IT University
of Copenhagen

Scalability of Web Systems

Correctness of Distributed Systems

Jonathan Fürst <jonf@itu.dk>

Outline

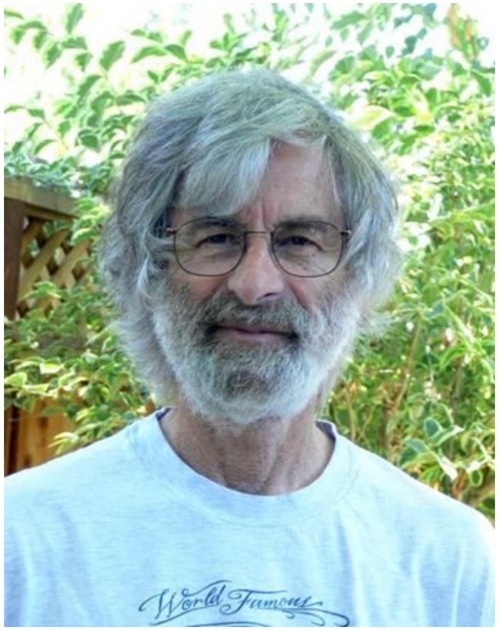
A. Lecture

1. Challenges of Distributed Systems
2. Formal Methods
3. Testing
4. Applied Methods in Industry
5. Tools: Debugging, Tracing, Profiling

B. Hands-on

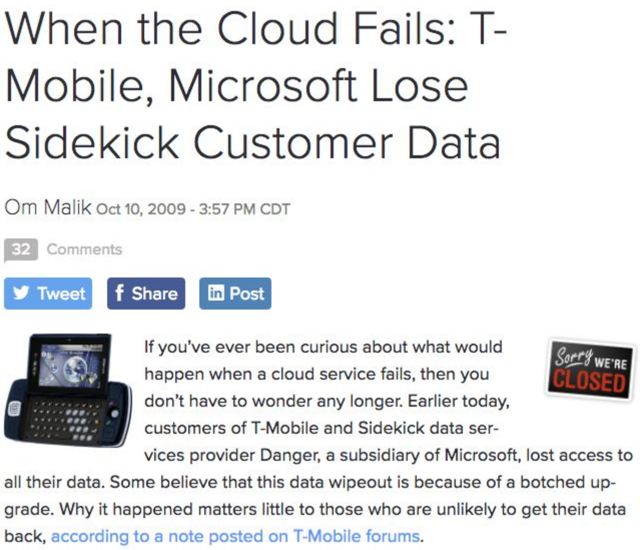
- Debugging, Testing, Profiling & Tracing Exercises (all in Go)

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable." --Leslie Lamport



ACM Turing Award (2013)

Some example of distributed systems going bad



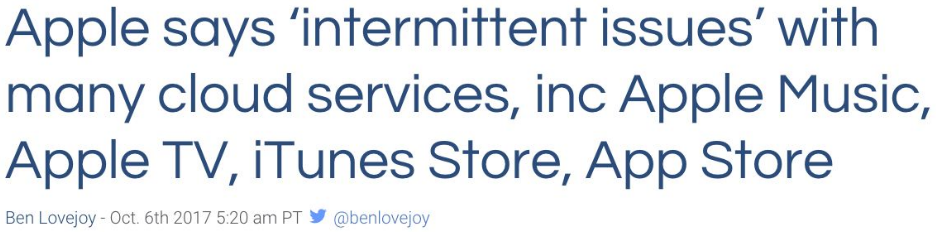
When the Cloud Fails: T-Mobile, Microsoft Lose Sidekick Customer Data

Om Malik Oct 10, 2009 - 3:57 PM CDT

32 Comments

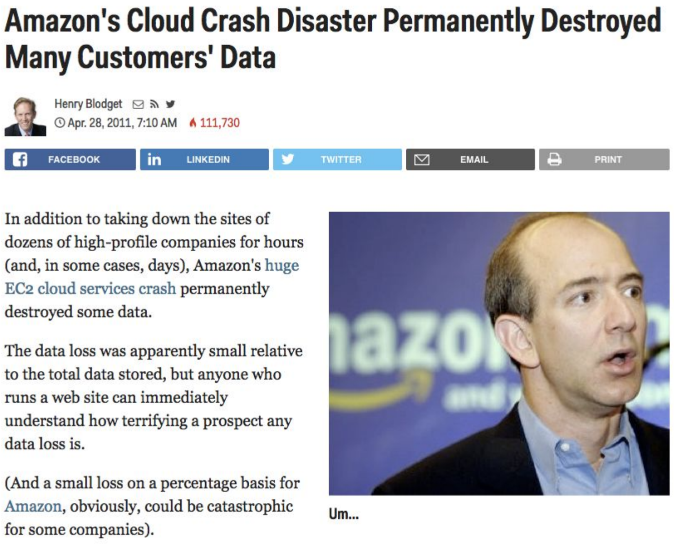
Tweet Share Post

If you've ever been curious about what would happen when a cloud service fails, then you don't have to wonder any longer. Earlier today, customers of T-Mobile and Sidekick data services provider Danger, a subsidiary of Microsoft, lost access to all their data. Some believe that this data wipeout is because of a botched upgrade. Why it happened matters little to those who are unlikely to get their data back, according to a note posted on T-Mobile forums.



Apple says 'intermittent issues' with many cloud services, inc Apple Music, Apple TV, iTunes Store, App Store

Ben Lovejoy - Oct. 6th 2017 5:20 am PT @benlovejoy



Amazon's Cloud Crash Disaster Permanently Destroyed Many Customers' Data

Henry Blodget
Apr. 28, 2011, 7:10 AM 111,730

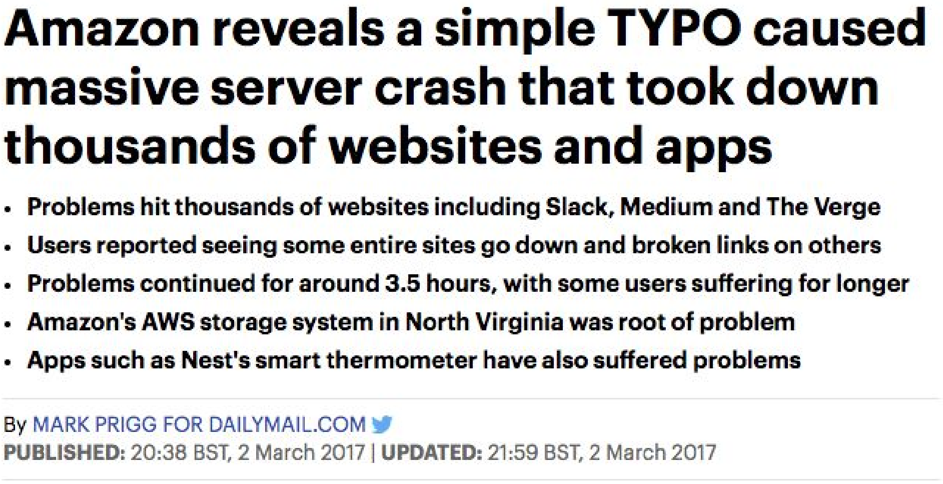
FACEBOOK LINKEDIN TWITTER EMAIL PRINT

In addition to taking down the sites of dozens of high-profile companies for hours (and, in some cases, days), Amazon's huge EC2 cloud services crash permanently destroyed some data.

The data loss was apparently small relative to the total data stored, but anyone who runs a web site can immediately understand how terrifying a prospect any data loss is.

(And a small loss on a percentage basis for Amazon, obviously, could be catastrophic for some companies).

Um...



Amazon reveals a simple TYPO caused massive server crash that took down thousands of websites and apps

- **Problems hit thousands of websites including Slack, Medium and The Verge**
- **Users reported seeing some entire sites go down and broken links on others**
- **Problems continued for around 3.5 hours, with some users suffering for longer**
- **Amazon's AWS storage system in North Virginia was root of problem**
- **Apps such as Nest's smart thermometer have also suffered problems**

By MARK PRIGG FOR DAILYMAIL.COM
PUBLISHED: 20:38 BST, 2 March 2017 | **UPDATED:** 21:59 BST, 2 March 2017

Challenges in Distributed Systems

Multiple processes, multiple languages on multiple nodes:

- Failures or partial failures of nodes
- Ordering of inputs
- Many states
- Concurrency
- Nondeterminism
- Timing
- No central view
- Unbounded inputs

Hierarchies of Failures

Hierarchies of failures according to the behavior that is perceived at the system's interface:

- Byzantine/Arbitrary Failures
  - Fail by doing whatever I want.
- Authentication detectable byzantine failures
  - Can't forge other's messages.
- Performance/Timing Failures
  - Fail by delivering message too soon/too late.
- Omission Failures
  - Fail by dropping messages (reply "indefinitely late")
- Crash Failures
  - Fail by stopping (e.g., after suffering from an omission failure)

Source: http://alvaro-videla.com/2013/12/failure-modes-in-distributed-systems.html

See also: Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem

Faults, Errors and Failures

- A **fault** is the initial root cause, which could be a hardware malfunction, a software bug, or a misconfiguration.
- A fault can produce abnormal behaviors referred to as **errors**, such as system call error return or Java exceptions.
- Some of the errors will have no user-visible side-effects or may be appropriately handled by software.
- Other errors manifest into a **failure**, where the system malfunction is noticed by end users or operators.

More of the same

- Deadlock

  => stuck on waiting for resource to be unlocked.
- Livelock/starvation

  => work on sth that is not the right thing
- Under specification

  => receive message that you were not expecting
- Over Specification

  => code path which is not used, nobody goes there

Conclusion

- Difficult to approach and many factor in play.
- Want to gain confidence of correct system behavior **now and later.**
- Behavior is **aggregated** => cannot just test single components.

=> We need some methods that help us to achieve **system correctness**.

System Correctness

Many Strategies

1. Design
2. Formal Methods
3. Testing
4. Tools: Debugging, Tracing, Profiling

Of course also: coding practices, established processes etc...

Design: Designing for Resilience

For many web systems availability is paramount:

- Alphabet Inc. makes $3246 per second in revenue, and would lose $973,756 if services ceased for 5 minutes.
- “Facebook experienced a **30-minute sitewide outage** in 2014, costing them half a million dollars. This incident saw Facebook’s **stock plummet by approximately 0.8%,** and there was a sharp increase in traffic on Twitter.”
- “In March of this year, a large number of sites were crippled for four hours when AWS (Amazon Web Services) went down due to a single employee’s error. One estimation from AXIOS claims that this incident cost upwards of **$150 million** for the companies and services involved.”

=> **Resilience** is the ability of a system to **adapt** or **keep working** when challenges occur.

https://investorplace.com/2017/09/tech-giant-outage-google-netflix-facebook-amazon-ggsyn/

Design: Designing for Resilience

Some good practices:

- Redundancies are key
  - Redundancies of resources, execution paths, checks, replication of data, replay of messages, anti-entropy **build resilience**.
  - Capacity planning matters
- Not all complexity is bad
  - Adding resilience may come at the cost of other desired goals (e.g., performance, simplicity, cost)
- Leverage engineering best practices
  - Resilience and testing are correlated => Test.
  - Versioning

=> More about this in the **next lecture (availability)**. Also, see Ines Sombra's talk on "Architectural Patterns of Resilient Distributed Systems" for more.

Formal Methods

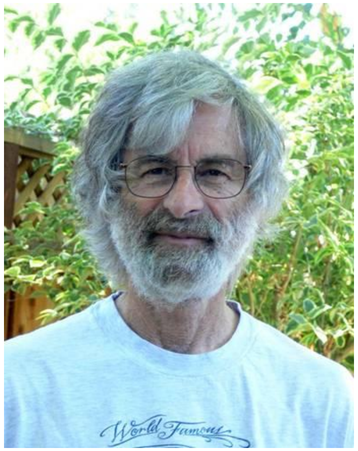
Formal Methods

- Human Assisted Proofs
  - Considered slow and hard to use
    => Safety critical domains (TLA+, Coq, Isabelle)
- Model Checking
  - State-Machine of properties and transitions. Particularly used in protocols (TLA+, Modist, Spin,...)
- Lightweight Formal Methods
  - Best of both worlds (Alloy, SAT)

Human Assisted Proofs: TLA+

- Temporal logic of actions:
  - Logic which combines temporal logic with a logic of actions.
  - Right logic to express liveness properties with predicates about a system’s current & future state.
- TLA+:
  - A formal specification language used to design, model, document, and verify concurrent/distributed systems. TLA+ verifies **all traces** exhaustively.
  - Alternative: PlusCalc (similar to C)

=> Slowly making its way to industry.



https://lamport.azurewebsites.net/tla/tla.html

TLA+ video course: http://lamport.azurewebsites.net/video/videos.html

Model Checking

Stages:

1. Initial Design => 2. Model Checker => 3. Implementation

- **SPIN**: Model of system design and requirements (properties) as input. Checker tells us if they hold.
- If not, a **counterexample** is produced (system run that violates the requirement)
- ProMeLa (Process Meta Language) to describe models of distributed systems (c-like)

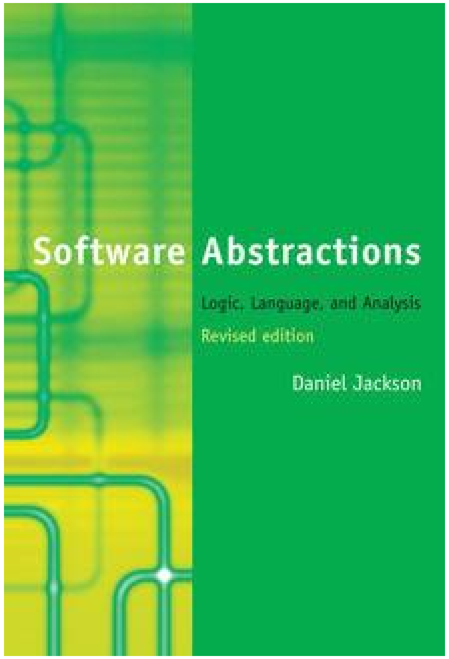
=> Model Checking in Analysis, Design and Coding Phases

http://spinroot.com/spin/whatispin.html

Lightweight Formal Methods / Agile Modeling

- **Alloy**: solver that takes constraints if a model and finds structures that satisfy them
- Can be used to both, **explore** the **model** by generating sample structures and to **check properties** by generating counterexamples.

=> Developed ecosystem: Java, Ruby and more, **used a lot**.



http://alloy.mit.edu/alloy/

Testing

What is the output of having some testing?

Simple: We test to gain confidence that our system is doing the right thing.

Testing Categories

- Top Down
    - Fault injections, input generators
- Bottom-Up
    - Lineage driven fault injections
- White/Black Box
    - We know (or not) about the system

Benefits:

- Pay-as-you-go: **gradually increase confidence**
- **Sacrifice rigor** (less certainty) for something reasonable.
- Problem: Challenged by large state space.

Types of Testing

Small	Medium	Complex
● Unit ● Integration (maybe)	● Unit ● Integration ● System ● Acceptance	● Unit ● Integration ● System ● Acceptance ● Compatibility ● Fault Injection ● Stress/performance ● Canary ● Regression

Paper 1: Simple Testing Can Prevent Most Critical Failures

An Analysis of Production Failures in Distributed Data-Intensive Systems

- Randomly picked 198 user reported failures on popular distributed data-intensive systems issue trackers: Cassandra, HBase, Hadoop Distributed File System (HDFS), Hadoop MapReduce and Redis.
- Then: White Box / Static Analysis of these failures.
- **Aspirator**: Tool capable of finding bugs (JVM). 121 new bugs and 379 bad practices.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems

Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm, *University of Toronto*

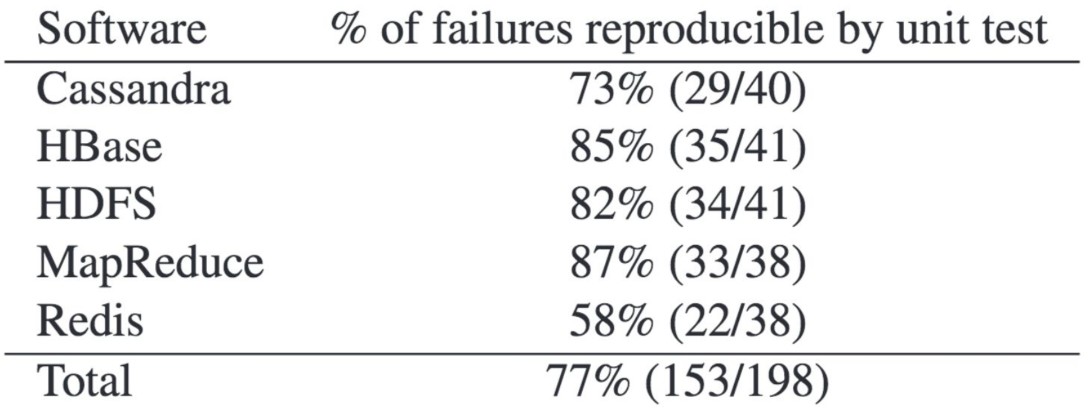
https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

Paper Download

Paper 1: Simple Testing Can Prevent Most Critical Failures

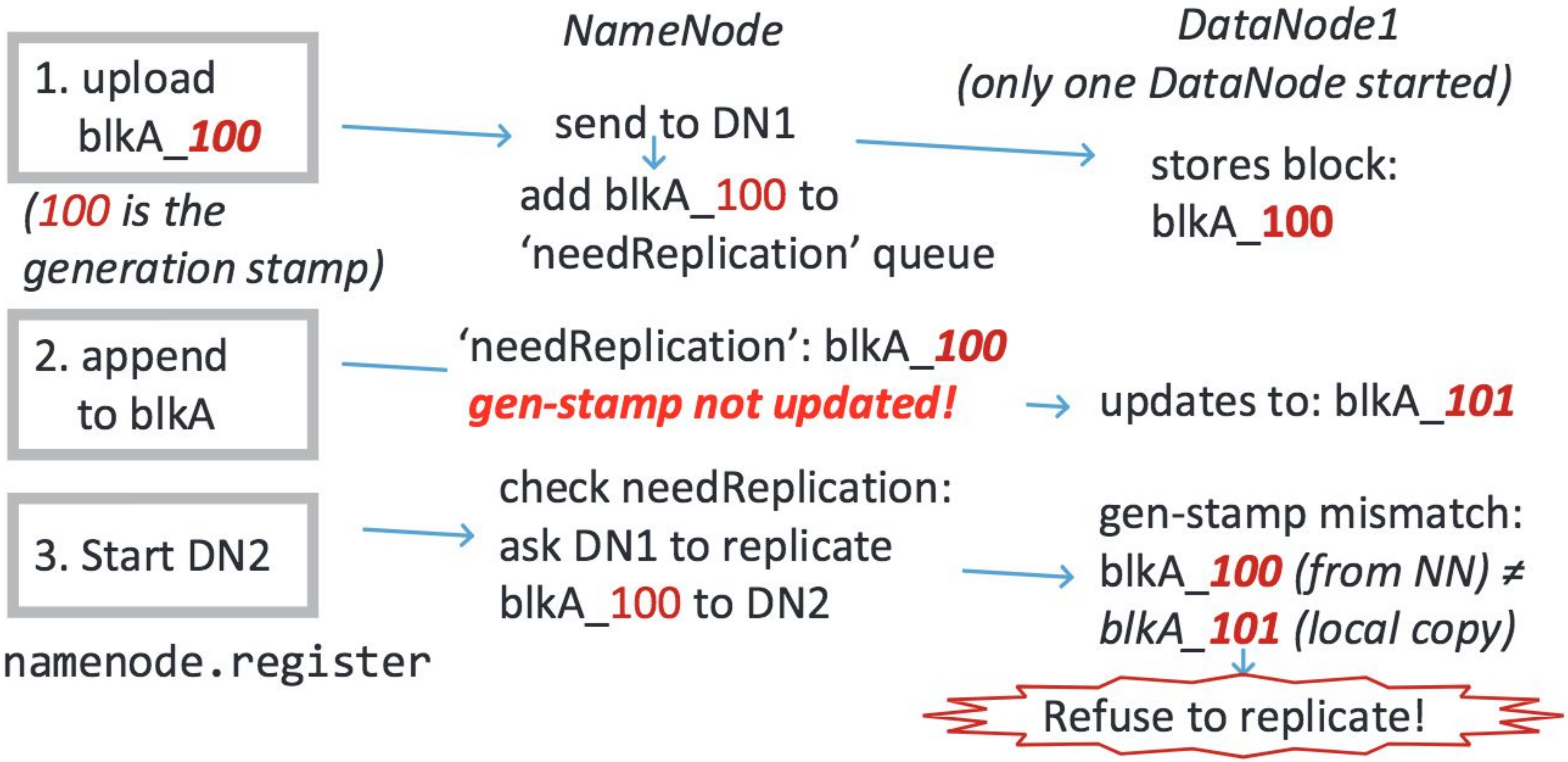
Main Insights

- Failures require only three nodes to reproduce.
- However, multiple inputs needed **in correct order** to replicate these failures (for 90% of failures <=3 inputs).
- Faulty error handling code culprint for nearly all catastrophic failures. Complex sequences.
- Logs were very useful (76% printed explicit failure related error messages).



Software	% of failures reproducible by unit test
Cassandra	73% (29/40)
HBase	85% (35/41)
HDFS	82% (34/41)
MapReduce	87% (33/38)
Redis	58% (22/38)
Total	77% (153/198)

Paper 1: Multiple inputs and correct permutation



NameNode

DataNode1

(only one DataNode started)

1. upload blkA_**100**

(100 is the generation stamp)

send to DN1

add blkA_100 to 'needReplication' queue

stores block: blkA_**100**

2. append to blkA

'needReplication': blkA_***100***

gen-stamp not updated!

updates to: blkA_***101***

3. Start DN2

namenode.register

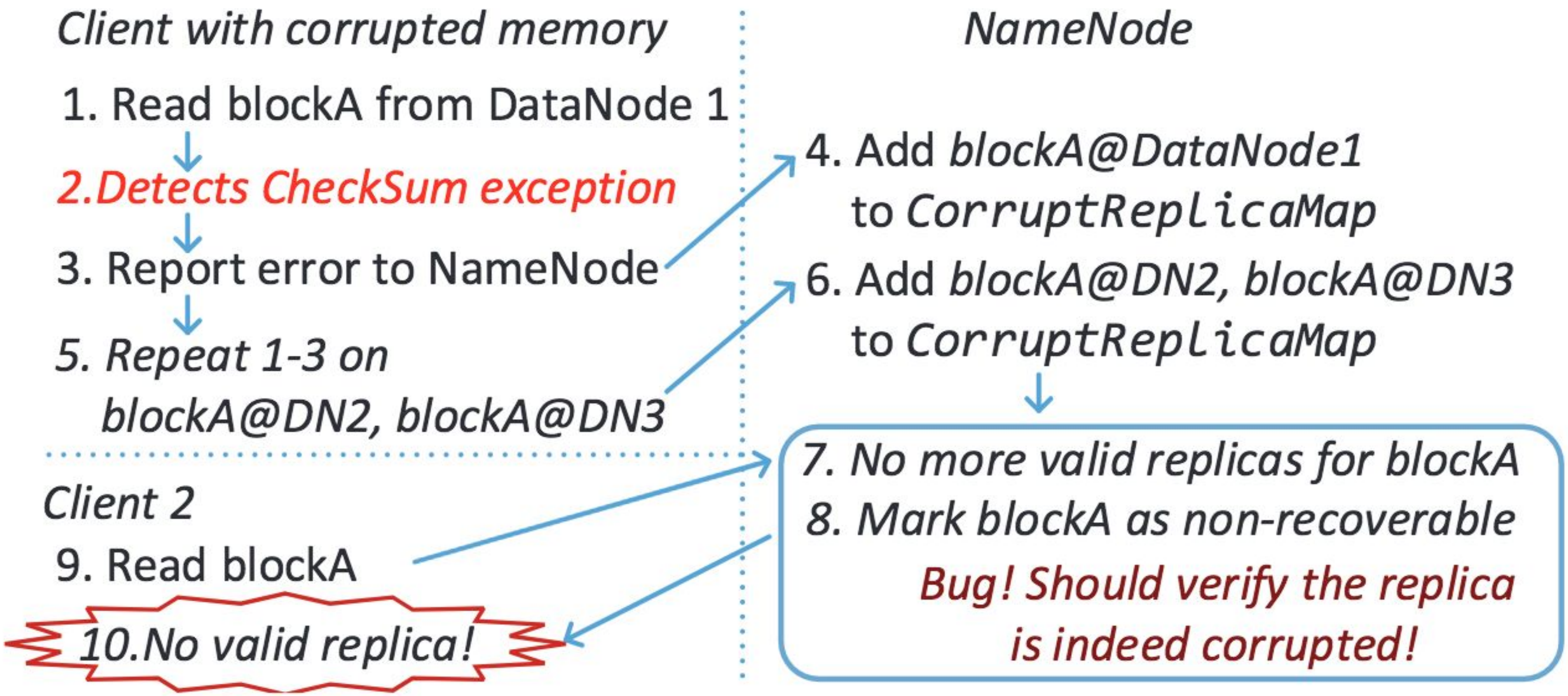
check needReplication: ask DN1 to replicate blkA_100 to DN2

gen-stamp mismatch: blkA_***100*** *(from NN)* ≠ *blkA_**101** (local copy)*

Refuse to replicate!

- Three input events are required for the failure to manifest.
- Order matters!

Paper 1: Complex Bugs



Client with corrupted memory

1. Read blockA from DataNode 1

2.Detects CheckSum exception

3. Report error to NameNode

5. Repeat 1-3 on blockA@DN2, blockA@DN3

Client 2

9. Read blockA

10.No valid replica!

NameNode

4. Add *blockA@DataNode1* to *CorruptReplicaMap*

6. Add *blockA@DN2, blockA@DN3* to *CorruptReplicaMap*

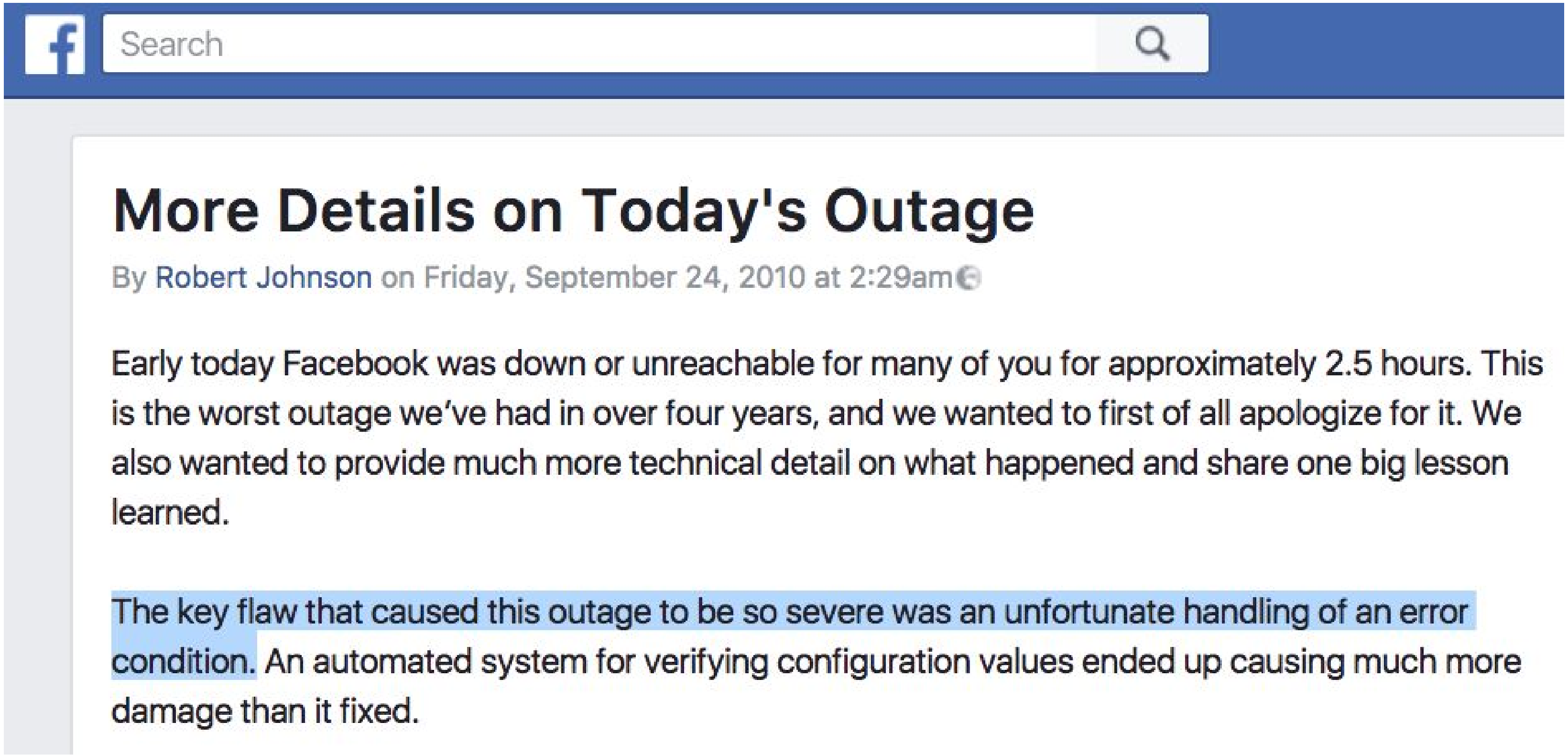
7. No more valid replicas for blockA

8. Mark blockA as non-recoverable

Bug! Should verify the replica is indeed corrupted!

- Cannot trust in anything => byzantine kind of failure

Paper 1: In Industry



More Details on Today's Outage

By Robert Johnson on Friday, September 24, 2010 at 2:29am

Early today Facebook was down or unreachable for many of you for approximately 2.5 hours. This is the worst outage we've had in over four years, and we wanted to first of all apologize for it. We also wanted to provide much more technical detail on what happened and share one big lesson learned.

The key flaw that caused this outage to be so severe was an unfortunate handling of an error condition. An automated system for verifying configuration values ended up causing much more damage than it fixed.

Facebook: More Details on Today's Outage

Paper 1: In Industry

“...Two factors caused the situation in this EBS cluster to degrade further during the early part of the event. First, the nodes failing to find new **nodes did not back off aggressively enough when they could not find space, but instead, continued to search repeatedly**. There was also a race condition in the code on the EBS nodes that, with a very low probability, caused them to fail when they were concurrently closing a large number of requests for replication...”
-- Amazon EC2 outage 2011

Paper 2: Bottom-Up / Molly: Lineage driven fault injection

Top-Down approach: Perturb and observe behavior of complex system.

=> most commonly through **fault injection**.

However, search space is very big:

- Poorly suited to discovering rare counterexamples involving complex combinations of multiple instances and types of faults (e.g., a network partition followed by a crash failure).
- Might discover counterexamples that are not related to any meaningful fault tolerance bugs.

Paper 2: Bottom-Up / Molly: Lineage driven fault injection

MOLLY’s approach:

- Reasons backwards from correct system outcomes and determines if a failure could have prevented it.
- Molly **only injects the faults it can prove might affect an outcome.**
- Counterexamples + Lineage visualizations to **help you understand why**.
- If fault-tolerance bugs exist for a particular configuration, MOLLY finds them rapidly using an order of magnitude fewer executions than random fault injection.
- Otherwise, MOLLY certifies that the code is bug-free for that configuration.

See paper for more.

Formal Methods and Testing in Industry

Paper 1: Paxos Made Live - An Engineering Perspective

Goal:

- Replace the current commercial database of Chubby with a database that uses Paxos for finding consensus (Chubby is a fault tolerant system at Google that provides distributed locking mechanism and small file storage (e.g., metadata). It is used by many systems at Google, e.g., GFS, Bigtable).

- See paper for more.
- And for more on Chubby, see The Chubby lock service for loosely-coupled distributed systems

Paper 1: Paxos Made Live - An Engineering Perspective

Problem:

Fault-tolerant algorithms are notoriously hard to express correctly, even worse when intermingled with all other system code.

Approach:

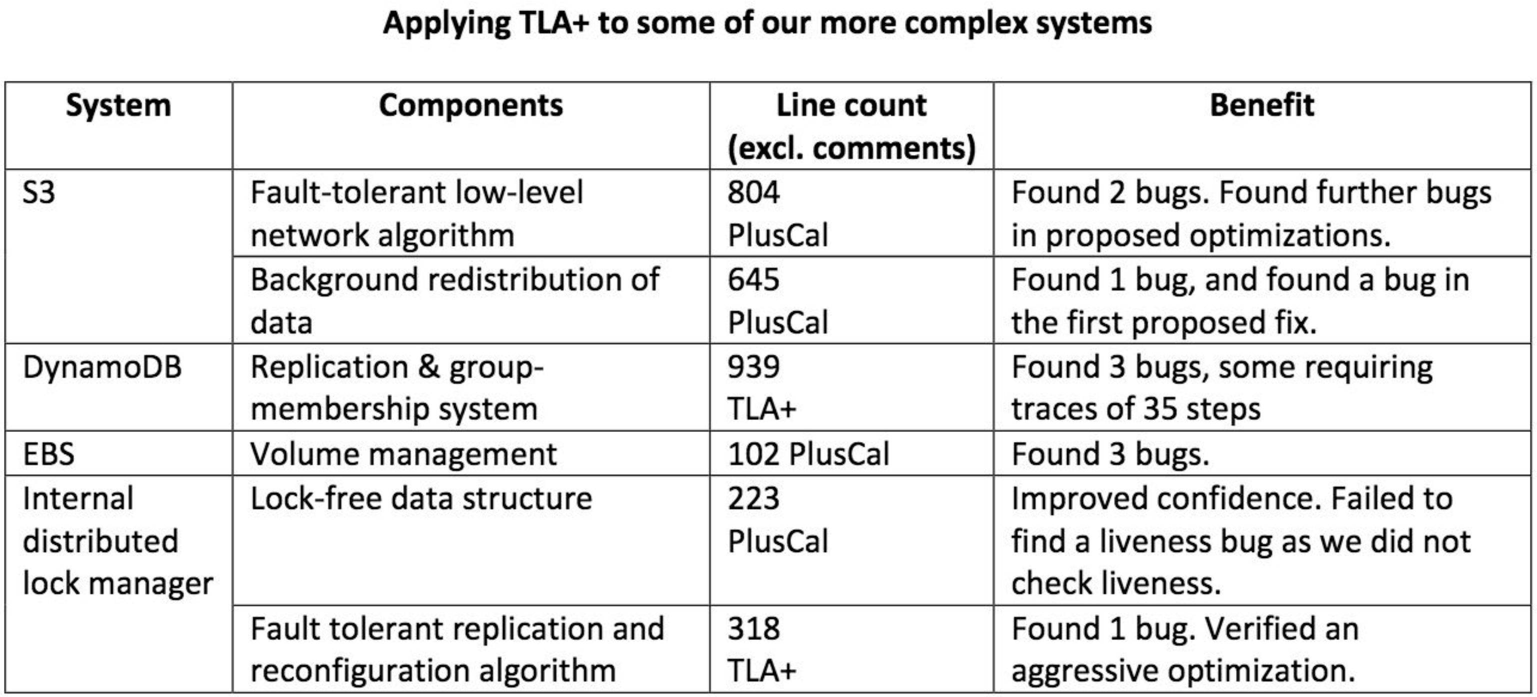
- State machine specification language and compiler to translate specs to C++.
- Core algorithm: **2 explicit state machines**
- **Test safety vs. liveness mode** for tests.
  All tests start in safety and inject random faults. Tests turned to **liveness** mode to verify system is not deadlocked. **Repeatable**.

Paper 2: Use of Formal Methods at Amazon Web Services

- Precise description of system in TLA+ (PlusCal language, similar to C).
- Used it in six large complex real-world systems. 7 teams use TLA+.
- Found subtle bugs and confidence to make **aggressive optimizations** w/o sacrificing correctness.
- Use formal specification **to teach** system to new engineers.

amazon

Paper 2: Use of Formal Methods at Amazon Web Services



Applying TLA+ to some of our more complex systems

System	Components	Line count (excl. comments)	Benefit
S3	Fault-tolerant low-level network algorithm	804 PlusCal	Found 2 bugs. Found further bugs in proposed optimizations.
	Background redistribution of data	645 PlusCal	Found 1 bug, and found a bug in the first proposed fix.
DynamoDB	Replication & group-membership system	939 TLA+	Found 3 bugs, some requiring traces of 35 steps
EBS	Volume management	102 PlusCal	Found 3 bugs.
Internal distributed lock manager	Lock-free data structure	223 PlusCal	Improved confidence. Failed to find a liveness bug as we did not check liveness.
	Fault tolerant replication and reconfiguration algorithm	318 TLA+	Found 1 bug. Verified an aggressive optimization.

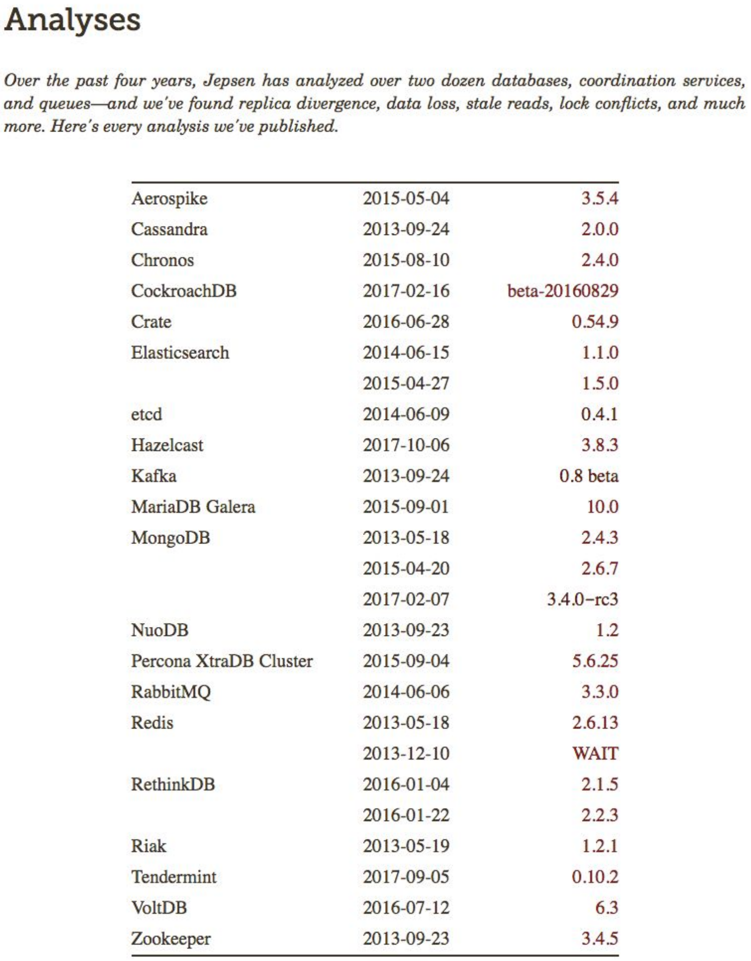
Also TLA+ has been used to design Microsoft's Cosmos DB.

See also: Foundations of Azure Cosmos DB with Dr. Leslie Lamport

Jepsen Tool

- Tool for testing the performance of distributed systems undergoing network partitions.
- How well is data consistency preserved when network connections between nodes are interrupted?
- Basic working: Jepsen spins up a cluster of hosts to serve as a distributed database system, simulates a network partition, then sees how the system manages database operations under such conditions.
- Numerous analyses of distributed systems on https://jepsen.io/analyses

https://github.com/jepsen-io/jepsen



Analyses

Over the past four years, Jepsen has analyzed over two dozen databases, coordination services, and queues—and we've found replica divergence, data loss, stale reads, lock conflicts, and much more. Here's every analysis we've published.

Aerospike	2015-05-04	3.5.4
Cassandra	2013-09-24	2.0.0
Chronos	2015-08-10	2.4.0
CockroachDB	2017-02-16	beta-20160829
Crate	2016-06-28	0.54.9
Elasticsearch	2014-06-15	1.1.0
	2015-04-27	1.5.0
etcd	2014-06-09	0.4.1
Hazelcast	2017-10-06	3.8.3
Kafka	2013-09-24	0.8 beta
MariaDB Galera	2015-09-01	10.0
MongoDB	2013-05-18	2.4.3
	2015-04-20	2.6.7
	2017-02-07	3.4.0-rc3
NuoDB	2013-09-23	1.2
Percona XtraDB Cluster	2015-09-04	5.6.25
RabbitMQ	2014-06-06	3.3.0
Redis	2013-05-18	2.6.13
	2013-12-10	WAIT
RethinkDB	2016-01-04	2.1.5
	2016-01-22	2.2.3
Riak	2013-05-19	1.2.1
Tendermint	2017-09-05	0.10.2
VoltDB	2016-07-12	6.3
Zookeeper	2013-09-23	3.4.5

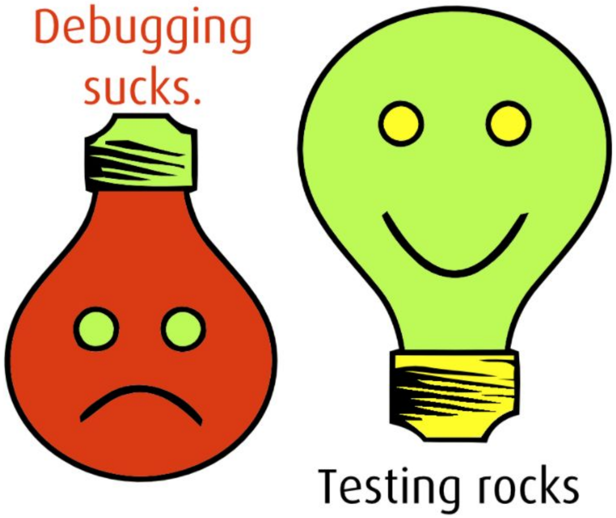
Conclusion: Visibility

- Visual aggregation of test results
- Target architecture and legacy versions greatly increases state space
- Make errors a big deal
- Alerts and monitoring
- Logs:
    - Logs are verbose and mostly awful, but they are very useful
    - Mind verbosity and configuration. Use different modes if tests repeatable.

Conclusions

Test the full distributed system: **client**, **system** and **provisioning code**.

- Have decent tests, even simple unit tests help to avoid catastrophic failures in complex distributed systems.
- Invest in visibility
- Formal methods actually are used in industry for big, complex distributed systems.



Tools: Debugging, Tracing, Profiling

Definitions

Debugging is the process of finding and resolving of defects or problem within the program that prevent correct operation of computer software or a system.

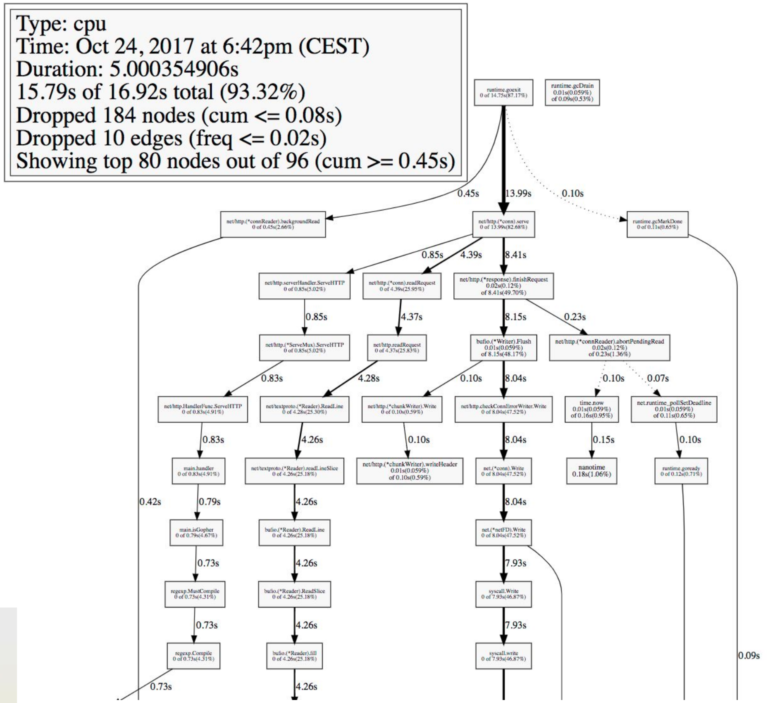
Profiling updates summary statistics of execution when an event occurs. It uses the occurrence of an event to keep track of statistics of performance metrics. These statistics are maintained at runtime as the process executes and are stored in profile data files when the program terminates. Profiles are subsequently analyzed to present the user with aggregate summaries of the metrics.

Tracing, on the other hand, records a detailed log of timestamped events and their attributes. It reveals the temporal aspect of the execution. This allows the user to see when and where routine transitions, communication and user defined events take place. Thus, it helps in understanding the behavior of the parallel program.

Profiling

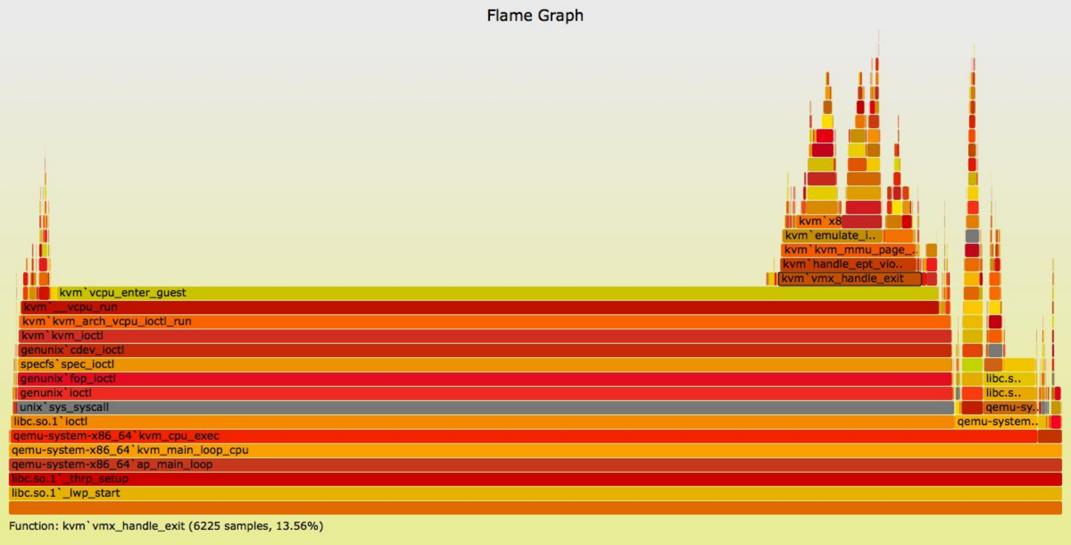
In Go:

- Go Benchmarks
- pprof
- Visualisation using Flame Graphs



```
Type: cpu
Time: Oct 24, 2017 at 6:42pm (CEST)
Duration: 5.000354906s
15.79s of 16.92s total (93.32%)
Dropped 184 nodes (cum <= 0.08s)
Dropped 10 edges (freq <= 0.02s)
Showing top 80 nodes out of 96 (cum >= 0.45s)
```

runtime.goexit 0 of 14.75s(87.17%)
runtime.gcDrain 0.01s(0.059%) of 0.09s(0.53%)
0.45s
13.99s
0.10s
net/http.(*connReader).backgroundRead 0 of 0.45s(2.66%)
net/http.(*conn).serve 0 of 13.99s(82.68%)
runtime.gcMarkDone 0 of 0.11s(0.65%)
0.85s
4.39s
8.41s
net/http.serverHandler.ServeHTTP 0 of 0.85s(5.02%)
net/http.(*conn).readRequest 0 of 4.39s(25.95%)
net/http.(*response).finishRequest 0.02s(0.12%) of 8.41s(49.70%)
0.85s
4.37s
8.15s
0.23s
net/http.(*ServeMux).ServeHTTP 0 of 0.85s(5.02%)
net/http.readRequest 0 of 4.37s(25.83%)
bufio.(*Writer).Flush 0.01s(0.059%) of 8.15s(48.17%)
net/http.(*connReader).abortPendingRead 0.02s(0.12%) of 0.23s(1.36%)
0.83s
4.28s
0.10s
8.04s
0.10s
0.07s
net/http.HandlerFunc.ServeHTTP 0 of 0.83s(4.91%)
net/textproto.(*Reader).ReadLine 0 of 4.28s(25.30%)
net/http.(*chunkWriter).Write 0 of 0.10s(0.59%)
net/http.checkConnErrorWriter.Write 0 of 8.04s(47.52%)
time.now 0.01s(0.059%) of 0.16s(0.95%)
net.runtime_pollSetDeadline 0.01s(0.059%) of 0.11s(0.65%)
0.83s
4.26s
0.10s
8.04s
0.15s
0.10s
main.handler 0 of 0.83s(4.91%)
net/textproto.(*Reader).readLineSlice 0 of 4.26s(25.18%)
net/http.(*chunkWriter).writeHeader 0.01s(0.059%) of 0.10s(0.59%)
net.(*conn).Write 0 of 8.04s(47.52%)
nanotime 0.18s(1.06%)
runtime.goready 0 of 0.12s(0.71%)
0.42s
0.79s
4.26s
8.04s
main.isGopher 0 of 0.79s(4.67%)
bufio.(*Reader).ReadLine 0 of 4.26s(25.18%)
net.(*netFD).Write 0 of 8.04s(47.52%)
0.73s
4.26s
7.93s
regexp.MustCompile 0 of 0.73s(4.31%)
bufio.(*Reader).ReadSlice 0 of 4.26s(25.18%)
syscall.Write 0 of 7.93s(46.87%)
0.73s
4.26s
7.93s
0.09s
regexp.Compile 0 of 0.73s(4.31%)
bufio.(*Reader).fill 0 of 4.26s(25.18%)
syscall.write 0 of 7.93s(46.87%)
0.73s
4.26s

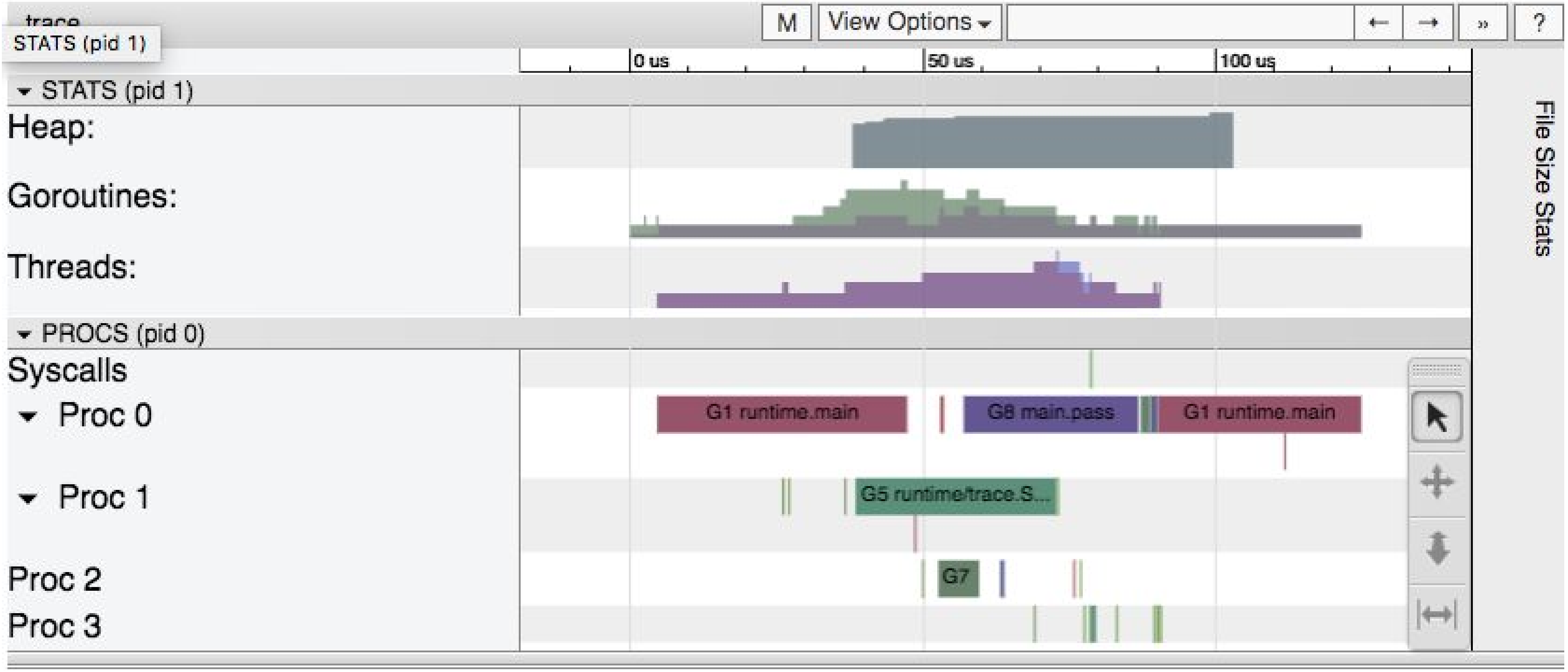


Function: kvm`vmx_handle_exit (6225 samples, 13.56%)

Tracing

In Go:

- Using go tool trace (since Go > 1.5)



Debugging: GDB - The GNU Debugger

- Can principally be used with Go.
- See https://golang.org/doc/gdb for an intro to GDB and Go.
- gcflags "-N-l" build flag to prevent compiler optimizations (e.g., aggregate variables, functions, etc.)

However:

GDB does not understand Go programs well. The stack management, threading, and runtime contain aspects that differ enough from the execution model GDB expects that they can confuse the debugger, even when the program is compiled with gccgo. As a consequence, although GDB can be useful in some situations, it is not a reliable debugger for Go programs, particularly heavily concurrent ones.

-

Debugging: GDB - The GNU Debugger

Demo?

Debugging: Delve

- Delve is a new (2015) debugger for the Go programming language.
- Works much better with goroutines.
- (Nearly) same commands as in GDB.
- There are plugins for most editors, IDEs.

=> We will use Delve in the exercise session.

https://github.com/derekparker/delve

Hands-on

Hands-on

Debugging, Testing, Profiling & Tracing Exercises (all in Go)

- https://github.com/jf87/scalable_web_systems/session-05

References

Some slides adapted from Ines Sombra’s talk on Testing in a Distributed World.