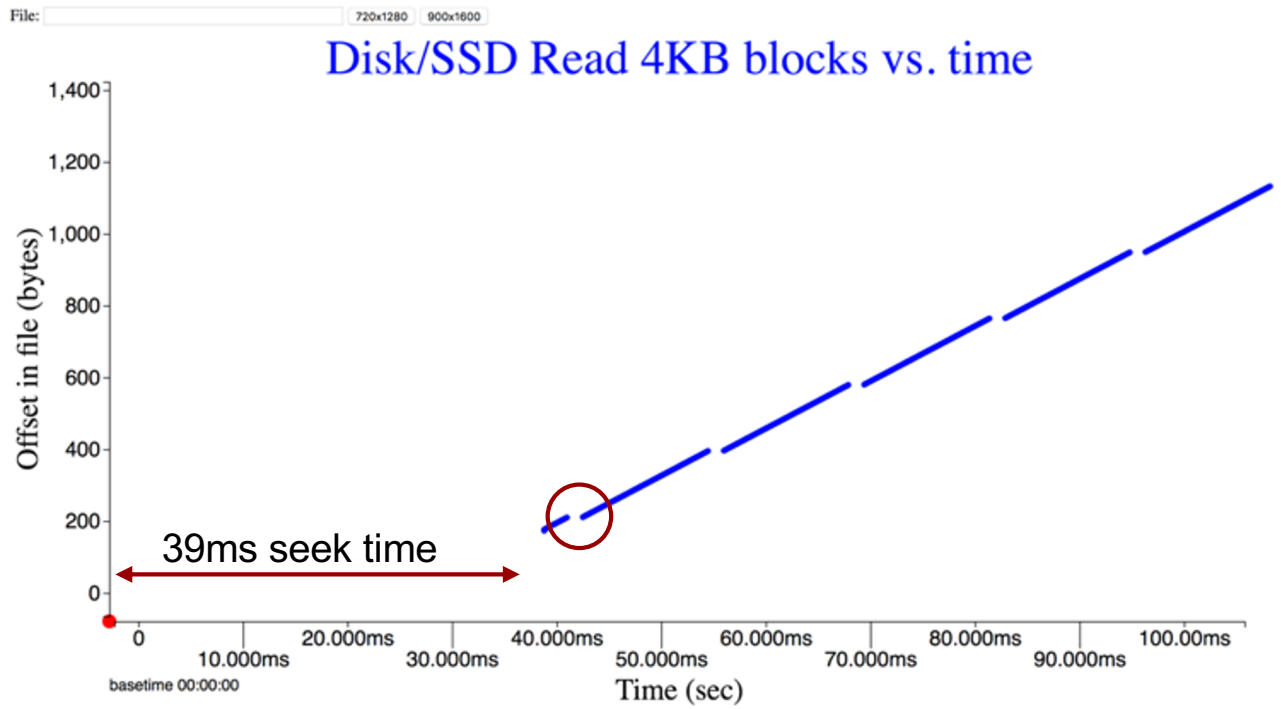
To get more accurate successive-block timings, we also look at the *expected* very-next-block every time around the loop instead of just once per 10,000 looks.

Disk read timing, an actual result

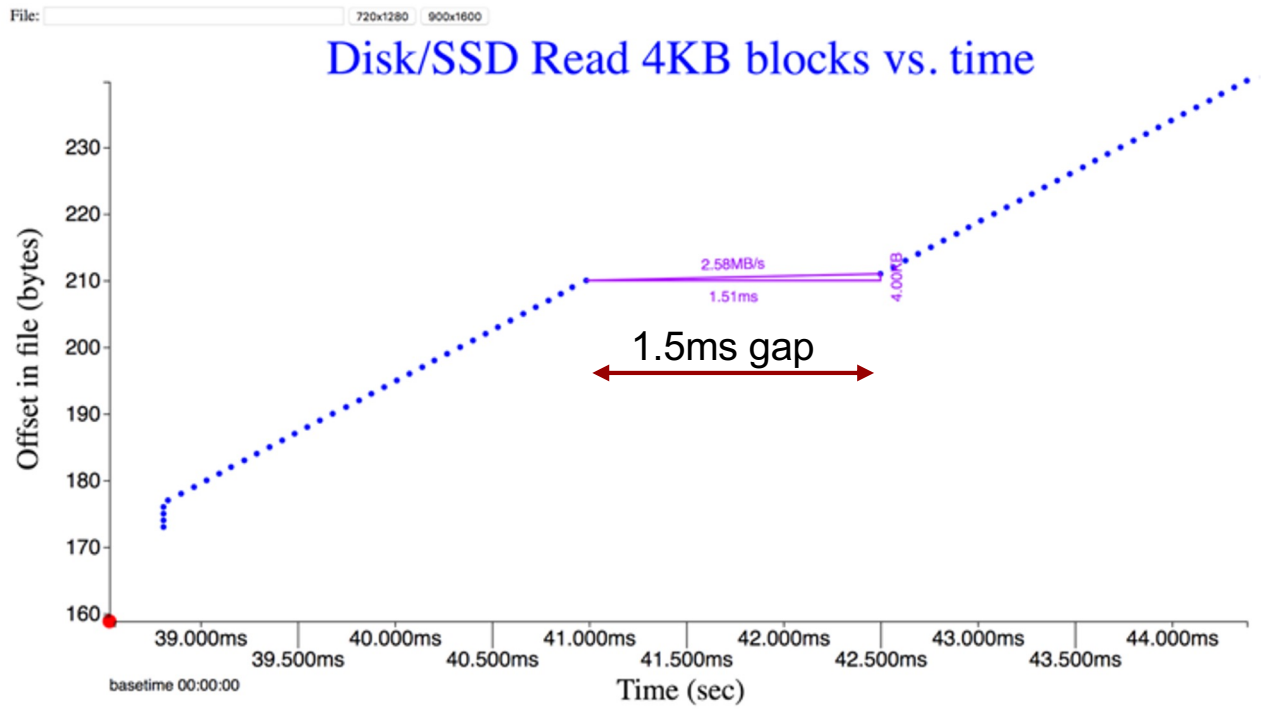
Plotted times look a bit like this, with each dot a 4KB block, x = sec, y = byte offset of block.



Disk read, Initial seek



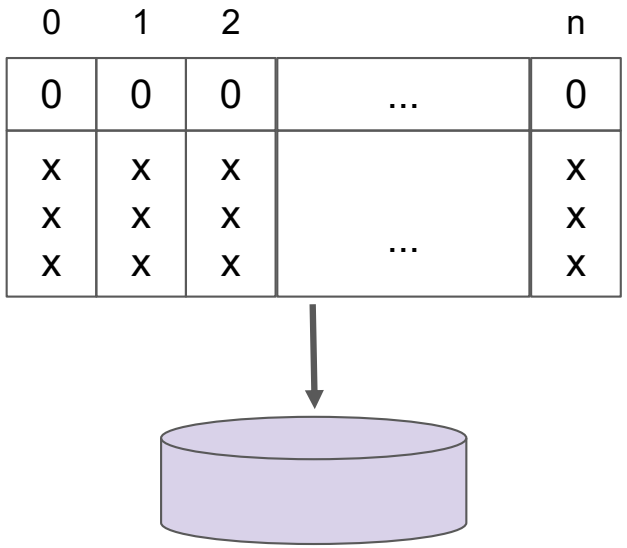
Disk read, Track-to-track seek/rotate/servo



Disk writes

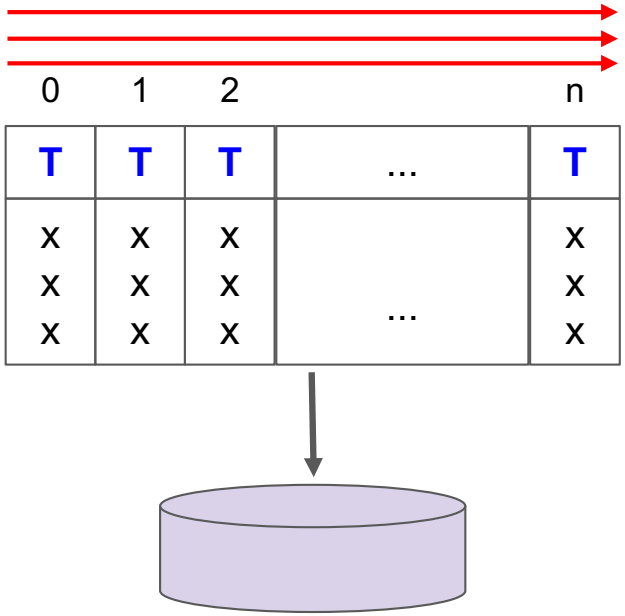
Disk write timing

[1] Random bits in buffer, zeros in first word, do **asynchronous** write to disk



Disk write timing, overall design

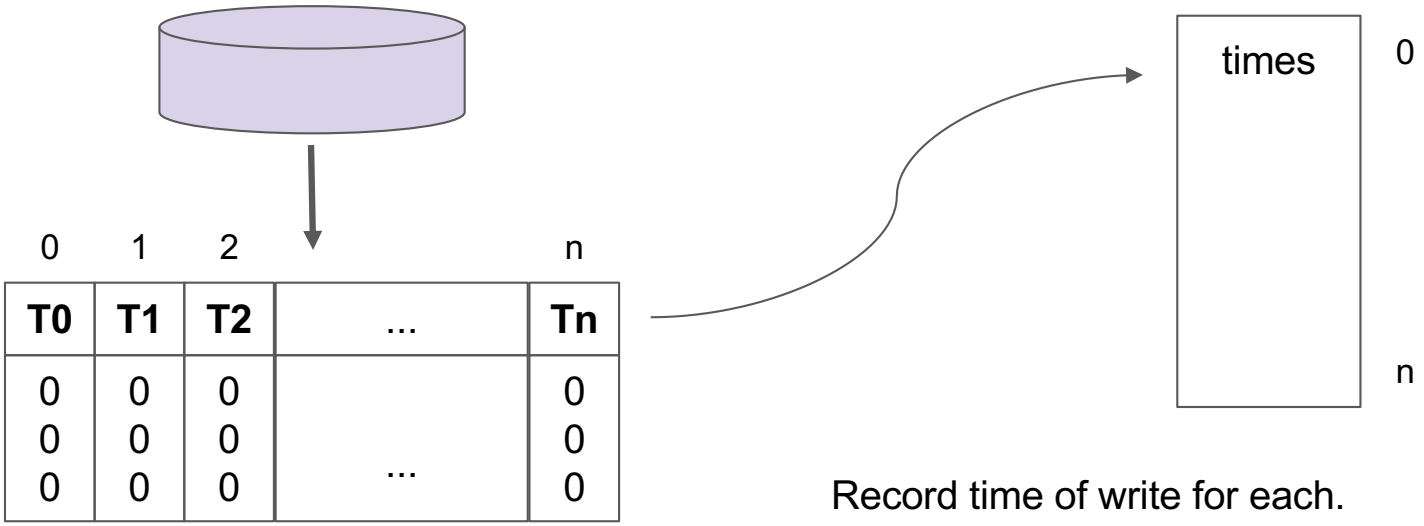
[2] Random bits in buffer, zeros in first word, do **asynchronous** write to disk



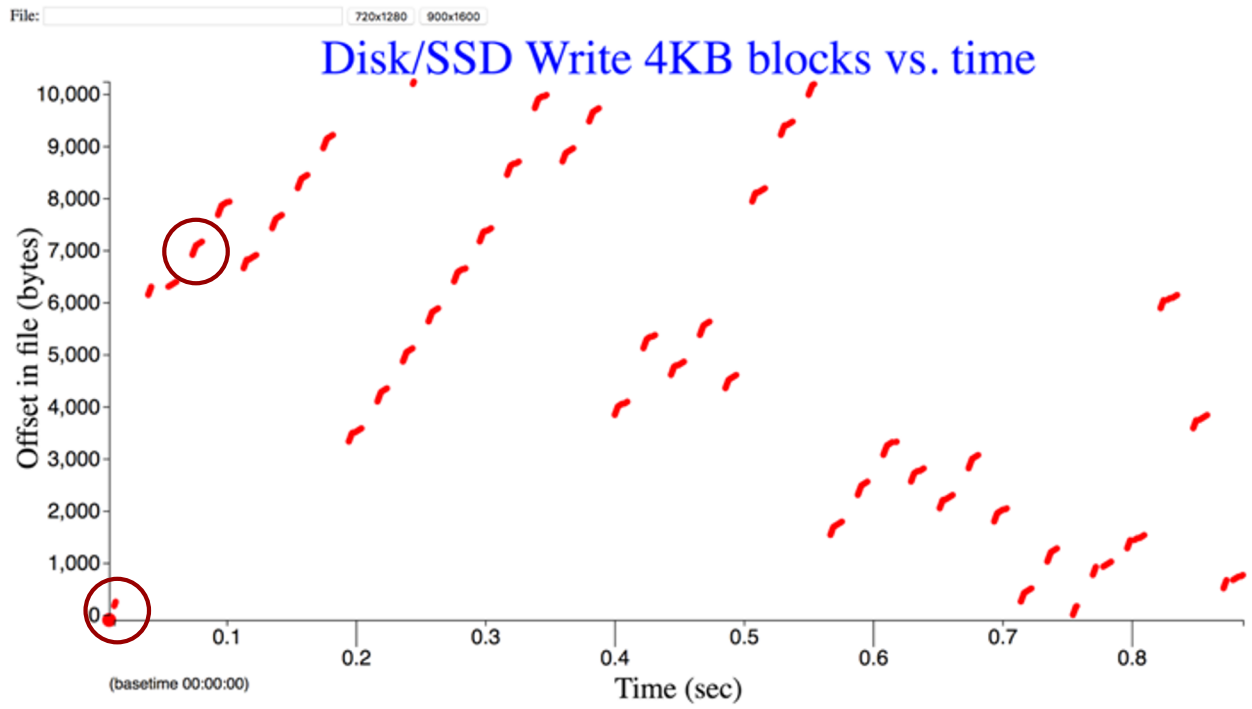
During the write, repeatedly write the current time into the first word.

Disk write timing

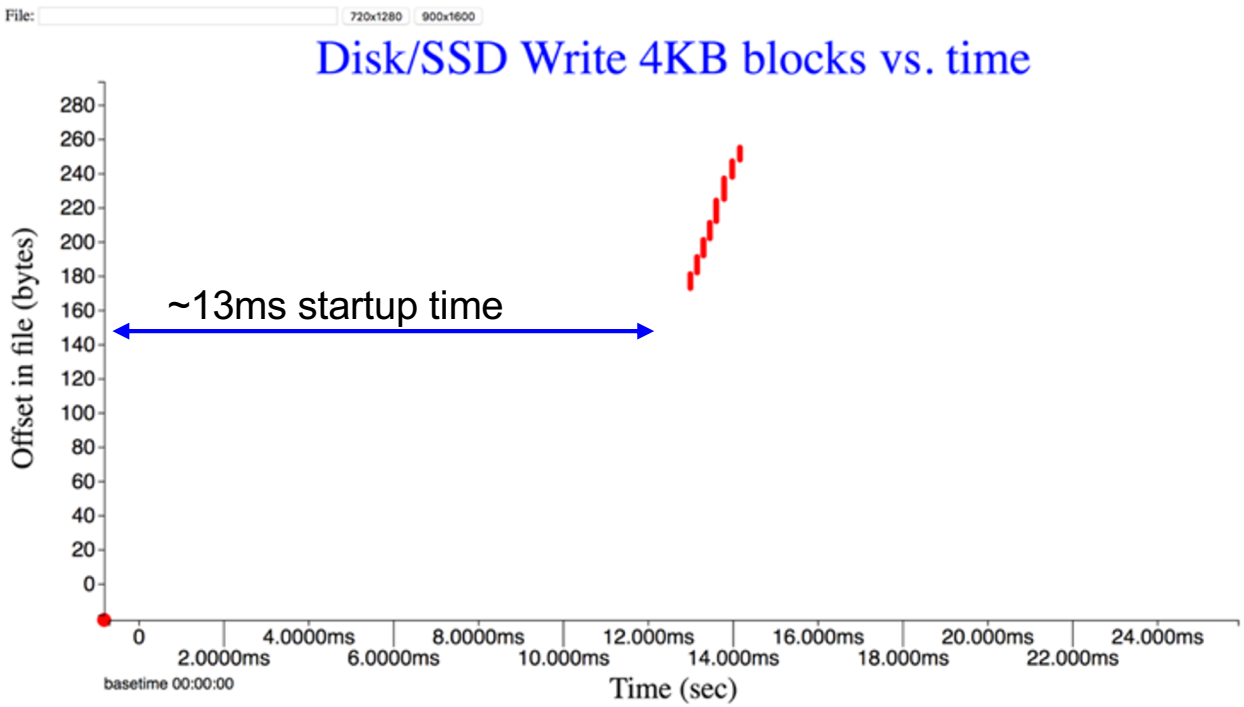
[3] Read back the file and see what time values got copied to disk



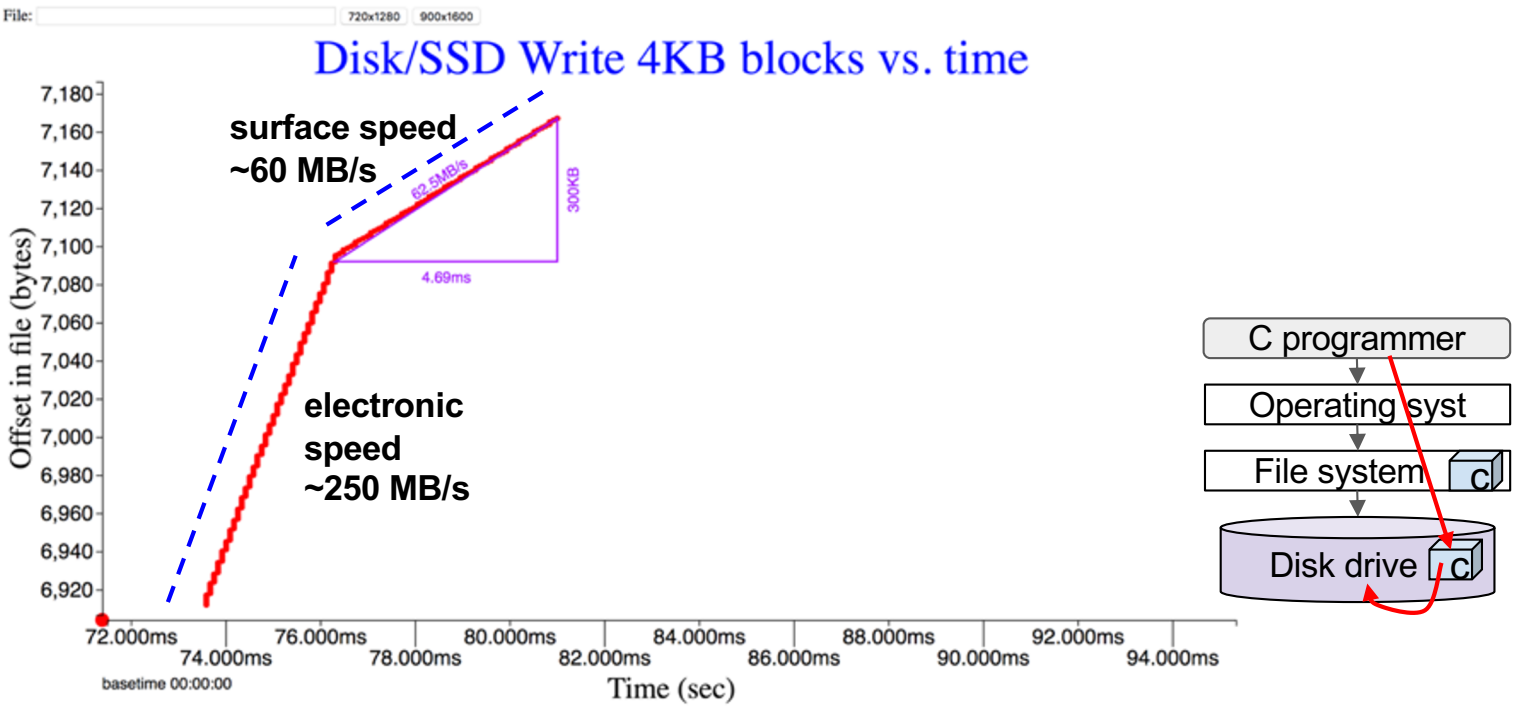
Disk write timing, an actual result



Disk write, startup to first block copied



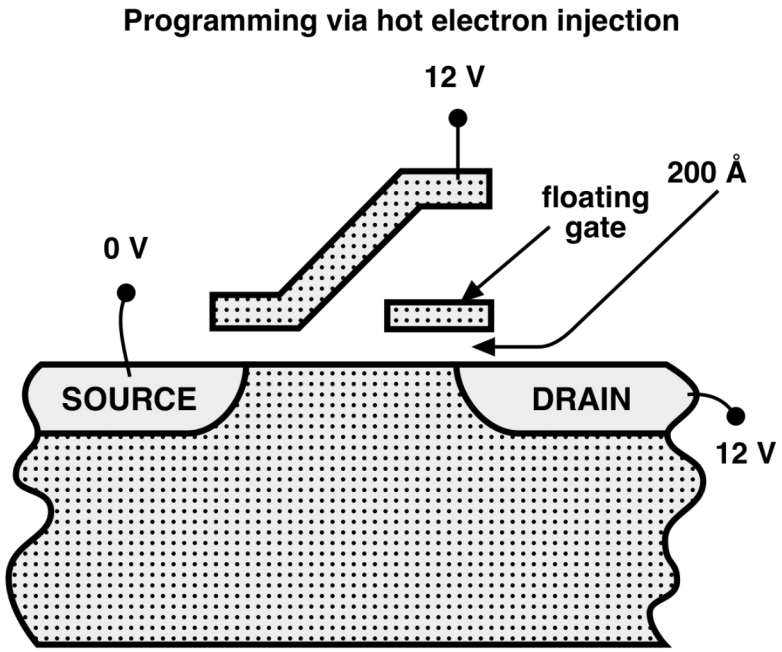
Disk write, one 1MB transfer

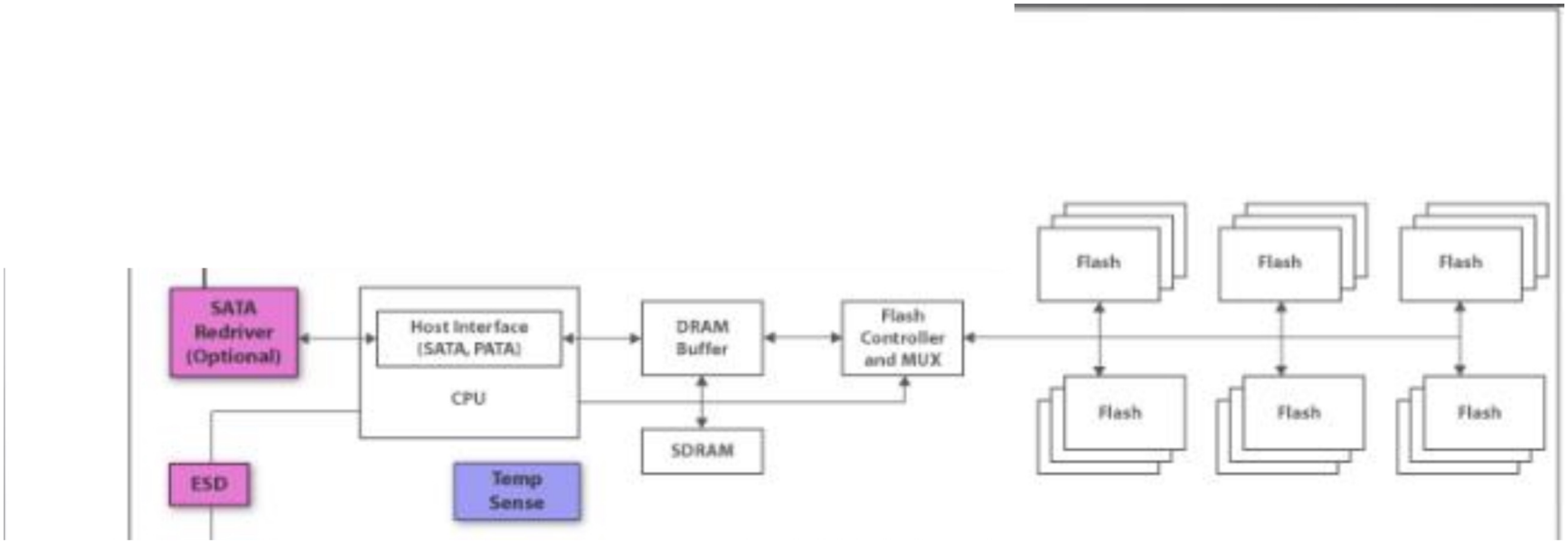


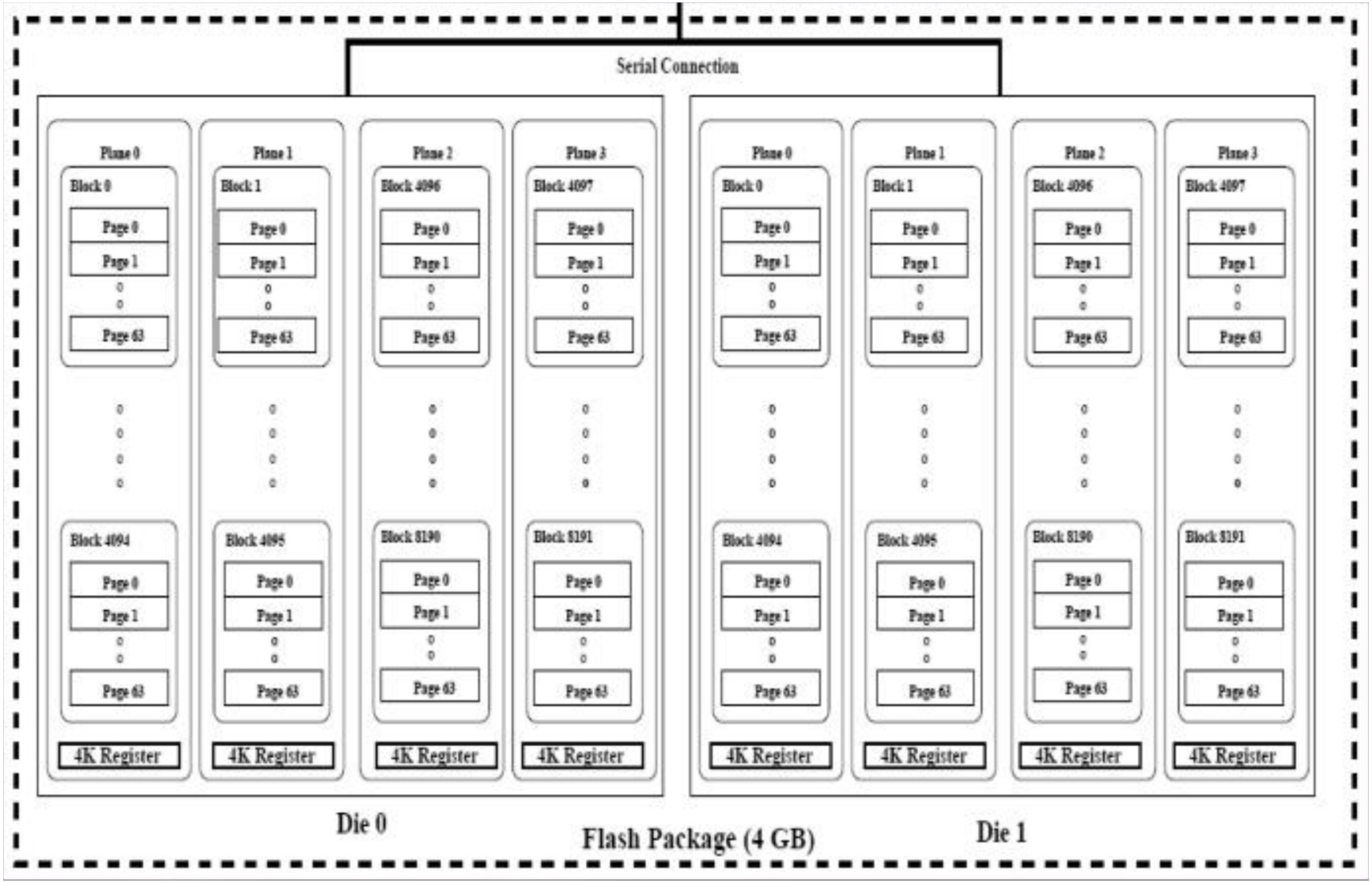
SSD timing

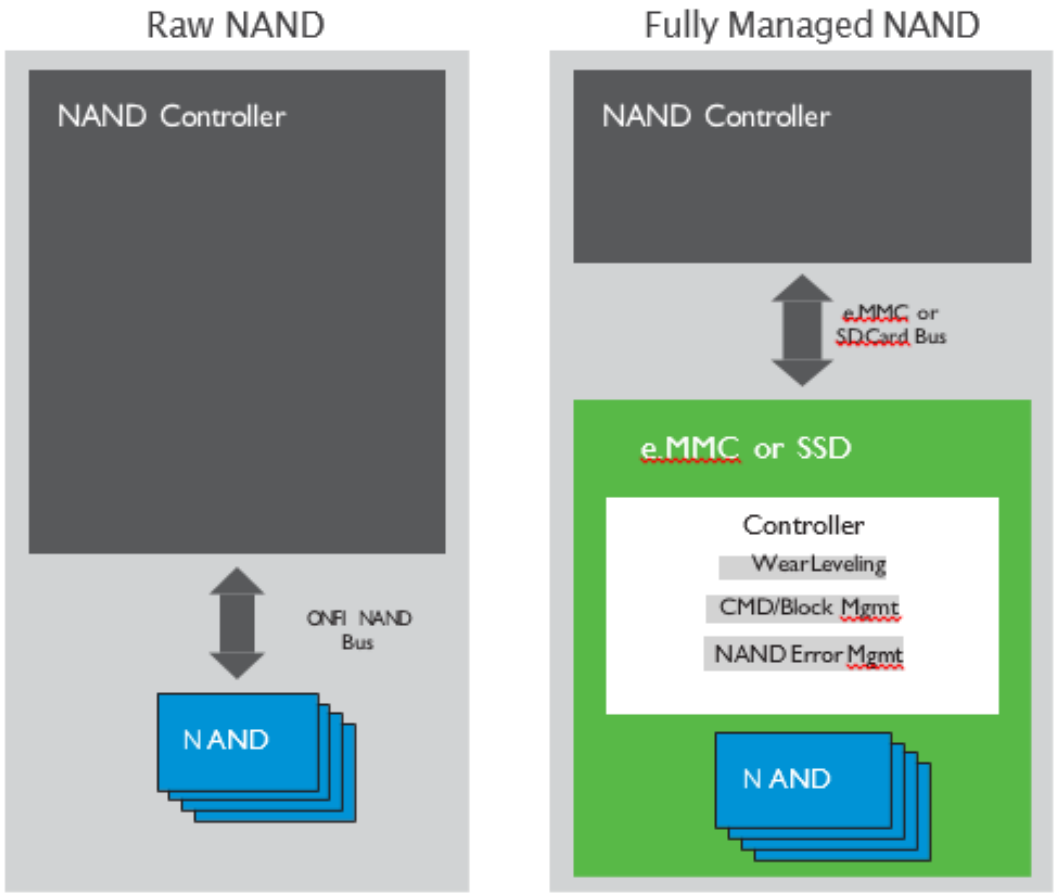
About SSDs

One bit per transistor, stored in a *floating gate*




SATA
Redriver
(Optional)
Host Interface
(SATA, PATA)
CPU
DRAM
Buffer
SDRAM
Flash
Controller
and MUX
ESD
Temp
Sense
Flash
Flash
Flash
Flash
Flash
Flash


Serial Connection
Plane 0
Plane 1
Plane 2
Plane 3
Block 0
Block 1
Block 4096
Block 4097
Page 0
Page 1
Page 63
Block 4094
Block 4095
Block 8190
Block 8191
4K Register
Die 0
Flash Package (4 GB)
Die 1


Raw NAND
NAND Controller
ONFI NAND Bus
NAND
Fully Managed NAND
NAND Controller
e.MMC or SD Card Bus
e.MMC or SSD
Controller
Wear Leveling
CMD/Block Mgmt
NAND Error Mgmt
NAND

About SSDs

Erase cycle: drive a block of bits (4,8,16KB) to all-ones

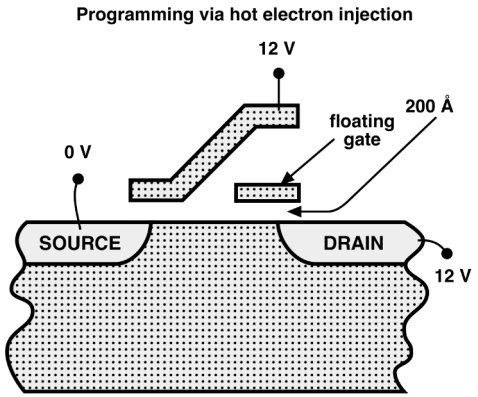
O(10 msec)

Write cycle: selectively drive some bits in a block to zeros

O(1 msec)

Read cycle: sense the floating gate state

O(100 usec)



Disk read and write timings

The mystery3.cc program has the structure but a little missing from the *write-timing* work. You get to add it, O(10) lines of code, and then run on both disk and SSD.

Everyone will get roughly similar results, but they will differ in detail at each run.

Networks

The Four Fundamental Resources

CPU

Memory

Disk/SSD

Network 

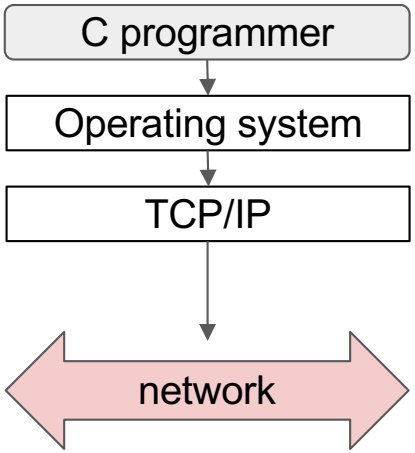
Jeff Dean's 'Numbers Everyone Should Know'

L1 cache reference	0.5 ns	O(1) ns
Branch mispredict	3.0 ns	O(10) ns
L2 cache reference	4.0 ns	O(10) ns
Mutex lock/unlock	17.0 ns	O(10) ns
Main memory reference	100.0 ns	O(100) ns
Compress 1K bytes with Zippy/Snappy	2,000.0 ns	O(1) us
Read 1 MB sequentially from memory	4,000.0 ns	O(10) us
Send 2K bytes over 1 Gbps network	**20,000.0 ns**	**O(10) us**
Read 1 MB sequentially from SSD	62,000.0 ns	O(10) us
Round trip within same datacenter	**500,000.0 ns**	**O(1) ms**
Read 1 MB sequentially from spinning disk	947,000.0 ns	O(10) ms
Disk seek	3,000,000.0 ns	O(10) ms
Read 1 MB sequentially from network	**10,000,000.0 ns**	**O(10) ms**
Send packet CA->Netherlands->CA	**150,000,000.0 ns**	**O(100) ms**

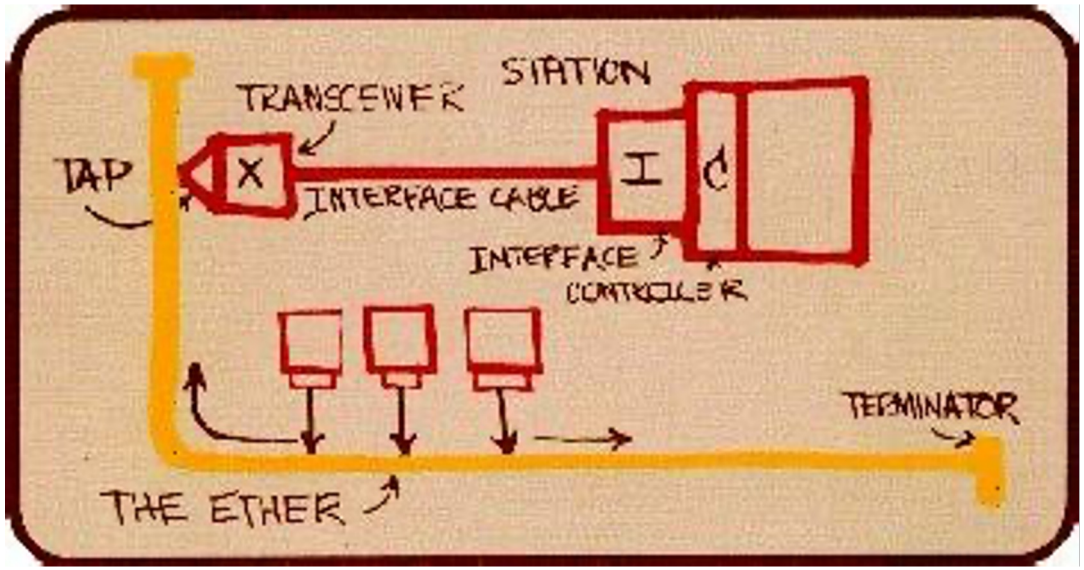
Taken from a mashup of two slides stacks and some updated numbers from URL below
http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/people/jeff/stanford-295-talk.pdf
http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf
https://kousiknath.medium.com/must-know-numbers-for-every-computer-engineer-6338a12c292c
https://norvig.com/21-days.html#answers

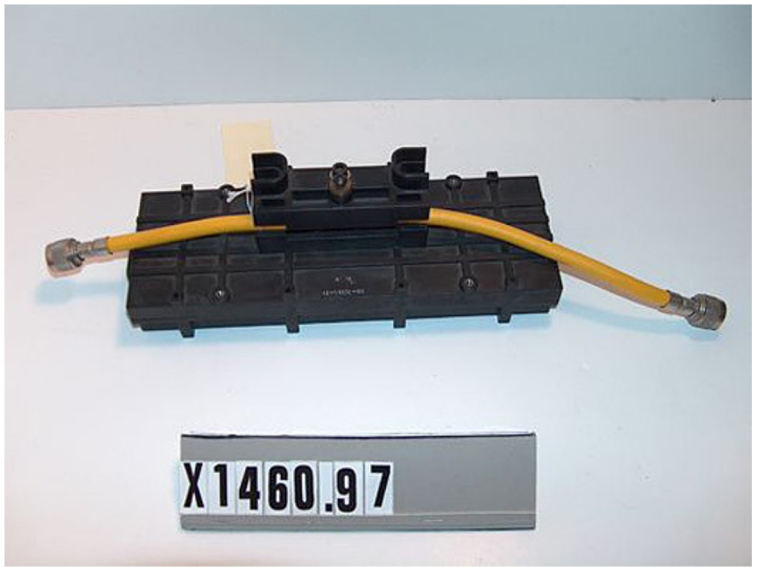
A bit about networks

Several layers of software involved



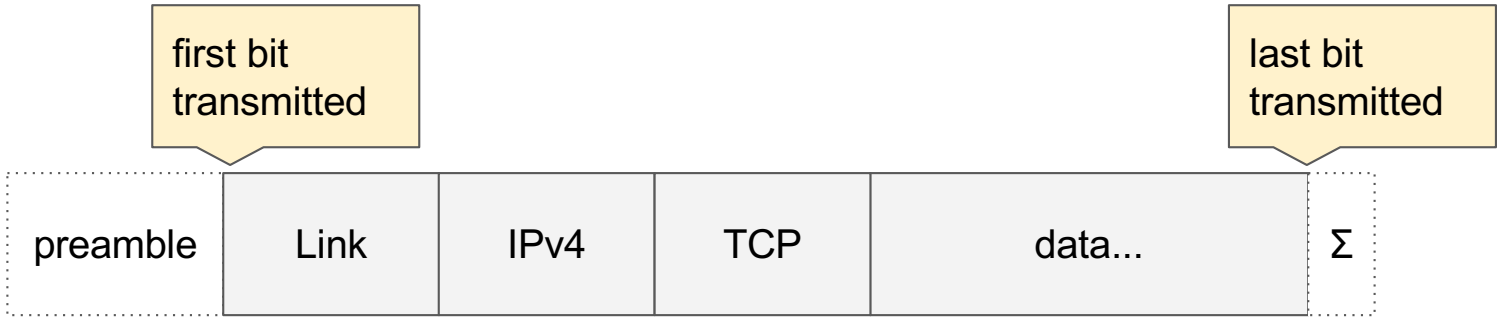
Ethernet, 1973 and 1980



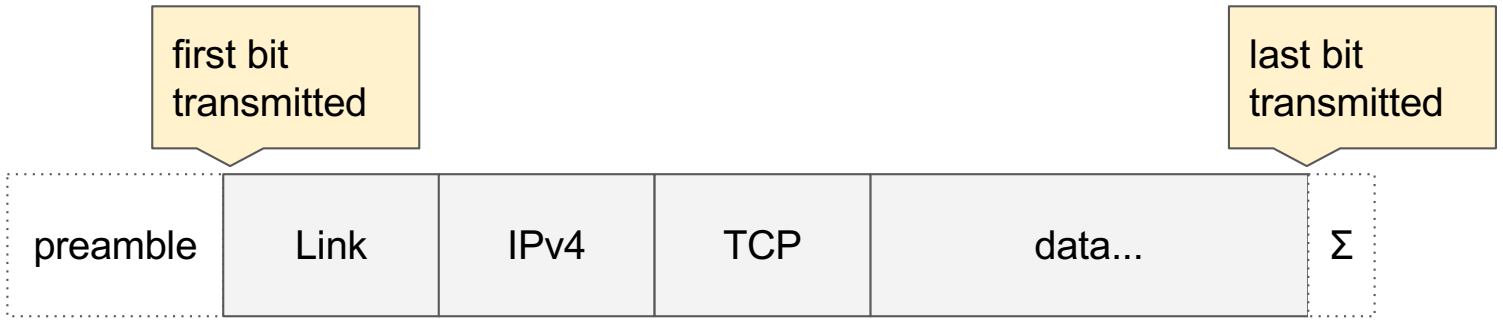


Ethernet packet summary

Individual packets <= 1536 bytes: switched, routed, and interleaved.
Store and forward



Ethernet packet summary

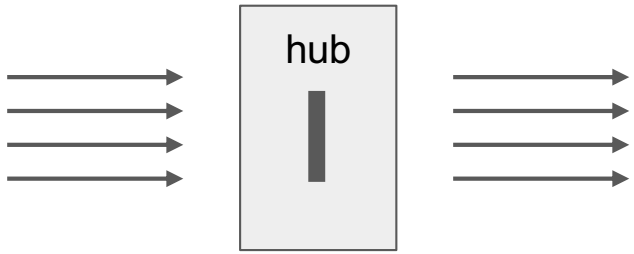


Link: Destination/source MAC address

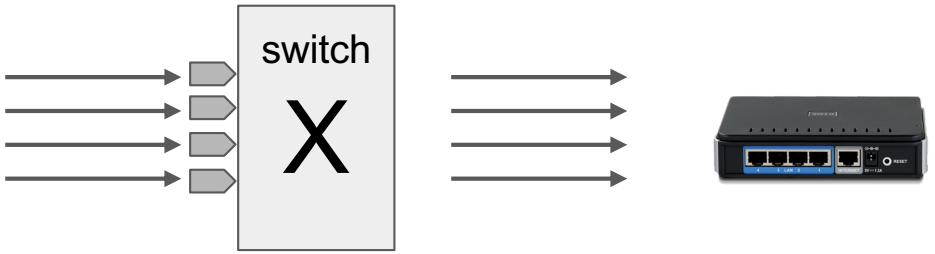
IP: Source/destination IPv4 address

TCP: Source/destination port numbers

Ethernet hub, switch, router

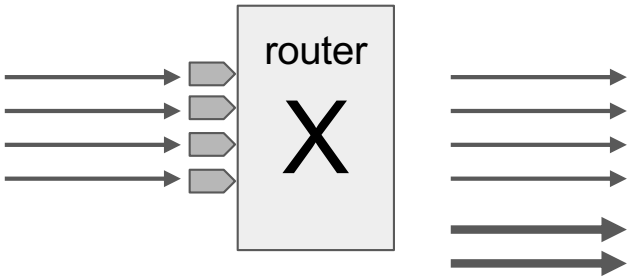


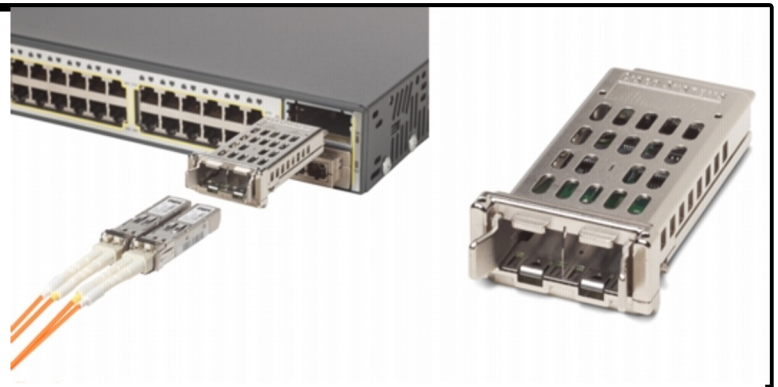
Hub Connects all inputs to outputs at once; single path inside



Switch Store and forward; crossbar can connect multiple paths at once, based on MAC address alone

Ethernet hub, switch, router

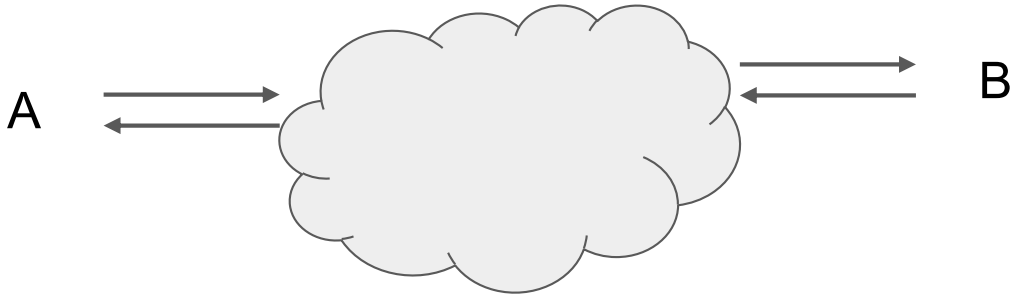




Router Store and forward; deeper buffers; higher-level protocols (IPv4, IPv6, VLAN,...) for complex routing decisions; different speed connections; enforces allowed connection rules.

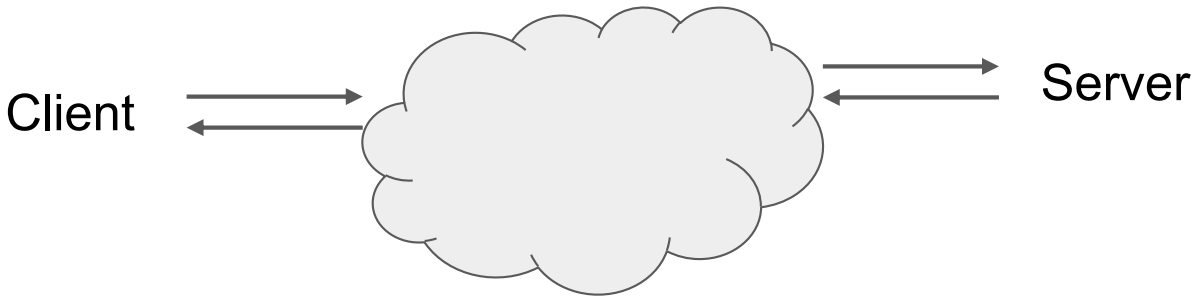
A non-blocking router can connect all inputs at once

TCP/IP socket summary



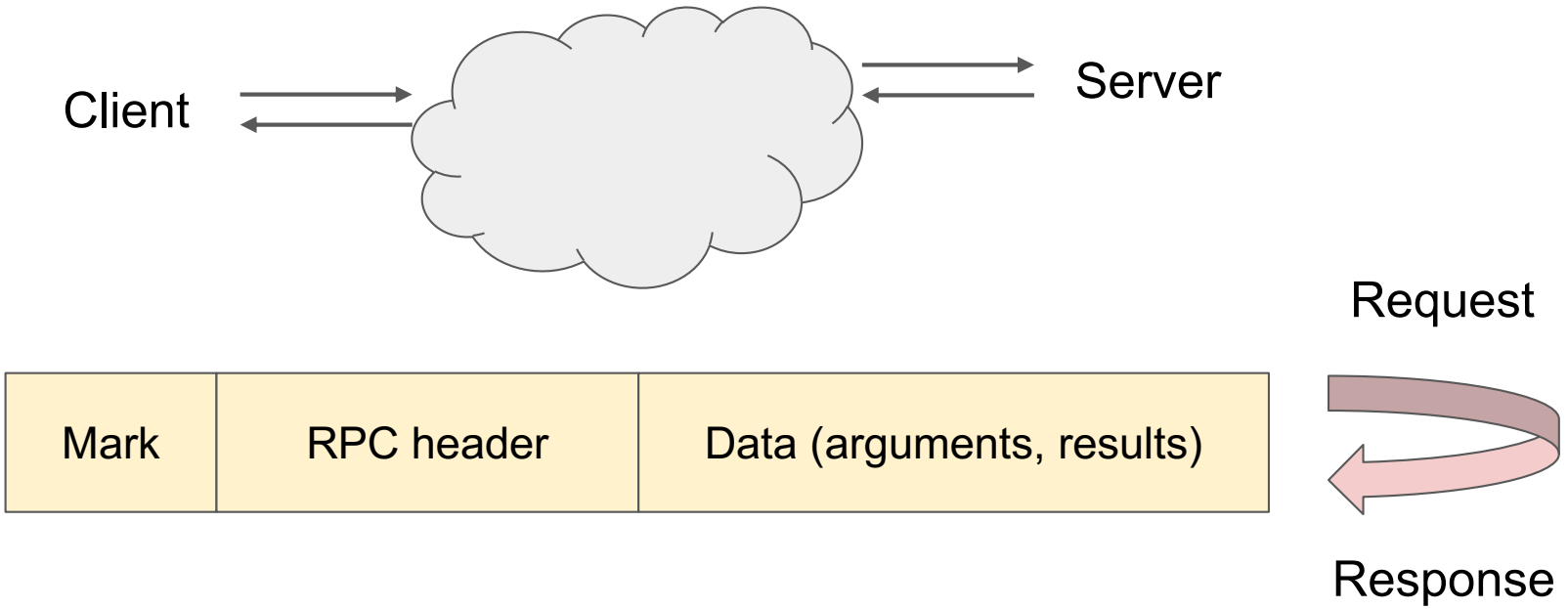
In each direction, a reliable byte stream is delivered, hiding the details of packets, retransmission, acknowledgements. TCP stream is identified by IP:port tuple

Client/Server RPC summary

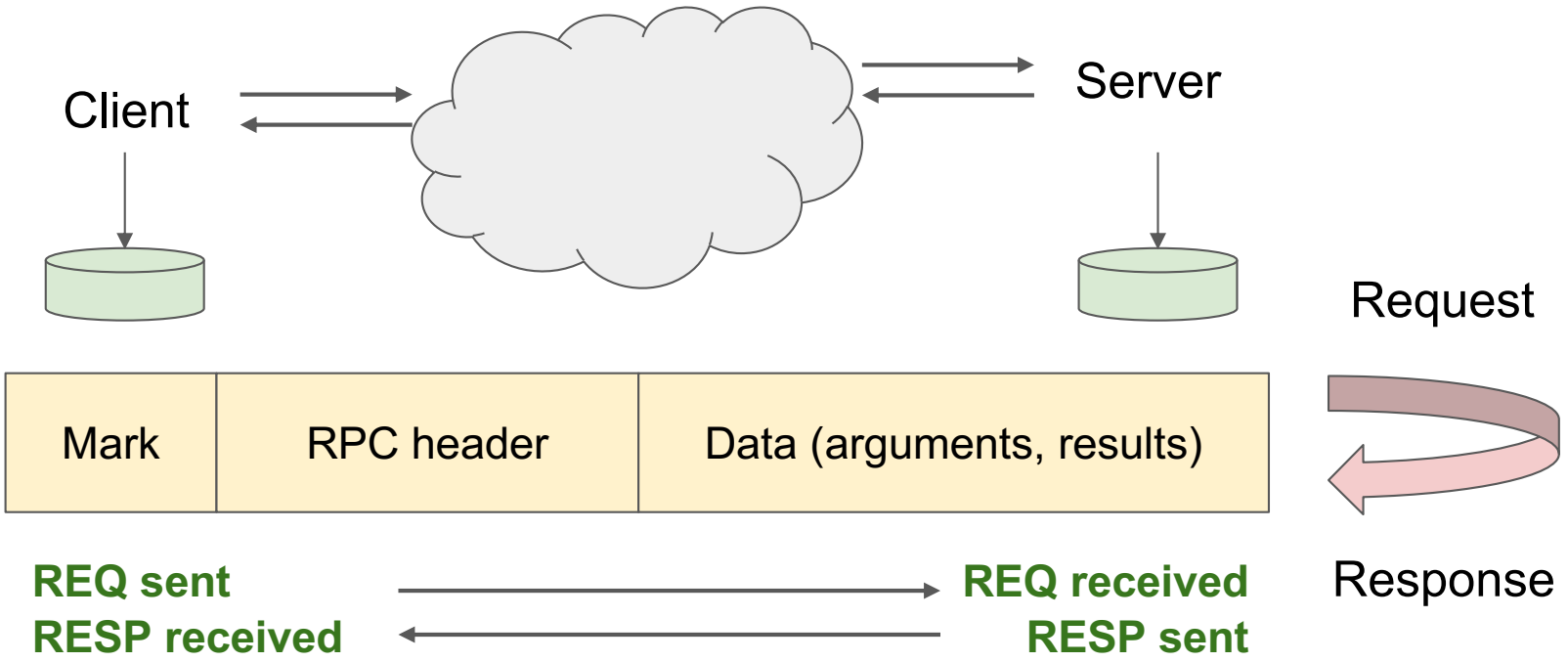


Client sends a remote procedure call request to server
Server receives request, operates on it
Server sends response
Client receives response

RPC message **for our labs**



RPC logging for our labs



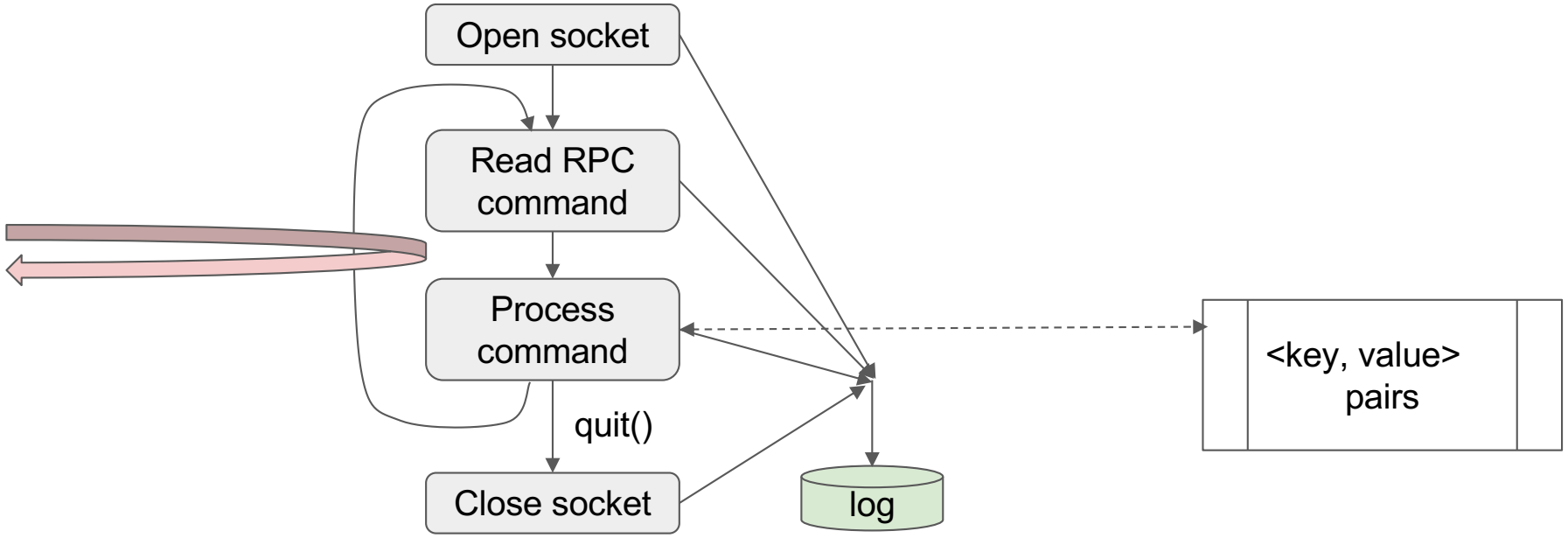
RPC simple server2

The server accepts RPCs that read or write key-value pairs in RAM

The implemented methods are:

- Ping(data), just returns data with no key-value action
- Write(key, value), remembers the pair
- Read(key), returns value
- Delete(key), removes key and its value
- Stats(), returns some TBD statistics on server usage
- Reset(), erases all key-value pairs
- Quit(), stops the server and all its threads.

RPC simple server2

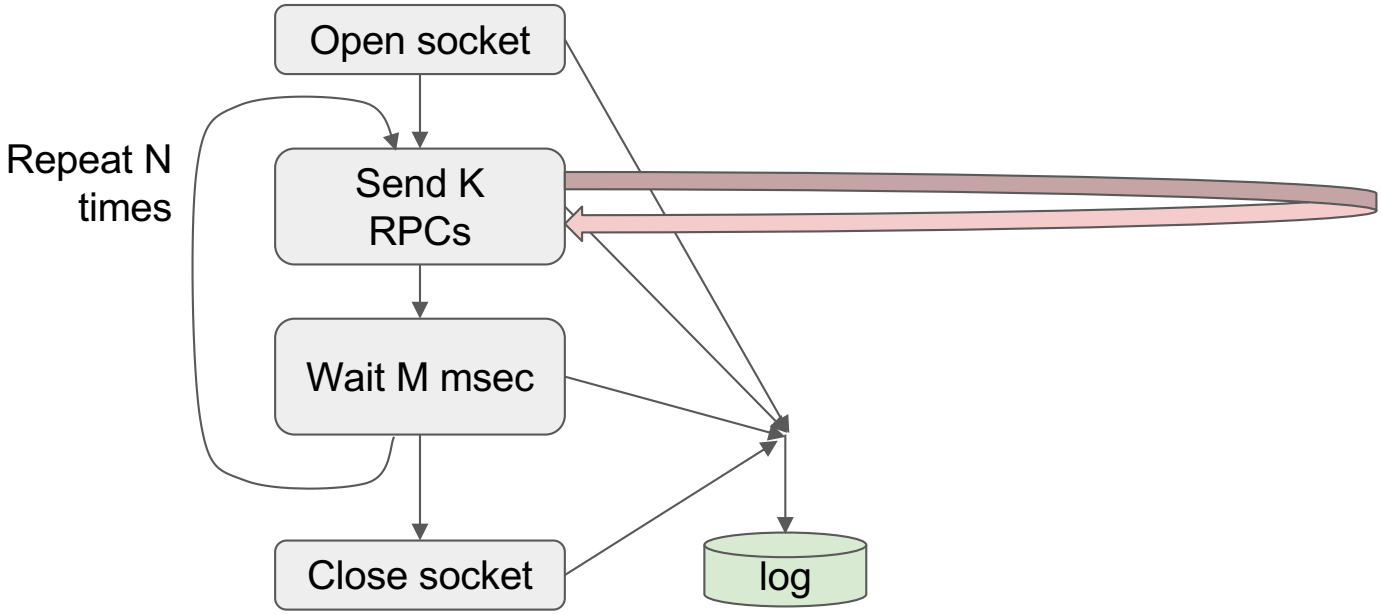


RPC simple client2

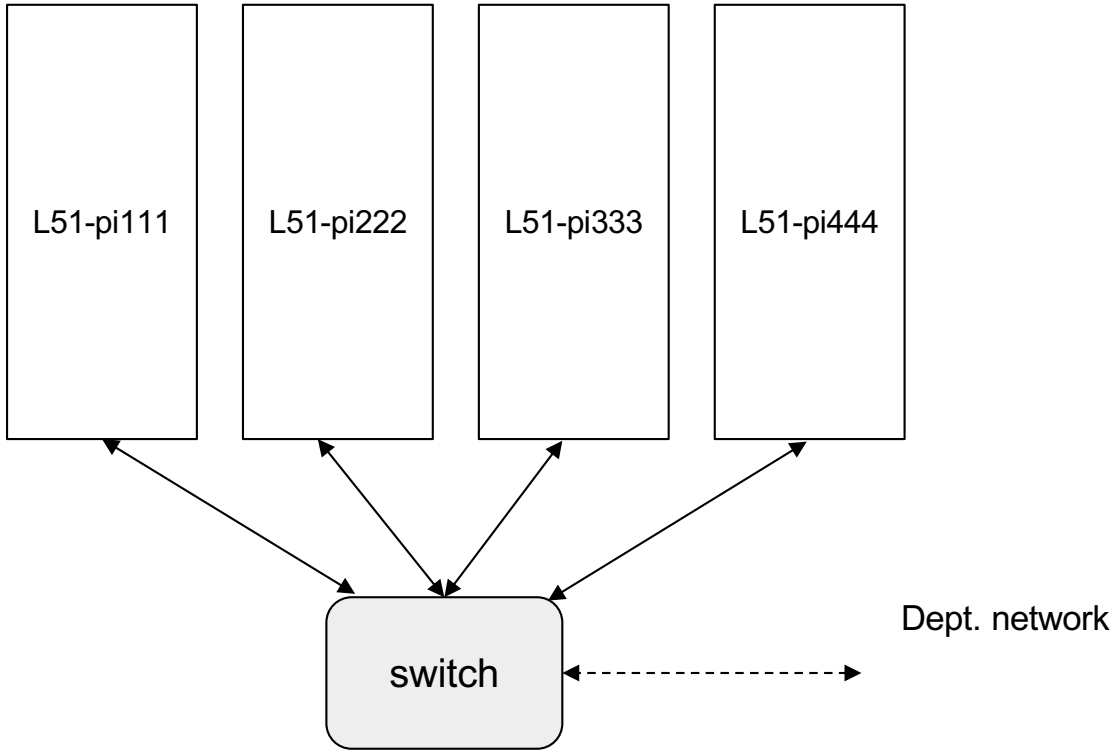
The client sends RPCs that read or write key-value pairs in RAM
It implements a simple pair of nested loops:

```
for Rep do {
    for K do {
        send RPC
    }
    wait M msec
}
```

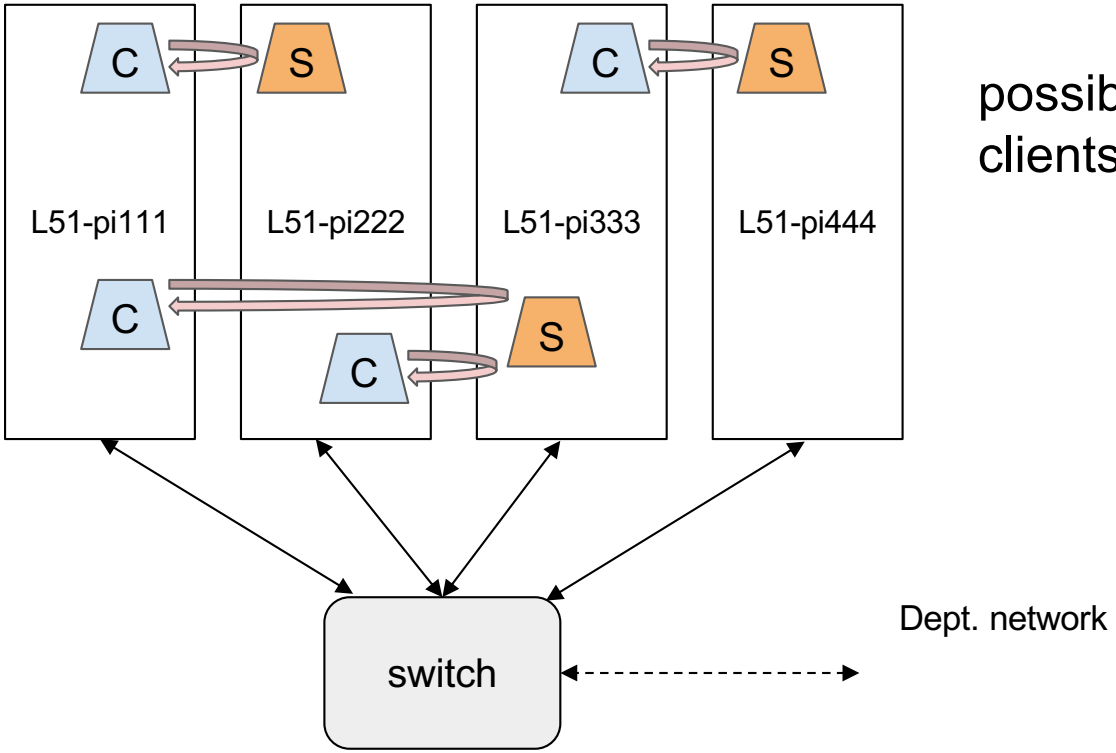
RPC simple client2



Each of your four-node lab setups



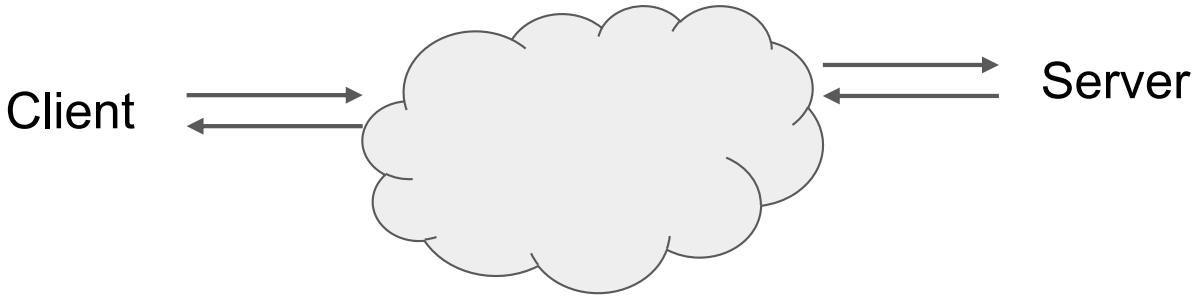
Each of your four-node lab setups



possible multiple clients/servers

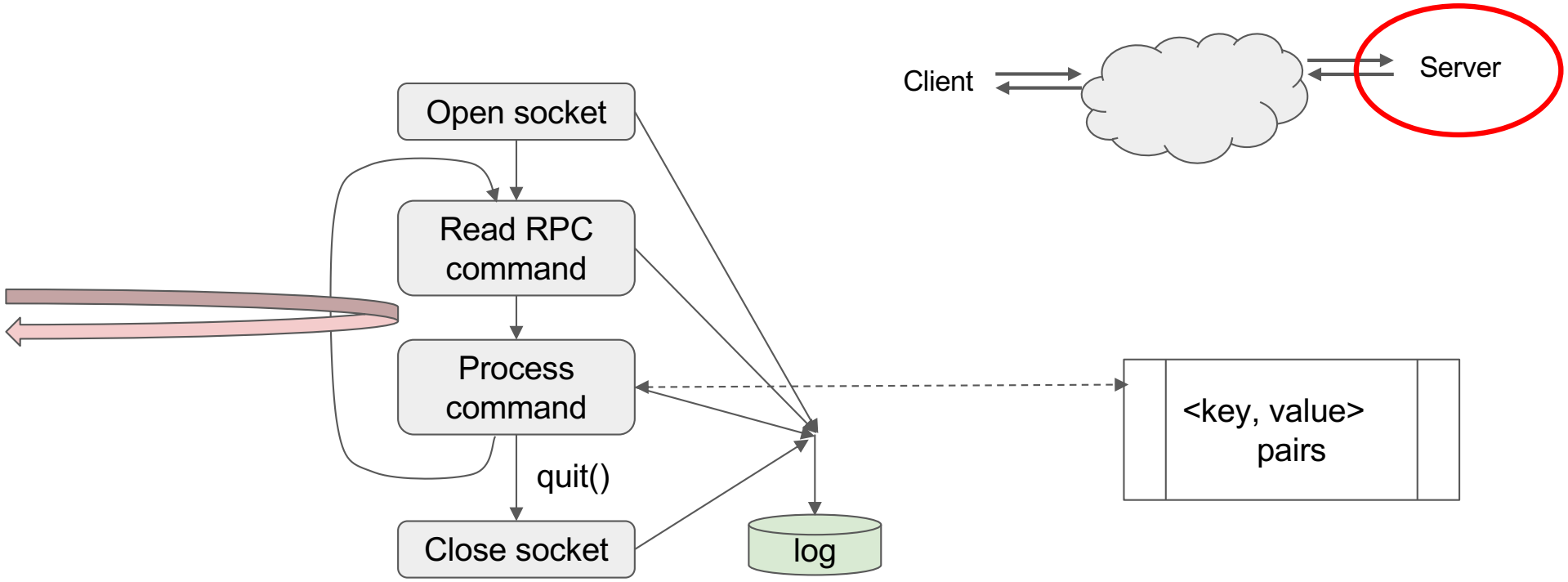
TCP/IP, Byte streams

Client/Server RPC summary

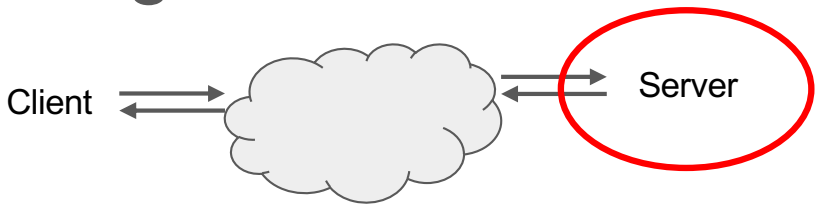


Client sends a remote procedure call request to server
Server receives request, operates on it
Server sends response
Client receives response

RPC simple server4



TCP/IP socket creation and listening on **server**



Open socket
for(;;) {
    Connect to client
    for(;;) {
        Receive request; Log request [2]
        Process request
        Send response; Log response [3]
    }
    Close connection
}
Close socket

TCP/IP socket creation and listening on **server**

Open socket

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0)      // TCP/IP
server_addr.port = portnum
bind(sockfd, server_addr)
```

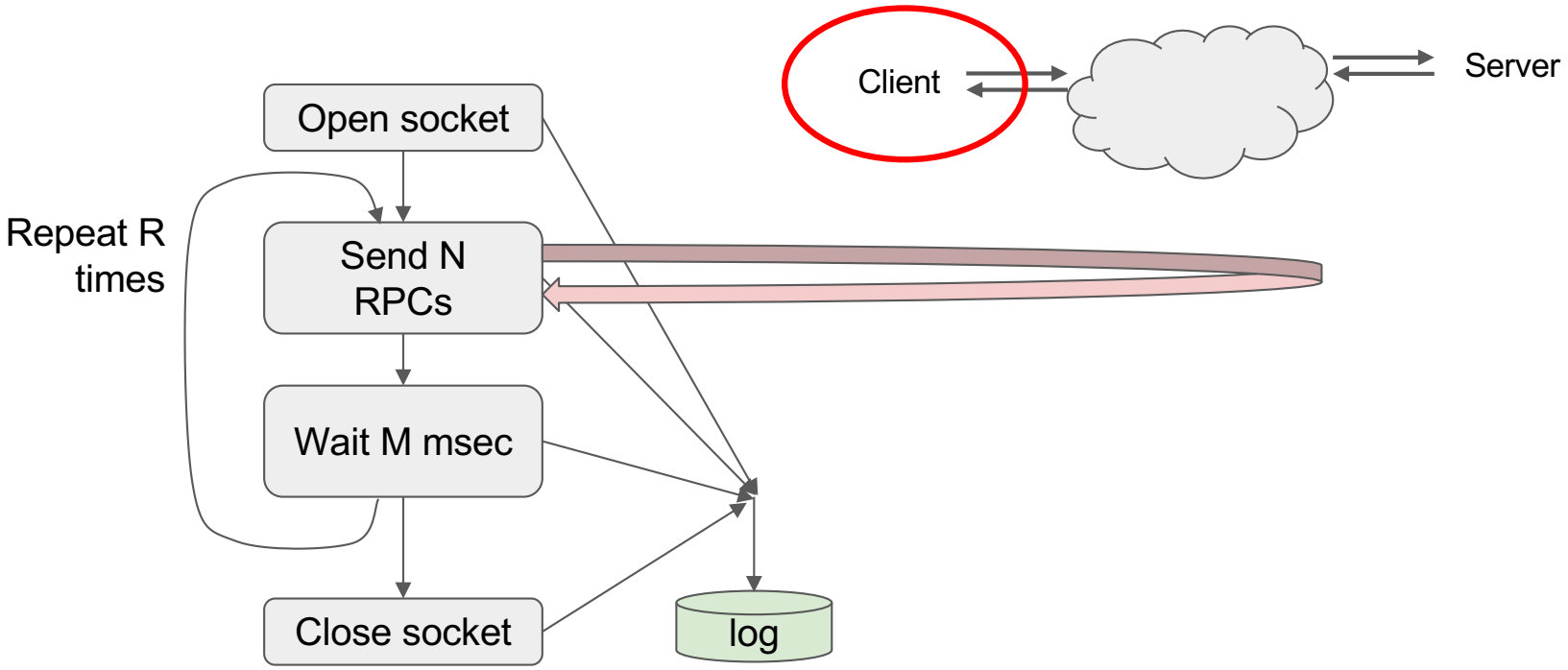
Connect to client

```
listen(sockfd, 5)
int acceptsock = accept(sockfd, client_addr)  // Blocks until a client shows up
```

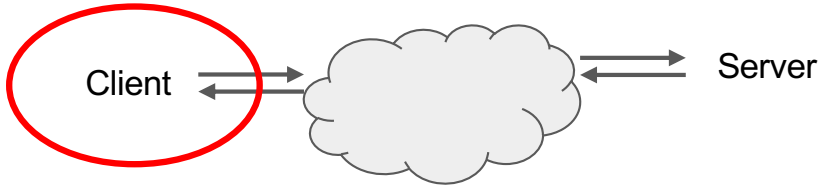
Close connection: close(acceptsock)

Close socket: close(sockfd)

RPC simple client4



TCP/IP socket creation and connection on **client**



Connect to server

```
for(;;) {
        Build request
        Send request; Log request[1]
    (wait for server to respond)
        Receive response; Log response[4]
}
```

Close connection

TCP/IP socket creation and connection on **client**

Connect to server

getaddrinfo(server_name, server_port, server_addr)

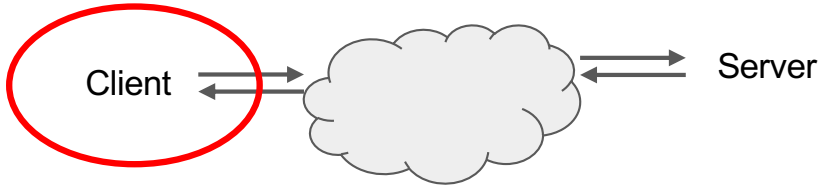
int sockfd = socket(server_addr)

connect(sockfd, server_addr)

Close connection

close(sockfd)

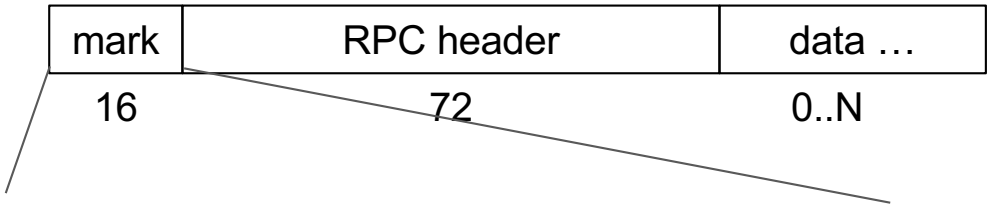
TCP/IP socket creation and connection on **client**



Connect to server
for(;;) {
    Build request
    Send request; Log request[1]
    (wait for server to respond)
    Receive response; Log response[4]
}
Close connection

Simple RPC format in detail

Each RPC request and response message has this format:



The mark part consists of four uint32 values

- signature, always 3519354853u
- **length of RPC header**, currently 72 bytes
- **length of data**, 0..N bytes
- checksum = signature + ((headerlen << 20) ^ datalen)

More on RPC header in a little while

Summary of where we are headed today

TCP/IP socket creation and listening on server

TCP/IP socket creation and connection on client

Sending byte streams

Receiving byte streams

RPC requests

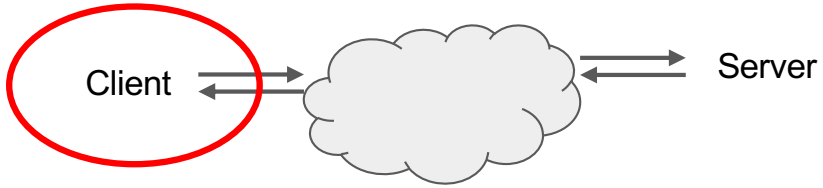
RPC responses

Multiple server threads

Software locks

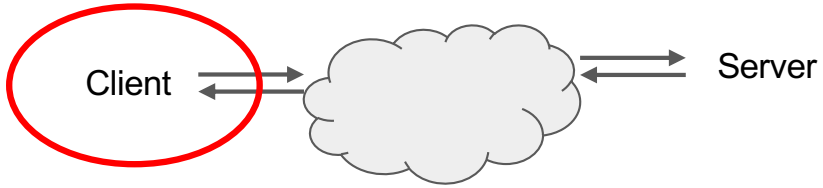
Logging

Sending byte streams on the **client**



Connect to server
for(;;) {
    Build request
    Send request; Log request[1]
    (wait for server to respond)
    Receive response; Log response[4]
}
Close connection

Sending byte streams on the **client**



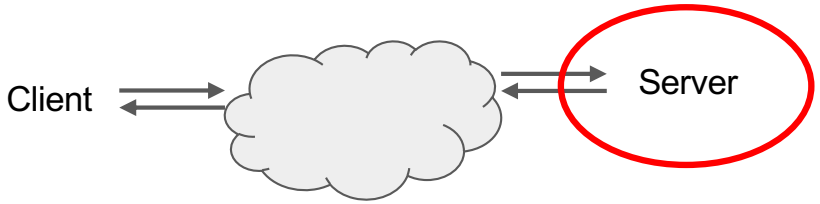
Send one RPC

```
write(sockfd, RPCMarker, 16);
write(sockfd, RPCHeader, 72);
write(sockfd, data, N1);
```

Receive one RPC

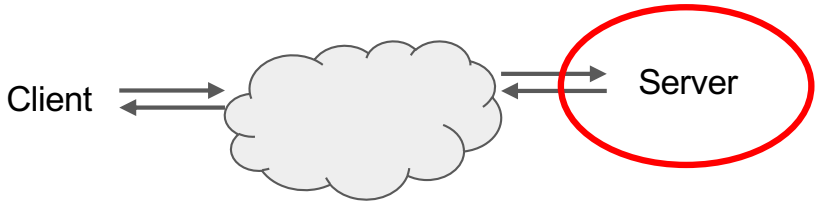
```
ReadExactly(sockfd, marker buffer, 16);
ReadExactly(sockfd, header buffer, 72);
ReadExactly(sockfd, data buffer, N2);
```

Receiving byte streams on the **server**




```
Open socket
for(;;) {
        Connect to client
        for(;;) {
                Receive request; Log request [2]
                Process request
                Send response; Log response [3]
        }
        Close connection
}
Close socket
```

Receiving byte streams on the **server**



Receive one RPC

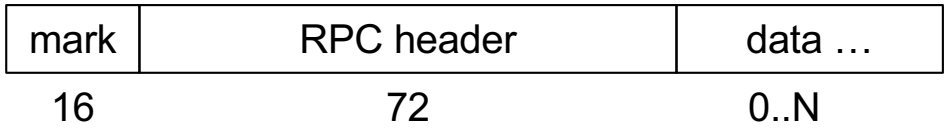
```
ReadExactly(sockfd, marker buffer, 16);
ReadExactly(sockfd, header buffer, 72);
ReadExactly(sockfd, data buffer, N1);
```

Send one RPC

```
write(sockfd, RPCMarker,16);
write(sockfd, RPCHeader,72);
write(sockfd, data,N2);
```

RPC requests, responses

RPC requests and responses



ping(key) => key

write(key, value) => nil

read(key) => value

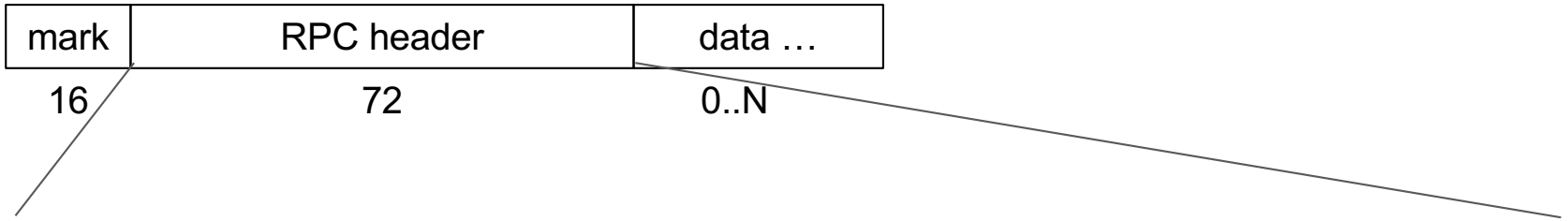
delete(key) => nil

reset() => nil

stats() => string

quit() => nil

RPC header



timestamps: req_send, req_rcv, resp_send, resp_rcv

type: request, response

machines: client IP:port, server IP:port

IDs: RPC ID, Parent ID

method: ping, read, etc.

status: success, fail, various reasons

User actions for dynamic HTML in this prac.

- Mouse drag to pan left-right, up-down
- Mouse wheel to zoom in-out
- Red dot at lower left resets

- Shift-click-drag to measure.
- End with unshift-unclick to keep, unclick-unshift to toss

- Zoom way in to see individual 4KB block dots

Use screenshots or "SVG Crowbar 2" from github to capture specific views.