Concurrency Analysis for Distributed Systems

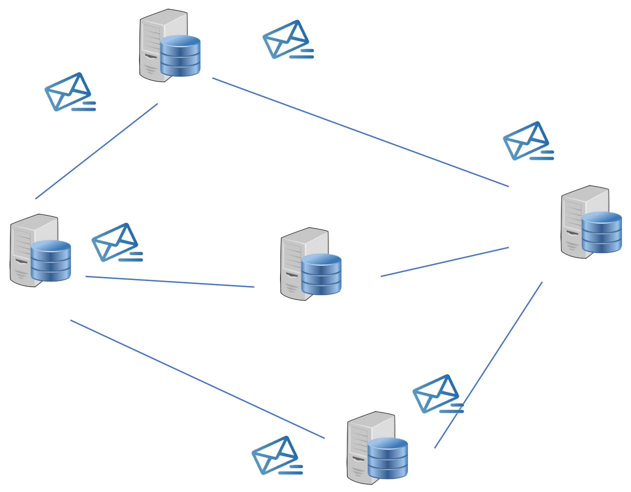
CS4405 – Analysis of Concurrent and Distributed Programs

Burcu Kulahcioglu Ozkan

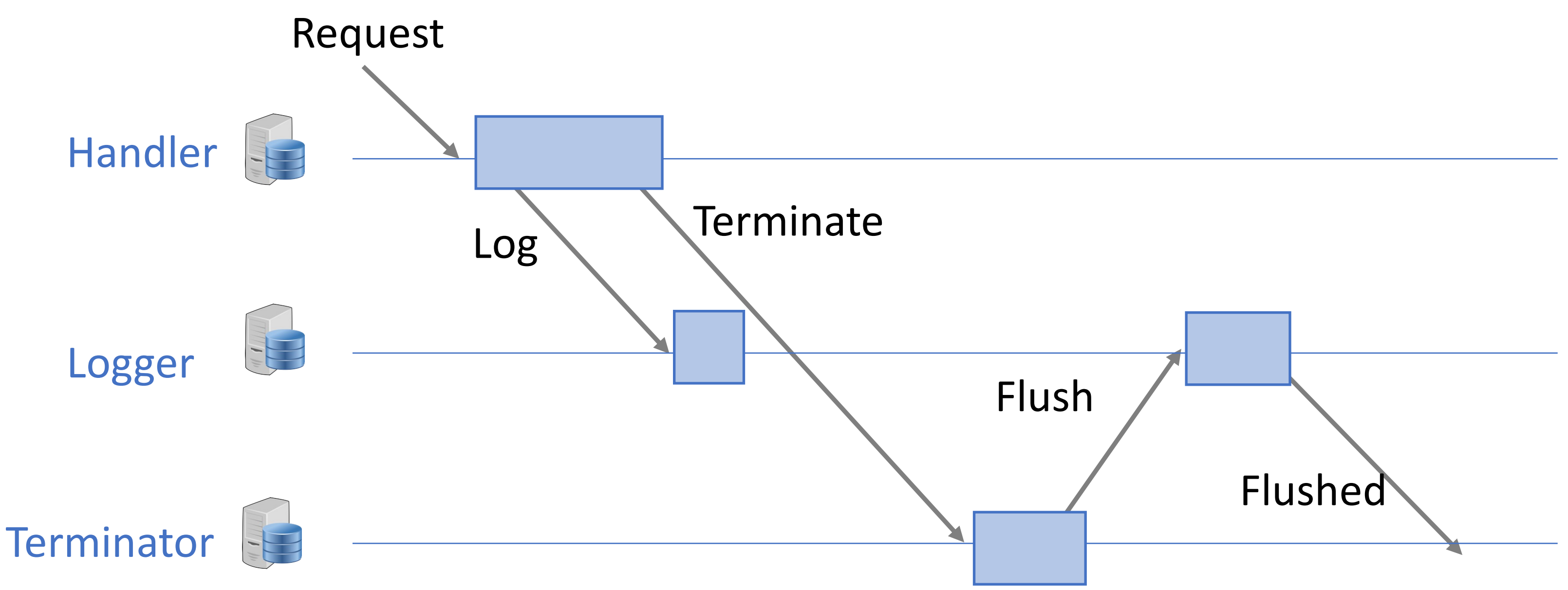
TU Delft

Revisit: Events in distributed systems

- 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

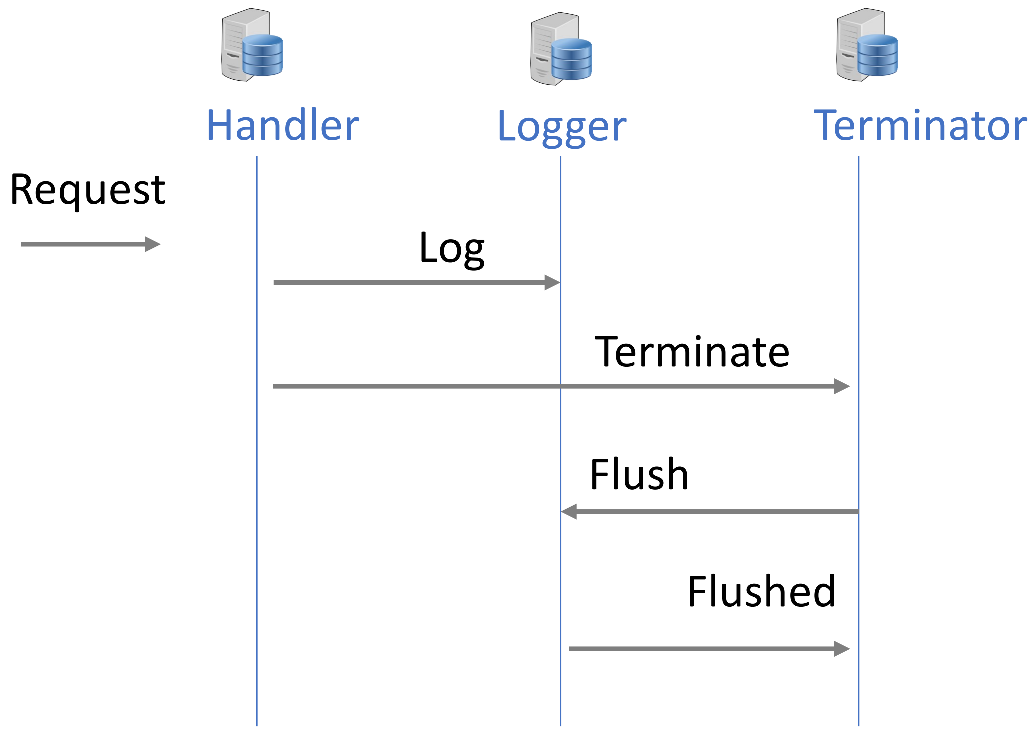


An example execution



A simplified version of a bug found in a performance testing tool Gatling [2018]

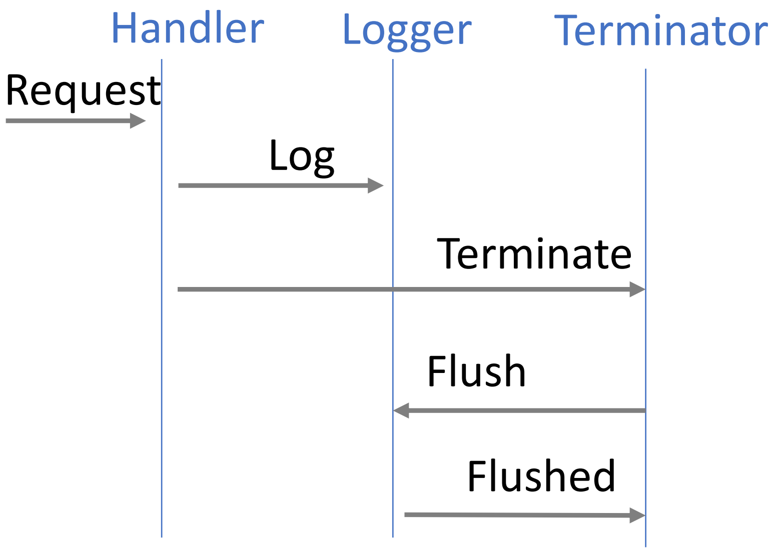
An example execution



A simplified version of a bug found in a performance testing tool Gatling [2018]

Model of distributed systems

- $Nodes$: the set of nodes/processes
- $Msgs$: the set of all messages
- $Events$: $\langle recv, send, msg \rangle$ s.t.
  - $recv \in Nodes$
  - $send \in Nodes$
  - $msg \in Msgs$
- $recv\ (e), send(e), msg(e)$



Model of distributed systems

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

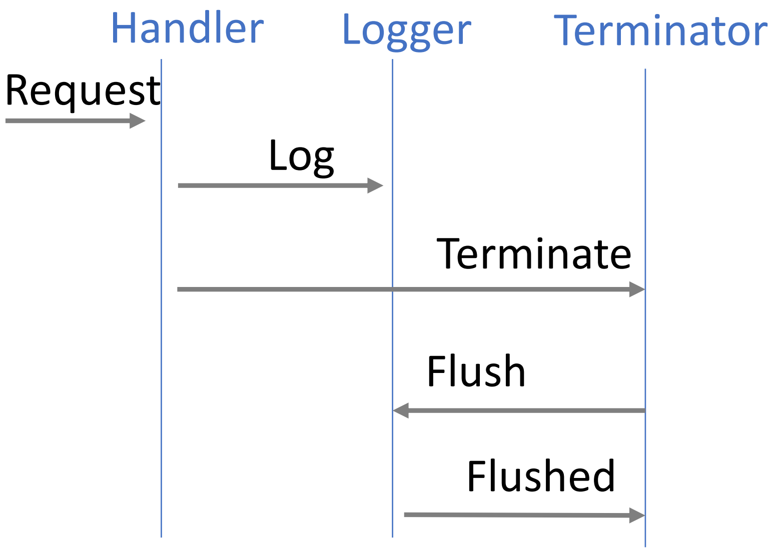
Model of distributed systems

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

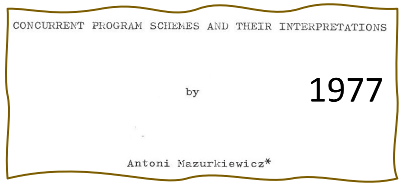
is called a schedule



An example schedule:

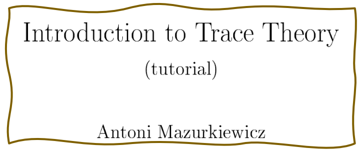
$$\langle Handler: e_0 = \langle Handler, Client, Request\rangle\rangle,$$
$$\langle Logger: e_1 = \langle Logger, Handler, Log\rangle\rangle,$$
$$\langle Terminator: e_2 = \langle Terminator, Logger, Terminate\rangle\rangle,$$
$$\langle Logger: e_3 = \langle Logger, Terminator, Flush\rangle\rangle,$$
$$\langle Terminator: e_4 = \langle Terminator, Logger, Flushed\rangle\rangle$$

Mazurkiewicz trace theory for concurrent systems


CONCURRENT PROGRAM SCHEMES AND THEIR INTERPRETATIONS

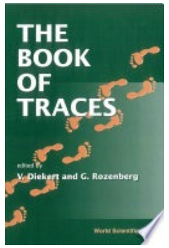
by

Antoni Mazurkiewicz*


Introduction to Trace Theory

(tutorial)

Antoni Mazurkiewicz



- A mathematical description of the behavior of concurrent systems
  - Formulated by A. Mazurkiewicz in 1977
- The linear event schedule is not a faithful representation of a concurrent system's behavior
  - Two events a and b may appear adjacent in a schedule, while they are really performed concurrently within the system
  - Sequential observations, nonsequential causality

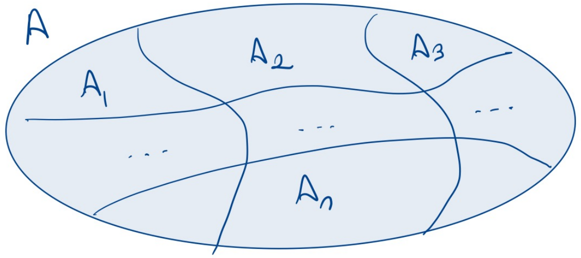
Concurrent program schemes and their interpretations, A. Mazurkiewicz, 1977
Trace theory, A. Mazurkiewicz, Advanced Course on Petri Nets, 1986
Theory of Traces, I.J. Aalsberg, G. Rozenberg, Theoretical Computer Science, 1988
The book of traces, V. Diekert, G. Rozenberg, 1995

Mazurkiewicz trace theory for concurrent systems

- Trace theory introduces independence relation I between the events:
  - Given the set of events (alphabet) Σ, and $a, b \in \Sigma$
  - $D \in (\Sigma \times \Sigma)$ is a symmetric and reflexive dependence relation
  - $I \in (\Sigma \times \Sigma)$ is a symmetric and irreflexive dependence relation
  - $D \cup I = \Sigma \times \Sigma$ and $D \cap I = \emptyset$

- Independent events can commute:
  - If $(a, b) \in I$, then the schedules $\mathrm{x}_1 ab x_2$ and $\mathrm{x}_1 ba x_2$ are equivalent

Dependence relation partitions the set of schedules into equivalence classes called traces



Traces in distributed systems

- A schedule induces a partial ordering among events Σ, captured by a binary dependence relation $D \subseteq \Sigma \times \Sigma$

- Dependence Relation: Let e_i and e_j be respectively the ith and jth events in a schedule. $(e_i, e_j) \in D$ iff:
  - either (i) ∃k : i ≤ k < j such that recv(e_i) = recv(e_k) and e_j is transitively causally dependent on e_k ;
  - or (ii) recv(e_i) = recv(e_j).

- Given a schedule, two adjacent events that are independent can be permuted without changing the behavior of the execution

Traces in distributed systems

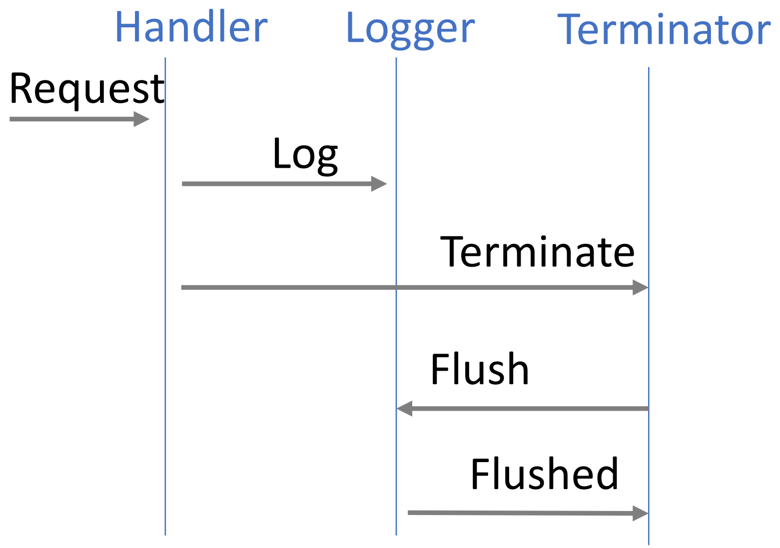
- Happens-before relation $\rightarrow$ for an execution $S = e_1 e_2 \dots e_n$ is the smallest relation on $\Sigma \times \Sigma$ such that:
  - if $i \leq j$ and e_i is dependent with e_j , then $e_i \rightarrow e_j$
  - $\rightarrow$ is transitively closed.

- Race Relation: Two events e_i and e_j are racy iff:
  - $(e_i, e_j) \in D$
  - e_i and e_j may be co-enabled.

- Reordering the execution of racy events may result in different program behaviors

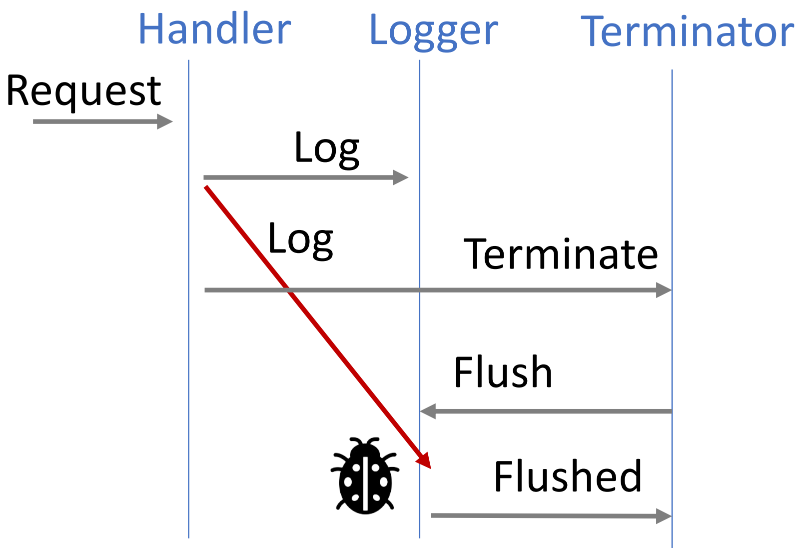
Exercise: Revisit the example

- Dependent/independent events
- Example schedules
- Traces
- Racy events



Order violations

- Racy events may cause order violations or atomicity violations

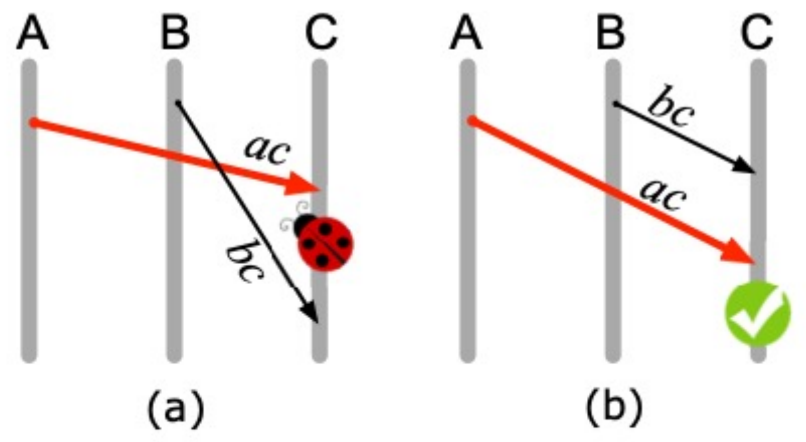


Order violations

Hadoop Map/Reduce / MAPREDUCE-3274

Race condition in MR App Master Preemtion can cause a dead lock

https://issues.apache.org/jira/browse/MAPREDUCE-3274



B sends to C a task-init message (bc_{init})
Soon afterwards, A sends to C a task-kill preemption message (ac_{kill})
However, ac_{kill} arrives before bc_{init} and thus is incorrectly ignored by C
The bug would not manifest if ac_{kill} arrives after bc_{init}

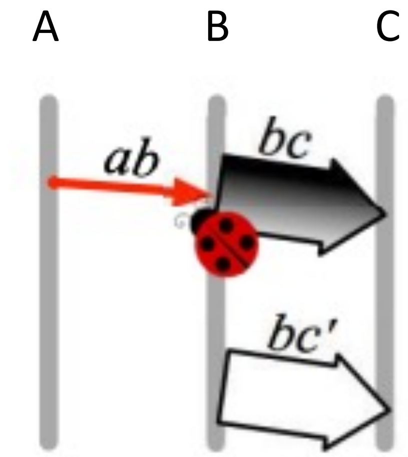
TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems.
T. Leesatapornwongsa, J. F. Lukman, S. Lu, H. S. Gunawi. ASPLOS 2016.

Atomicity violations

Hadoop Map/Reduce / MAPREDUCE-5009

Killing the Task Attempt slated for commit does not clear the value from the Task commitAttempt member

https://issues.apache.org/jira/browse/MAPREDUCE-5009

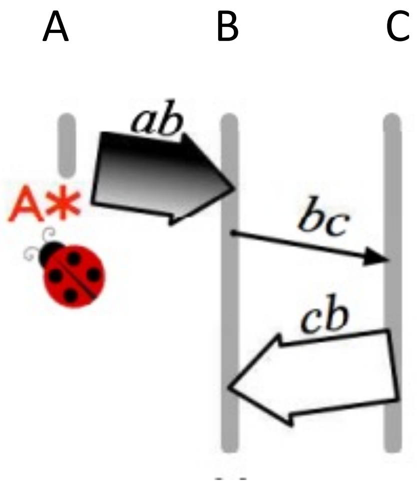


When B is in the middle of a commit transaction, transferring task output data bc to C,
A sends a kill preemption message ab to B, preempting the task without resetting commit states on C.
The system is never able to finish the commit.
Then B later reruns the task and tries to commit to C with bc', C throws a double-commit exception.
This failure would not happen if the kill message ab comes before or after the commit transaction bc.

TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems.
T. Leesatapornwongsa, J. F. Lukman, S. Lu, H. S. Gunawi. ASPLOS 2016.

https://issues.apache.org/jira/browse/MAPREDUCE-3858

Fault tolerance bugs: Process crash



A is sending a task's output ab to B but A crashes in the middle, leaving the output half-sent.
The system is unable to recover from this untimely crash
B detects the fault and reruns the task at C (via bc) and later when C re-sends the output cb, B throws an exception.
This bug would not manifest, if the crash happens before/after the output transfer ab.

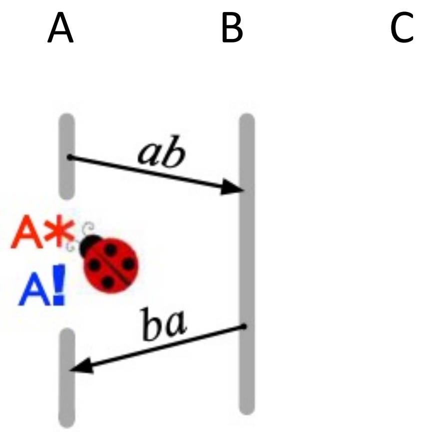
TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems.
T. Leesatapornwongsa, J. F. Lukman, S. Lu, H. S. Gunawi. ASPLOS 2016.

Hadoop Map/Reduce / MAPREDUCE-3186

User jobs are getting hanged if the Resource manager process goes down and comes up while job is getting executed.

https://issues.apache.org/jira/browse/MAPREDUCE-3186

Fault tolerance bugs: Process crash & recovery



A sends a job ab to B. While B is processing, A crashes and reboots losing its in-memory job info. B sends a job-commit message ba but A throws an exception because it does not have the job info. The bug would not manifest if A reboots later: if A is still down when B sends ba commit message, B will realize the crash and cancel the job before A reboots. A would repeat the job correctly.

TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems.
T. Leesatapornwongsa, J. F. Lukman, S. Lu, H. S. Gunawi. ASPLOS 2016.

Concurrency bugs in distributed systems

- Reported summary of 104 distributed concurrency bugs from four cloud-scale datacenter distributed systems, Cassandra, Hadoop MapReduce, Hbase and ZooKeeper.

	Ordering	Atomicity	Fault	Reboot
CA	4	4	6	5
HB	13	9	8	1
MR	25	4	5	3
ZK	4	8	7	5
All	46	25	26	14

Table 2. #DC bugs triggered by timing conditions (§3.1). *The total is more than 104 because some bugs require more than one triggering condition. More specifically, 46 bugs (44%) are caused only by ordering violations, 21 bugs (20%) only by atomicity violations, and 4 bugs (4%) by multiple timing conditions (as also shown in Figure 3a).*

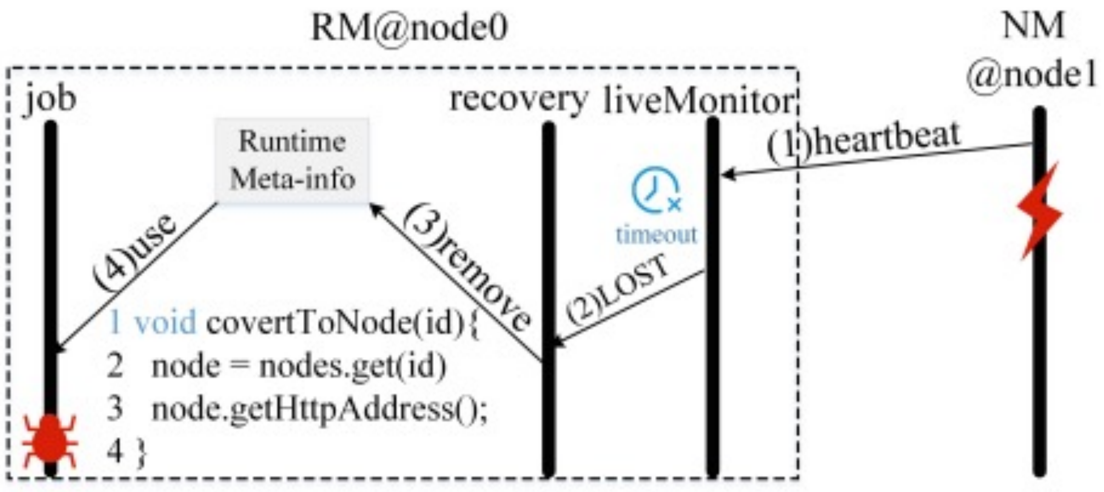
TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems.
T. Leesatapornwongsa, J. F. Lukman, S. Lu, H. S. Gunawi. ASPLOS 2016.

Hadoop YARN / YARN-5918

Handle Opportunistic scheduling allocate request failure when NM is lost

https://issues.apache.org/jira/browse/YARN-5918

Distributed fault tolerance + shared memory bug



Two nodes are involved: *ResourceManager* (RM) at node0 and the *NodeManager* (NM) at node1.

1. NM @node1 sends heartbeat message to RM @node0 when it is alive. After node1 crashes, no heartbeat message will be sent.
2. The *liveMonitor* thread in node0 detects the crash of node1 after a timeout period. A LOST event is dispatched to the recovery thread.
3. The recovery thread removes node1 from nodes, a shared data structure to record all available nodes.
4. Another running thread job tries to get resources of NM @node1.

CrashTuner: detecting crash-recovery bugs in cloud systems via meta-info analysis.
J. Lu, C. Liu, L. Li, X. Feng, F. Tan, J. Yang, L. You. SOSP 2019.

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	Status:	
Priority:	Critical	Resolution:	
Affects Version/s:	2.0.2-alpha	Fix Version/s:	
Component/s:	ha, hdfs-client		
Labels:	None		
Target Version/s:	2.0.3-alpha		
Hadoop Flags:			

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

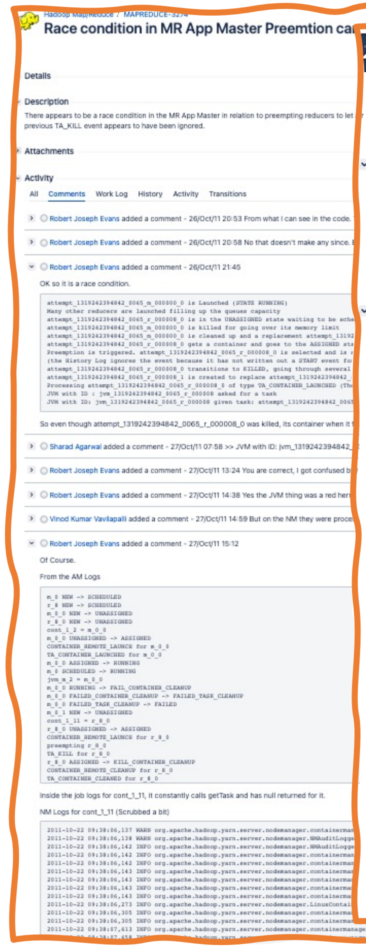
HBase / HBASE-2849
HBase clients cannot recover

Core Server / SERVER-38084
MongoDB hangs when a part of a replica set

ZooKeeper / ZOOKEEPER-4003
Zookeeper server breakdown Frequently

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

All Comments Work Log History Activity Transitions

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.

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

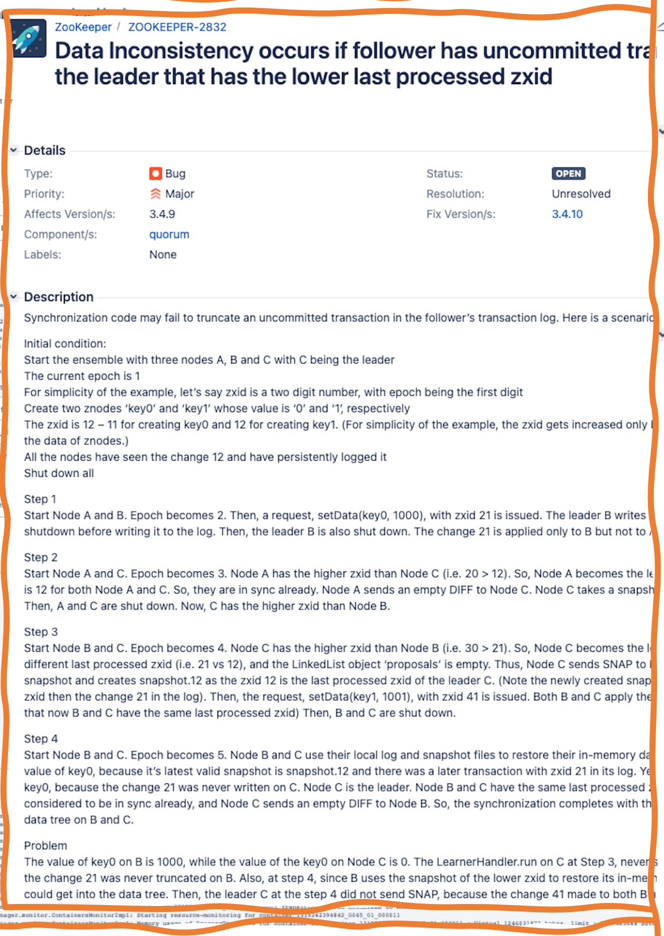
Robert Joseph Evans added a comment - 27/Oct/11 15:12

Of Course.

From the AM Logs

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)


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

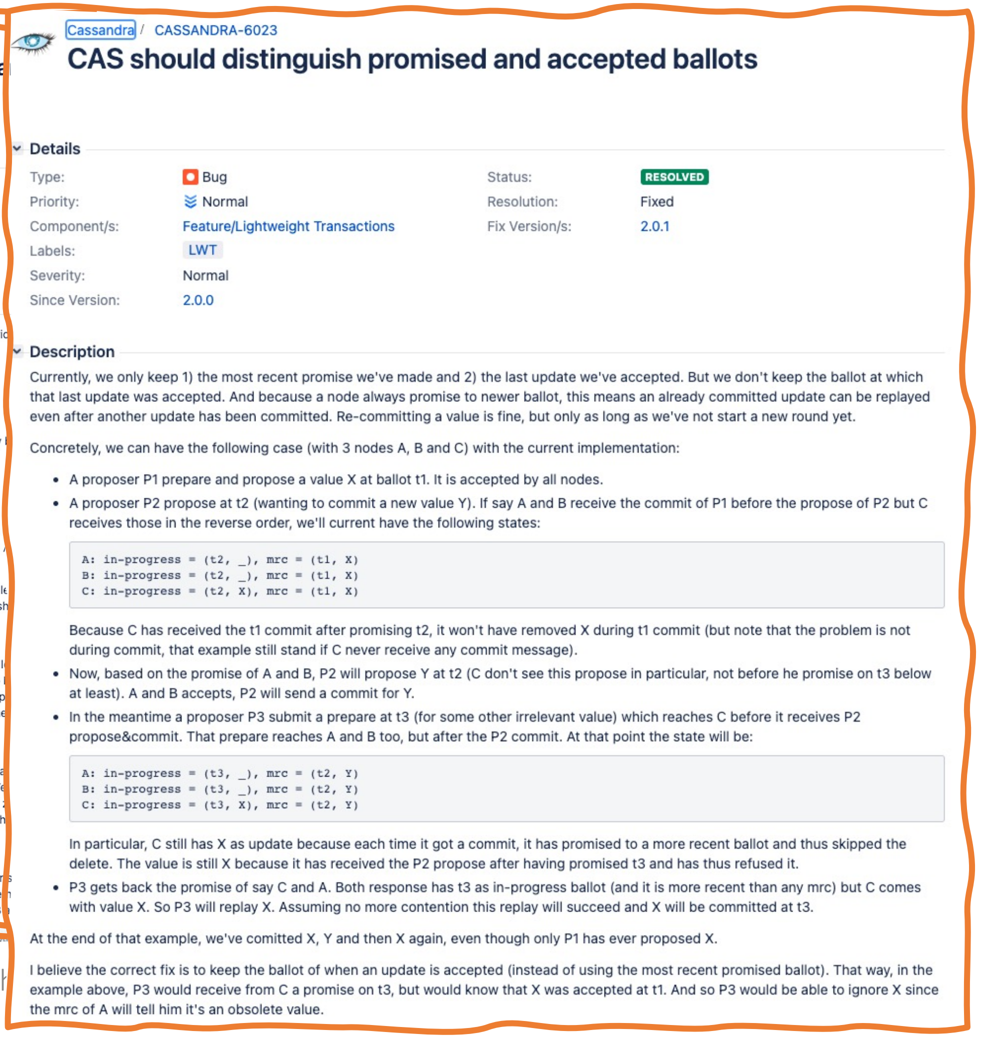
Step 1
Start Node A and B. Epoch becomes 2. Then, a request, setData(key0, 1000), with zxid 21 is issued. The leader B writes
shutdown before writing it to the log. Then, the leader B is also shut down. The change 21 is applied only to B but not to

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.

Take aways

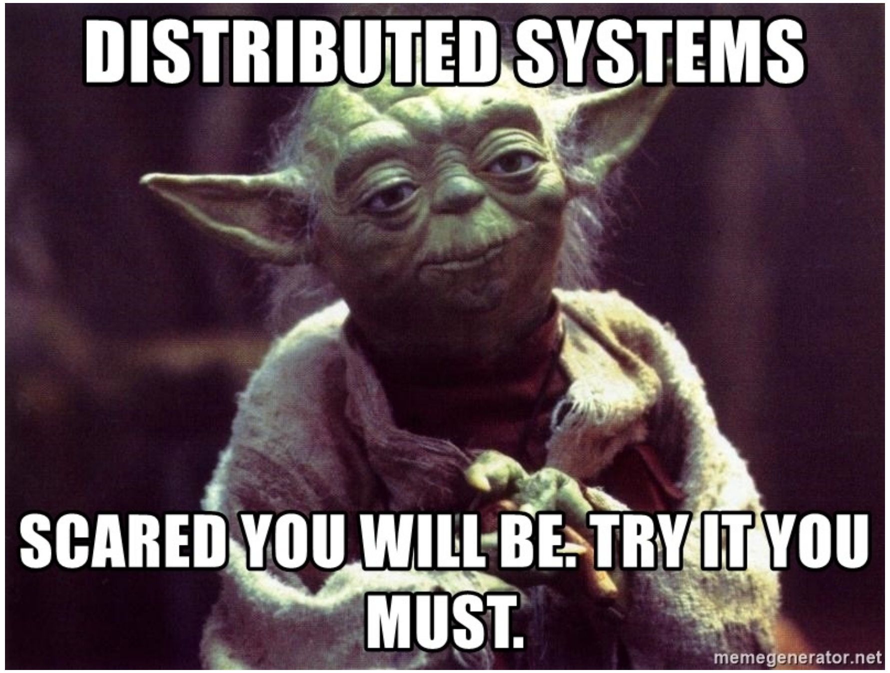


Image source: https://memegenerator.net/

- Distributed systems are notoriously hard to design and implement correctly
  - Complex interaction between concurrent components
  - Requires reasoning about concurrency and fault tolerance
- We need software analysis methods to verify correctness or detect errors
  - Concurrency analysis of distributed system events and failures

More at the last lecture "active research directions" ☺