OCC, MVCC & Multi-site atomicity

IPADS, Shanghai Jiao Tong University

https://www.sjtu.edu.cn

Optimistic concurrency control (OCC)

Problem of 2PL for before-or-after: locking overhead + deadlock

- Pessimistically execute the TX to avoid race conditions

Executing TXs optimistically w/o acquiring the lock

- Checks the results of TX before it commits
- If violate serializability, then **aborts & retries**

First proposed in 1981, widely used today (even in hardware impl. !)

HEKATON Microsoft SQL Server

App Engine

CouchDB

Review: Optimistic Concurrency Control

Phase 1: Concurrent local processing

- Reads data into a read set
- Buffers writes into a write set

Phase 2: Validation serializability in critical section

- Validates whether serializability is guaranteed:
- Has any data in the read set been modified?

Phase 3: Commit the results in critical section or abort

- Aborts: aborts the transaction if validation fails
- Commits: installs the write set and commits the transaction

Review: OCC Executes a Transaction in 3 Phases

Phase 1:

- Reads data into a read set
- Buffers writes into a write set

```
...
tx.begin();
...
tx.read(A)
...
tx.commit();
...
```

```
val_a = read(A)
read_set.add(val_a)
```

Review: OCC Executes a Transaction in 3 Phases

Phase 1:

- Reads data into a read set
- Buffers writes into a write set

What about a second read?

- Read from the-read-set!
- Why? Need to provided repeated read!

```
...
tx.begin();
...
tx.read(A)
tx.read(A)
tx.commit();
...
```

```
if A in read_set:
    return read_set[A
```

Review: OCC Executes a Transaction in 3 Phases

Phase 1:

- Reads data into a read set
- Buffers writes into a write set

Question

- Why do we need to update the readset? Is It necessary?

```
...
tx.begin();
...
tx.read(A)
tx.write(A)
tx.commit();
...
```

```
write_set[A] = ..
if A in read_set:
   read_set[A] = ..
```

Review: OCC Executes a Transaction in 3 Phases

Phase 1:

- Reads data into a read set
- Buffers writes into a write set

Question

- Why do we need to update the readset? Is It necessary?
- Goal: we need to ensure later read will see my write
- We can avoid updating the readset by checking the writeset during reads

```
...
tx.begin();
...
tx.read(A)
tx.write(A)      Write_set[A] = ..

tx.read(A)       if A in write_set:
                     return write_set[A
                 if A in read_set:
                     return read_set[A]
tx.commit(       ...
...
```

Review: OCC Executes a Transaction in 3 Phases

Phase 2:

- Validates whether serializability is guaranteed:
- Has any data in the read set been modified?

```
...
tx.begin();
...
tx.read(A)
...
tx.commit();
...
```

```
for d in read_set:
    if d has changed:
        abort()
```

Review: OCC Executes a Transaction in 3 Phases

Phase 3:

- Aborts: aborts the transaction if validation fails
- Commits: installs the write set and commits the transaction

```
...
tx.begin();
...
tx.read(A)
...
tx.commit();
...
```

```
for d in read-set:
    if d has changed:
        abort()
for d in write_set:
    write(d)
```

OCC Executes a Transaction in 3 Phases

Phase 3:

- Aborts: aborts the transaction if validation fails
- Commits: installs the write set and commits the transaction

Phase 2 & 3 should execute in a critical section

- Otherwise, what if a value has changed during validation?

```
...
tx.begin();
...
tx.read(A)
...
tx.commit();
...
```

Critical section

```
for d in read_set:
   if d has changed:
      abort()
for d in write_set:
   write(d)
```

How to implement the critical section for phase 2 & 3?

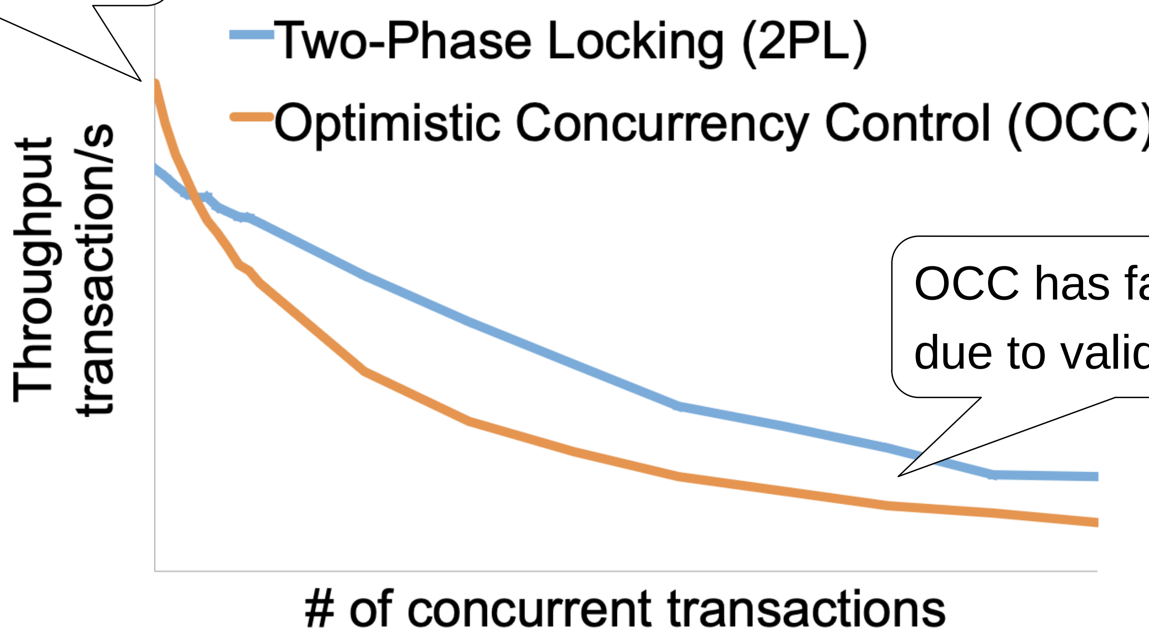
Use two-phase locking + read validation

- Only lock the write set, after that,
- Check the read set has not changed & locked

```
def validate_and_commit() // phase 2 & 3 with before-or-
after
    for d in sorted(write-set):
        d.lock()
    for d in read-set:
        if d has changed or d has been locked:
            abort()
    for d in write-set:
        write(d)
    // release the locks
    ...
```

2PL vs. OCC: in a nutshell

OCC reduced locking overhead



OCC has false aborts due to validation fails

Source: Extracting More Concurrency from Distributed Transactions [OSDI'14]

OCC & hardware transactional memory

Hardware Transactional Memory

A CPU feature for writing concurrent programs

CPU guarantees the before-or-after atomicity of memory reads/writes

– i.e., no race conditions & no need for 2PL & (Software-implemented) OCC!

Intel's HTM named as RTM

– Restricted Transactional Memory

– First released in Haswell processor

Other implementations also exist

– E.g., ARM Transactional Memory Extension (TME)

Let’s look at how to use HTM (RTM)

ISA support for transactional memory

– The ISA is very similar to what we have been described for TXs

Recall: Writing with TX in software

– Use `TX.begin` to mark when a TX starts

– Use `TX.commit` to commit the TX

In RTM, the concept is similar (new assemble code)

– Use `xbegin` to mark an RTM execution start

– Use `xend` to mark an RTM end

Programming with RTM

If transaction starts successfully

– Do work protected by RTM, and then try to commit

CPU detects race condition & aborts if necessary

– If abort, CPU rollbacks to line xbegin, return an abort code

Manually abort inside a transaction

```
if _xbegin() ==
_XBEGIN_STARTED:
     if conditions:
     _xabort()
   critical code
     _xend()
else
   fallback routine
```

Programming with RTM

If transaction starts successfully

- Do work protected by RTM, and then try to commit

CPU detects race condition & aborts if necessary

- If abort, CPU rollbacks to line `xbegin`, return an abort code

Manually abort inside a transaction

```
if _xbegin() ==
_XBEGIN_STARTED:
    if conditions:
    _xabort()
   critical code (access x)
    _xend()
else
   abort case
```

Another process

```
x = 1
```

Programming with RTM

If transaction starts successfully

– Do work protected by RTM, and then try to commit

CPU detects race condition & aborts if necessary

– If abort, CPU <u>rollbacks</u> to line `xbegin`, return an <u>abort code</u>

Manually abort inside a transaction

```
if _xbegin() ==
_XBEGIN_STARTED:
    if conditions:
    _xabort()
   critical code
    _xend()
else
   abort case
```

RTM: pros & cons

The benefits of RTM

- Memory operations between `xbegin() & xend()`satisfy before-of-after
- Much easier to program than 2PL or OCC (in most cases)
- In some cases, the performance is better

```
extern std::vector<u64> data; // shared by multiple threads
if _(xbegin() == _XBEGIN_STARTED)
    data.push(12);
else
    ?
```

Drawbacks

- Cannot guarantee success

Problem: RTM cannot guarantee success

No guaranteed success. So handling the abort case is more complex

- E.g., we cannot use a simple retry-base strategy
- Otherwise, we may encounter livelock

Why RTM cannot guarantee success?

```
Beginning:
    if _xbegin() == _XBEGIN_STARTED
        /* do some critical work */
        _xend()
    else
        goto beginning
```

Simply Retry

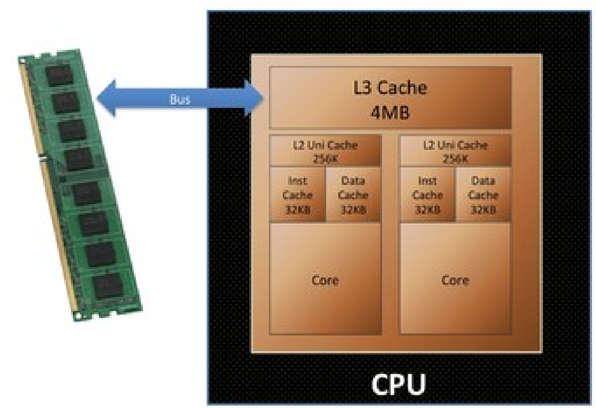
Fun facts about the implementation of RTM

Intel implements RTM using optimistic concurrency control

- OCC itself does not guarantee success

RTM's OCC is implemented on the CPU hardware, which has restrictions

- Use CPU cache to track the **read/write sets of CPU reads/writes**
- Use **cache coherence** protocols to detect conflicts



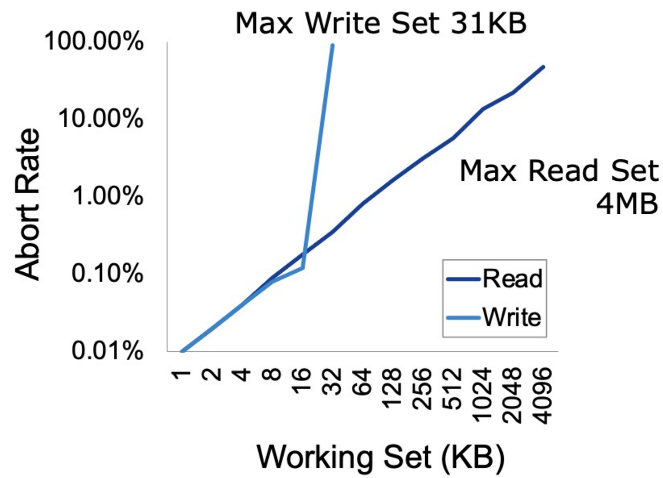
RTM: limited working set

CPU has limited cache capacity

- If the read/write sets exceed the cache size, the RTM will unconditionally abort

How big is the RTM read/write sets?

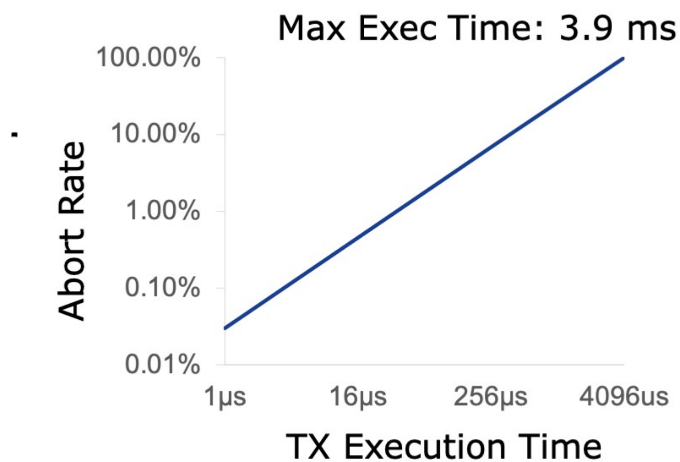
- Depends on various factors
  - Hardware setup (e.g., CPU cache size)
  - Access pattern: read or write
  - L1 cache tracks all the writes
  - L2 or L3 to tracks all the reads
  - Why not using L2 or L3 to track reads/writes?



RTM: limited execution time

The longer the execution, the higher the probability a TX aborts

– Root cause: CPU interrupts will also unconditionally abort the TX
– Why will interrupt cause an RTM to abort? Context switch will pollute the cache



Fallback path: make RTM finally commits

Problem: RTM cannot guarantee success

- Similar to OCC, code in RTM can be frequently aborted due to conflicts
- It can also be aborted due to hardware restrictions described above

Must switch to a fallback path after *some* retries (e.g., using a counter)

- E.g., fallback to locking; if fallbacks frequently, no performance benefits

```
if _xbegin() == _XBEGIN_STARTED
    if lock.held()
        xabort()
      /* do some critical work */
    _xend()
else
    lock.acquire()
```

check lock status to avoid conflict with fallback handlers on other CPU

Switch to Pessimistic Sync.

HTM on transactions

Naïve: using HTM to execute the in-memory transaction

- Smallbank: transfer &. Audit
- TPC-C: much complex, insert 10+ orders and update 10+ stocks

Silo@SOSP'13: the fastest in-memory OCC implementation

Short summary of OCC & RTM

OCC is yet another classic protocol for before-or-after atomicity

- Whose idea has even been adopted by hardware designers

Hardware support for transactional memory

- Easy programming model for the programmer (no need for locking)
  - As long as the programmer do the in-memory computations, e.g., in-memory database
- Good performance if using properly
  - No need for locking & atomic operations
- However, the programmer should handle its pitfalls

OCC & 2PL are bad when TXs are long running w/ many reads

OCC: abort due to read validation fails

2PL: read will hold the lock and block others

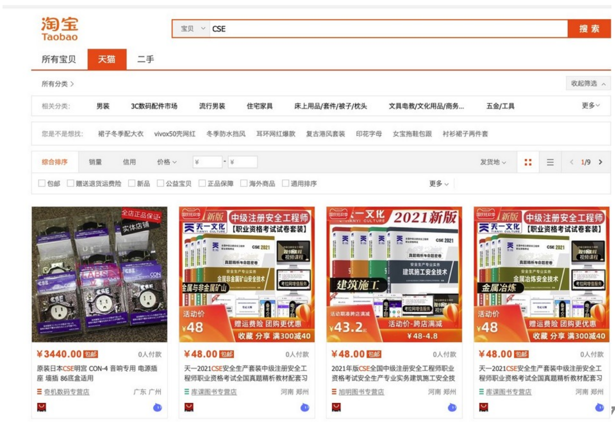
Yet, reads are common in applications

e.g., for Taobao, we rarely add items, but extensively read the items

- Scenario: long-running read-only or read-mostly transactions

run analytics on a companion data,
while update a piece of data

long running read-only transaction



Can we avoid validation for reads?

Abort in OCC

```
T1:
Print(A+B
)
```

```
T2:
B = 2
A = 3
```

Time	T1	T2
	Read(A_{T0})	
	Read(B_{T0})	
		Write(A_{T2})
		Write(B_{T2})
	Validate(A)	

Abort!

Abort in OCC

```
T1:
Print(A+B
)
```

```
T2:
B = 2
A = 3
```

Time	T1	T2
	Read(A_{T0})	
	Read(B_{T0})	
		Write(B_{T2})
	Validate(A)	

Do we necessarily need to abort T1?

Abort!

Abort in OCC case (2)

```
T1:
Print(A+B)
```

```
T2:
B = 2
A = 3
```

Time	T1	T2
	Read(A_{T0})	
		Write(A_{T2})
		Write(B_{T2})
	Read(B_{T2})	
	Validate(A)	
	Abort!	

Question: can we avoid aborting T1?

Abort in OCC case (2)

```
T1:
Print(A+B
)
```

```
T2:
B = 2
A = 3
```

Time	T1	T2
	Read(A_{T0})	
		Write(A_{T2})
		Write(B_{T2})
	Read($\mathbf{B}_{\mathbf{T0}}$) ~~Read($\mathbf{B}_{\mathbf{T2}}$)~~	Overwrite

Idea: multi-versioning concurrency control

Each data has multiple-versions instead of a single version

- A set of version represents a snapshot of the database
  - E.g., $A_{T0} + B_{T0}$
- The version ~= the time a TX makes the updates

Data X

```
Struct Data {
 value :  u8[…]
 lock : lock_t
}
```

VersionedData X_0 X_1

```
Struct Data {
 value: List<VersionedData>
 lock : lock_t
}
```

```
Struct VersionedData {
 value: u8[…]
 version: u64
}
```

Idea: multi-versioning concurrency control

Read

- Only read from a consistent snapshot at **a start time**

Write

- Install a new version of the data instead of overwriting the existing one
- Version ~= the **commit time** of the TX

Goal: avoid race conditions on reading a snapshot

Data X

```
Struct Data {
 value :  u8[…]
 lock : lock_t
}
```

VersionedData X_0 X_1

```
Struct Data {
 value: List<VersionedData>
 lock : lock_t
}
```

```
Struct VersionedData {
 value: u8[…]
 version: u64
}
```

Get the start and commit time

Requirement: the counter reflects TX's serial execution order

- E.g., if T1 finishes before T2, it will has a smaller start & commit timestamp

Simplest (& most widely used) solution: global counter

- Using atomic fetch and add (FAA) to get at the TX's begin & commit time
- TX Begin: use FAA to get the start time
- TX Commit: use FAA to get the commit time

We will introduce more advanced timestamps in later lectures

- Not using a global counter is challenging because de-centralized time (e.g., physical time) is unsynchronized

Try #1: Optimize OCC w/ MV (incomplete)

Acquire the start time

Phase 1: Concurrent local processing

- Reads data belongs to the snapshot closest to the start time
- Buffers writes into a write set

Acquire the commit time

Phase 2: Commit the results in critical section

- Commits: installs the write set with the commit time

Compared to the OCC, no validation is need!

Try #1: Optimize OCC w/ MV (incomplete)

```
Commit(tx):
   for record in tx.write_set:
       lock(record)
   let commit_ts = FAA(global_counter)
   for record in tx.write_set:
       record.insert_new_version(commit_ts, ...)
       unlock(record)


Get(tx, record):


   for version,value in
record.sort_version_in_decreasing():
       if version <= tx.start_time:
          return value
```

Partial snapshot

```
T1:
Print(A+B)
```

T1

```
T2:
B = 2
A = 3
```

T2

Global counter g (initial 0)

A: A_0
B: B_0

1 Start_time = FAA(g)

Read(A) = A_0

2 Commit_time = FAA(g)

Write(B, B_2)

A: A_0
B: **B_2** B_0

Read(B) = B_0

Write(A, A_2)

A: **A_2** A_0
B: **B_2** B_0

No validation needs here (for T1) !

Problem: partial updated snapshot

Partial snapshot example

```
T1:
Print(A+B)
```

```
T2:
B = 2
A = 3
```

Global counter (initial 0) **T1** **T2**

Needs to be atomic

1 Commit_time = FAA(g)

Write(B, B_1)

A: A_0
B: B_0

A: A_0
B: B_1 B_0

2 start_time = FAA(g)

Read(A) = A_0

Read(B) = B_1

Write(A, A_1)

A: A_1 A_0
B: B_1 B_0

Missing the updates! We need T2's write being before-or-after atomic!

Enforcing the write atomicity of snapshot w/ locking

```
Commit(tx):
   for record in tx.write_set:
       lock(record)
   let commit_ts = FAA(global_counter)
   for record in tx.write_set:
       record.insert_new_version(commit_ts, ...)
       unlock(record)


Get(tx, record):
   while record.is_locked():
         pass
   for version,value in
record.sort_version_in_decreasing():
       if version <= tx.start_time:
          return value
```

Example revisit

Question: can we reorder get commit time and locking?

Global counter (initial 0) | **T1** | **T2**

Lock A, B

1 ← Commit_time = FAA(g)

A: A_0
B: B_0

Write(B, B_1)

Unlock B

A: A_0
B: $\mathbf{B_1}$ B_0

2 ← start_time = FAA(g)

Read(A) = ?

Cannot read! Because A is locked!

A: A_0
B: $\mathbf{B_1}$ B_0

Write(A, A_1)

Unlock A

A: $\mathbf{A_1}$ A_0
B: $\mathbf{B_1}$ B_0

Read(A) = $\mathbf{A_1}$

Read(B) = $\mathbf{B_1}$

What about writes?

```
T1:
A += 1
```

```
T2:
A += 1
```

Global counter (initial 0)	T1	T2
1	Start_time = FAA(g) Read(A_0) = 0	
2		Start_time = FAA(g) Read(A_0) = 0
3	Commit_time = FAA(g) Write ($A_{3,}$ 1)	
4		Commit_time = FAA(g) Write ($A_{4,}$ 1)

MVCC ensures no race for reads, but not writes

Race conditions of reads are isolated via snapshots

But we still needs to rule out race conditions between reads and writes

- We do so by validating the writes at the commit time

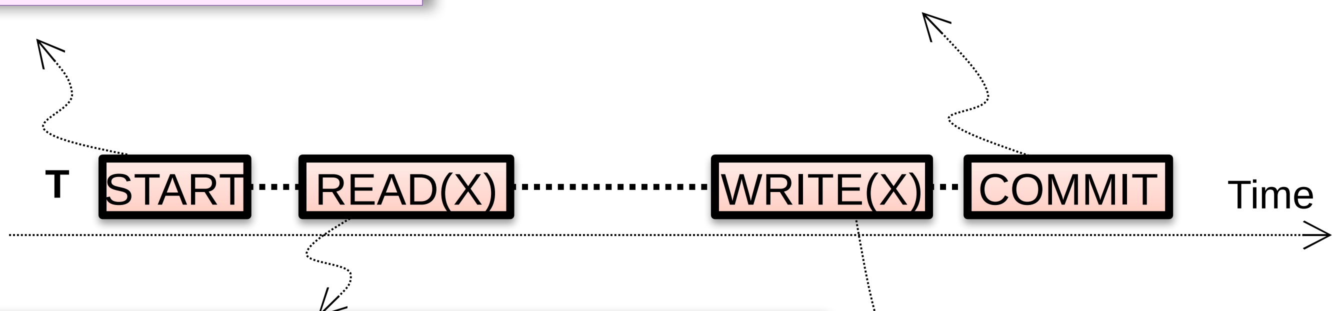
The validation is simple:

- During commit time, check whether another TX has installed a new snapshot after the committing TX's start time

Put it together, MVCC so far

T is assigned a start timestamp T.sts

T is assigned a commit timestamp T.cts
System checks all data within T.wset,
if T.sts < X.cts, then abort **T**
Update all data within T.wset with T.cts



T reads the biggest version of X(i), such that X(i).cts <= T.sts

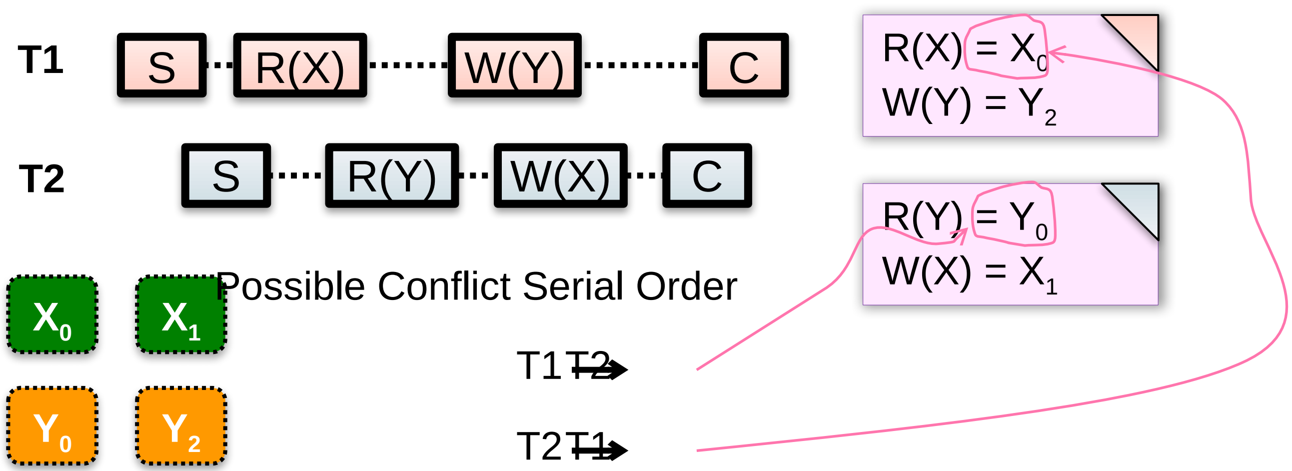
T buffers writes to X and adds X to its write-set, T.wset += {X}

Question: do our current MVCC method guarantee before-or-after?

Write skew anomaly

Our MVCC differs from serializability due to one **anomaly**

Describe the anomaly and also give a concrete application for which the anomaly is undesirable.



Fixing the anomaly

The simplest way is to validate the read-set in read-write TX

- Essentially fallbacks to OCC for read-write TX
- But read-only TX can still enjoy the benefits from MVCC
  - Never aborts & no validations

Usually being ignored in practice (Snapshot isolation)

- The MVCC without the read validation is also called **snapshot isolation (SI)**
- Though the idea of MVCC is first proposed in SI, its usage is not restricted to it, e.g., we can have MV-2PL and MV-OCC; & we will see it later

Multi-site transaction & Multi-site atomicity

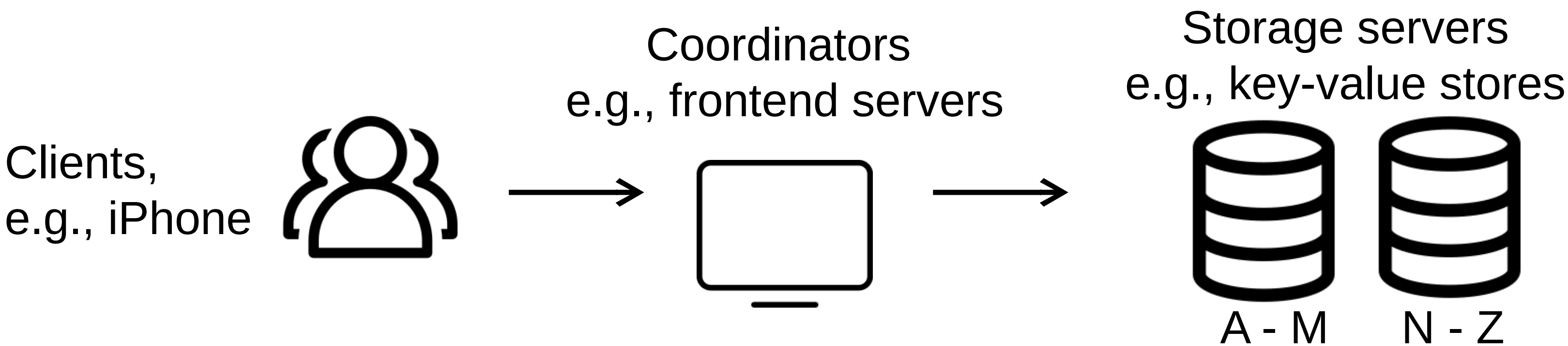
Multi-site transaction: what if the data is distributed?

The data accessed by TXs are stored on multiple machines

- i.e., a single site cannot store all the bank accounts

The setup

- Client + coordinator + two servers
- One server handles bank accounts A-M
- The other server handles bank accounts N-Z
- Coordinator and servers all have logs to ensure single transaction atomicity



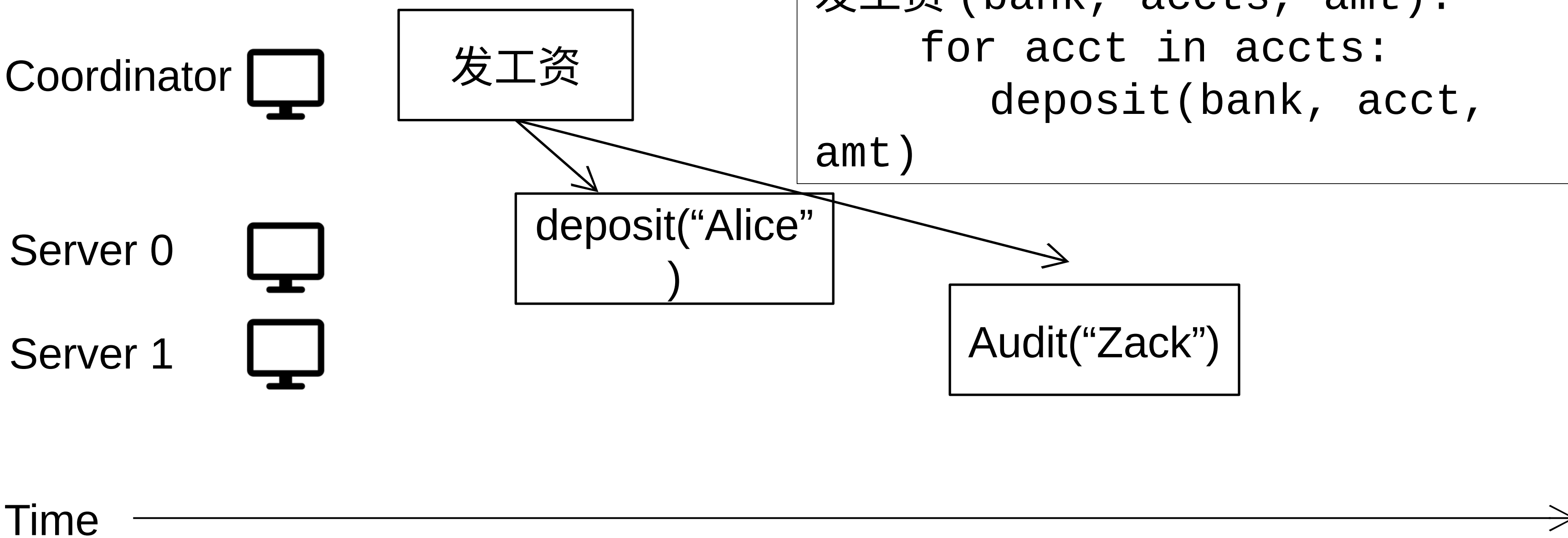
Multi-site transaction

```
Deposit(bank, a, amt):
  bank[a] += amt
```

e.g., coordinator sends multiple deposit to different servers

– They use RPCs to send requests to the server to execute transaction

```
发工资(bank, accts, amt):
    for acct in accts:
       deposit(bank, acct, 
amt)
```

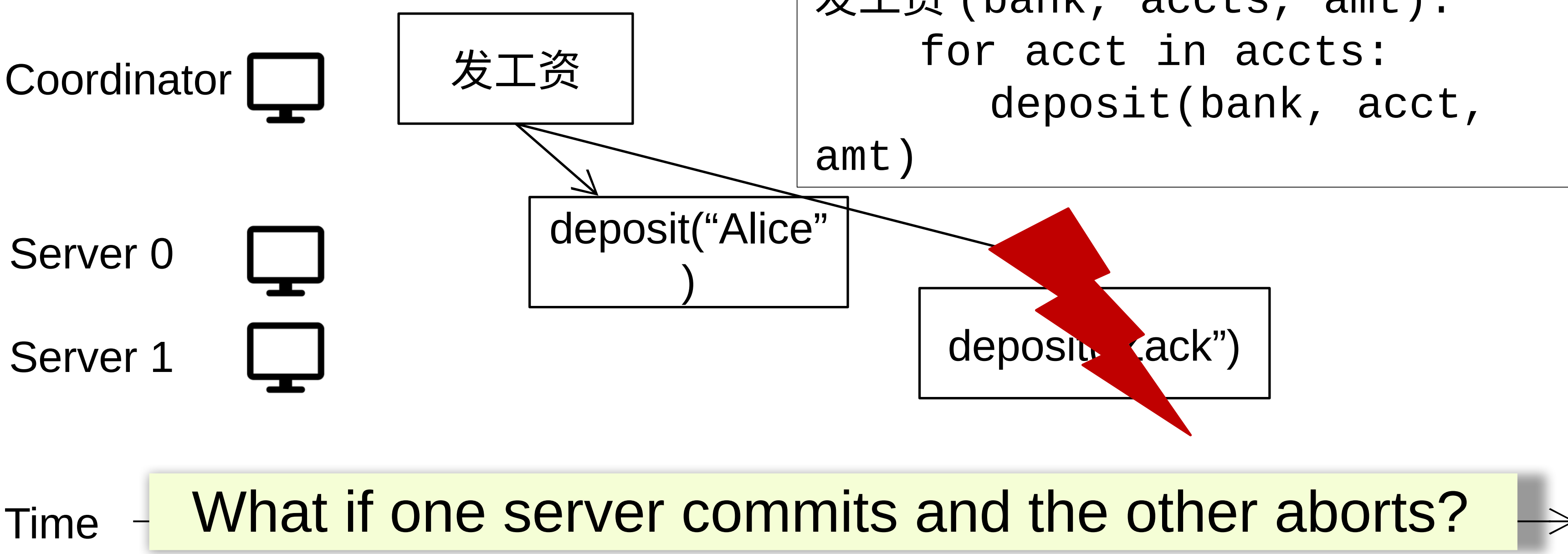


Multi-site transaction

Coordinator sends multiple deposit to different servers

– They use RPCs to send requests to the server to execute transaction

```
发工资(bank, accts, amt):
    for acct in accts:
        deposit(bank, acct,
amt)
```



What if one server commits and the other aborts?

Goal: all sites either all commits or all aborts

Two-phase Commit: a way to ensure multi-site atomicity

High-layer TX

- The high-level view of the TX

Low-layer TX

- Specific reads and writes that executed on a single machine

Higher-layer transaction coordinates the execution of lower-layer TXs

- All low-layer TX either all commits or all aborts

```
发工资(bank, accts, amt):
    tx.high_begin()
    for acct in accts:
        deposit(bank, acct,
amt)
    tx.high_com
```

High-layer TX

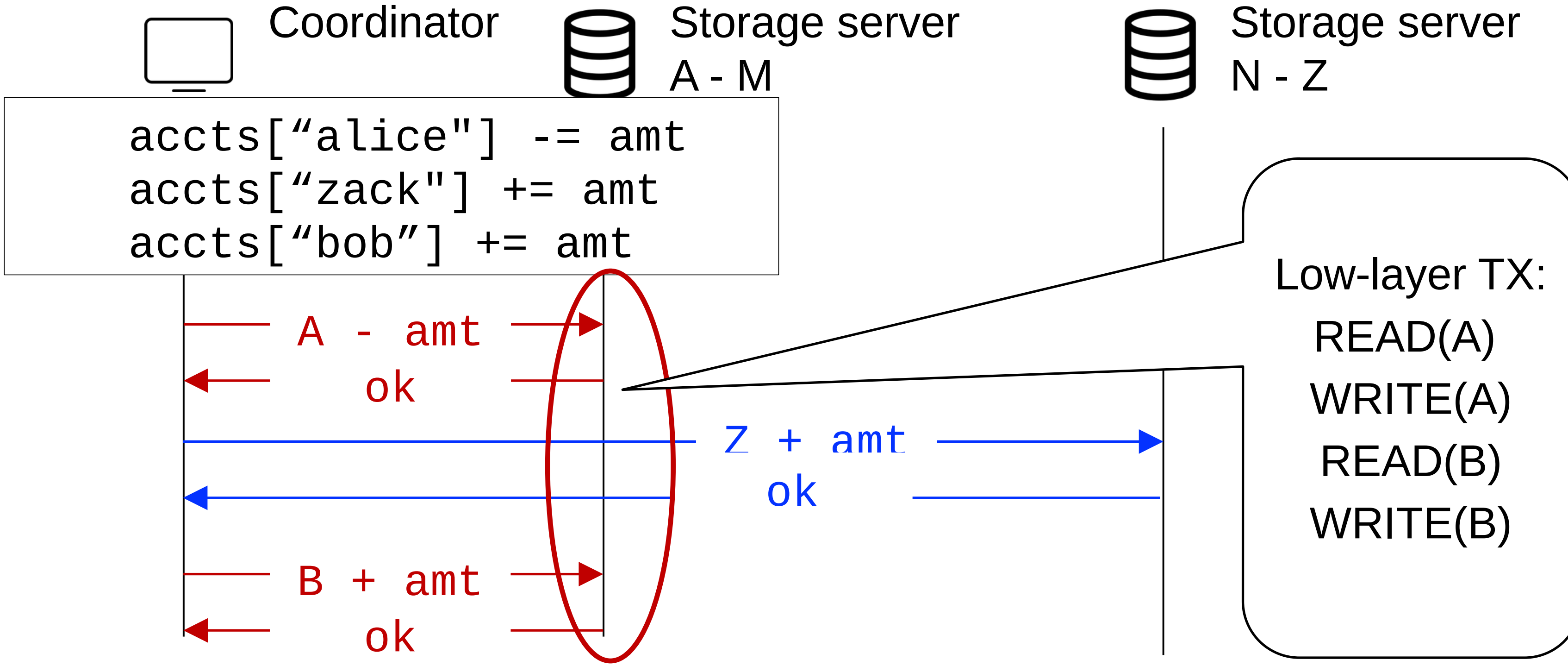
```
Deposit(bank, a, amt):
    tx.low_begin()
    bank[a] += amt
    tx.low_commit()
```

Low-layer TX

We also aggregate scattered accesses as low-layer TX

The coordinator can send scattered reads/writes to a site

– The reads and writes on that site also form a low-layer TX



Two-phase Commit: a way to ensure multi-site atomicity

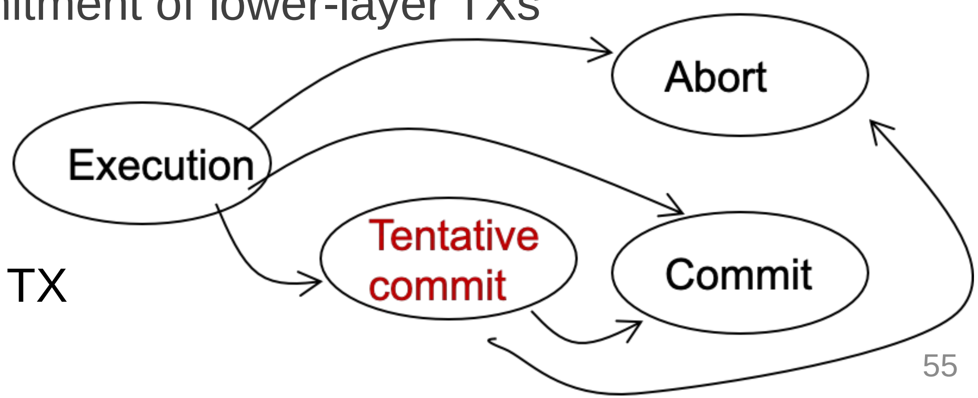
Phase-1: preparation / voting

- Delay the commitment of low-layer TXs
- Lower-layer transactions either abort or *tentatively* committed
- Higher-layer transaction evaluate lower situation

Phase-2: commitment

- The high-layer decides whether low-layer TXs will commit or abort
- It will also coordinate the commitment of lower-layer TXs

State transitions of a TX



Two-phase Commit: a way to ensure multi-site atomicity

High-layer TX

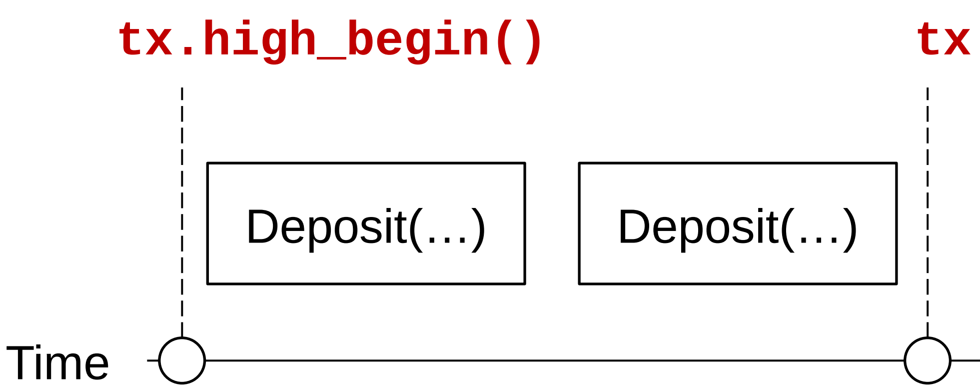
```
发工资(bank, accts, amt):
    tx.high_begin()
    for acct in accts:
        deposit(bank, acct, amt)
    tx.high_commit()
```

High_begin

- Mark itself as a high-level transaction

High_commit

- Send prepare messages to the low-level transactions to check whether it can commit




```
for each deposit:
    if deposit.abort: abort()
for each deposit:
    deposit.commit()
```

We need extension to the low-layer TX's lo

Low-layer TX

```
Deposit(bank, a, amt):
    tx.begin()
    bank[a] += amt
    tx.commit()
```

Recall: the WAL logging in lecture 10

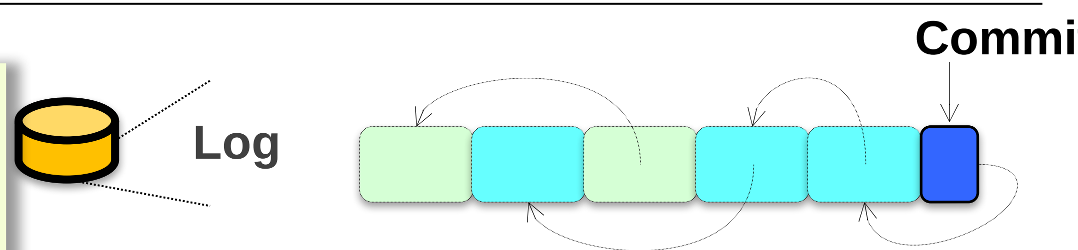
– At commit point, append a commit record to the log

```
...
log.append("TX {id}
commit”).sync()
...
```

Question

– Can we directly append the commit record to the lower-level transaction log?

No! The high-level transaction can abort



Logging rule of WAL under 2PC

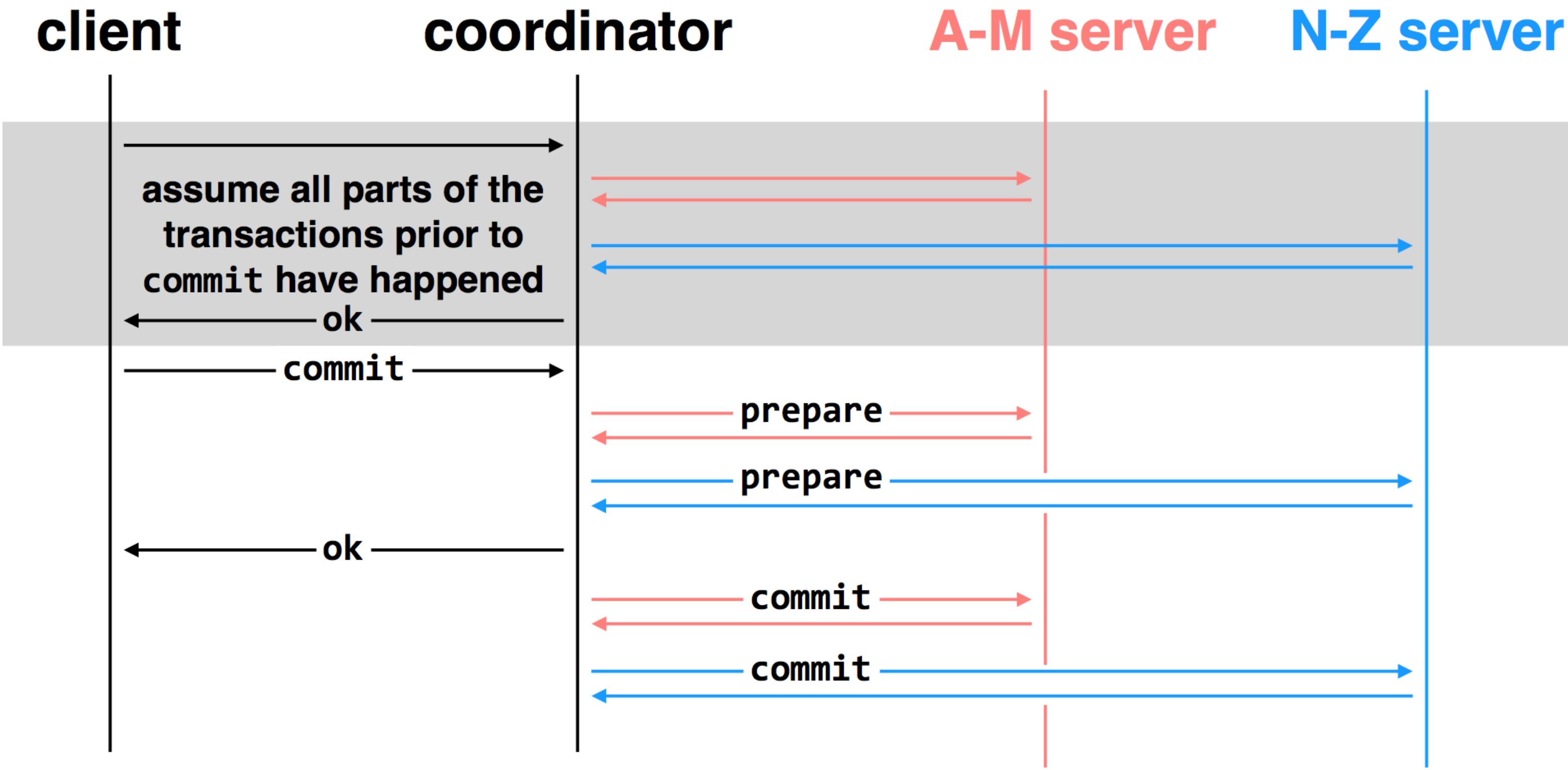
Low-layer transaction

- REDO-UNDO log entries: like normal
- Commit log entry -> Tentative commit log entry **(PREPARED)**
  - Contains a reference to the high-layer TX
  - In case of failure: ask the high-layer TX's coordinator to see if I can commit

High-layer transaction （Responsible for commit of low-layer TXs)

- Log the prepare log as a commitment of a high-layer TX

Multi-site transactions under two-phase commit



Idea: the coordinator is responsible for committing all the TXs

How? By sending RPCs or Messages to all the other servers

Challenge: unreliable communications, coordinator & worker

Multiple-site Atomicity

Principles:

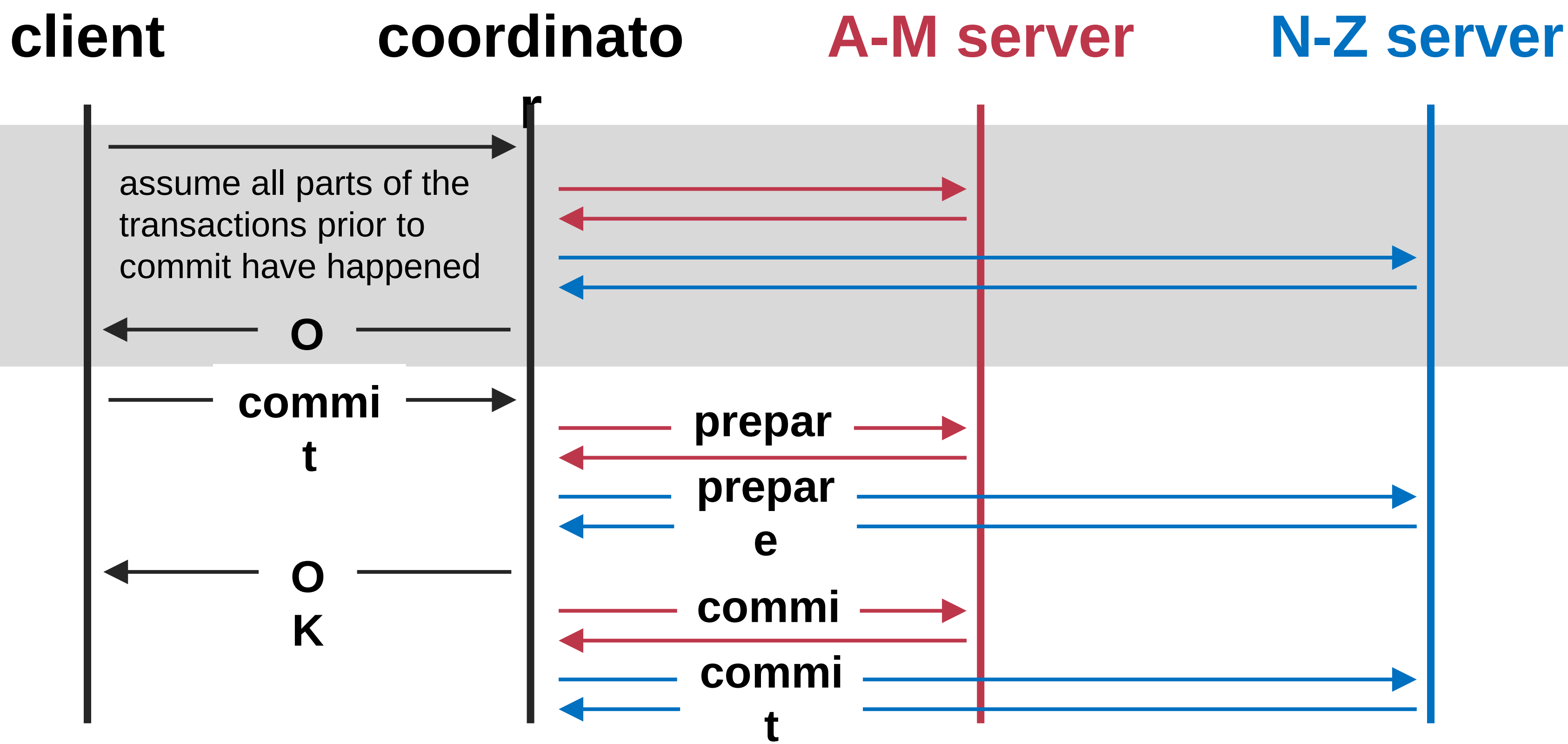
- Following the coordinator's decision
- Log sufficient state to tolerate failures (e.g., the coordinator's decision)

Coordinator (do the decision & maintain the state)

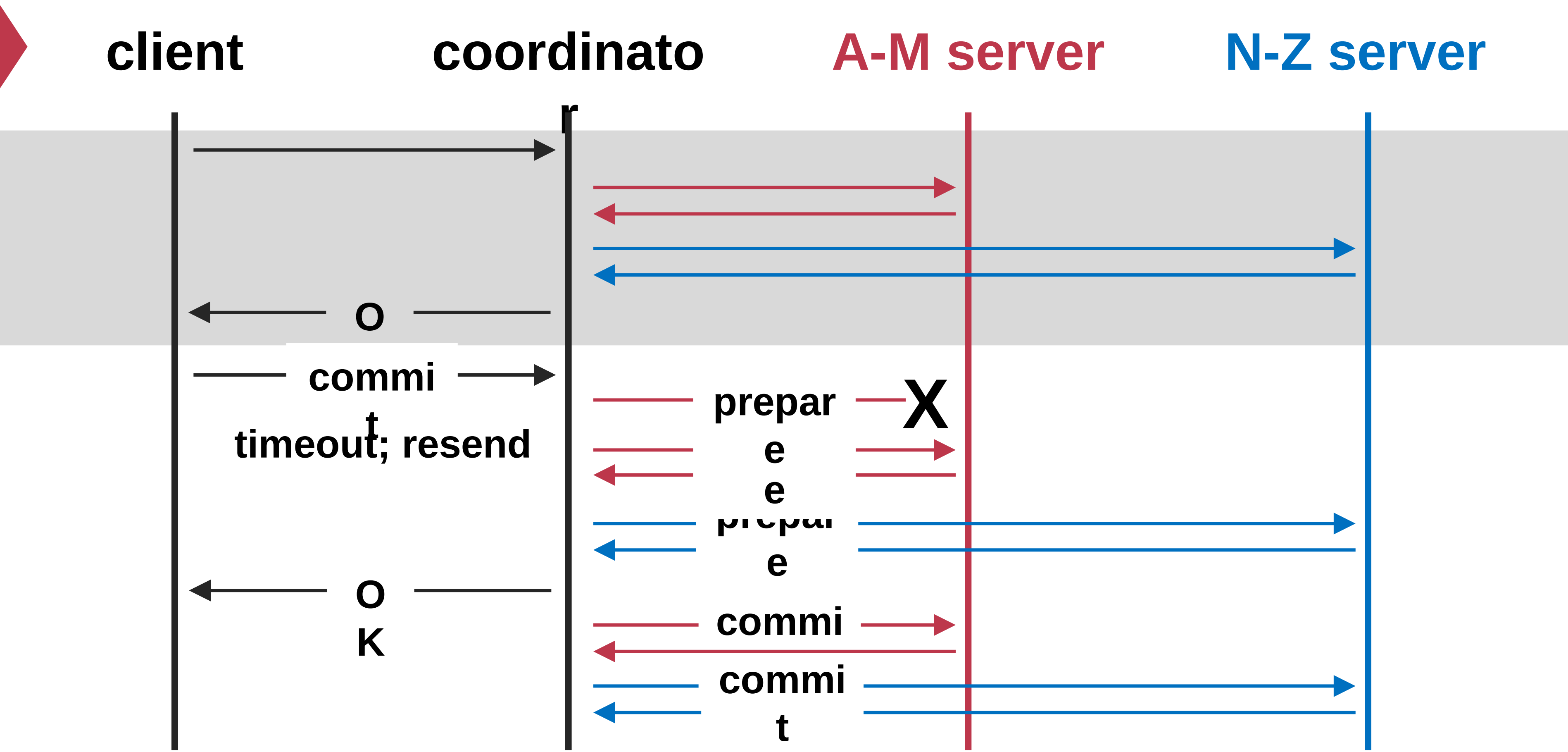
- Collect some **ABORT** or nothing: **ABORT** or retry
- Collect all **COMMIT**: then **COMMIT**

Worker passively react to the coordinator's actions

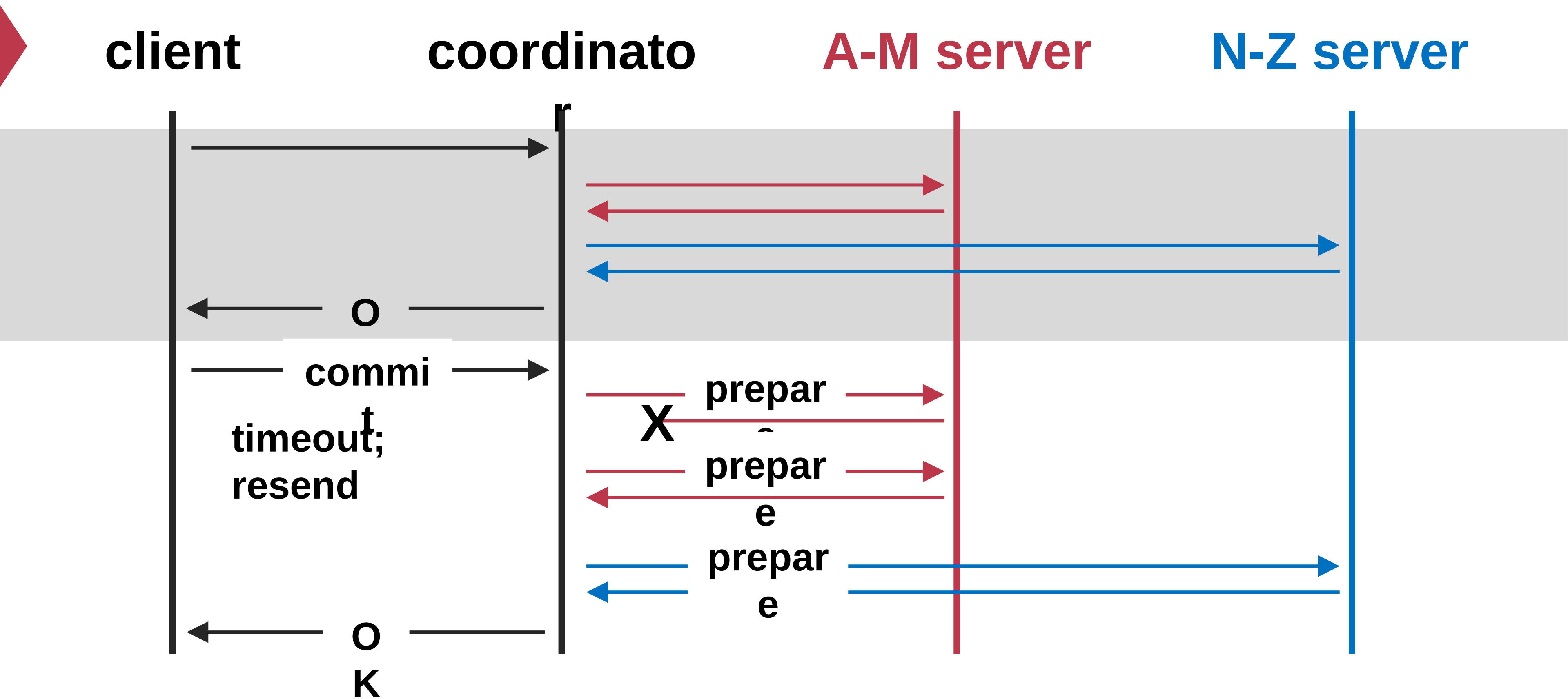
- If no message is sent from the coordinator, just wait
- When receive **COMMIT**: then **COMMIT**



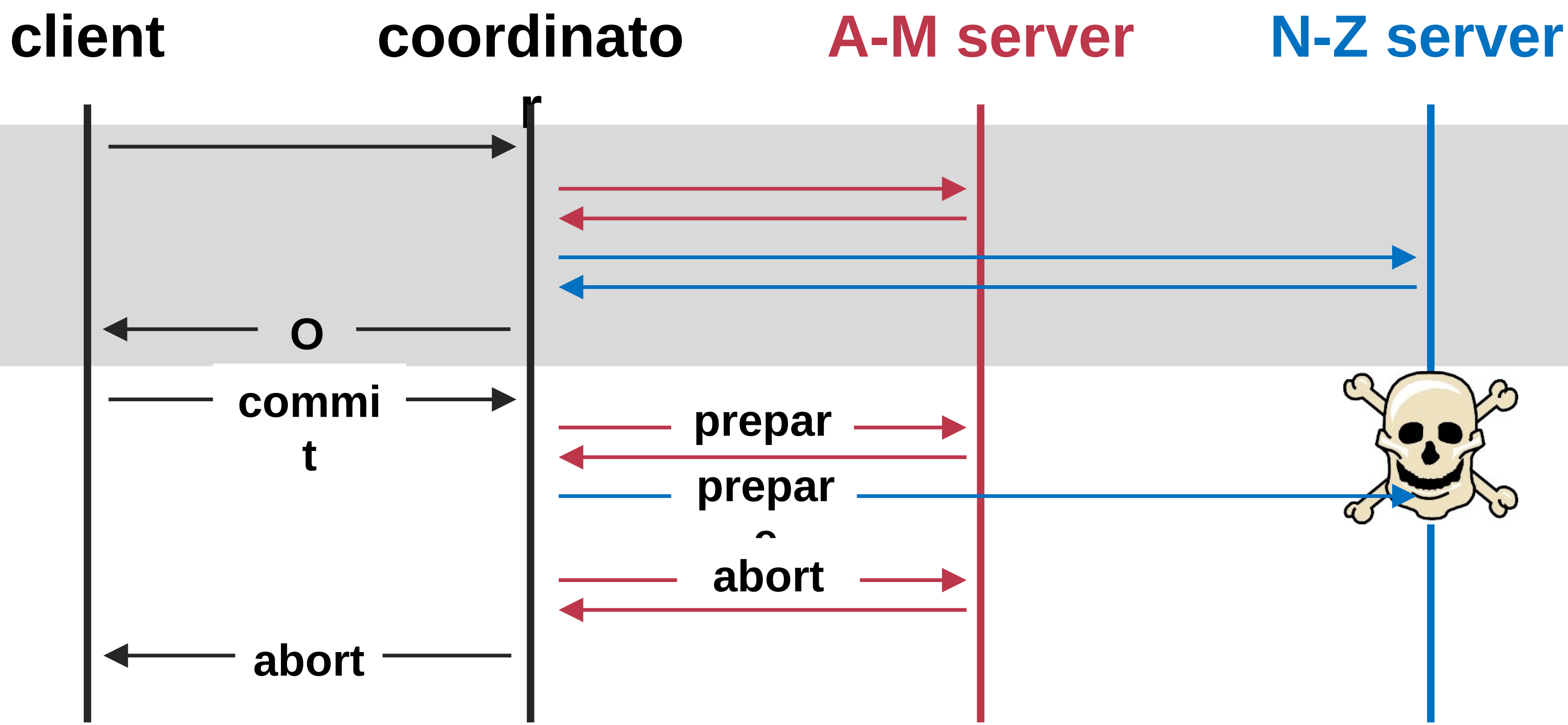
two-phase commit: **nodes agree that they are ready to commit before committing**



failure: lost prepare

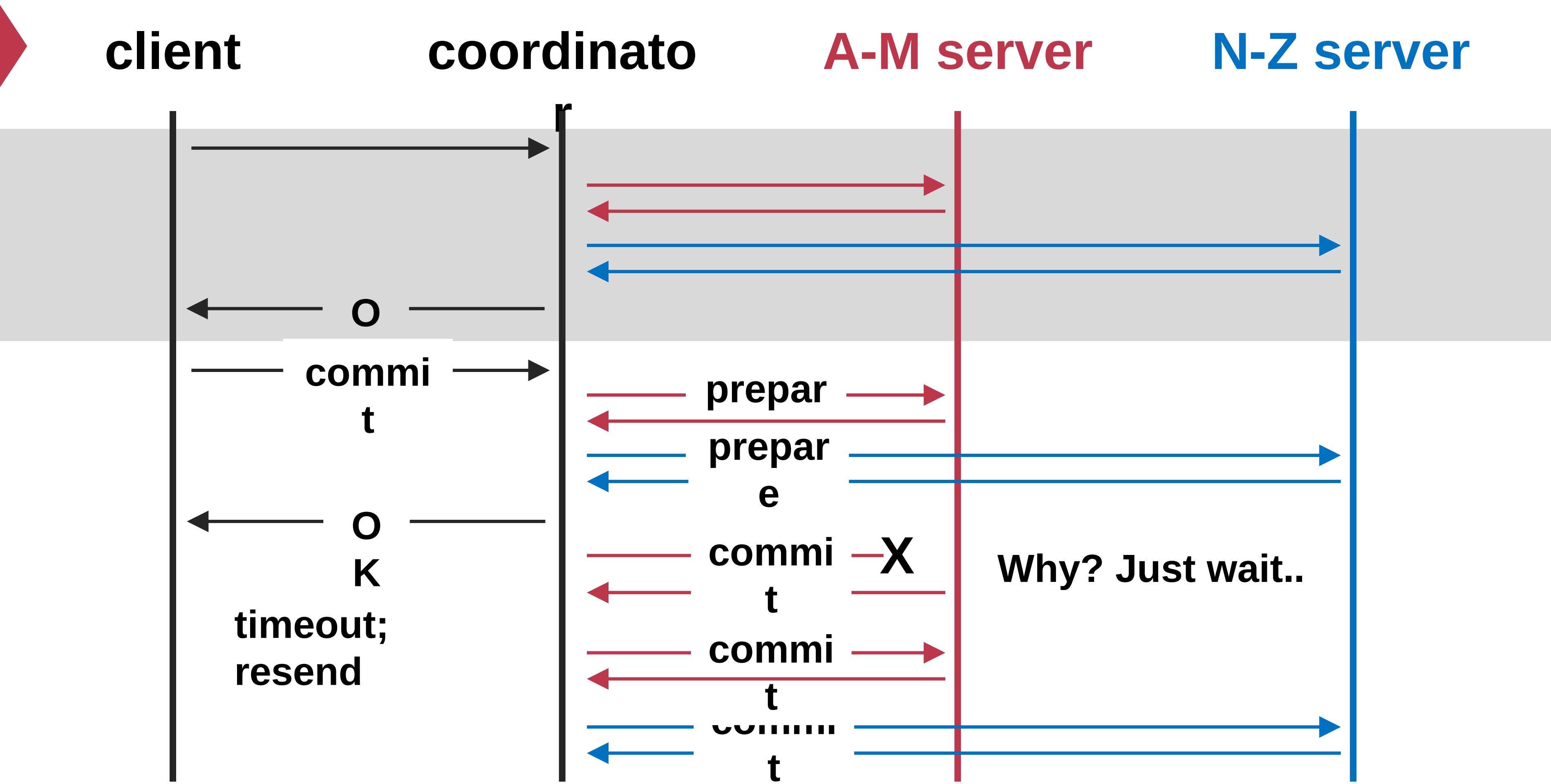


failure: lost **ACK** for **prepare**

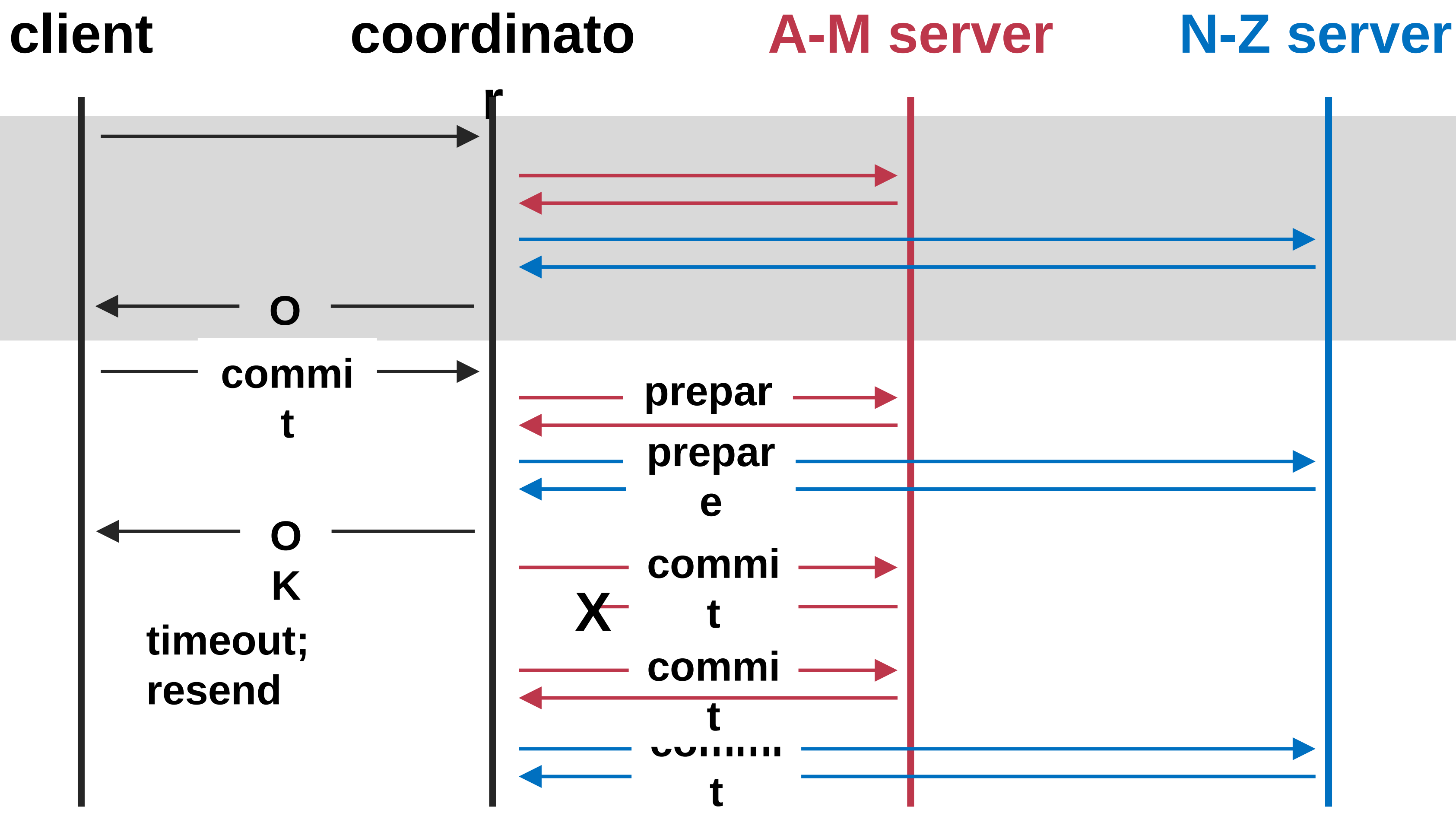


failure: worker failure during prepare

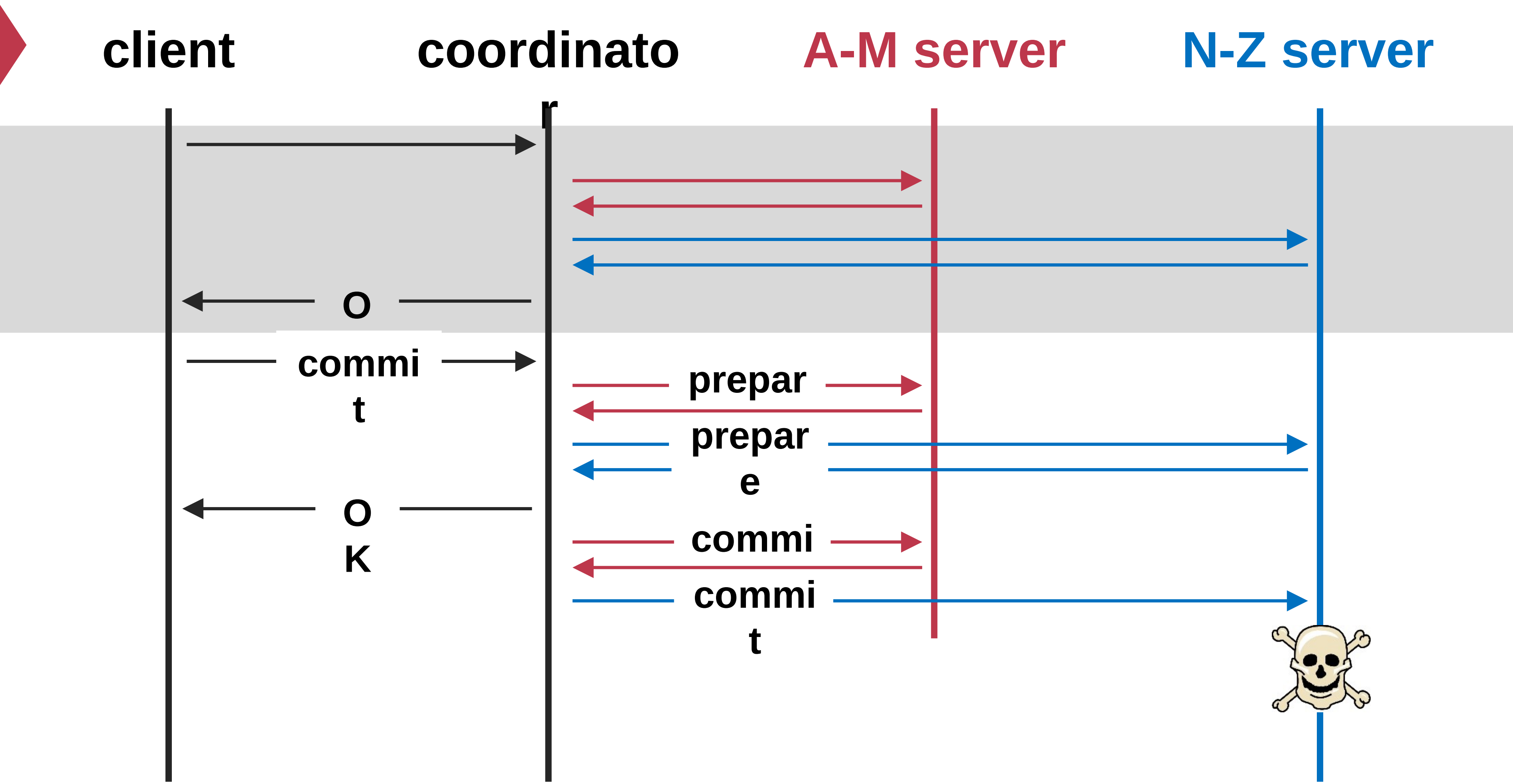
coordinator can safely abort transaction, will send explicit abort messages to live workers



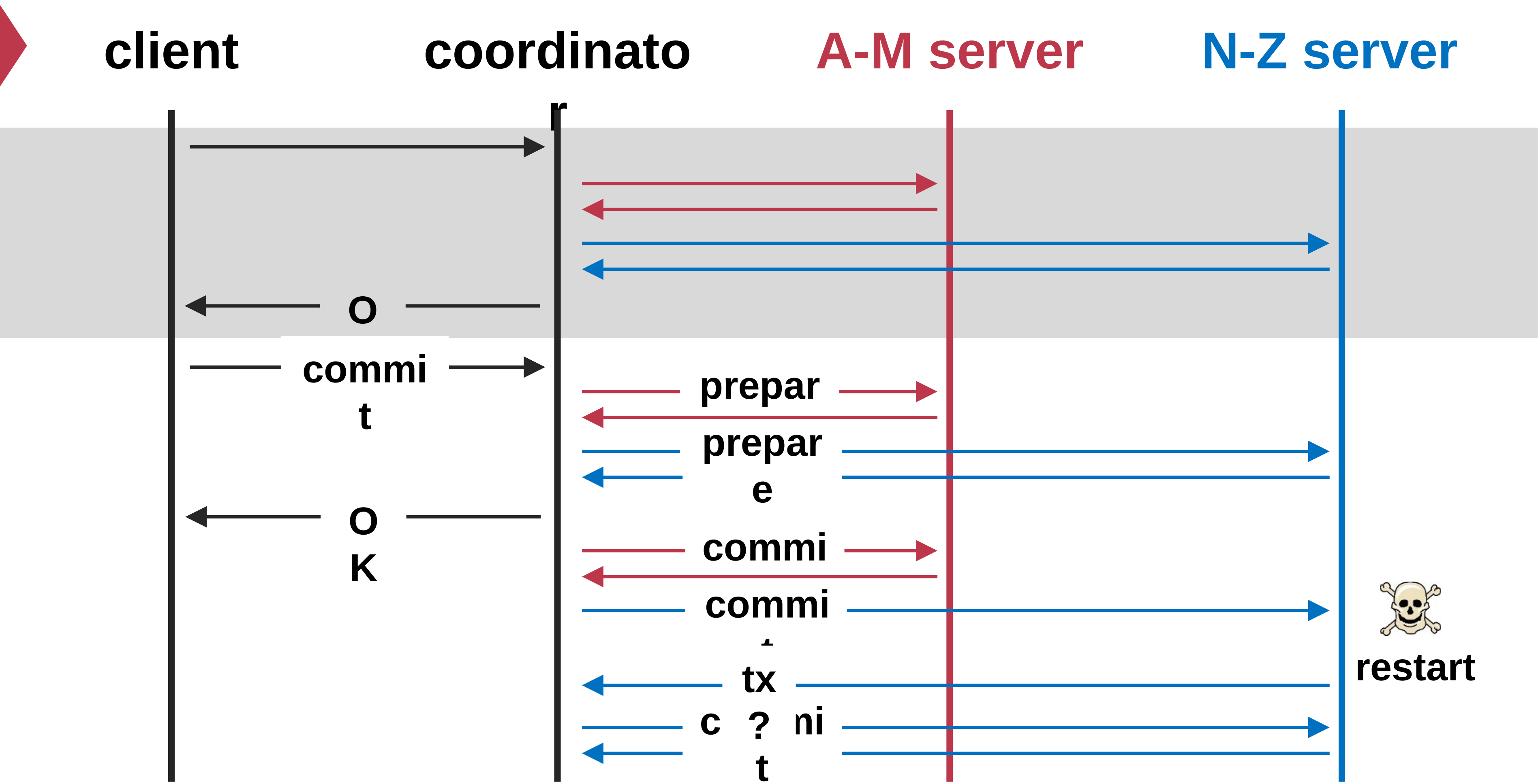
failure: lost commit message



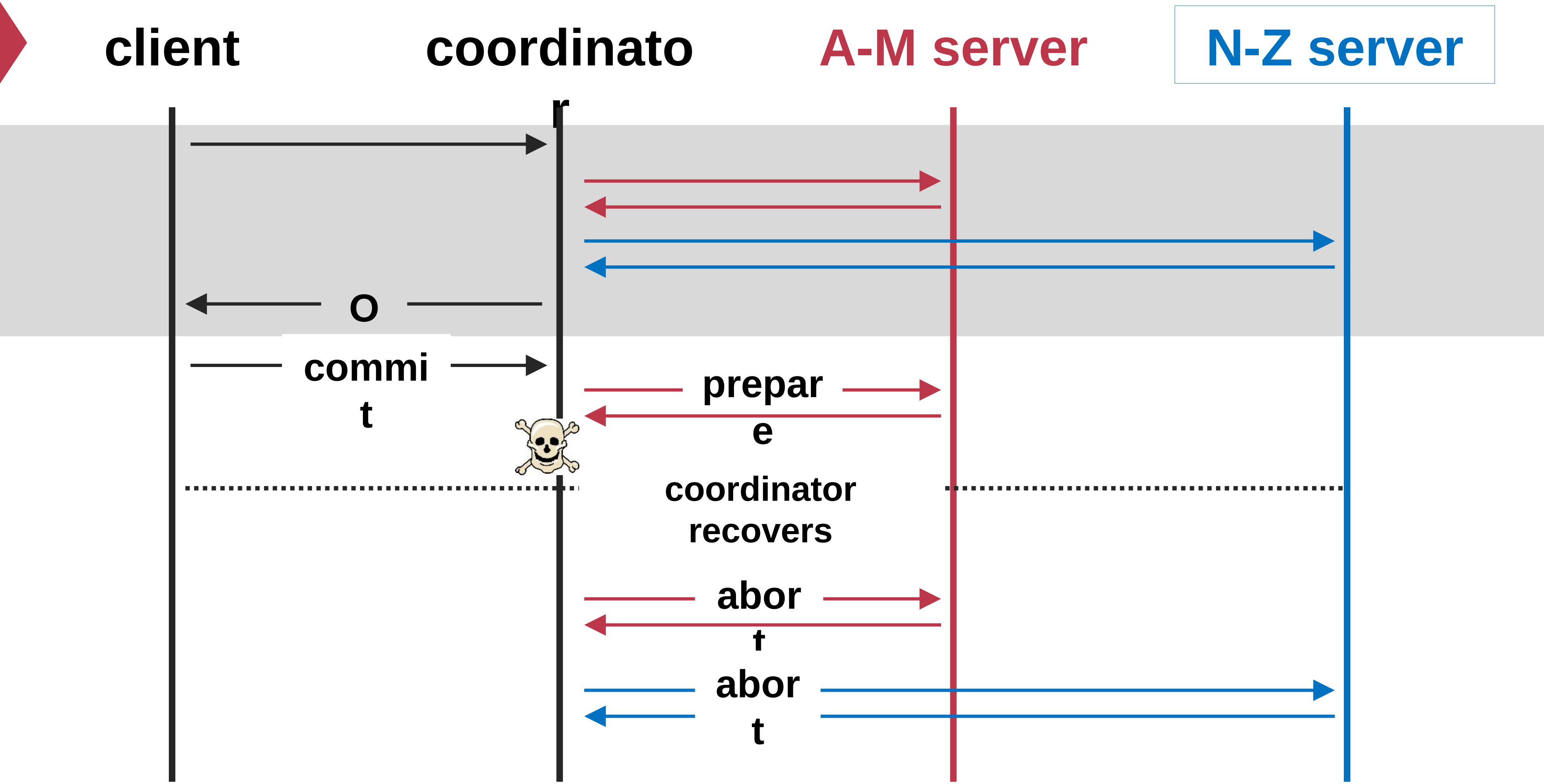
failure: lost **ACK** of **commit** message



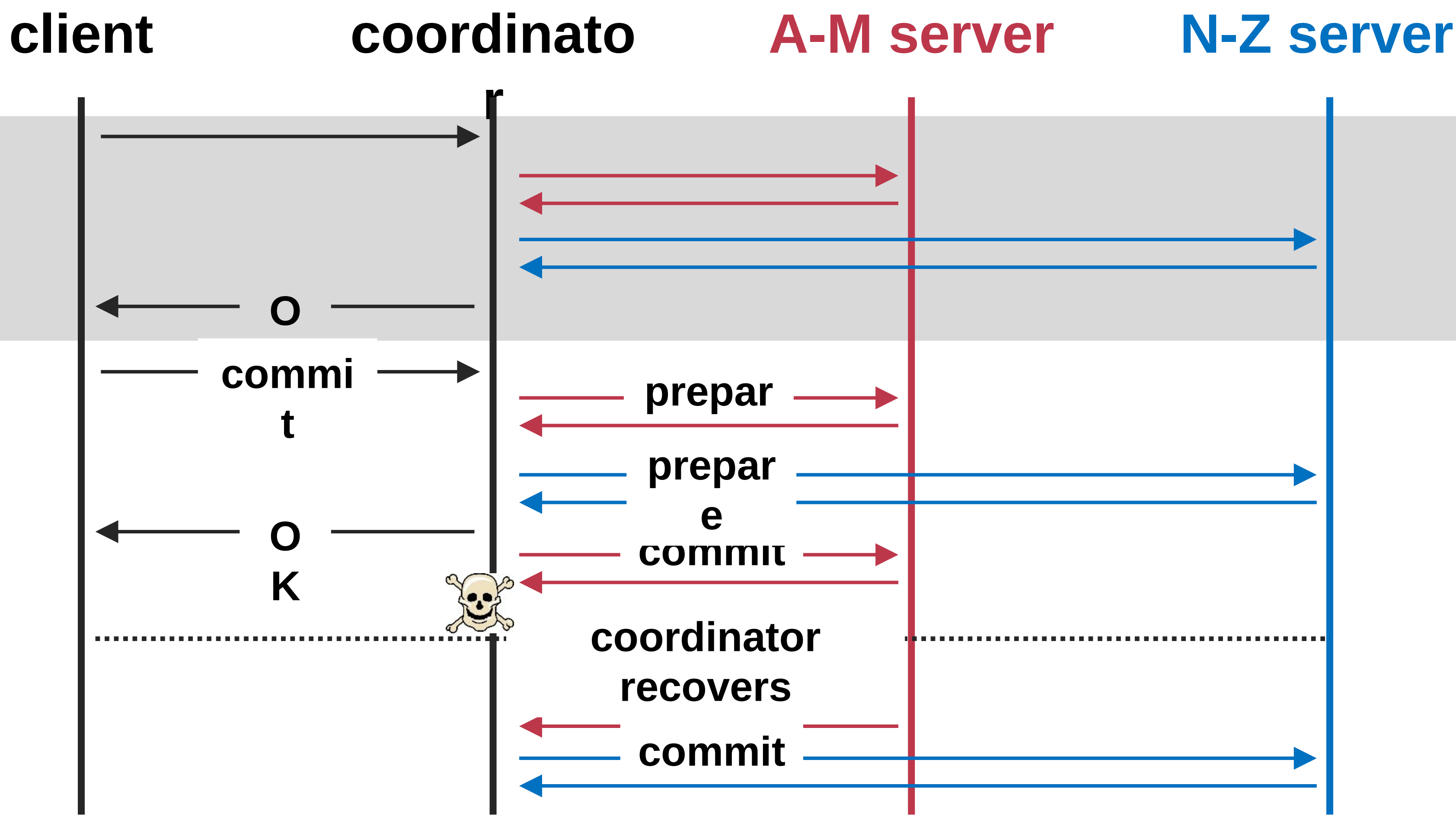
failure: worker failure during commit



failure: worker failure during commit



failure: coordinator failure during prepare



failure: coordinator failure during commit

Must log the decision before sending the commit messages

What about 2PL or OCC in 2PC?

Nearly identical

- 2PL: each low-layer TX cannot release its lock until the high-layer TX decides to commit
- OCC: the validation & commit phases are done by the coordinator

Summary of 2-phase Commit

Two-phase commit allows us to achieve **multi-site atomicity**: transaction remains atomic even when they require communication with multiple machines

In two-phase commit, failures prior to the commit point can be aborted. If workers (or the coordinator) fail after the commit point, **they recover into the PREPARED state**, and complete the transaction

Remaining challenge: availability under coordinator failures

- Follow the coordinator's decision simplifies achieving multi-site atomicity, but what if the coordinator crashes for a long time?
- We will see how to make the coordinator available in the next lecture