TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Programming Distributed Systems

09 Testing Distributed Systems

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Summer Term 2018

Why is it so difficult to test distributed systems?

Challenges

- Multiple sources of non-determinism
  - Scheduling
  - Network latencies
- Testing fault-tolerance requires to introduce faults
  - Typically not captured by testing frameworks
- Complexity of systems is high
  - No centralized view
  - Multiple interacting components
  - Correctness of components is often not compositional
- Formulating correctness condition is non-trivial
  - Consistency criteria
  - Timing and interaction
- Some situations to test occur after a significant amount of time and interaction
  - E.g. Timeouts, back pressure

Test support for Distributed Systems

We will discuss three approaches in detail:

1. Jepsen
2. ChaosMonkey
3. Molly

Jepsen

- Test tool for safety of distributed databases, queueing systems, consensus systems etc.
- Black-box testing by randomly inserting network partition faults
- Developed by Kyle Kingsbury, available open-source
- Approach
  1. Generate random client operations
  2. Record history
  3. Verify that history is consistent with respect to the model

Example: Jepsen Analysis for MongoDB

- MongoDB is a document-oriented database
- Primary node accepting writes and async replication to other nodes

Test scenario:

- 5 nodes, n_1 is primary
- Split into two partitions (n_1, n_2 and n_3, n_4, n_5) $\Rightarrow$ n_5 becomes new primary
- Heal the partition

How many writes get lost?

In Version 2.4.1. (2013)

> *Writes completed in 93.608 seconds 6000 total 5700 acknowledged 3319 survivors 2381 acknowledged writes lost!*

Even when imposing writes to majority:

> *6000 total 5700 acknowledged 5701 survivors 2 acknowledged writes lost! 3 unacknowledged writes found!*

- In Version 3.4.1 all tests are passed (when using the right configuration with majority writes and linearizable reads) !!

Why Is Random Testing Effective for Partition Tolerance Bugs? [2]

Typical scenarios where bugs manifest:

- *k-Splitting*: Split network into k distinct blocks (typically $k = 2$ or $k = 3$)
- *(k,l)*-Separation: Split subsets of nodes with specific role
- *Minority isolation*: Constraints on number of nodes in a block (e.g. leader is in the smaller block of a partition)

With high probability, $O(\log n)$ random partitions simultaneously provide full coverage of partitioning schemes that incur typical bugs.

ChaosMonkey

Unleash a wild monkey with a weapon in your data center (or cloud region) to randomly shoot down instances and chew through cables[1]

- Built by Netflix in 2011 during their cloud migration
- Testing for fault-tolerance and quality of service in turbulent situations
- Random selection of instances in the **production environment** and deliberately put them out of service
  - Forces engineers to built resilient systems
  - Automation of recovery

[1] http://principlesofchaos.org

Principles of Chaos Engineering[2]

Discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production

- Focus on the measurable output of a system, rather than internal attributes of the system
  - Throughput, error rates, latency percentiles, etc.
- Prioritize disturbing events either by potential impact or estimated frequency.
  - Hardware failures (e.g. dying servers)
  - Software failures (e.g. malformed messages)
  - Non-failure events (e.g. spikes in traffic)
- Aim for authenticity by running on production system
  - But reduce negative impact by minimizing blast radius
- Automize every step

[2] http://principlesofchaos.org

The Simian Army[3]

Shutdown instance Shuts down the instance using the EC2 API. The classic chaos monkey strategy.

Block all network traffic The instance is running, but cannot be reached via the network

Detach all EBS volumes The instance is running, but EBS disk I/O will fail.

Burn-CPU The instance will effectively have a much slower CPU.

Burn-IO The instance will effectively have a much slower disk.

Fill Disk This monkey writes a huge file to the root device, filling up the (typically relatively small) EC2 root disk.

[3] https://github.com/Netflix/SimianArmy/wiki/The-Chaos-Monkey-Army

Kill Processes This monkey kills any java or python programs it finds every second, simulating a faulty application, corrupted installation or faulty instance.

Null-Route This monkey null-routes the 10.0.0.0/8 network, which is used by the EC2 internal network. All EC2 <-> EC2 network traffic will fail.

Fail DNS This monkey uses iptables to block port 53 for TCP & UDP; those are the DNS traffic ports. This simulates a failure of your DNS servers.

Network Corruption This monkey corrupts a large fraction of network packets.

Network Latency This monkey introduces latency (1 second +- 50%) to all network packets.

Network Loss This monkey drops a fraction of all network packets.

Molly: Lineage-driven fault injection[1]

- Reasons backwards from correct system outcomes & determines if a failure could have prevented this outcome
- Only injects the failures that might affect an outcome
- Yields counter examples + lineage visualization
- Works on a model of the system defined in Dedalus (subset of Datalog language with explicit representation of time)

Molly - main idea

User provides program, precondition, postcondition and bounds (number of time steps to execute, maximum number of node crashes, maximum time until which failures can happen)

1. Execute program without faults
2. Find all possible explanations for the given result by reasoning backwards ("lineage")
3. Find faults that would invalidate all possible explanation (using SAT solver)
4. Run program again with injected faults
5. If new run satisfies precondition but not postcondition: report failure
6. Otherwise: Repeat until all explored

Sounds all very complex, right?

Simple Testing Can Prevent Most Critical Failures[3]

- Study of 198 randomly sampled user-reported failures from five distributed systems (Cassandra, HBase, HDFS, MapReduce, Redis)

  Almost all catastrophic failures (48 in total – 92%) are the result of incorrect handling of non-fatal errors explicitly signaled in software.

Symptom	all	catastrophic
Unexpected termination	74	17 (23%)
Incorrect result	44	1 (2%)
Data loss or potential data loss*	40	19 (48%)
Hung System	23	9 (39%)
Severe performance degradation	12	2 (17%)
Resource leak/exhaustion	5	0 (0%)
Total	198	48 (24%)

Table 2: Symptoms of failures observed by end-users or operators. The right-most column shows the number of catastrophic failures with "%" identifying the percentage of catastrophic failures over all failures with a given symptom. *: examples of potential data loss include under-replicated data blocks.

Check list to prevent errors

- Error handlers that ignore errors (e.g. just contain a log statement)
- Error handlers with “TODO”s or “FIXME”s
- Error handlers that take drastic action

$\Rightarrow$ Simple code inspections would have helped!

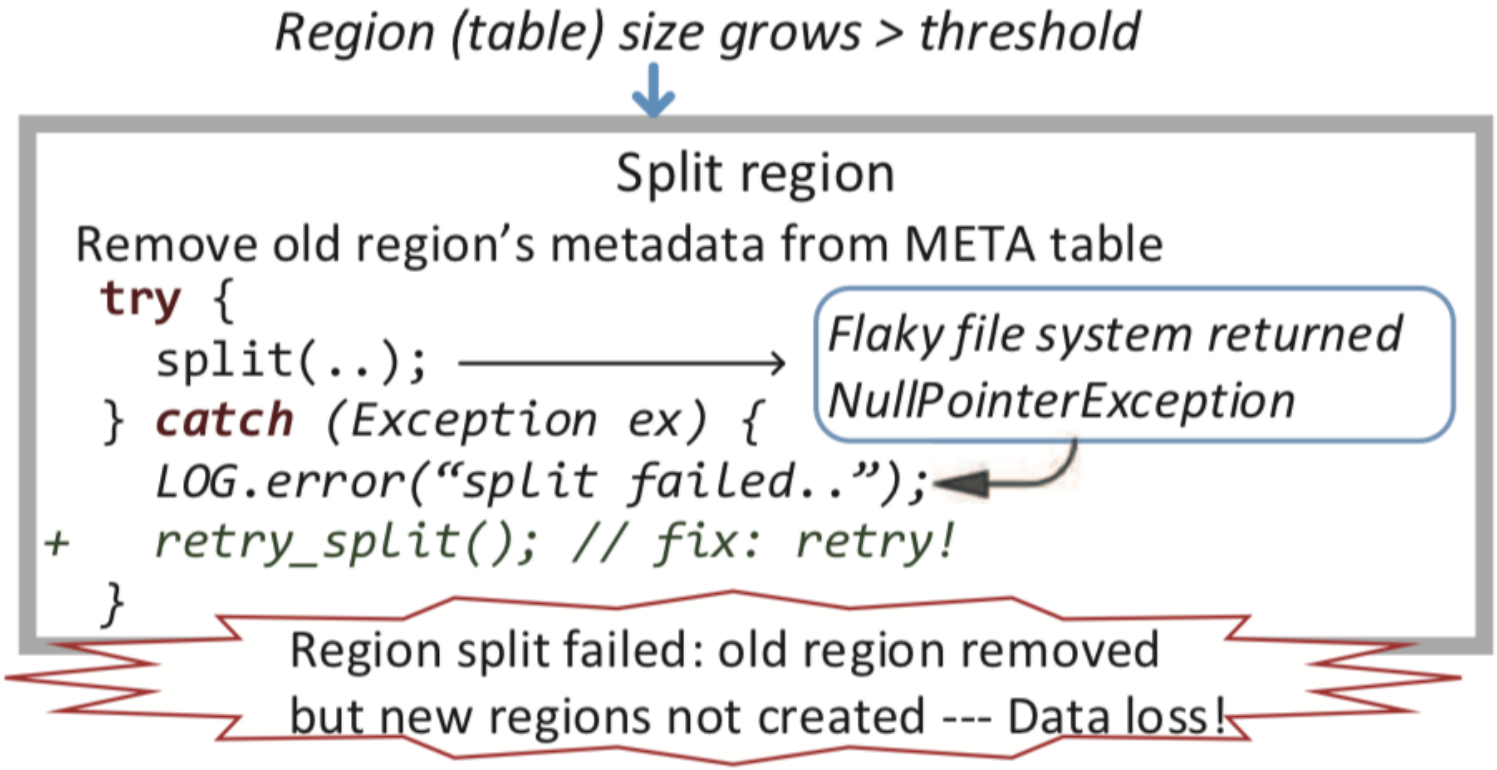

Figure 7: A data loss in HBase where the error handling was simply empty except for a logging statement. The fix was to retry in the exception handler.

User: MapReduce jobs hang when a rare Resource Manager restart occurs. *I have to ssh to every one of our 4000 nodes in a cluster and try to kill all the running Application Manager.*

Patch:

```
 catch (IOException e) {
-  // TODO
    LOG("Error event from RM: shutting down..");
+  // This can happen if RM has been restarted. Must clean up.
+  eventHandler.handle(..);
 }
```

Figure 9: A catastrophic failure in MapReduce where developers left a "TODO" in the error handler.

```
      try {
        namenode.registerDatanode();
+     } catch (RemoteException e) {
+       // retry.
      } catch (Throwable t) {
        System.exit(-1);
      }
```

RemoteExcepion is thrown due to glitch in namenode

Only intended for IncorrectVersionException

Figure 8: Entire HDFS cluster brought down by an over-catch.

No excuse for no test!

- A majority of the production failures can be reproduced by a unit test.
- It is not necessary to have a large cluster to test for and reproduce failures.
  - Almost all of the failures are guaranteed to manifest on no more than *3 nodes*
  - A vast majority will manifest on no more than 2 nodes.
- Most failures require no more than three input events to get them to manifest.
- Most failures are deterministic given the right input event sequences.

Beyond testing: Formal Methods and Verification

- Human-assisted proofs
  - Proof-assistants like Coq, Isabelle, TLA+
  - Non-trivial
- Model checking
  - TLA+ (Temporal Logic of Actions), developed by Leslie Lamport
    - Has been used to specify and verify Paxos, Raft, different services at Amazon and Microsoft, etc.
    - State-machine of properties and transitions
    - Based on invariance specifications
    - Problem: Exhaustively checks all reachable states
  - Concuerror: Stateless model checking for Erlang programs

Want to learn more?

A very comprehensive overview on testing and verification of distributed systems can be found here:
https://asatarin.github.io/testing-distributed-systems/

Further reading I

[1] Peter Alvaro, Joshua Rosen und Joseph M. Hellerstein. "Lineage-driven Fault Injection". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Hrsg. von Timos K. Sellis, Susan B. Davidson und Zachary G. Ives. ACM, 2015, S. 331–346. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2723711. URL: http://doi.acm.org/10.1145/2723372.2723711.

[2] Rupak Majumdar und Filip Niksic. "Why is random testing effective for partition tolerance bugs?". In: *PACMPL* 2.POPL (2018), 46:1–46:24. DOI: 10.1145/3158134. URL: http://doi.acm.org/10.1145/3158134.

Further reading II

[3] Ding Yuan u. a. "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, S. 249–265. ISBN: 978-1-931971-16-4. URL: http://dl.acm.org/citation.cfm?id=2685048.2685068.