13

Meteor Strike

(CS739 Fall ‘15 Midterm)

A meteor has struck the Earth, and all that is left are these brief essays/pictures/etc. on how to build distributed computer systems. Apparently, distributed computer systems are important, so we need to understand how to build them. Although we still have not figured out how to “grow food” or even what the fancy word “agriculture” means, we are hoping that distributed systems will be enough to feed society and move humanity forward after this tragic accident.

Can you help us reconstruct the truth from these fragments? We hope so, because you’re the only one left who seems to know anything about “computers” at all. Good luck!

Meteor Striking the Earth
(Artist’s* Rendition)

NAME: ____________________________

STUDENT ____________________________

*: *Not a good Artist.*

1. There is a paper on something apparently called **Remote Procedure Call**. Here is the text we recovered:

> "When the callee machine receives this packet the appropriate procedure is invoked. When the procedure returns, a result packet containing the same **call identifier**, and the results, is sent back to the caller.
>
> The call identifier consists of the calling machine identifier, a machine-relative identifier of the calling process, and a sequence number."

What is this call identifier used for? Why are all the pieces of it needed?

machine ID + process ID => activity

uniquely IDs comm endpoint

useful for server to track
info about comm stream
from that process

sequence #: used to detect duplicates (per activity)

assumes 1 msg outstanding @ a time

2. We have also uncovered a paper seemingly about "fast user-level networks". This **U-net** system seems to be the key to making distributed systems work efficiently. The fragment we found from that paper is here:

> "… receive queues are similarly allocated such that the host can poll them without crossing the I/O bus, while send and free queues are actually placed in [can't read this part] and mapped into user-space such that …"

What issue are the authors talking about here? What are these "receive, send, free" queues, and how are they placed in the SBA-200 implementation of U-net?

Host places msg desc on send queue to send

NIC places recv'd msg descs on recv queue

free queue for descs that are not in use

send, free => SBA-200 memory recv -> host memory

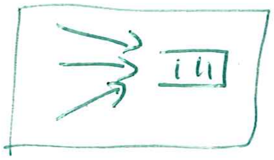
placed for efficiency

host code polls recv frequently (check for new messages)

card polls send queue a lot (for msgs to send)

3. Our deep research team has discovered a paper on an apparently critical technology, **TCP**. We have deduced that sometimes TCP suffered from an **incast** problem. **What is this incast problem?** What conditions must hold true for it to happen? (we believe there are three, but are not sure). Can you help us?

Problem:

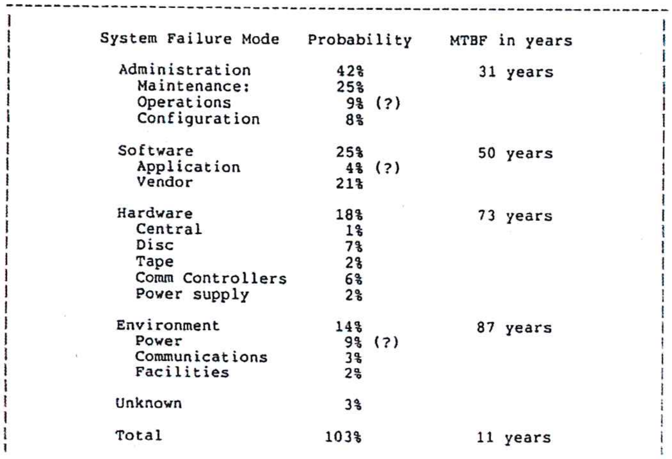


switch
many → 1 comm
overflows buffer
retry (slow)

Conditions:

1) High B/W, low latency switch w/ small buffers
2) Barrier sync requests (no new until all returned)
3) Small data returned per request

4. We have also found a famous paper on how to build fault tolerant systems. Apparently it is by a man whose name is **Jim** and whose last name is something also quite unmemorable. This paper is full of data on what is likely to fail. We have uncovered one such table:



System Failure Mode	Probability	MTBF in years
Administration	42%	31 years
Maintenance:	25%	
Operations	9% (?)	
Configuration	8%	
Software	25%	50 years
Application	4% (?)	
Vendor	21%	
Hardware	18%	73 years
Central	1%	
Disc	7%	
Tape	2%	
Comm Controllers	6%	
Power supply	2%	
Environment	14%	87 years
Power	9% (?)	
Communications	3%	
Facilities	2%	
Unknown	3%	
Total	103%	11 years

We also found the following text:

"The implications of these statistics are clear: the key to high availability is tolerating …"

Unfortunately, the rest of the system was actually hit by a piece of the meteor, and is now lost to history. What is **availability**? What are they keys to **high availability** according to the author?

$$A = \frac{MTBF}{MTBF + MTTR}$$

(Lots of other answers possible)

Key to High Availability is tolerating operations and software faults.

5. We found a paper that tells us how **disk drives** fail; it is by "Bianca Schroeder" and "Garth Gibson" - we think (but are not sure) that they were a rock duet who also wrote papers on system failure. Here is one of the pictures we recovered from their paper:

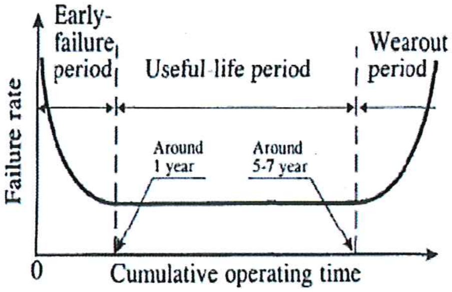


Figure 2: *Lifecycle failure pattern for hard drives [33].*

Why is this figure in the paper? Is this simply a summary of the main results, or perhaps just something related to guitars? Do you know what the main result (that relates to this figure) of the paper is?

General model of failure is "bath tub" => 1) wear in (lots of failures)
2) useful life (steady state)
3) wear out (more again)

Schroeder/Gibson found disks basically just wore out @ increasing rates (no 1, 2)

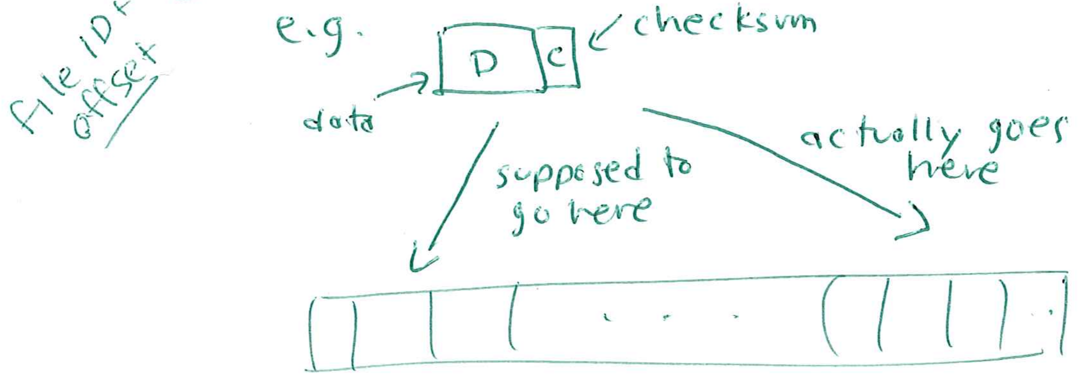
6. Another paper on **disk corruption** has been found, with this vexing sentence:

"A second component of the data integrity segment is **block identity information.** In this case, the fact that the file system is part of the storage system is utilized. The identity is the disk block's identity within the file system (e.g., this block belongs to inode 5 at offset 100)."

What is this **block identity information** used for? Why is it needed?

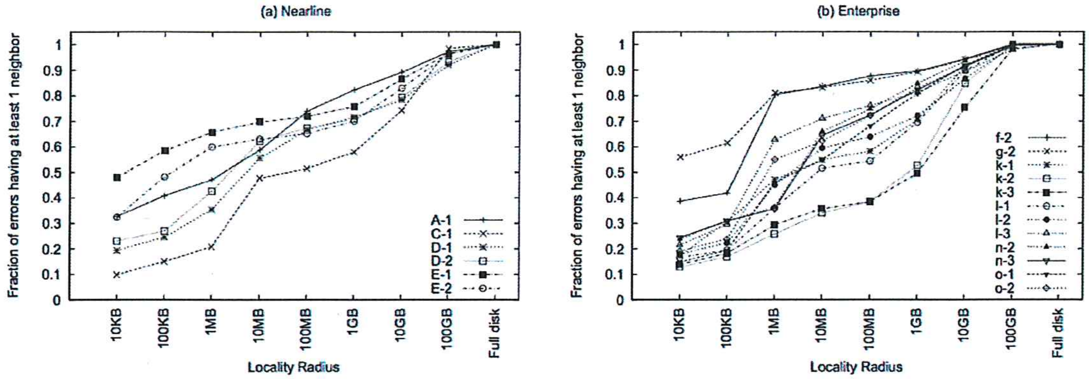
✓ Useful for misdirected write \ lost write

e.g.



simple checksum over data can't tell; it looks "correct"

7. These figures were located within a paper on the topic of **LSEs**.



First, what is an **LSE**? (explain)

4 (Latent Sector Error
Disk basically works, but a sector is inaccessible

Why were these two figures included in the paper?

3 (To see if LSEs exhibit locality
(they do)

Why are the authors differentiating between **nearline** and **enterprise** drives?

3 (To see if more expensive drives (enterprise)
fail @ a different rate
than inexpensive ones (nearline)

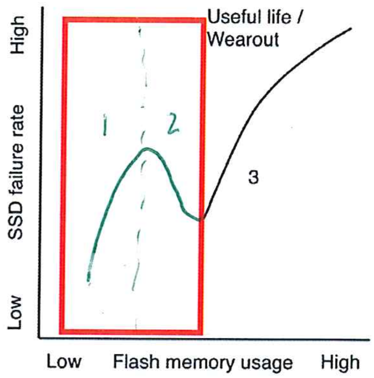
8. We have concluded that distributed systems have "bugs" in them, which is apparently quite a bad thing. We have also concluded we should figure out how to find the bugs automatically. Fortunately, we have found a technique that may help. Here is the snippet of a relevant paper:

"To scale **dmck** [distributed system model checking?], we introduce **semantic-aware model checking (SAMC)**, a white-box principle that takes simple semantic information of the target system and incorporates that knowledge..."

What problem do typical "model checkers" run into? How does semantic information help?

Model checkers ⇒ state explosion
(too many states to check)
Semantic info can reduce
redundant executions, find
bugs more quickly
[can fill in a little
more detail here]

9. An old paper on a technology known as **Flash-based SSDs** has also been discovered. Unfortunately, a part of an essential graph is missing (boxed in red). It looks like this:



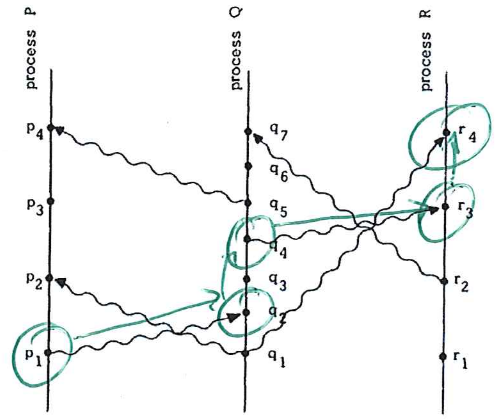
Can you fill in the missing parts of the graph? Why does the graph have the shape that it does?

1) early detection of bad blocks } from weak pool

now, mapped out by controller ← 2) early failure of bad blocks }

3) normal failure of blocks from stronger pool

10. Time was apparently salient in these distributed systems. So much so that one guy named **Lamport** made up some stuff about time and started drawing pictures. Here is one:



$p_1 \rightarrow q_2 \rightarrow q_4 \rightarrow r_3 \rightarrow r_4$

Lamport claims that there is something called a **happens-before** relation, and that some happens-before relations can be obviously seen from the figure, and worse, that happens-before is transitive. But, we can't figure this out. **Can you tell us what happens-before means?** Can you show us an example of the transitive nature of happens-before on this picture?

"happens before" relates events in a dist sys.

in same process, if a comes before b, a → b

if a is send and b is recv, a → b

it is transitive in that if a → b and b → c, a → c

example above

key: causality

if a → b, a can affect b

11. Amazingly, somebody improved on the genius Lamport and created **vector clocks**. Unfortunately, the diagram that was left was incomplete. Can you fill the values of these vectors, so we can figure out the rules? (we've put red boxes where the missing values are).

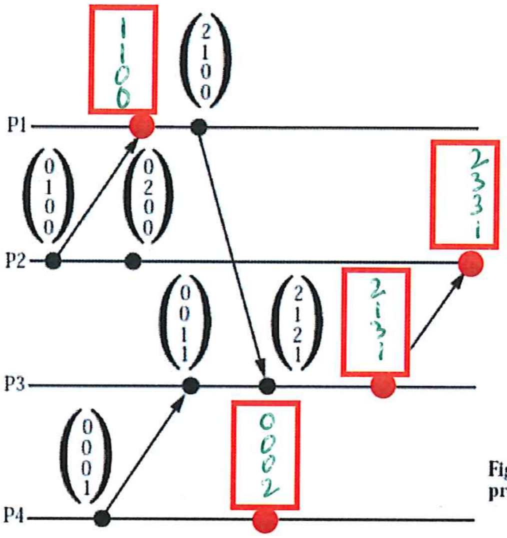


Fig. 13. Time propagation

2

2

pts

2

2

In what way are vector clocks better than Lamport clocks? Do you know?

if $T_1 < T_2$,

$E(T_1) \to E(T_2)$

((can determine more about events by comparing times)

(Lamport clocks lose some of this info, and can't compare times meaningfully in all cases)

2 pts

12. We have also uncovered a distributed file system called **NFS**. Apparently this NFS was used quite a bit before the meteor hit. Everywhere we look, documents say that NFS writes are **synchronous**. What does this mean? Why is it needed? We think it slows systems down; can we remove this behavior from NFS?

Server writes are synchronous

That way, when finished, client knows it can discard write (no need to retry again)

w/ async server write, ack → client thinks it is done

⇒ server crash ⇒ data lost

Yes, it can be slow; hard to remove (but can optimize, e.g., NV Memory, or change protocol)

13. **NFS** also apparently implemented some weird feature to preserve “compatibility”. We don’t know what compatibility they are talking about; maybe you can help us make sense of this sentence?

> "…[what we did was] check in the client VFS remove operation if the file is open, and if so rename it instead of removing it…."

What feature are they talking about here? Why is this behavior necessary?

Unix semantics:
can remove open file;
writes should still work] w/in one process!

NFS hack to handle this:
rename it, leaves garbage file
(clean up later)

14. NFS wasn’t the only distributed file system in use before the meteor strike. At some incredible school in the “middle west” known as “the University of Wisconsin”, the people there used **AFS**, which is clearly different because you know A is different than N.

Here is the one cryptic quote we found about AFS:

> "In a conventional 4.2BSD system, a file has a unique, fixed-length name, its inode, and one or more variable-length pathnames that map to this inode. The routine that performs this mapping, namei, is usually one of the most heavily used and time consuming parts of the kernel. In our prototype, Venus was aware only of pathnames …"

What are the authors talking about here? What problem were they having? How did they fix it?

Files originally identified by full pathname
=) Too much server CPU work
(to traverse paths)
Solved with file IDs in AFS$_{v2}$
Clients traverse paths,
and eventually cache directories
(no server interaction @ all)

15. In the days before the strike, some systems ran "servers" and apparently making these servers as fast as possible was important. One early system to do this, we have learned, was called **Flash**. The following paragraph details a problem the Flash authors found:

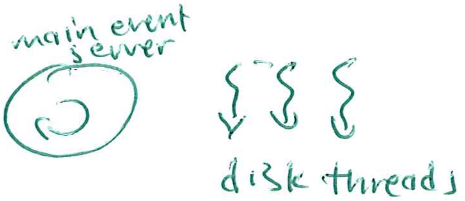
```
"In these operating systems, non-blocking read and write operations
work as expected on network sockets and pipes, but may actually block
when used on disk files."
```

Why do the authors think this is an important fact? What can be done about it when developing these high-performance servers?

if building event-based server,
must never block while handling event

disk I/O didn't have such an interface

thus, add disk threads/processes



give request, later check for done

16. The **Coda** paper we found uses the term **hoarding** to refer to some important activity. What is hoarding? Why is it important in this so-called Coda work?

Coda's meant to work offline

Thus, must fill cache while connected
w/ useful stuff before disconnected

This activity is called hoarding

(Some details about how
hoard caching is different
then typical LRU is nice)

17. We recently discovered another critical piece of information about how old distributed systems used to work, in an effort known as **Grapevine**. Here is the text we discovered:

"If a change message gets destroyed because of a software bug or equipment failure, there is a danger that a permanent inconsistency will result."

This sounds important! What does it mean?

→ w/o extra work, bug of h/w fault
could lead to inconsistency in
replicated data (event. consistency
propogation assumes
correct operation)

→ solved by periodically comparing
copies and merging to resolve

18. We have stumbled upon another critical idea known as **Leases**. Here is the fragment of paper we recently dug up:

"Short leases also minimize the false write-sharing that occurs."

What could this sentence mean? What is false write sharing, and why do short leases help get rid of them?

System w/ long lease leads to
lots of extra work, as
leases held by machine much
longer than needed

=) false write sharing problem
(next requester must wait,
or somehow invalidate)

shorter: less likely to happen
(but too short can be bad too;
lots of requests to extend)

19. One of our final discoveries relates to a system known as **Remus**. Unfortunately, the only thing we have found about Remus is this:

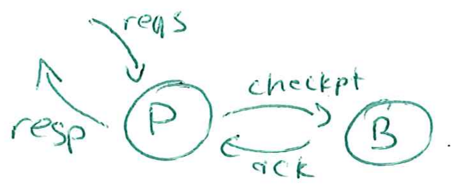
```
"Remus is …"
```

That's all we could find! Do you know anything about Remus? What is important about it? How does Remus work?

Remus is a VMM-based High Availability approach

It builds on live migration in VMMs to provide transparent P/B fault tolerance

Basic approach



requests come in

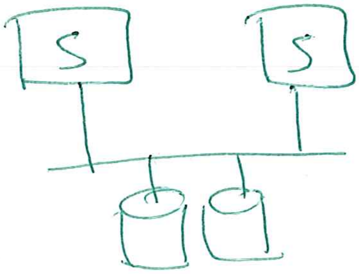
periodically, VMM sends mem/disk ckpt to Backup

once ack'd, primary can release responses

20. One last system has recently been chanced upon: **HA-NFS**. Our snippet does not include much information, alas:

```
"We separate the problem of network file server reliability into three
different sub-problems: server reliability, disk reliability, and
network reliability."
```

What do they mean by this confusing sentence? Can you explain this so we can build an HA-NFS? (once we figure out what that is?)



(note: each server disk may be RAID or whatever)

make disks reliable by allowing access from either server (OK when one is down)

make net reliable through redundancy (multiple connections)

make servers reliable via P/B for each other, use heartbeats to detect failure, and then take over if failed