Programming Distributed Systems

More on testing (Lineage-based Testing)

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Summer Term 2019

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

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 paths explored

Example: Getting Reliable Broadcast Right

Version 1 (wrong):

```
log(Node, Pload)          :- bcast(Node, Pload);
log(Node, Pload)@next     :- log(Node, Pload);
node(Node, Neighbor)@next :- node(Node, Neighbor);
log(Node2, Pload)@async   :- bcast(Node1, Pload),
                                 node(Node1, Node2);
```

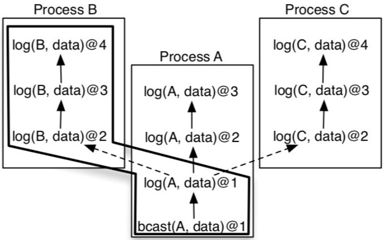
- Encoding in Dedalus as relations
- Computation is expressed via rules that describe how relations change over time
- First attribute: Location
- `@next`, `@async`: evolvement over time

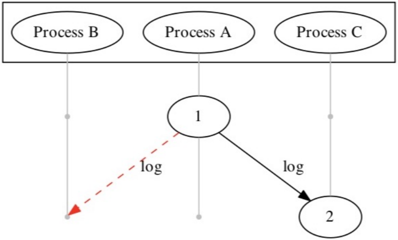
Correctness condition for Reliable Broadcast:

"If a correct node delivers a message, then all correct nodes receive it!"

```
missing_log(A, Pl) :- log(X, Pl), node(X, A), notin log(A, Pl);
pre(X, Pl)         :- log(X, Pl), notin crash(_, X, _);
post(X, Pl)        :- log(X, Pl), notin missing_log(_, Pl);
```

Example: Lineage graphs



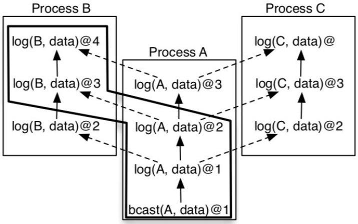


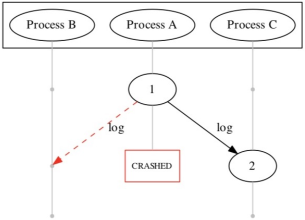
- Lineage for `log(B, Data)` is message between A and B

Example: Getting Reliable Broadcast Right, Retry

Version 2 (wrong): Add redundancy when sending!

```
bcast(N, P)@next :- bcast(N, P);
```



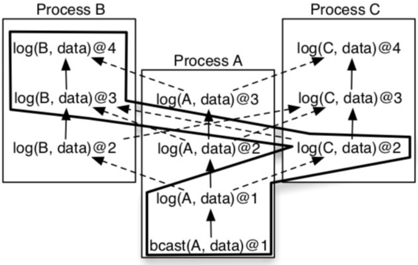


- Adversary crashes process and wins

Example: Getting Reliable Broadcast Right, Redundant

Version 3: Add redundancy on senders!

```
bcast(N, P)@next :- log(N, P);
```



- Adversay cannot make a move
- Programmer wins!

Sounds all very complex, right?

Simple Testing Can Prevent Most Critical Failures[2]

- 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

⇒ Simple code inspections would have helped!

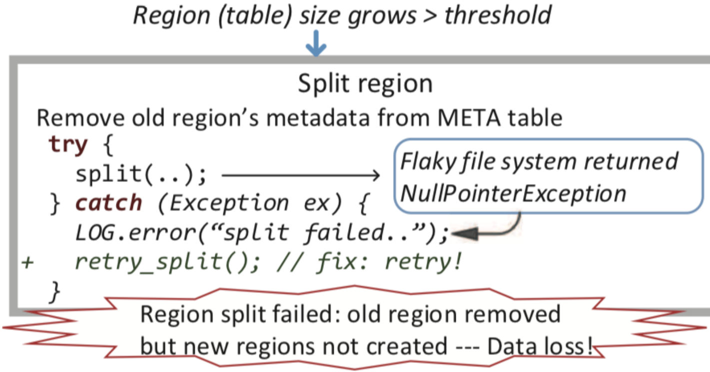


Figure 7: A data loss in HBase where the error handling was simply empty except for a logging statement. The fix was to retry in the exception handler.

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.

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

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.

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] 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.