DARE’23 – First Summer School on Distributed and Replicated Environments
STV’23 – Second Summer School on Security Testing and Verification

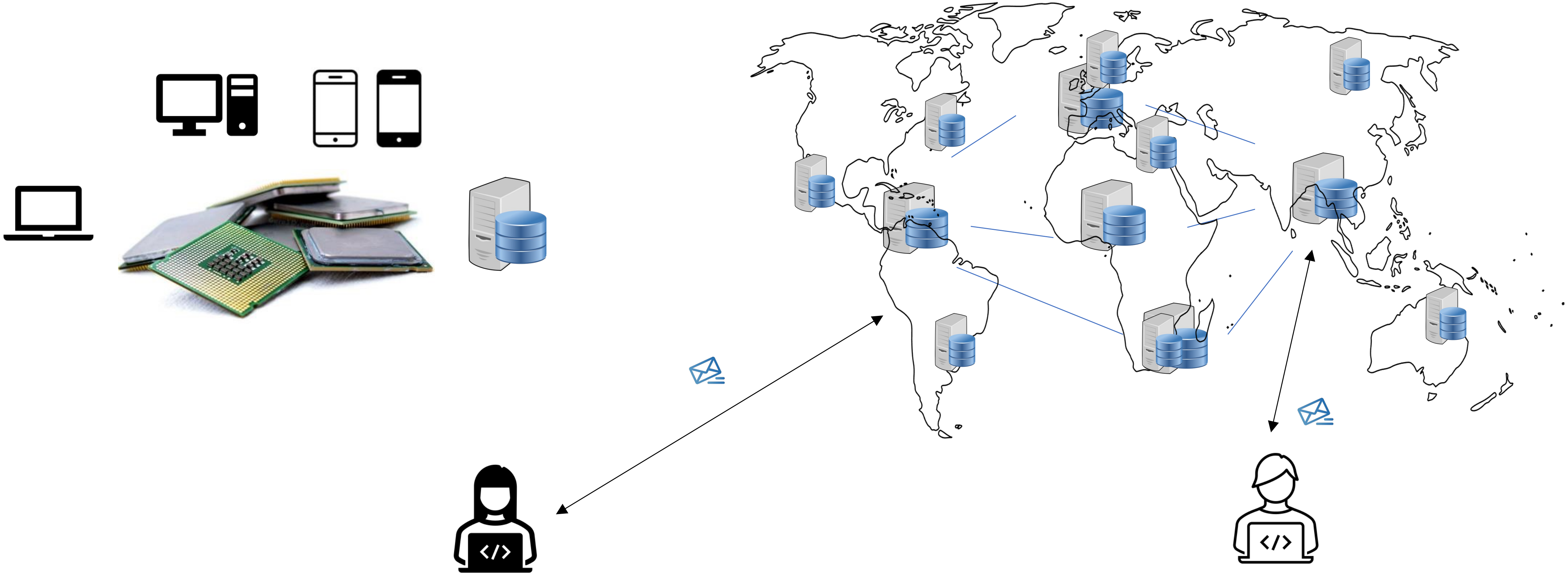
Testing Distributed System Implementations

Burcu Kulahcioglu Ozkan

b.ozkan@tudelft.nl
https://burcuku.github.io/home/

VUB, Brussels, 11 September 2023,

Ubiquitous concurrency and distribution



How familiar are you to concurrency?

- What kind of concurrent programs have you worked with?
- Have you encountered any heisenbugs?

Many bugs in distributed systems …


Cassandra / CASSANDRA-9794

Linearizable consistency for lightweight transactions is not achieved


Solr / SOLR-1144

replication hang


Kafka / KAFKA-382

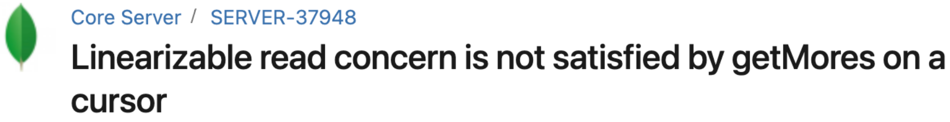
Write ordering guarantee violated


ActiveMQ / AMQ-6911

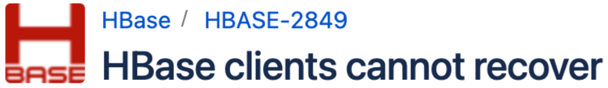
Constraint violation on failover (Postgresql)


ActiveMQ / AMQ-2780

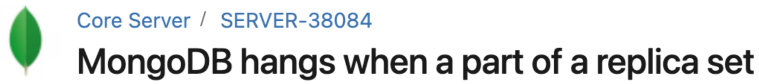
ActiveMQ not preserving Message Order


Core Server / SERVER-37948

Linearizable read concern is not satisfied by getMores on a cursor


HBase / HBASE-2849

HBase clients cannot recover


Core Server / SERVER-38084

MongoDB hangs when a part of a replica set


ZooKeeper / ZOOKEEPER-4003

Zookeeper server breakdown Frequently

Learning objectives

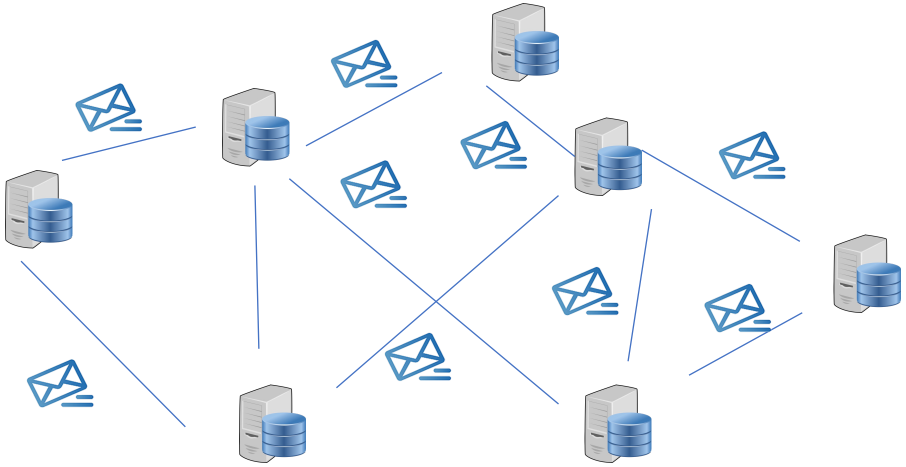
At the end of this lecture, you will be able to:

- Identify concurrency bugs in distributed systems
- Explain controlled concurrency testing for distributed systems
    - Systematic testing
    - Naïve random testing
    - Probabilistic Concurrency Testing (PCT)

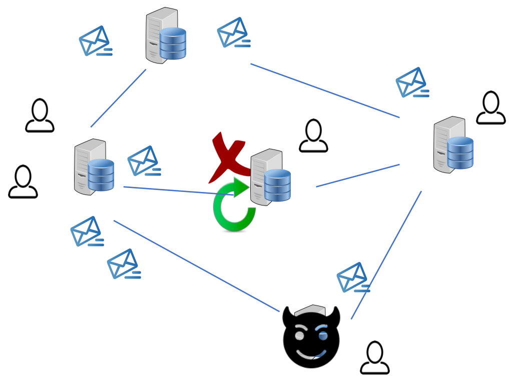
What is a distributed system?

- The processes/nodes in the system:
    - Are connected over network
    - Communicate by asynchronous messages

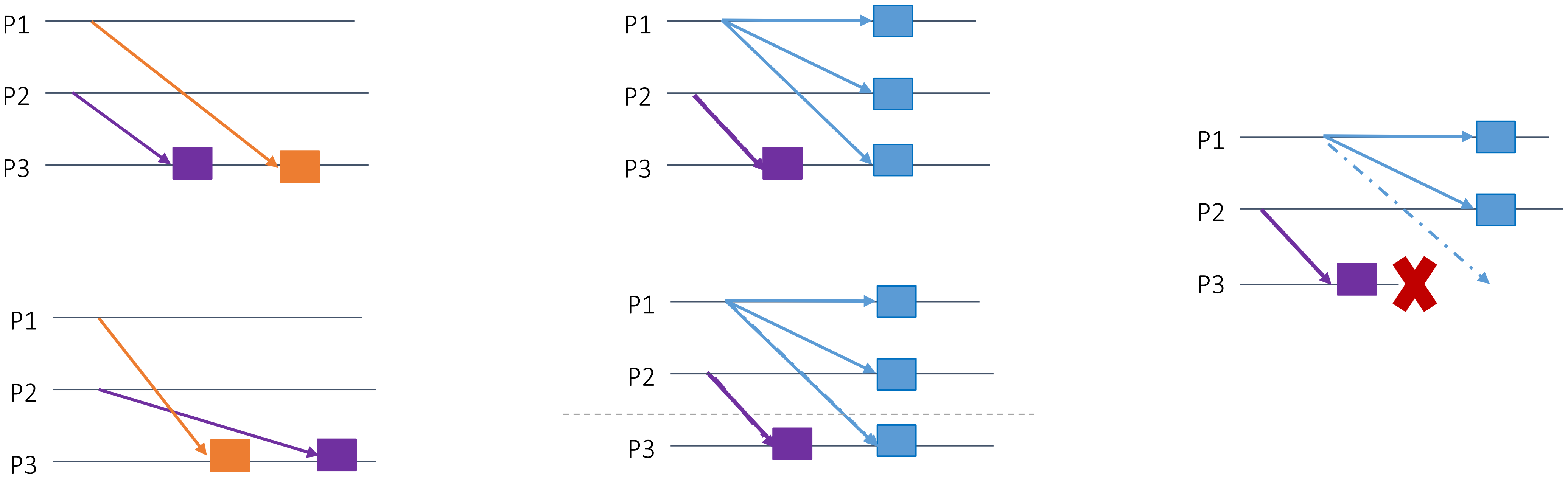
- Processes operate on their local memory and communicate by exchanging messages:
    - A process performs some local computation
    - A process sends a message
    - A process receives a message



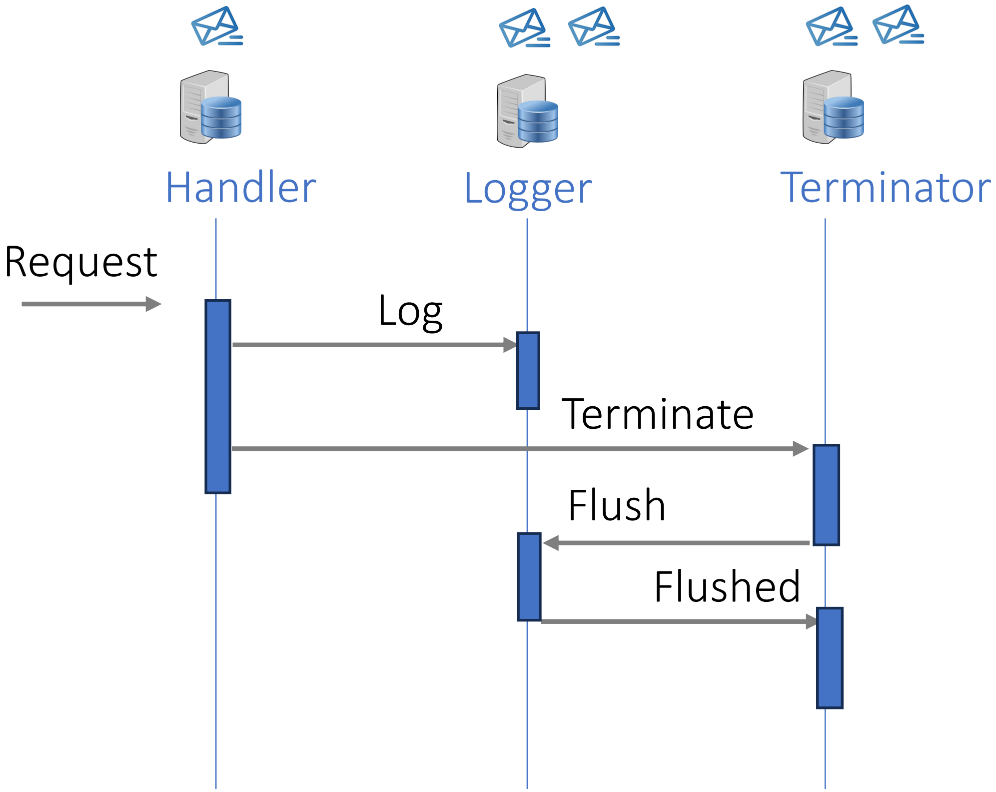
What can go wrong?



- Many components, many sources of nondeterminism



An example execution



A simplified version of a bug found in a performance testing tool Gatling [2018]
(modified example from ASE’13, OOPSLA’18)

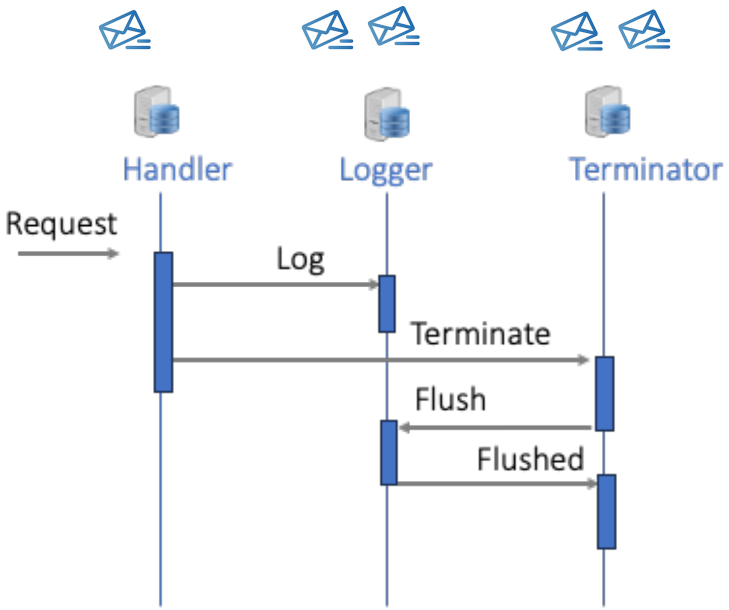
Model of distributed systems

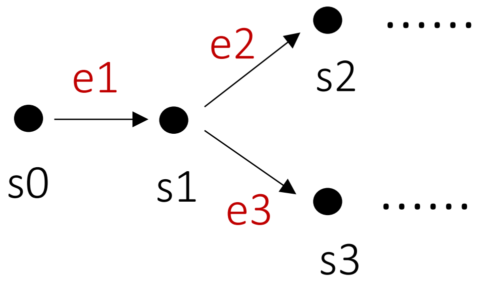
- $Nodes$:the set of nodes/processes
- $Msgs$: the set of all messages
- $Events$: $\langle recv, send, msg \rangle$

For simplicity, assume unique messages

and events as message delivery $Events$: $\langle msg \rangle$

- A state of the system is a map: $\mathrm{c}: Nodes \rightarrow 2^{\Sigma}$, from nodes to sets of enabled events
- A transition: $\mathrm{e} = \langle msg \rangle \in s(node)$
- The new state s' is obtained by removing e from $s(node)$ and adding $\mathrm{e_i}$ to $s(node_i)$ for each i: $s \xrightarrow{node:e} s'$





Model of distributed systems

- An execution is a sequence:

$$s_0 \xrightarrow{node_0 : e_0} s_1 \xrightarrow{node_1 : e_1} \ldots \xrightarrow{node_n : e_n} s_{n+1}$$

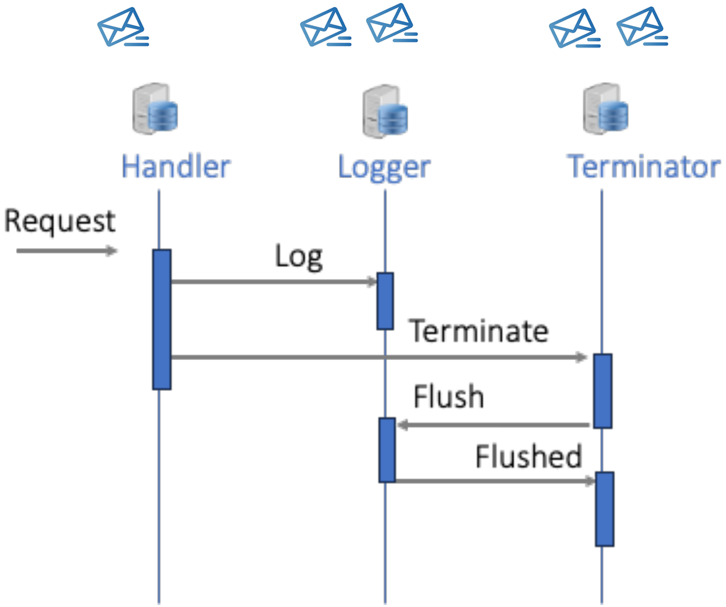
- The sequence $\langle node_0 : e_0 \rangle, \ldots \langle node_0 : e_0 \rangle$

is called a schedule

An example schedule:

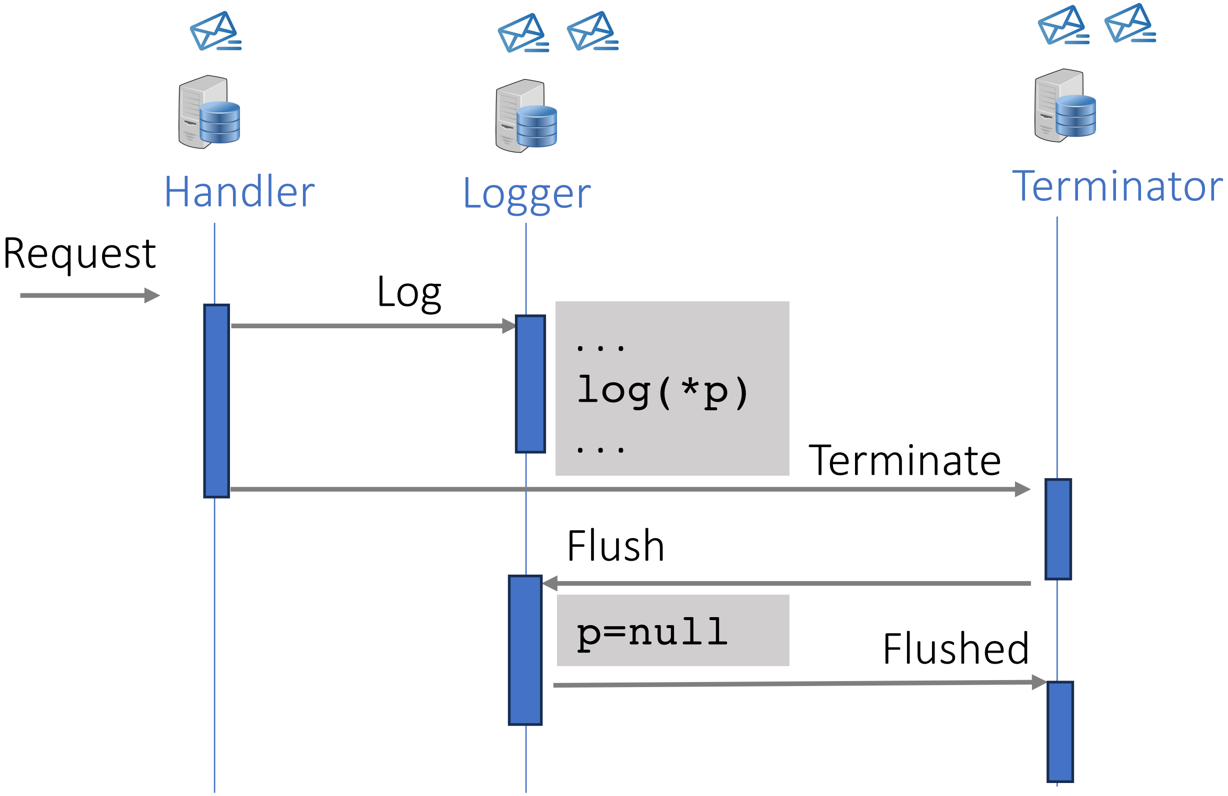
$$\begin{gathered}[\langle Handler: e_0 = \langle Request \rangle \rangle, \\ \langle Logger: e_1 = \langle Log \rangle \rangle, \\ \langle Terminator: e_2 = \langle Terminate \rangle \rangle, \\ \langle Logger: e_3 = \langle Logger, Terminator, Flush \rangle \rangle, \\ \langle Terminator: e_4 = \langle Terminator, Logger, Flushed \rangle \rangle]\end{gathered}$$

Simply: $[Request, Log, Terminate, Flush, Flushed]$



System behavior depends on the schedule

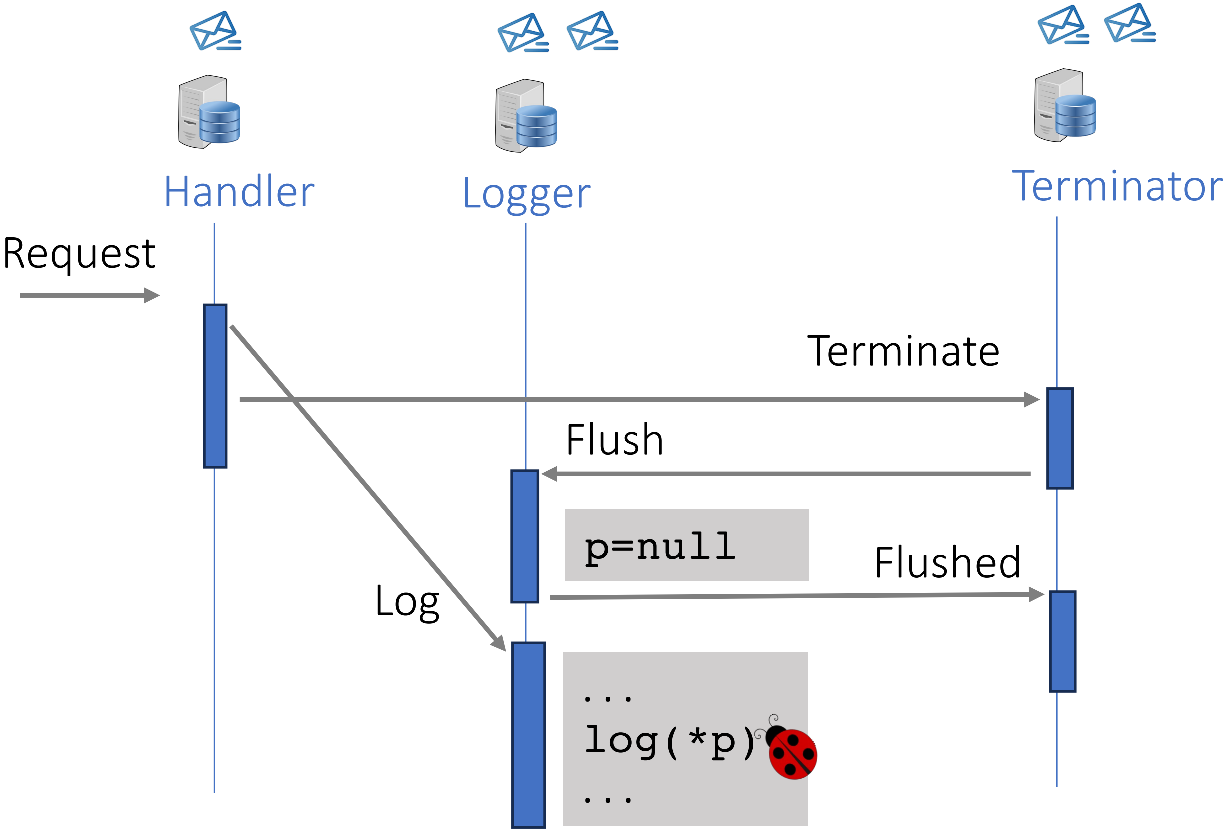
Revisit the example execution



Is it possible to hit NPE?

What is the buggy schedule?

Revisit the example execution – Order violation



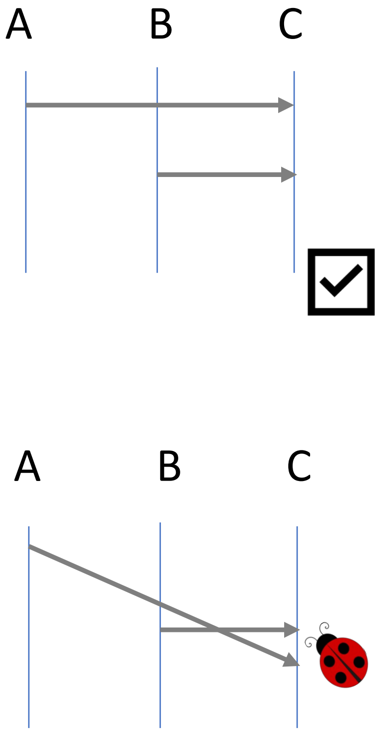
```
p=null
```

```
...
log(*p)
...
```

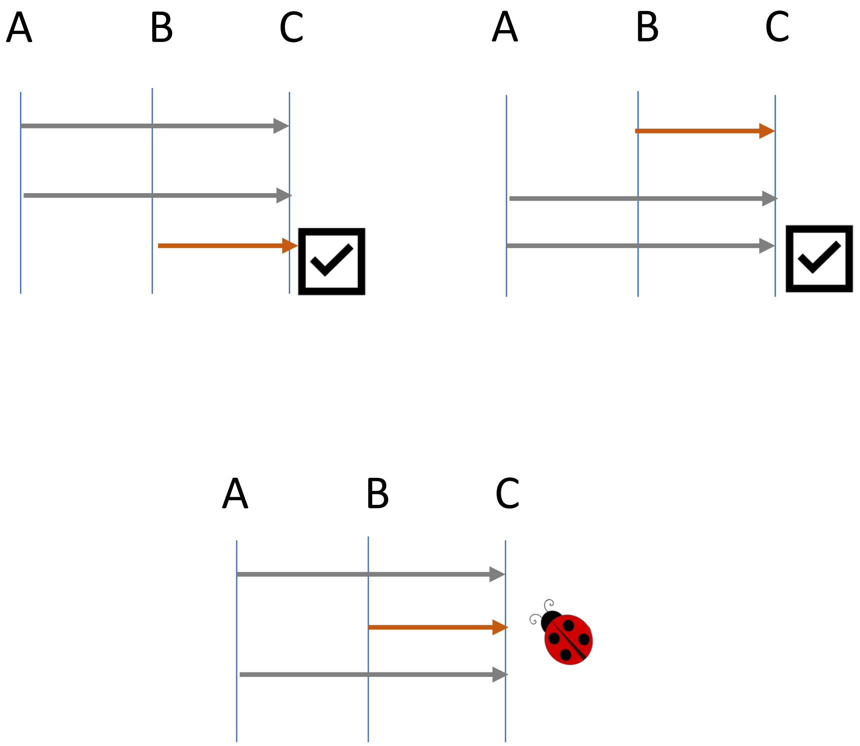
Correct: $Request, Log, Terminate, Flush, Flushed$

Buggy: $Request, Terminate, Flush, Flushed, Log$

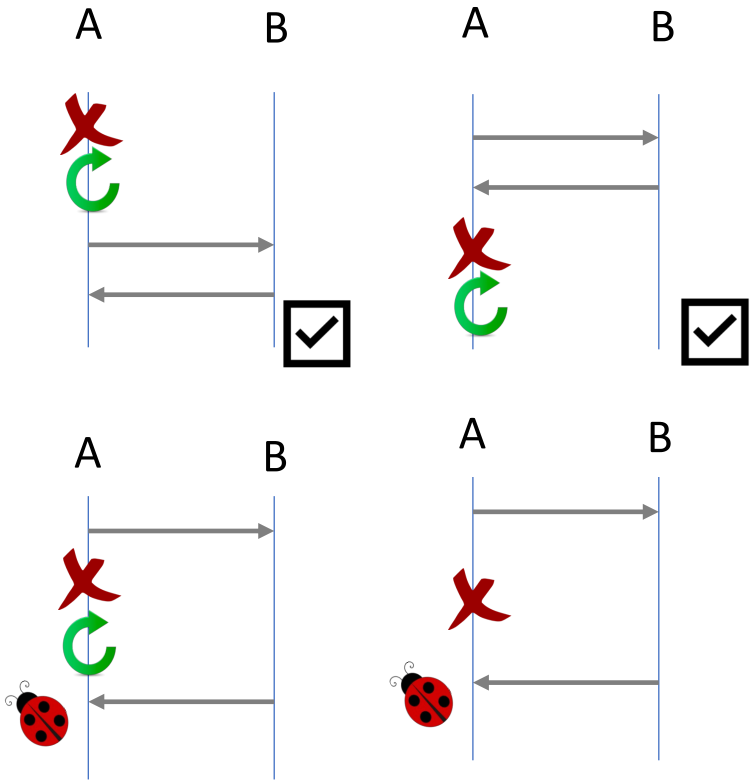
Concurrency and fault-tolerance bugs



Message order violation



Atomicity violation

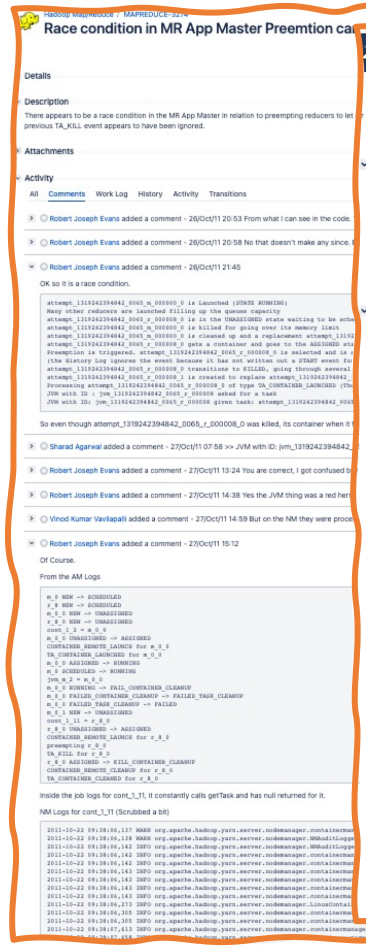


Process crash/recovery

A Taxonomy of Non-Deterministic Concurrency Bugs [Leesatapornwongsa et. Al., ASPLOS'16]

Concurrency bugs in large-scale systems are difficult to detect

Subtle execution scenarios with interleavings of many events, node crashes, network partitions



Hadoop Map/Reduce / MAPREDUCE-3274

Race condition in MR App Master Preemtion ca

Details

Description

There appears to be a race condition in the MR App Master in relation to preempting reducers to let
previous TA_KILL event appears to have been ignored.

Attachments

Activity

Robert Joseph Evans added a comment - 26/Oct/11 20:53 From what I can see in the code.

Robert Joseph Evans added a comment - 26/Oct/11 20:58 No that doesn't make any since.

Robert Joseph Evans added a comment - 26/Oct/11 21:45

OK so it is a race condition.

```
attempt_1319242394842_0065_m_000000_0 is Launched (STATE RUNNING)
Many other reducers are launched filling up the queues capacity
attempt_1319242394842_0065_r_000008_0 is in the UNASSIGNED state waiting to be sche
attempt_1319242394842_0065_m_000000_0 is killed for going over its memory limit
attempt_1319242394842_0065_m_000000_0 is cleaned up and a replacement attempt_13192
attempt_1319242394842_0065_r_000008_0 gets a container and goes to the ASSIGNED sta
Preemption is triggered. attempt_1319242394842_0065_r_000008_0 is selected and is
(the History Log ignores the event because it has not written out a START event fo
attempt_1319242394842_0065_r_000008_0 transitions to KILLED, going through several
attempt_1319242394842_0065_r_000008_1 is created to replace attempt_1319242394842_
Processing attempt_1319242394842_0065_r_000008_0 of type TA_CONTAINER_LAUNCHED (Th
JVM with ID : jvm_1319242394842_0065_r_000008 asked for a task
JVM with ID: jvm_1319242394842_0065_r_000008 given task: attempt_1319242394842_0065
```

So even though attempt_1319242394842_0065_r_000008_0 was killed, its container when it

Sharad Agarwal added a comment - 27/Oct/11 07:58 >> JVM with ID: jvm_1319242394842_

Robert Joseph Evans added a comment - 27/Oct/11 13:24 You are correct, I got confused b

Robert Joseph Evans added a comment - 27/Oct/11 14:38 Yes the JVM thing was a red herr

Vinod Kumar Vavilapalli added a comment - 27/Oct/11 14:59 But on the NM they were proce

Robert Joseph Evans added a comment - 27/Oct/11 15:12

Of Course.

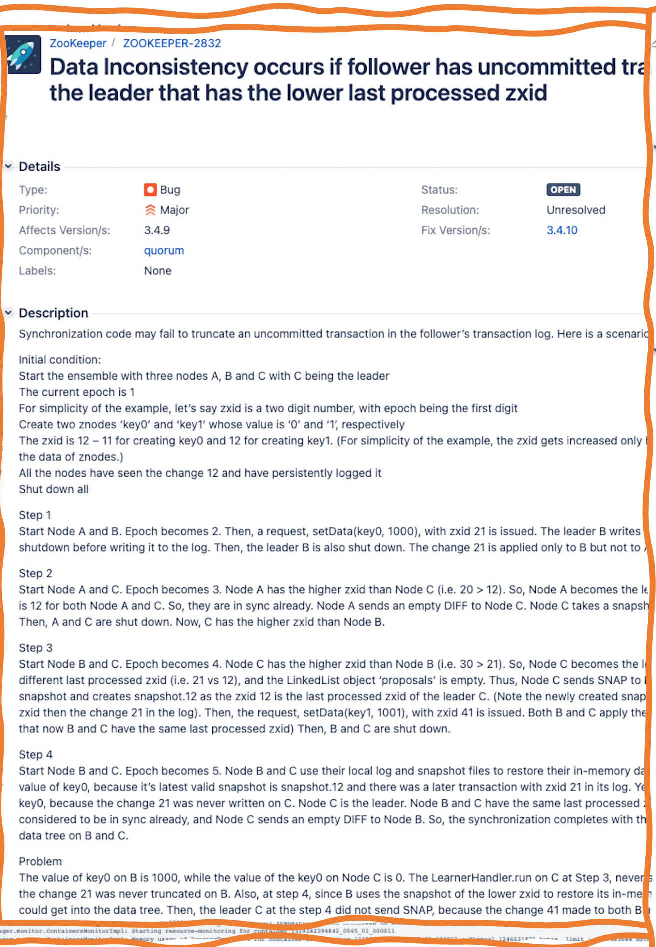
From the AM Logs

```
m_0 NEW -> SCHEDULED
r_8 NEW -> SCHEDULED
m_0_0 NEW -> UNASSIGNED
r_8_0 NEW -> UNASSIGNED
cont_1_2 = m_0_0
m_0_0 UNASSIGNED -> ASSIGNED
CONTAINER_REMOTE_LAUNCH for m_0_0
TA_CONTAINER_LAUNCHED for m_0_0
m_0_0 ASSIGNED -> RUNNING
m_0 SCHEDULED -> RUNNING
jvm_m_2 = m_0_0
m_0_0 RUNNING -> FAIL_CONTAINER_CLEANUP
m_0_0 FAILED_CONTAINER_CLEANUP -> FAILED_TASK_CLEANUP
m_0_0 FAILED_TASK_CLEANUP -> FAILED
m_0_1 NEW -> UNASSIGNED
cont_1_11 = r_8_0
r_8_0 UNASSIGNED -> ASSIGNED
CONTAINER_REMOTE_LAUNCH for r_8_0
preempting r_8_0
TA_KILL for r_8_0
r_8_0 ASSIGNED -> KILL_CONTAINER_CLEANUP
CONTAINER_REMOTE_CLEANUP for r_8_0
TA_CONTAINER_CLEANED for r_8_0
```

Inside the job logs for cont_1_11, it constantly calls getTask and has null returned for it.

NM Logs for cont_1_11 (Scrubbed a bit)

```
2011-10-22 09:38:06,137 WARN org.apache.hadoop.yarn.server.nodemanager.containerman
2011-10-22 09:38:06,138 WARN org.apache.hadoop.yarn.server.nodemanager.NMAuditLogge
2011-10-22 09:38:06,142 INFO org.apache.hadoop.yarn.server.nodemanager.NMAuditLogge
2011-10-22 09:38:06,142 INFO org.apache.hadoop.yarn.server.nodemanager.containerman
2011-10-22 09:38:06,142 INFO org.apache.hadoop.yarn.server.nodemanager.containerman
2011-10-22 09:38:06,143 INFO org.apache.hadoop.yarn.server.nodemanager.containerman
2011-10-22 09:38:06,143 INFO org.apache.hadoop.yarn.server.nodemanager.containerman
2011-10-22 09:38:06,143 INFO org.apache.hadoop.yarn.server.nodemanager.containerman
2011-10-22 09:38:06,143 INFO org.apache.hadoop.yarn.server.nodemanager.containerman
2011-10-22 09:38:06,273 INFO org.apache.hadoop.yarn.server.nodemanager.LinuxContai
2011-10-22 09:38:06,305 INFO org.apache.hadoop.yarn.server.nodemanager.containerman
2011-10-22 09:38:06,305 INFO org.apache.hadoop.yarn.server.nodemanager.containerman
2011-10-22 09:38:07,613 INFO org.apache.hadoop.yarn.server.nodemanager.containermanager.monitor.ContainersMonitorImpl: Starting resource-monitoring for container_1319242394842_0065_01_000011
```



ZooKeeper / ZOOKEEPER-2832

Data Inconsistency occurs if follower has uncommitted tra the leader that has the lower last processed zxid

Details

Type:	Bug	Status:	OPEN
Priority:	Major	Resolution:	Unresolved
Affects Version/s:	3.4.9	Fix Version/s:	3.4.10
Component/s:	quorum		
Labels:	None		

Description

Synchronization code may fail to truncate an uncommitted transaction in the follower's transaction log. Here is a scenario

Initial condition:
Start the ensemble with three nodes A, B and C with C being the leader
The current epoch is 1
For simplicity of the example, let's say zxid is a two digit number, with epoch being the first digit
Create two znodes 'key0' and 'key1' whose value is '0' and '1', respectively
The zxid is 12 – 11 for creating key0 and 12 for creating key1. (For simplicity of the example, the zxid gets increased only the data of znodes.)
All the nodes have seen the change 12 and have persistently logged it
Shut down all

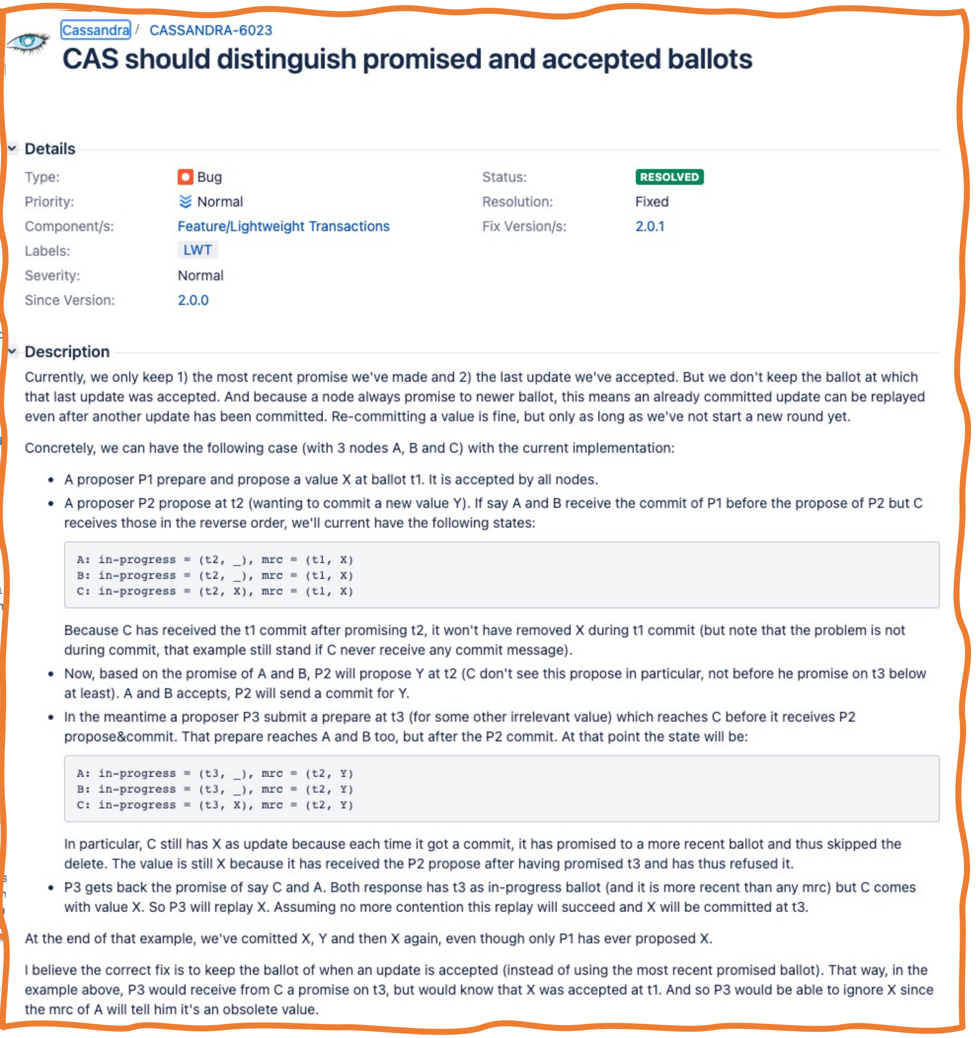
Step 1
Start Node A and B. Epoch becomes 2. Then, a request, setData(key0, 1000), with zxid 21 is issued. The leader B writes shutdown before writing it to the log. Then, the leader B is also shut down. The change 21 is applied only to B but not to

Step 2
Start Node A and C. Epoch becomes 3. Node A has the higher zxid than Node C (i.e. 20 > 12). So, Node A becomes the le is 12 for both Node A and C. So, they are in sync already. Node A sends an empty DIFF to Node C. Node C takes a snapsh Then, A and C are shut down. Now, C has the higher zxid than Node B.

Step 3
Start Node B and C. Epoch becomes 4. Node C has the higher zxid than Node B (i.e. 30 > 21). So, Node C becomes the l different last processed zxid (i.e. 21 vs 12), and the LinkedList object 'proposals' is empty. Thus, Node C sends SNAP to snapshot and creates snapshot.12 as the zxid 12 is the last processed zxid of the leader C. (Note the newly created snap zxid then the change 21 in the log). Then, the request, setData(key1, 1001), with zxid 41 is issued. Both B and C apply the that now B and C have the same last processed zxid) Then, B and C are shut down.

Step 4
Start Node B and C. Epoch becomes 5. Node B and C use their local log and snapshot files to restore their in-memory da value of key0, because it's latest valid snapshot is snapshot.12 and there was a later transaction with zxid 21 in its log. Ye key0, because the change 21 was never written on C. Node C is the leader. Node B and C have the same last processed considered to be in sync already, and Node C sends an empty DIFF to Node B. So, the synchronization completes with th data tree on B and C.

Problem
The value of key0 on B is 1000, while the value of the key0 on Node C is 0. The LearnerHandler.run on C at Step 3, never the change 21 was never truncated on B. Also, at step 4, since B uses the snapshot of the lower zxid to restore its in-me could get into the data tree. Then, the leader C at the step 4 did not send SNAP, because the change 41 made to both B



Cassandra / CASSANDRA-6023

CAS should distinguish promised and accepted ballots

Details

Type:	Bug	Status:	RESOLVED
Priority:	Normal	Resolution:	Fixed
Component/s:	Feature/Lightweight Transactions	Fix Version/s:	2.0.1
Labels:	LWT		
Severity:	Normal		
Since Version:	2.0.0		

Description

Currently, we only keep 1) the most recent promise we've made and 2) the last update we've accepted. But we don't keep the ballot at which that last update was accepted. And because a node always promise to newer ballot, this means an already committed update can be replayed even after another update has been committed. Re-committing a value is fine, but only as long as we've not start a new round yet.

Concretely, we can have the following case (with 3 nodes A, B and C) with the current implementation:

- A proposer P1 prepare and propose a value X at ballot t1. It is accepted by all nodes.
- A proposer P2 propose at t2 (wanting to commit a new value Y). If say A and B receive the commit of P1 before the propose of P2 but C receives those in the reverse order, we'll current have the following states:

  ```
  A: in-progress = (t2, _), mrc = (t1, X)
  B: in-progress = (t2, _), mrc = (t1, X)
  C: in-progress = (t2, X), mrc = (t1, X)
  ```

  Because C has received the t1 commit after promising t2, it won't have removed X during t1 commit (but note that the problem is not during commit, that example still stand if C never receive any commit message).
- Now, based on the promise of A and B, P2 will propose Y at t2 (C don't see this propose in particular, not before he promise on t3 below at least). A and B accepts, P2 will send a commit for Y.
- In the meantime a proposer P3 submit a prepare at t3 (for some other irrelevant value) which reaches C before it receives P2 propose&commit. That prepare reaches A and B too, but after the P2 commit. At that point the state will be:

  ```
  A: in-progress = (t3, _), mrc = (t2, Y)
  B: in-progress = (t3, _), mrc = (t2, Y)
  C: in-progress = (t3, X), mrc = (t2, Y)
  ```

  In particular, C still has X as update because each time it got a commit, it has promised to a more recent ballot and thus skipped the delete. The value is still X because it has received the P2 propose after having promised t3 and has thus refused it.
- P3 gets back the promise of say C and A. Both response has t3 as in-progress ballot (and it is more recent than any mrc) but C comes with value X. So P3 will replay X. Assuming no more contention this replay will succeed and X will be committed at t3.

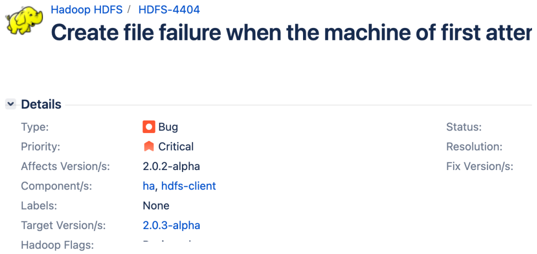
At the end of that example, we've comitted X, Y and then X again, even though only P1 has ever proposed X.

I believe the correct fix is to keep the ballot of when an update is accepted (instead of using the most recent promised ballot). That way, in the example above, P3 would receive from C a promise on t3, but would know that X was accepted at t1. And so P3 would be able to ignore X since the mrc of A will tell him it's an obsolete value.

Large-scale distributed system bugs in the wild

Cassandra / CASSANDRA-9794

Linearizable consistency for lightweight transactions is not achieved



Hadoop HDFS / HDFS-4404

Create file failure when the machine of first atter

Details

Type: Bug

Priority: Critical

Affects Version/s: 2.0.2-alpha

Component/s: ha, hdfs-client

Labels: None

Target Version/s: 2.0.3-alpha

Hadoop Flags:

Status:

Resolution:

Fix Version/s:

Solr / SOLR-1144

replication hang

Kafka / KAFKA-382

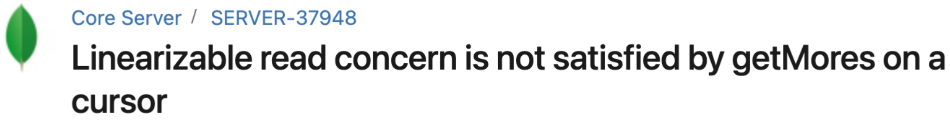
Write ordering guarantee violated

ActiveMQ / AMQ-6911

Constraint violation on failover (Postgresql)

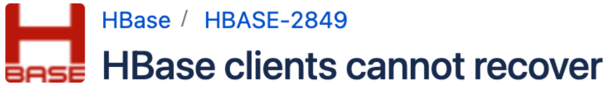
ActiveMQ / AMQ-2780

ActiveMQ not preserving Message Order



Core Server / SERVER-37948

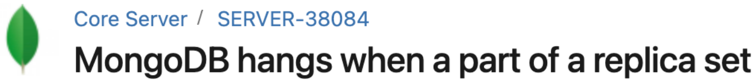
Linearizable read concern is not satisfied by getMores on a cursor



HBase / HBASE-2849

HBase clients cannot recover

Your bug report here ☺



Core Server / SERVER-38084

MongoDB hangs when a part of a replica set

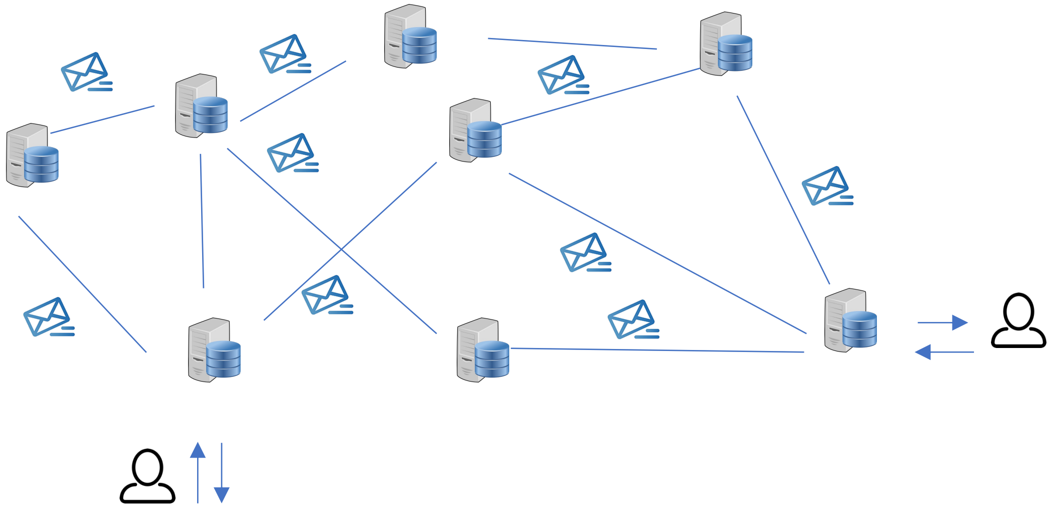
ZooKeeper / ZOOKEEPER-4003

Zookeeper server breakdown Frequently

It is hard to implement distributed systems correctly

The developers needs to reason about:

- Concurrency
- Asynchrony
- Network failures
- Partial (node) failures



Testing is practical method for discovering bugs

Learning objectives

At the end of this lecture, you will be able to:

- Identify concurrency bugs in distributed systems
- Explain controlled concurrency testing for distributed systems
    - Systematic testing
    - Naïve random testing
    - Probabilistic Concurrency Testing (PCT)

Challenges for testing distributed systems

(C0) Test oracle

- What is the correctness specification?

→ We assume it is provided
(e.g. unexpected exceptions, assertion violations, serializability of transactions, agreement of replicas)

(C1) Test harness discovery

- What are the requests/transactions to submit?

→ We randomly generate a few transactions
(small-scope hypothesis)

(C2) Enumerating executions

- What interleavings of events to exercise?

→ How to explore possible executions efficiently?
Combinatorial complexity!

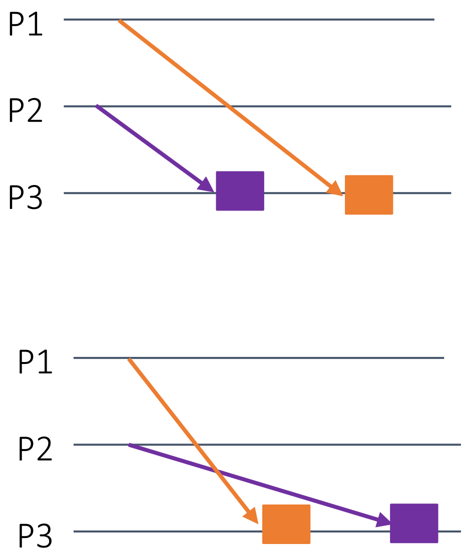
(C3) Improving interpretability

- Is the buggy trace easy to understand?

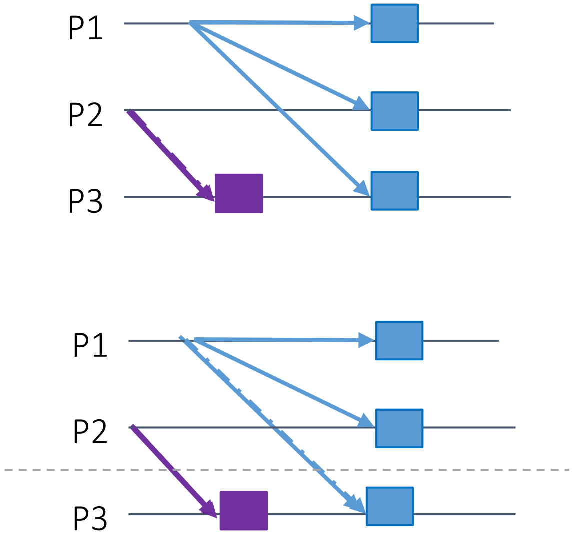
→ How to produce understandable traces?

Combinatorial complexity of possible interleavings

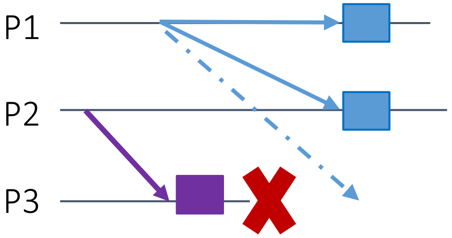
Concurrency



Network faults



Process/Node faults



(C2) Enumerating executions

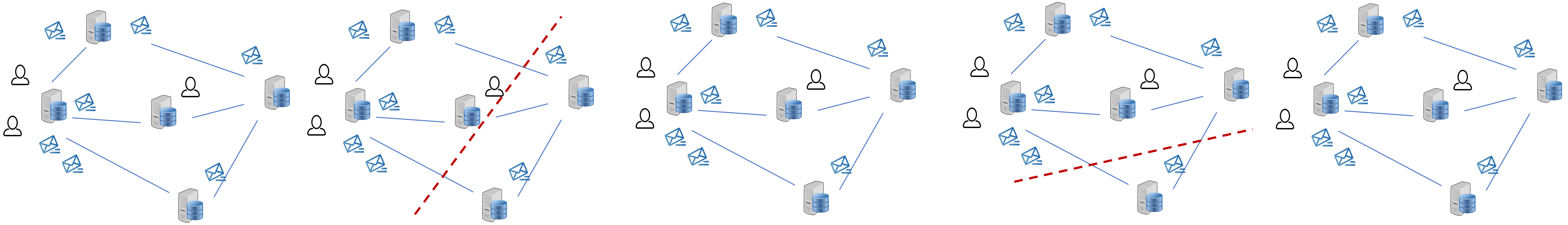
- What interleavings of events to exercise?

→ How to explore possible executions efficiently?
Combinatorial complexity!

What executions to test?

- Random fault-injection testing
  - Jepsen: Effective at finding fault-tolerance bugs
  - Theoretical explanation of the effectiveness [Majumdar & Niksic, POPL’18]

Example:



Run cluster

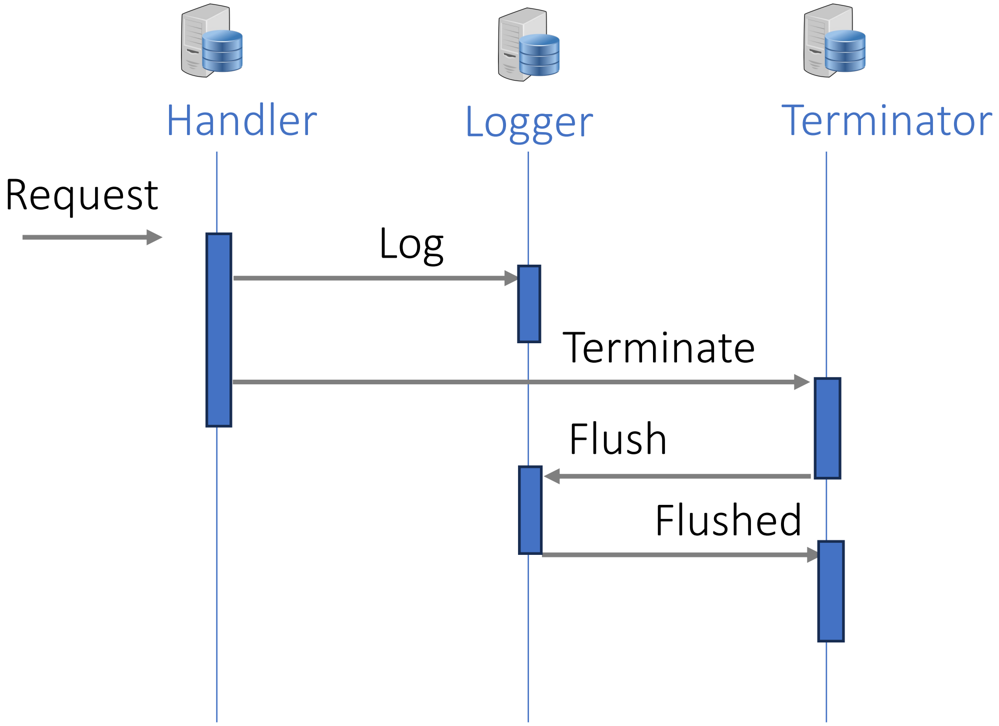
Partition the network

Recover the network

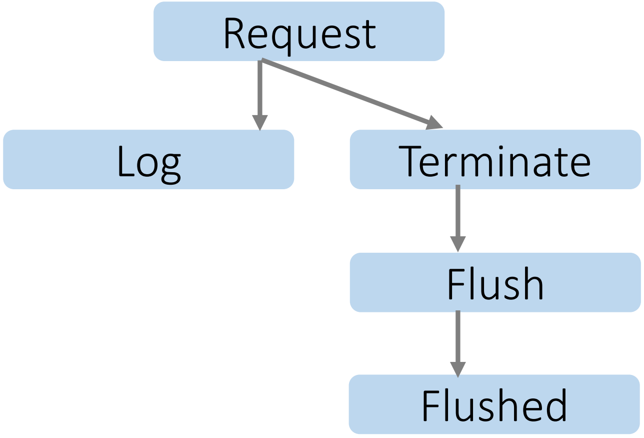
Partition the network

Recover the network
+ Check properties

Challenge: Mutual dependency between the schedule and system events



Upgrowing Poset:


Request

Log

Terminate

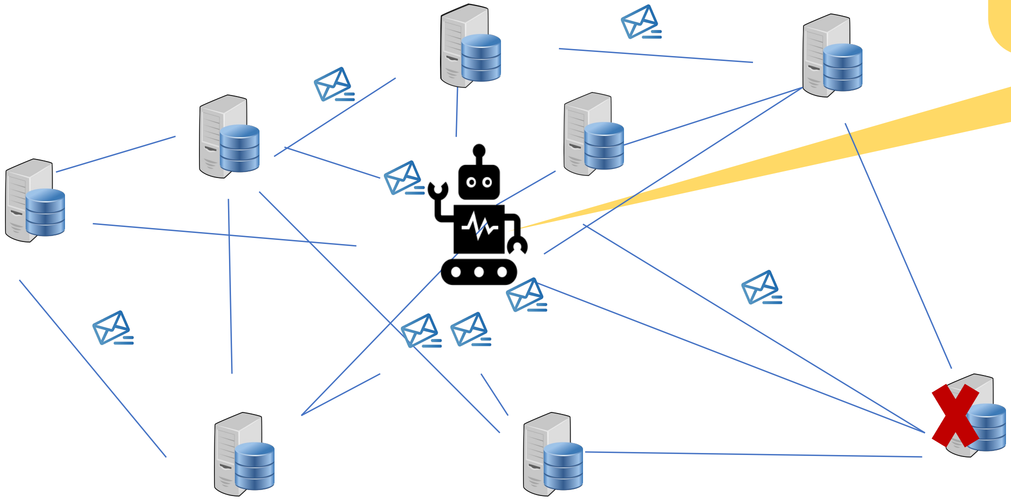
Flush

Flushed

Controlled concurrency + fault injection testing

- Control the non-determinism in the delivery order of messages and faults
- Reproduce a buggy execution for easier debugging
- Design testing strategies to explore different program executions
  - Delayed, reordered, lost messages
  - Process isolation, process crashes

What orderings of messages to schedule?
What faults to inject?
When to inject faults?



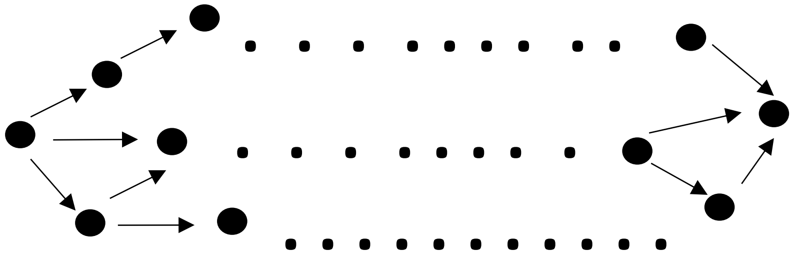
Learning objectives

At the end of this lecture, you will be able to:

- Identify concurrency bugs in distributed systems
- Explain controlled concurrency testing for distributed systems
  - Systematic testing
  - Naïve random testing
  - Probabilistic Concurrency Testing (PCT)

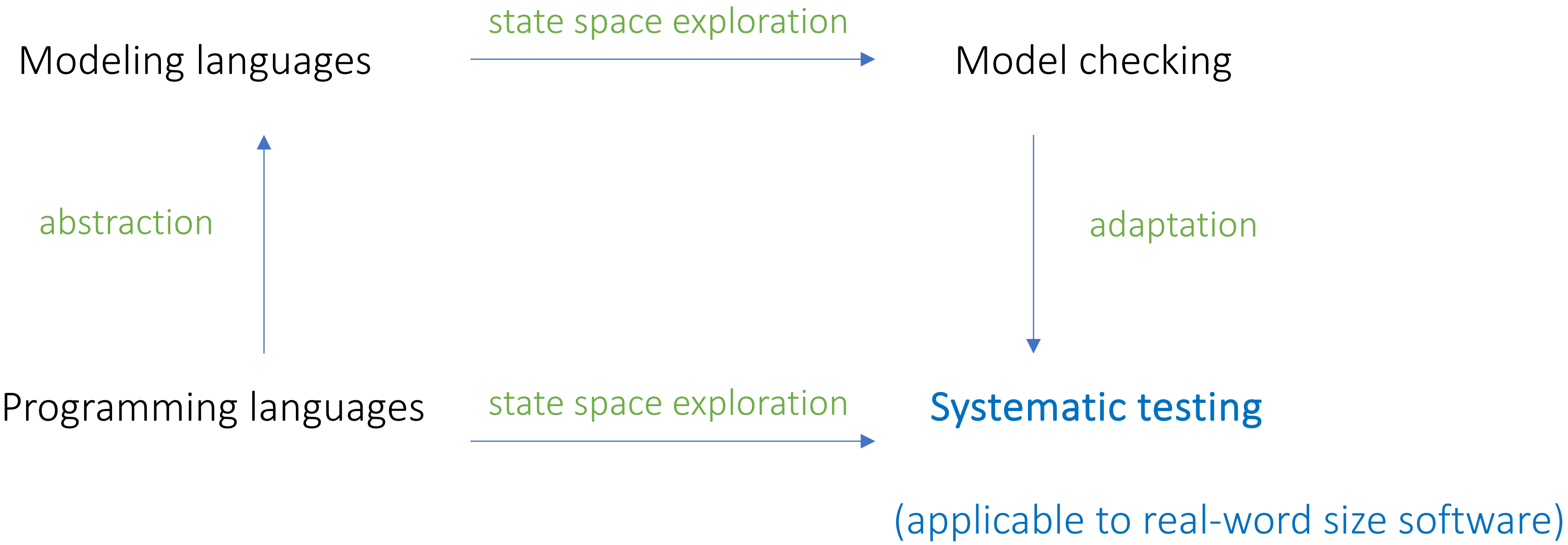
Enumerating executions: What interleavings of events to exercise?

- Systematic testing
  - Explore the state space systematically
  - Run time scheduler to exercise all possible sequences of events
  - Suffers from state space explosion problem



Systematic Testing

Combining Model Checking and Testing



Modeling languages

state space exploration

Model checking

abstraction

adaptation

Programming languages

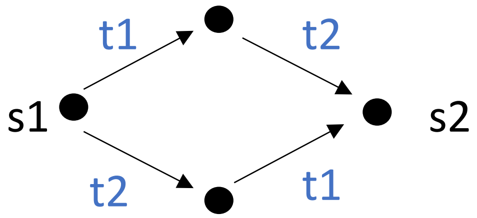
state space exploration

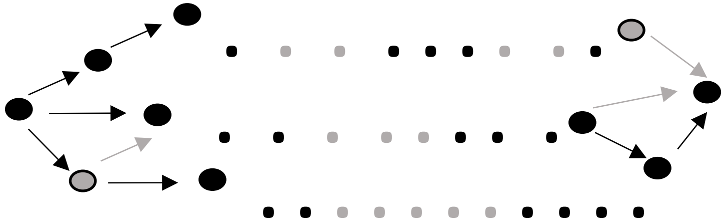
Systematic testing

(applicable to real-word size software)

Systematic Testing

- Partial order reduction (POR) to reduce the execution space
  - Exploits the commutativity of concurrent transitions
  - Based on the dependency relation between system transitions
  - Dependence relation: $(e_1, e_2) \in D$ iff:
    - They're causally dependent
    - $recv(e_1) = recv(e_2)$
- Dynamic POR (DPOR) dynamically tracks interactions between transactions [Flanagan & Godefroid, POPL'05]

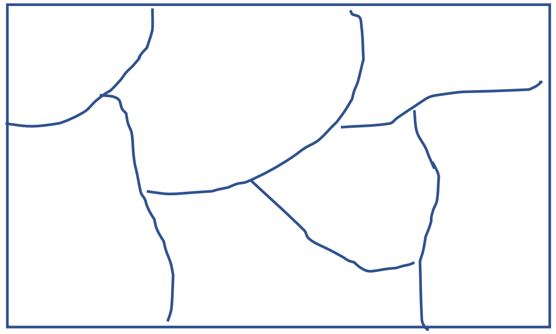




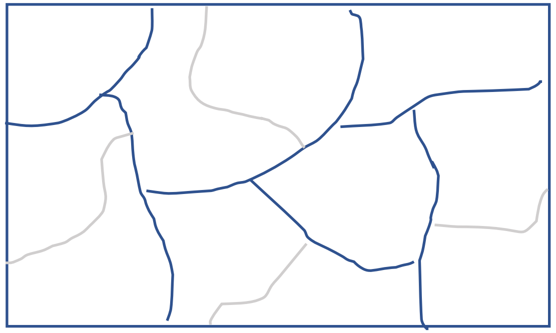
Partial Order Reduction in Distributed Systems

- Classical DPOR (e.g., MODIST [Yang et.al, NSDI'09])
  - Black box, exploits general properties of distributed systems

- Semantic-aware DPOR (e.g., SAMC [Leesatapornwongsa et. al., OSDI'14], FlyMC [Lukman et. al., EuroSys'19]):
  - White-box, exploits system specific semantic information



D partitions the state space into equivalence classes w.r.t $\equiv_D$



Equivalence w.r.t white box $\equiv_{WD}$

Black-box systematic testing is not scalable to large systems

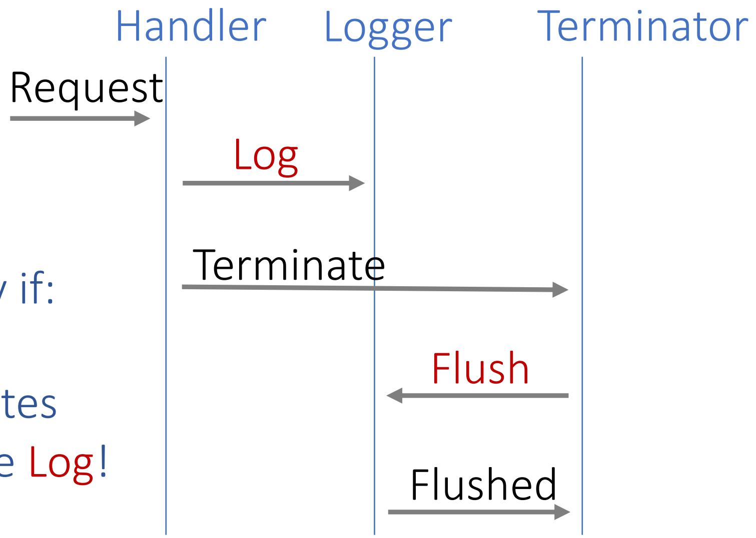
Learning objectives

At the end of this lecture, you will be able to:

- Identify concurrency bugs in distributed systems
- Explain controlled concurrency testing for distributed systems
    - Systematic testing
    - Naïve random testing
    - Probabilistic Concurrency Testing (PCT)

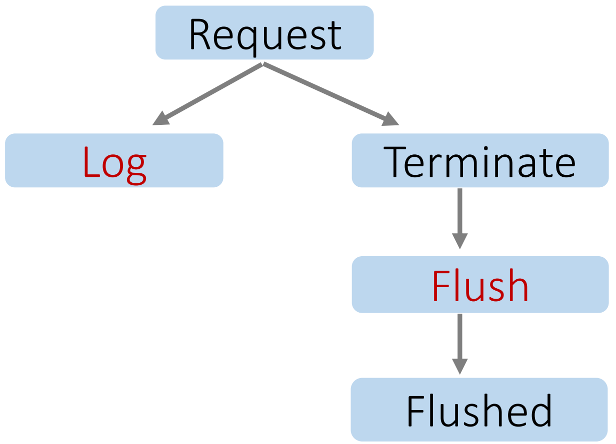
Naïve random testing

- Select the next event uniformly at random (random walk)
- What is the probability of naïve random testing to detect the bug?



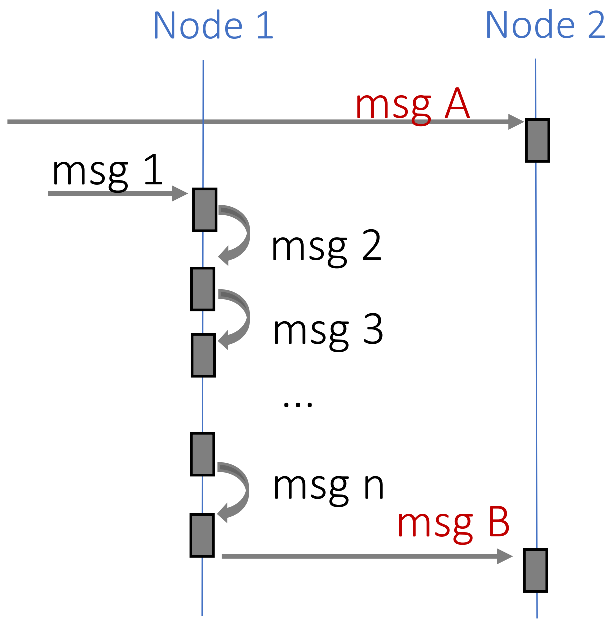
Buggy if: Flush executes before Log!

Upgrowing Poset:



Naïve random testing

- What is the probability of naïve random testing to detect the bug?



Buggy if: … msg B … msg A

Learning objectives

At the end of this lecture, you will be able to:

- Identify concurrency bugs in distributed systems
- Explain controlled concurrency testing for distributed systems
    - Systematic testing
    - Naïve random testing
    - Probabilistic Concurrency Testing (PCT)

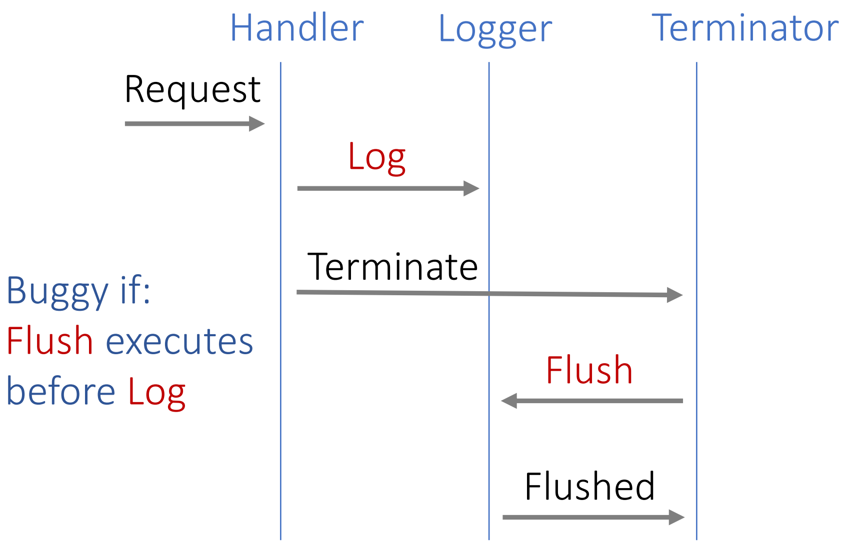
PCT for distributed systems is called "PCT with Chain Partitioning (PCTCP)".
The lecture refers to the algorithm as "PCT", as they are similar in essense.

Probabilistic Concurrency Testing (PCT)

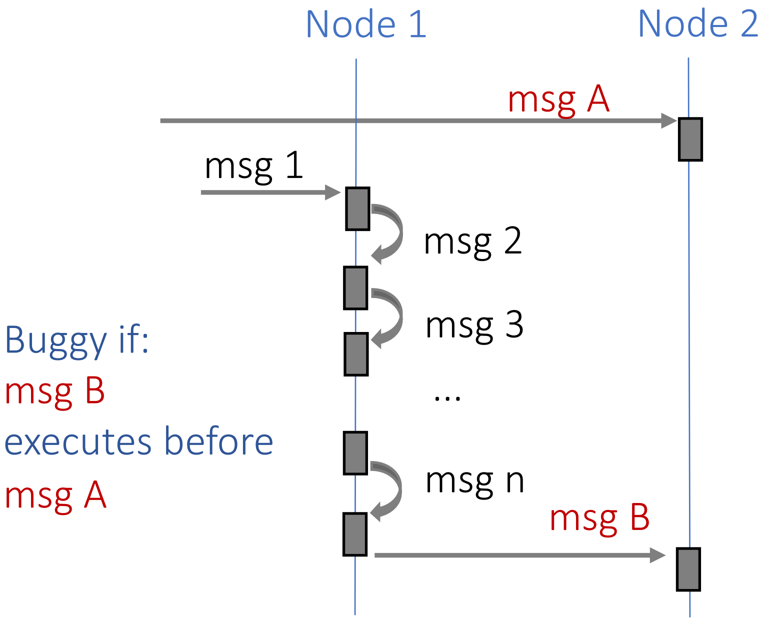
Can we provide a good probabilistic guarantee for detecting a bug?

- Observation: The example bug occurs in a single ordering requirement

Key idea: Characterization of concurrency bugs



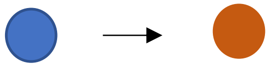
Buggy if:
Flush executes
before Log



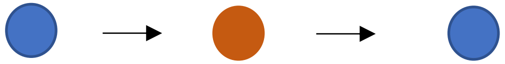
Buggy if:
msg B
executes before
msg A

Bug depth: Number of minimum ordering requirements between events

- $\langle e_1, e_2 \rangle$ e.g. order violation

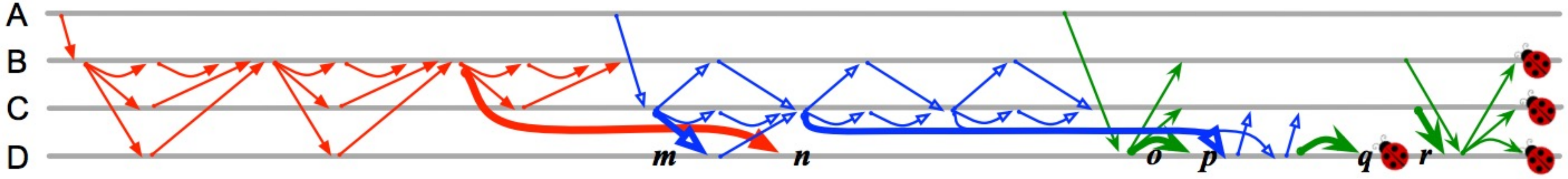


- $\langle e_1, e_2, e_3 \rangle$ e.g. atomicity violation



. . .

- $\langle e_1, \ldots, e_n \rangle$ more complicated bugs



Bug in Cassandra 2.0.0 *(img. from Leesatapornwongsa et. al. ASPLOS'16)*

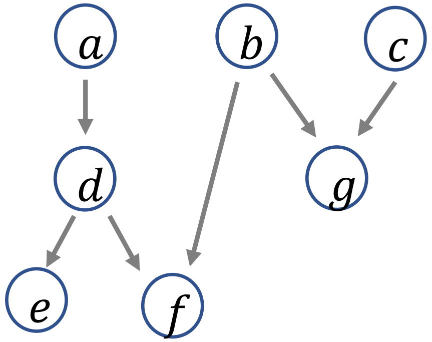
Strong hitting an event tuple

- A schedule α strongly hits $\langle e_0, \ldots, e_{d-1} \rangle$ if for all $e \in P$:

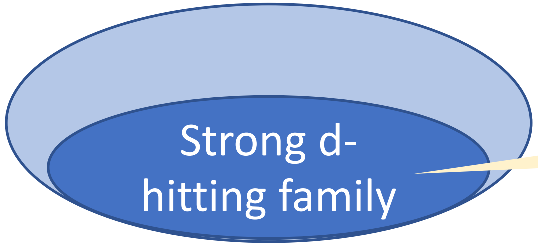
  $e \geq_\alpha e_i$ implies e is causally dependent on e_j for some $j \geq i$

$\alpha 1 = \ a, b, c, d, f, e, g$
strongly hits 1–tuple $\langle g \rangle$, 2–tuple $\langle e, g \rangle$

$\alpha 2 = \ a, b, c, d, f, g, e$
strongly hits 1–tuple $\langle e \rangle$, 2–tuple $\langle g, e \rangle$, 3-tuple $\langle d, g, e \rangle$



For each d-tuple, a **strong d-hitting family** has a schedule which strongly hits it.

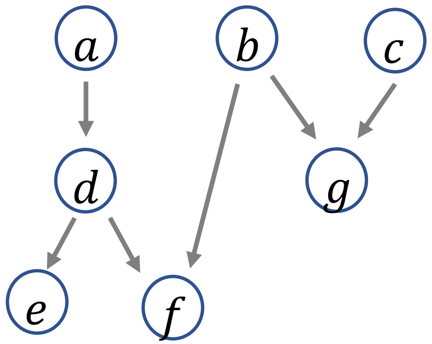


Challenge: How to sample uniformly from this set?

Details ☺

Challenge: How to sample uniformly at random from strong d-hitting family for distributed systems?

- Events form an upgrowing poset, revealed during execution
- Mutual dependency to the schedule



Schedule: $a\ d\ e\ b\ f\ c\ g$

- Build a schedule online
- For an arbitrary ordering

Use combinatorial results for posets!

Details ☺

Realizer and dimension of a poset

Realizer of P is a set of linear orders:

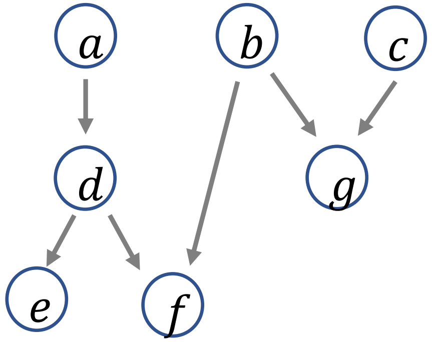
$$F_R = \{L_1, L_2, \dots, L_n\}$$

such that: $L_1 \cap L_2 \ \dots \ \cap L_n = P$

Dimension of P is the minimum size of a realizer

Realizer of size $\dim(P)$

- Covers all pairwise orderings!



$L_1 = a\ d\ e\ b\ f\ c\ g$

$L_2 = c\ a\ d\ e\ b\ g\ f$

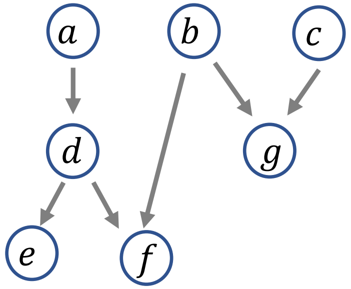
$L_3 = c\ b\ g\ f\ a\ d\ e$

$\dim(P) = 3$

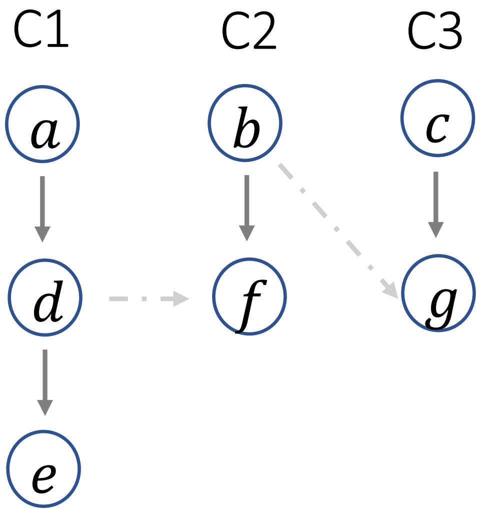
Details ☺

Adaptive chain covering ~ Online dimension algorithm

Decompose P into chains



Compute linear extensions of P



$L1 = c\ b\ g\ a\ d\ f\ e$

$L2 = c\ a\ d\ e\ b\ g\ f$

$L3 = a\ d\ e\ b\ f\ c\ g$

This is a strong 1-hitting family!

Adaptive chain covering ~ Strong 1-hitting family ~ Online dimension algorithm

[Felsner’97, Kloch’07]

Details ☺

Strong d-hitting family ~ Adaptive chain covering

[Felsner, Kloch] Strong 1-hitting family ~ Adaptive chain covering

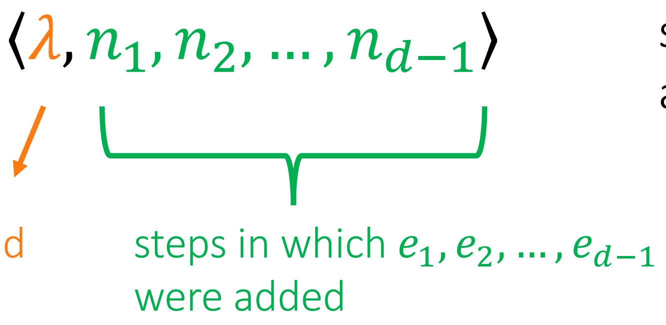
$$hit(w) = adapt(w)$$

[Our main result] Strong $\boldsymbol{d}$-hitting family ~ Adaptive chain covering

$$hit_d(w,n) \leq adapt(w)\binom{n}{d-1}(d-1)!$$

n: number of events
d: bug depth

Index the schedules in the strong d-hitting family by:



$$\langle \lambda, n_1, n_2, \dots, n_{d-1} \rangle$$

chain id

steps in which $e_1, e_2, \dots, e_{d-1}$ were added

strongly hits $e_0 \in Chain(\lambda)$ and $e_1, e_2, \dots, e_{d-1}$

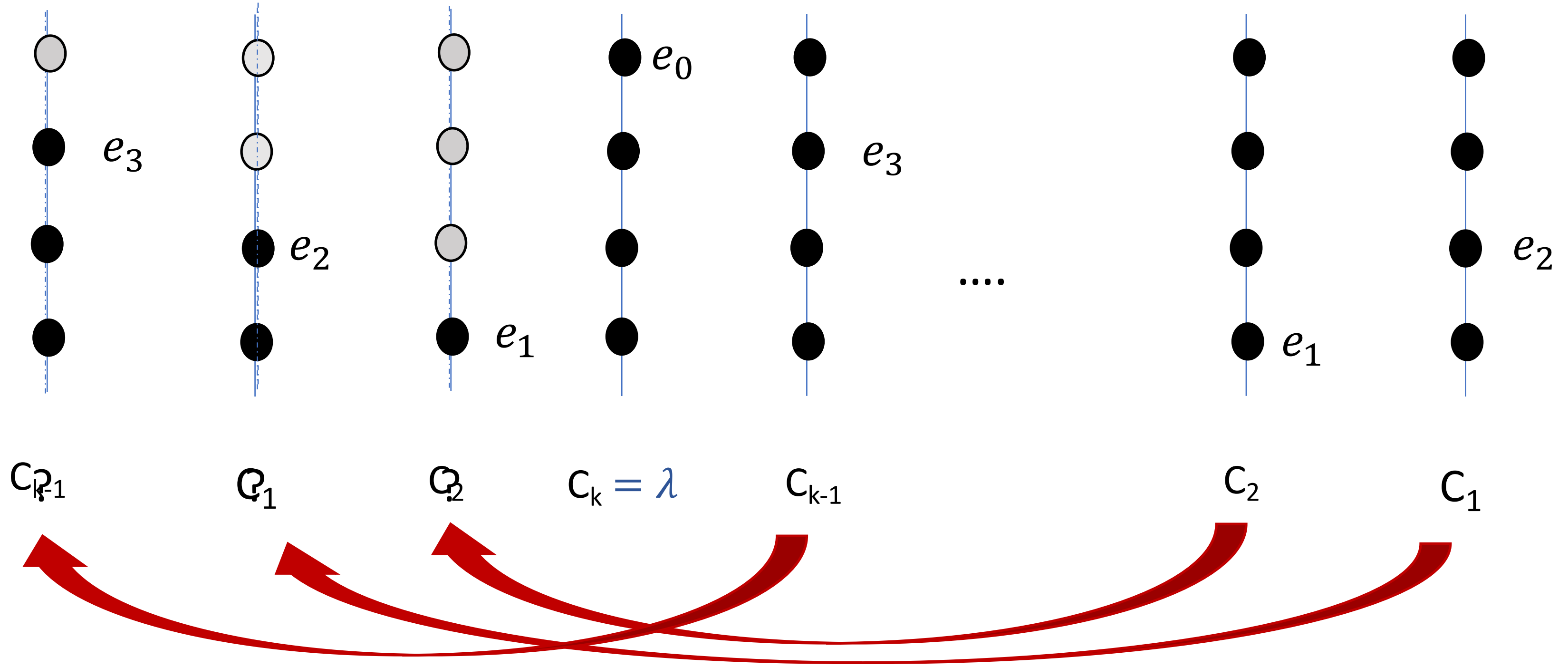
Sample from this set of schedules!

PCT(CP) - The Algorithm

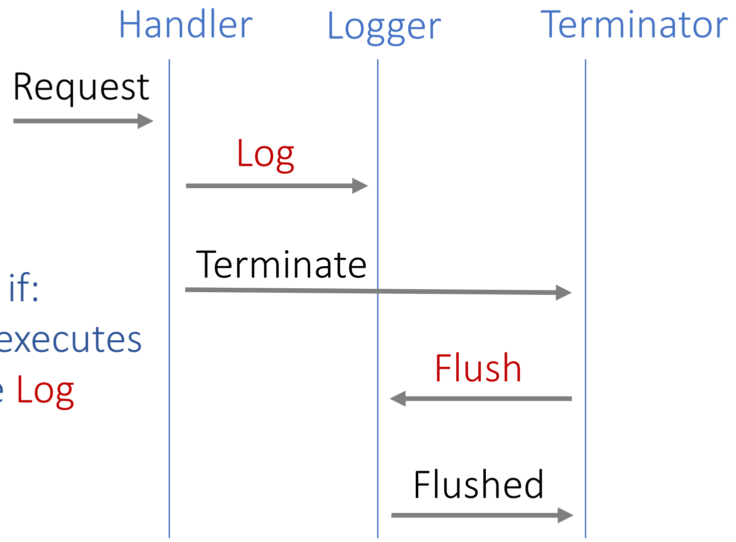
Generates randomly a schedule index $\langle \lambda, n_1, n_2, \ldots, n_{d-1} \rangle$:

- Randomly generate a $(d-1)$-tuple: $\langle n_1, n_2, \ldots, n_{d-1} \rangle$
- Partition P into chains online
- Assign random distinct initial priorities $> \ d$
- Reduce priority at: $\langle e_1, e_2, \ldots, e_{d-1} \rangle$ to $(d-i-1)$ for e_i

strongly hits $e_0 \in Chain(\lambda)$
and $e_1, e_2, \ldots, e_{d-1}$

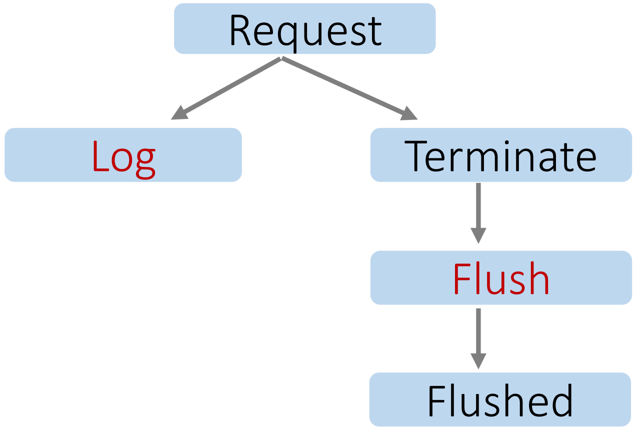


Probabilistic Concurrency Testing (PCT) – Example 1



Buggy if:
Flush executes before Log

Upgrowing Poset:



Online chain partitioning:

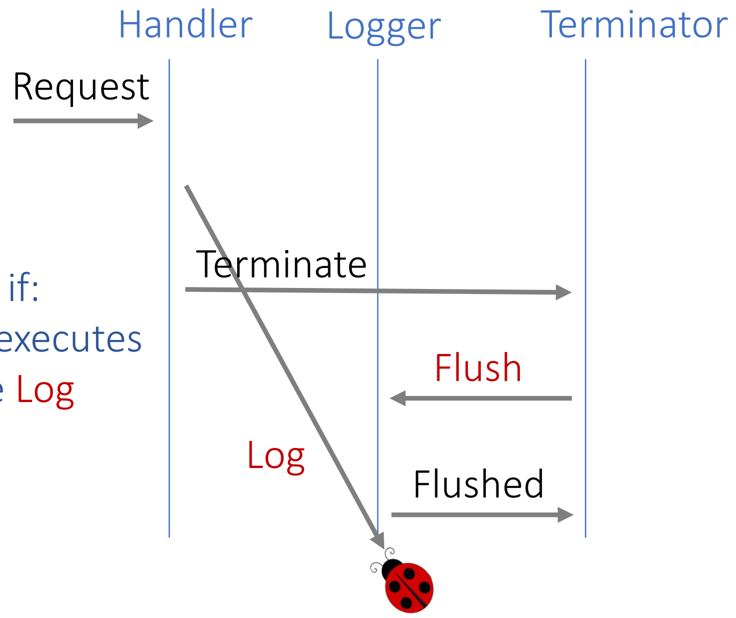
The program is decomposed into causally dependent chains of events:

$$C1 = [Request, Log]$$
$$C2 = [Terminate, Flush, Flushed]$$

$$priority(C1) > priority\ (C2)$$

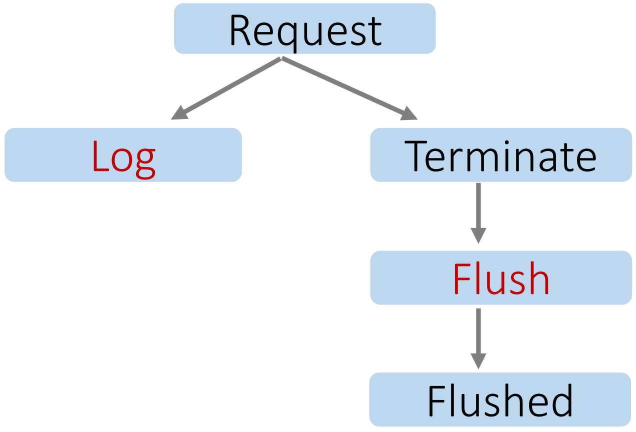
$$Schedule = [Request, Log, Terminate, Flush, Flushed]$$

Probabilistic Concurrency Testing (PCT) – Example 1


Handler
Logger
Terminator
Request
Terminate
Flush
Log
Flushed

Buggy if:
Flush executes
before Log

Upgrowing Poset:


Request
Log
Terminate
Flush
Flushed

The program is decomposed into causally dependent chains of events:

Online chain partitioning:

$$C1 = [Request, Log]$$
$$C2 = [Terminate, Flush, Flushed]$$

$$priority(C2) > priority\ (C1)$$

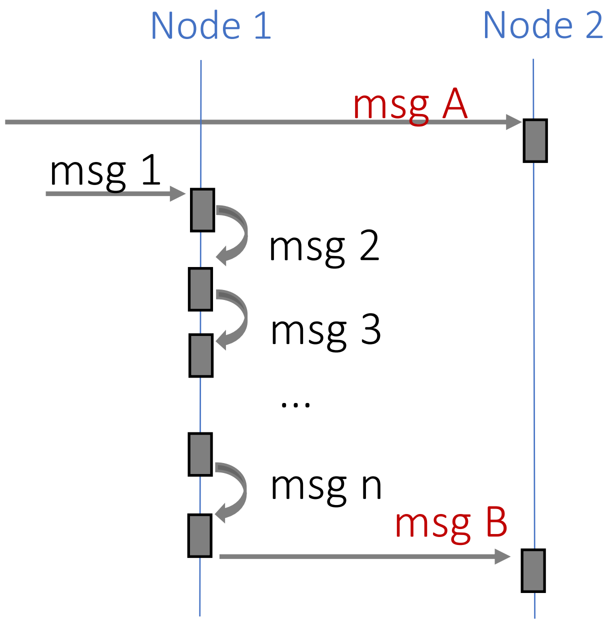
$$Schedule = [Request, Terminate, Flush, Flushed, Log]$$

Naive random: $1/4$

PCT: $1/2$

Probabilistic Concurrency Testing (PCT) – Example 2

- What is the probability of PCT to detect the bug?



Buggy if: … msg B … msg A

Online chain partitioning

Chain1 = msg A

Chain2 = msg 1 → msg 2 → … → msg n → msg B

PCT assigns random priorities to chains:

priority(Chain1) > priority(Chain2)

msg A msg 1 msg 2 … msg n msg B ✓

priority(Chain2) > priority(Chain1)

msg 1 msg 2 … msg n msg B msg A ✗

Naive random: $1/2^{n+1}$ PCT: ? $1/2$

PCT: Random testing with nontrivial probabilistic guarantees

- PCT result for multithreaded programs (linear orders) [Burckhardt et. al., ASPLOS'2010]

- PCT(CP): Generalizes the guarantees to distributed systems (posets) [K.O. et. al, OOPSLA'18]

  "Randomized testing of distributed systems with probabilistic guarantees"

  Covered in this lecture

  PCTCP hits a bug with a prob. $\frac{1}{adapt(w)n^{d-1}}$

  Generalizes the PCT result $\frac{1}{k\ n^{d-1}}$

  $adapt(w)$: online width

  k: number of threads

- Trace-aware PCT (taPCT): Partial order reduction + PCT [K.O. et. al, OOPSLA'19]

- PCT for Weak Memory (PCTWM): Extends the results for SC to weak memory [Gao et. al, ASPLOS'23]

Challenges for testing distributed systems

(C0) Test oracle

- What is the correctness specification?

→ We assume it is provided

(e.g. exceptions, assertion violations, serializability of transactions, agreement of replicas)

(C1) Test harness discovery

- What are the requests/transactions to submit?

→ We randomly generate a few transactions

(small-scope hypothesis)

(C2) Enumerating executions

- What interleavings of events to exercise?

→ How to explore possible executions efficiently?

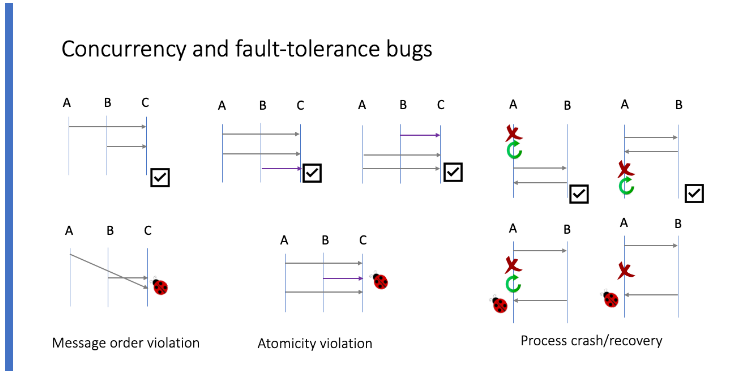
Combinatorial complexity!

(C3) Improving interpretability

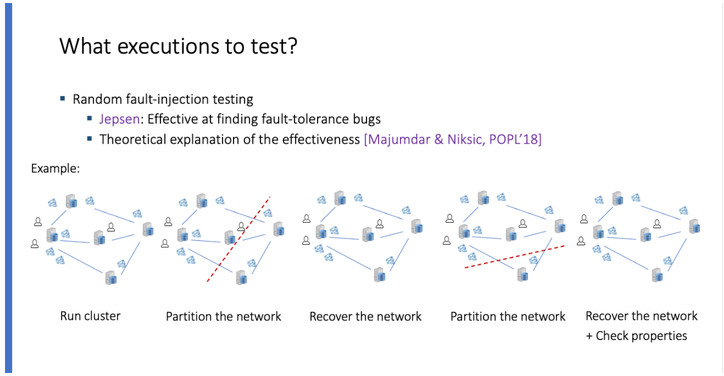
- Is the buggy trace easy to understand?

→ How to produce understandable traces?

Summary:



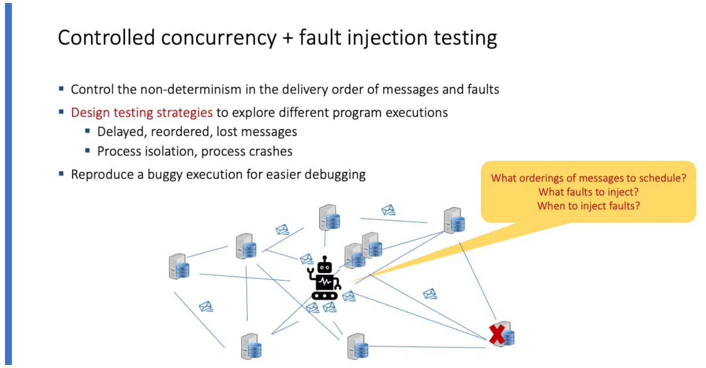
Concurrency and fault-tolerance bugs



What executions to test?

- Random fault-injection testing
  - Jepsen: Effective at finding fault-tolerance bugs
  - Theoretical explanation of the effectiveness [Majumdar & Niksic, POPL'18]

Example:



Controlled concurrency + fault injection testing

- Control the non-determinism in the delivery order of messages and faults
- Design testing strategies to explore different program executions
  - Delayed, reordered, lost messages
  - Process isolation, process crashes
- Reproduce a buggy execution for easier debugging

What orderings of messages to schedule?
What faults to inject?
When to inject faults?

In this lecture, we covered:

- Concurrency and fault-tolerance bugs in distributed systems
- Controlled concurrency testing for detecting such bugs:
  - Systematic testing
  - Naïve random testing
  - Probabilistic Concurrency Testing (PCT)

Questions?