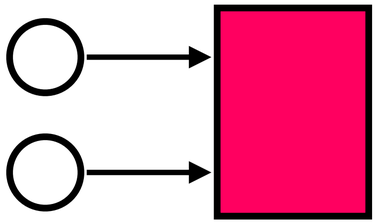
6.033 Spring 2021

Lecture #18: Distributed Transactions

getting atomicity across machines

Katrina LaCurts | lacurts@mit.edu | 6.033 2021

our goal is to build **reliable systems from unreliable components**. we want to build systems that serve many clients, store a lot of data, perform well, all while keeping availability high



transactions — which provide **atomicity** and **isolation** — make it easier for us to reason about failures

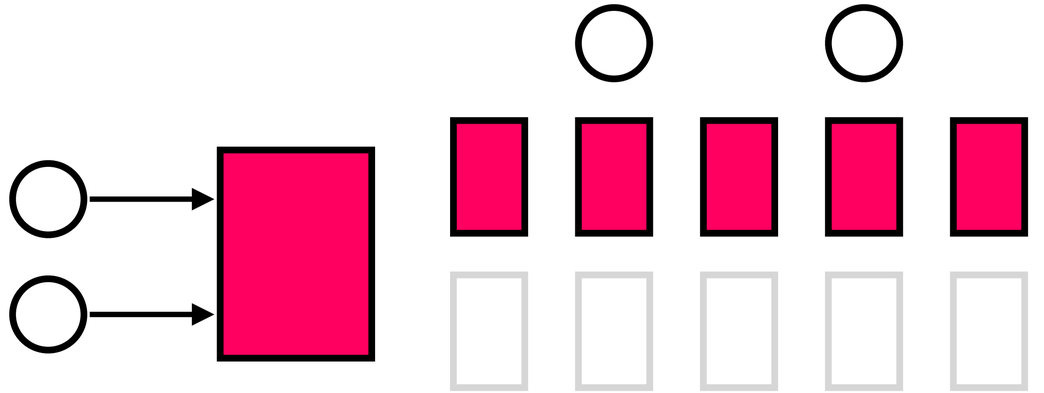
our job in lecture is to understand how a system *implements* these two abstractions.
how do our systems guarantee atomicity? how do they guarantee isolation?

atomicity: provided by **logging**, which gives better performance than shadow copies* at the cost of some added complexity

* shadow copies *are* used in some systems

isolation: provided by **two-phase locking**

our goal is to build **reliable systems from unreliable components**. we want to build systems that serve many clients, store a lot of data, perform well, all while keeping availability high



transactions — which provide **atomicity** and **isolation** — make it easier for us to reason about failures

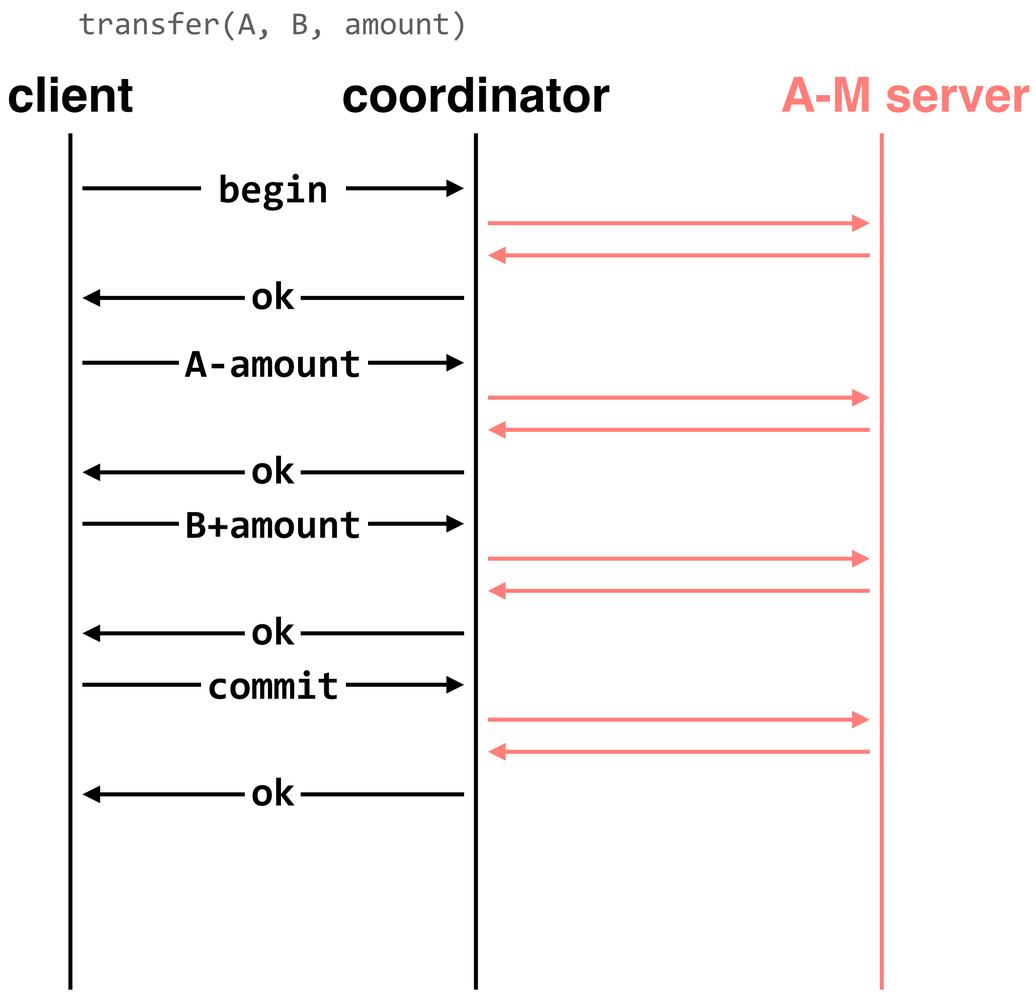
our job in lecture is to understand how a system *implements* these two abstractions.
how do our systems guarantee atomicity? how do they guarantee isolation?

atomicity: provided by **logging**, which gives better performance than shadow copies* at the cost of some added complexity

* shadow copies *are* used in some systems

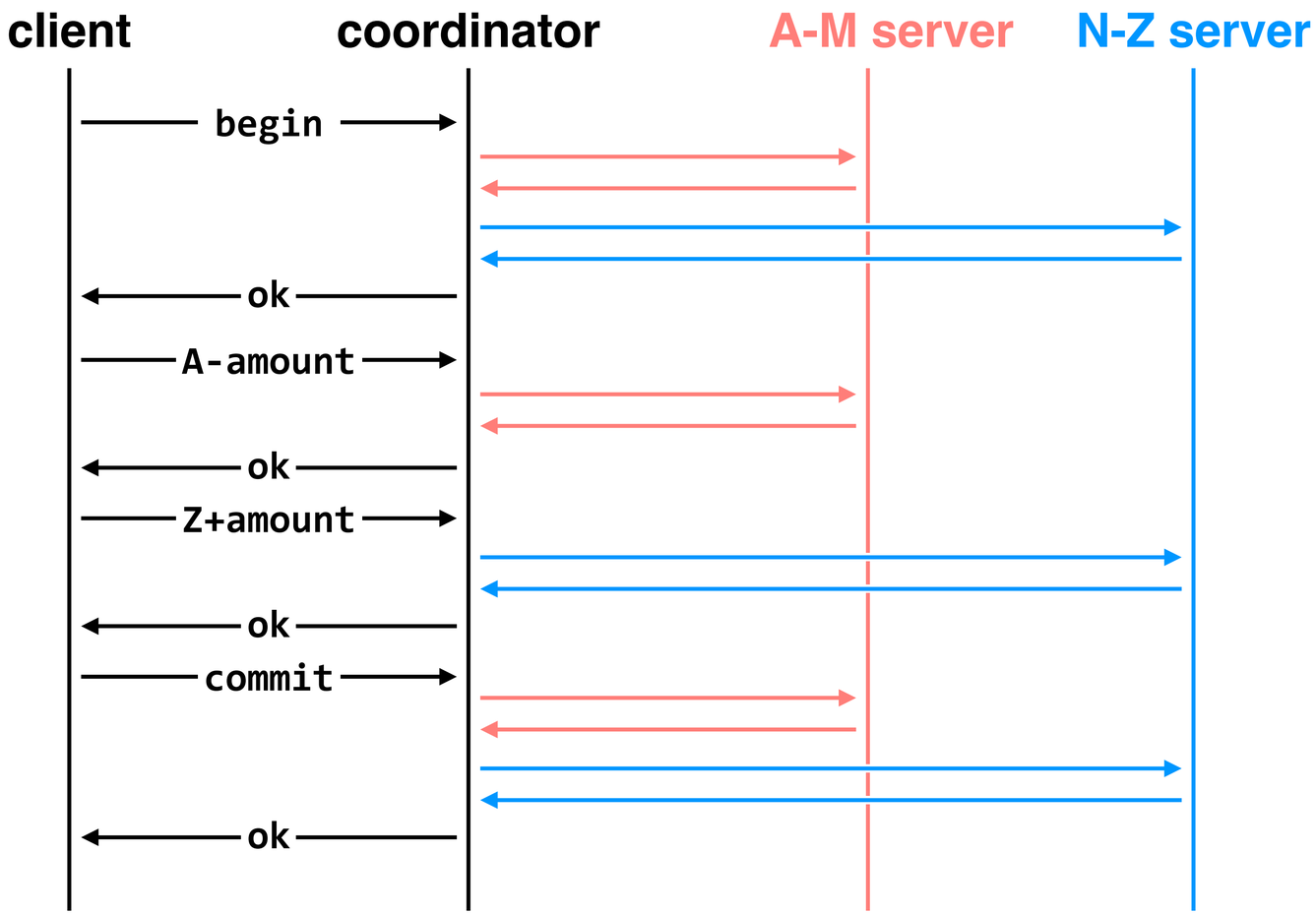
isolation: provided by **two-phase locking**

transactions across multiple machines (no failures yet)

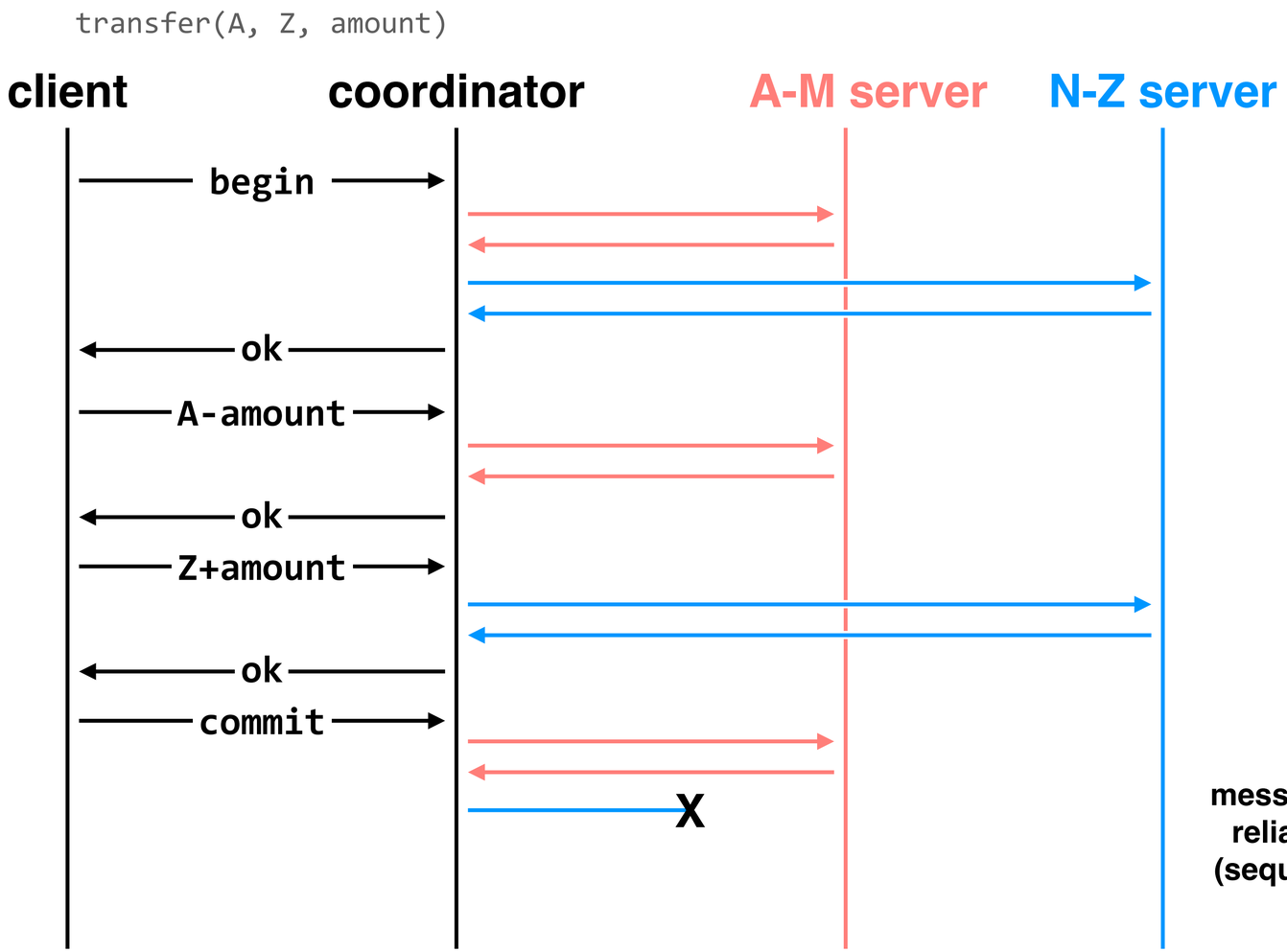


transactions across multiple machines (no failures yet)

```
transfer(A, Z, amount)
```



transactions across multiple machines (now with failures)



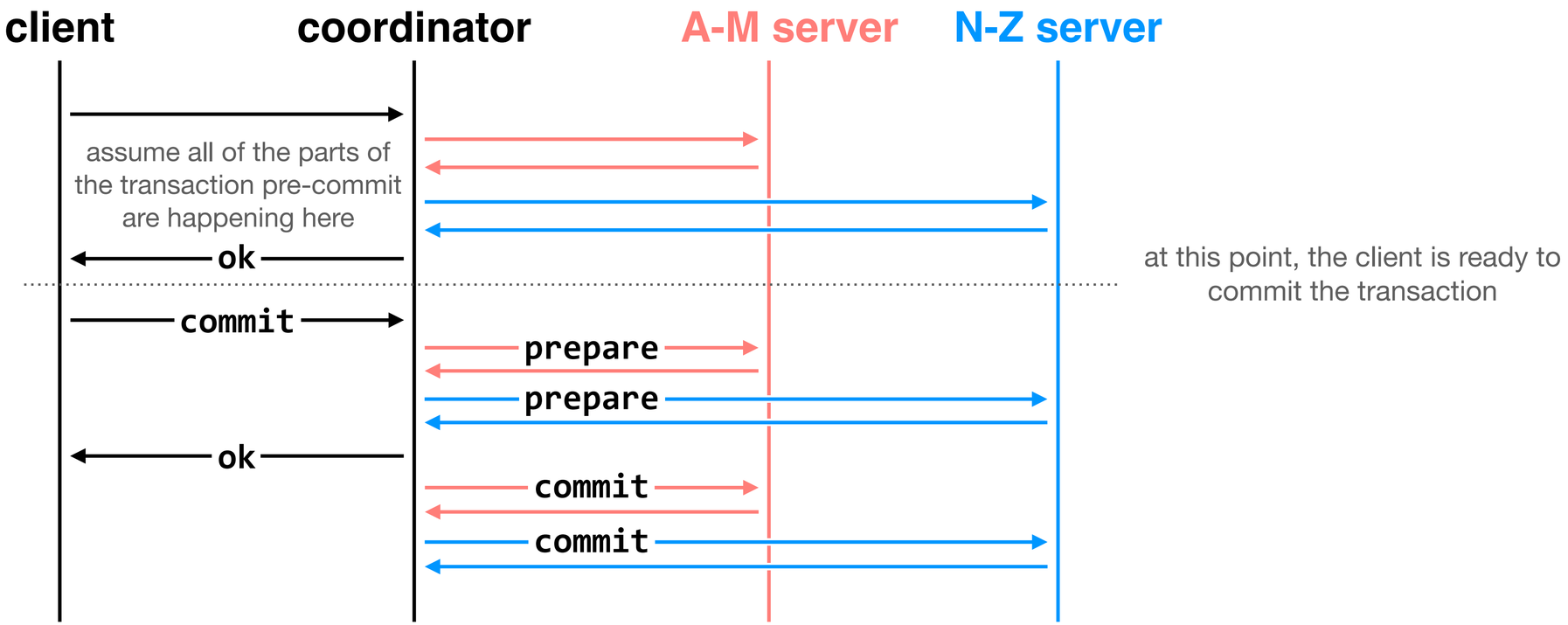
goal: develop a protocol that can provide **multi-site atomicity** in the face of all sorts of failures

(message loss, message reordering, worker failure, coordinator failure)

message failures solved with reliable transport protocol (sequence numbers + ACKs)

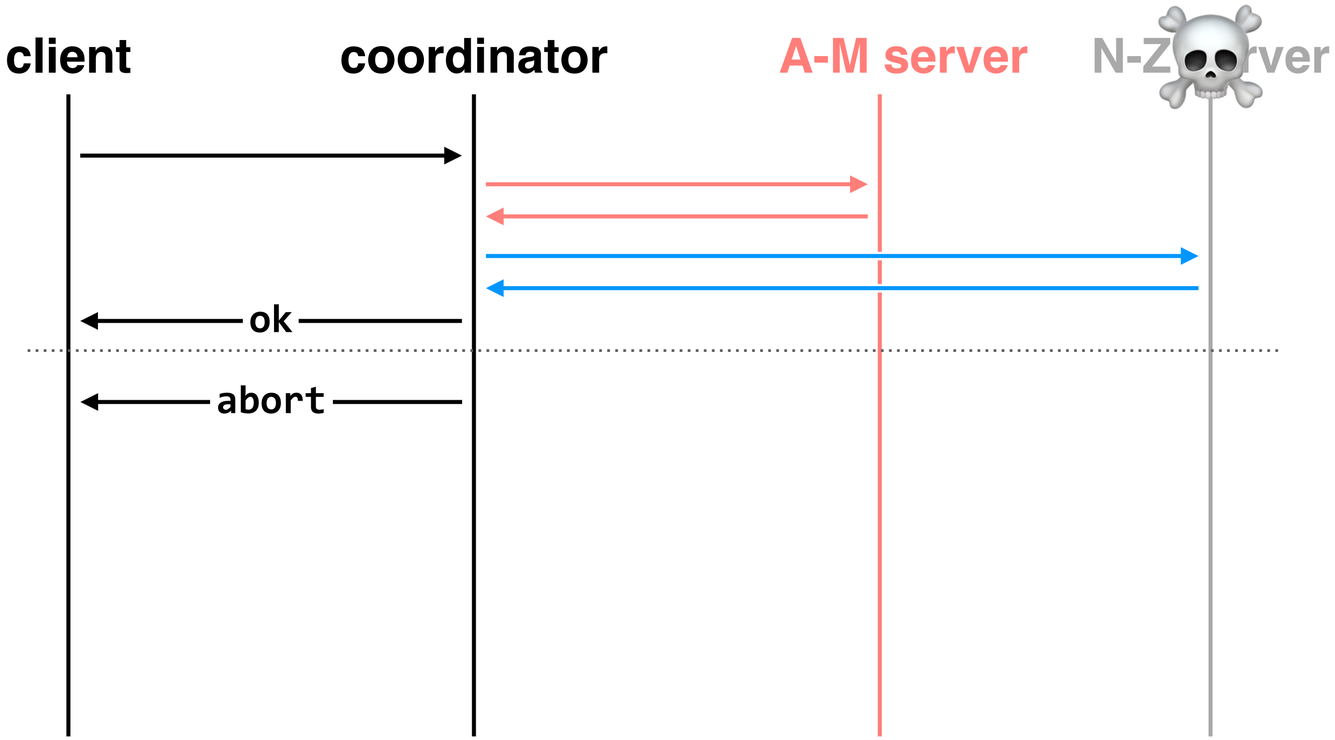
problem: one server committed, the other did not
(we'd have a similar problem if the N-Z server crashed)

two-phase commit: nodes agree that they're ready to commit before committing



to understand why this protocol provides atomicity — and specifically why we need two phases — we need to examine how it behaves under a variety of different types of failures

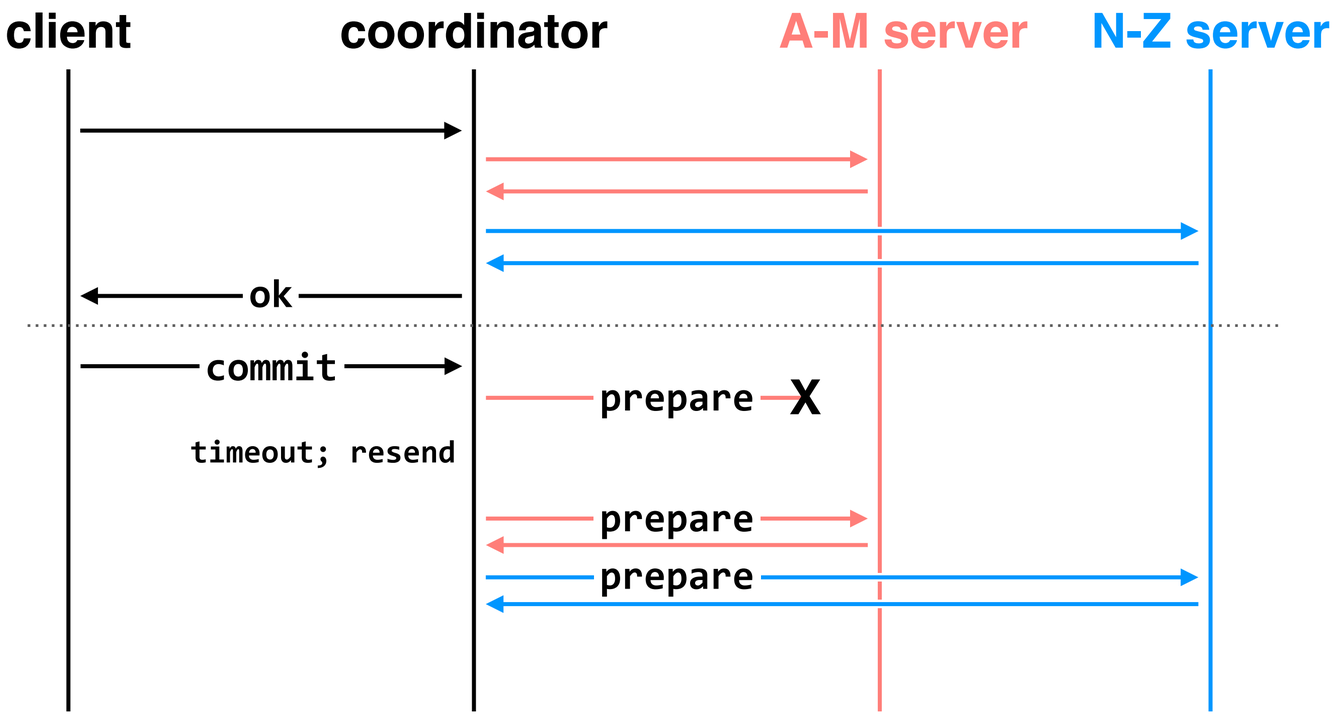
two-phase commit: nodes agree that they're ready to commit before committing



you can assume that the coordinator detects failures with a HELLO protocol, or something similar

worker failure before prepare phase:
coordinator can safely abort transaction without additional communication to workers

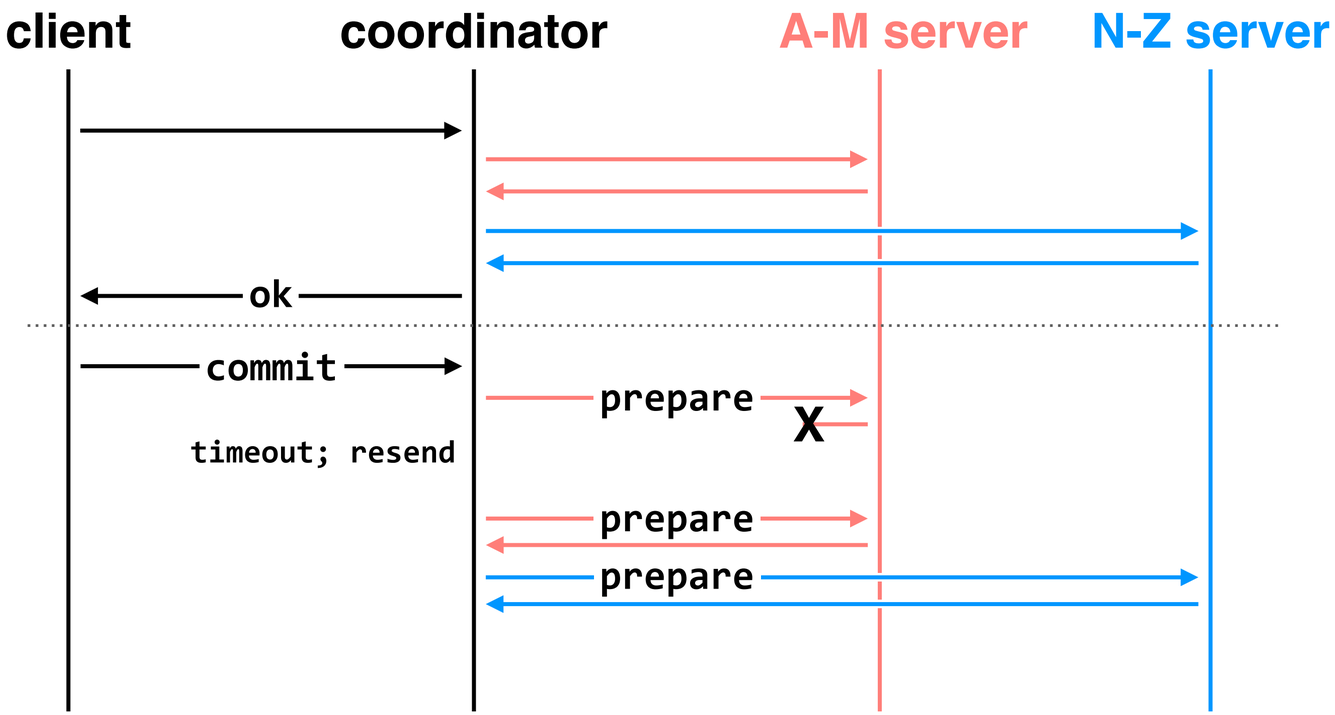
two-phase commit: nodes agree that they're ready to commit before committing



message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction without additional communication to workers

two-phase commit: nodes agree that they're ready to commit before committing

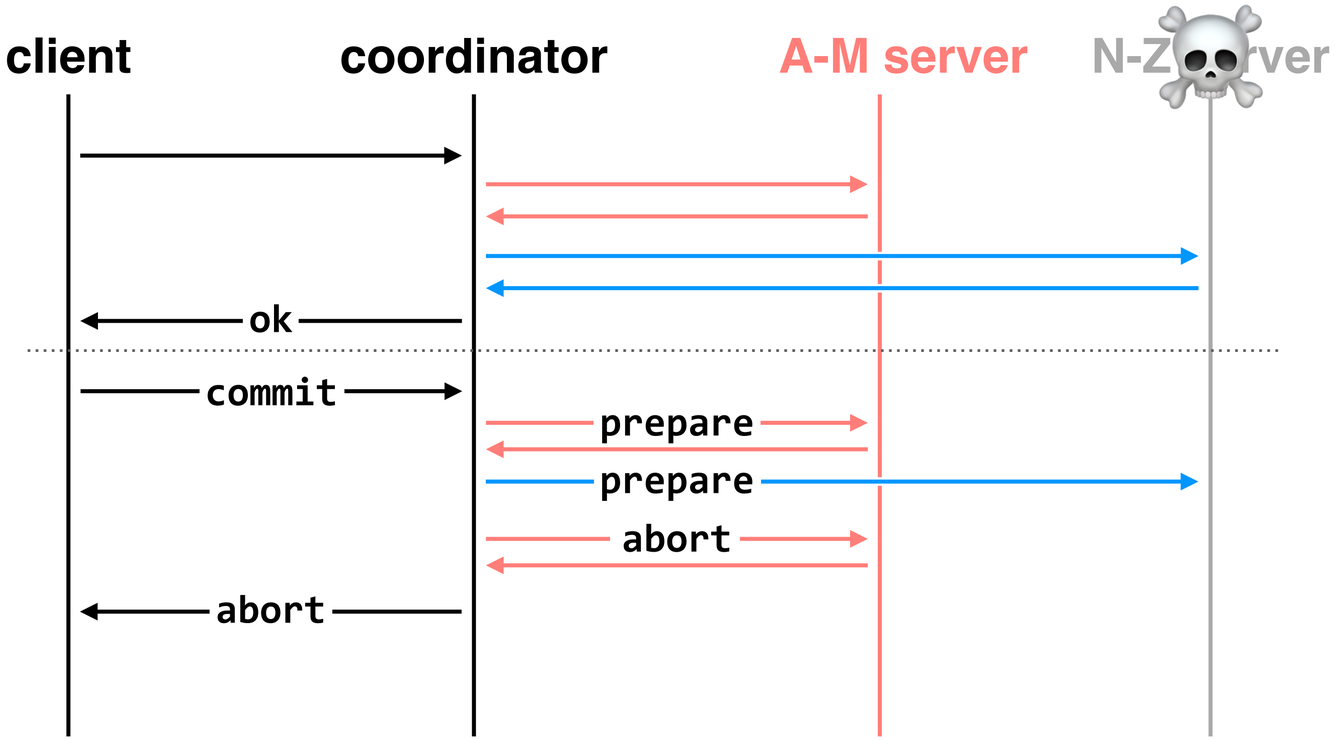


thanks to sequence numbers, A-M will ACK the second prepare message but not reprocess it

message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction without additional communication to workers

two-phase commit: nodes agree that they're ready to commit before committing

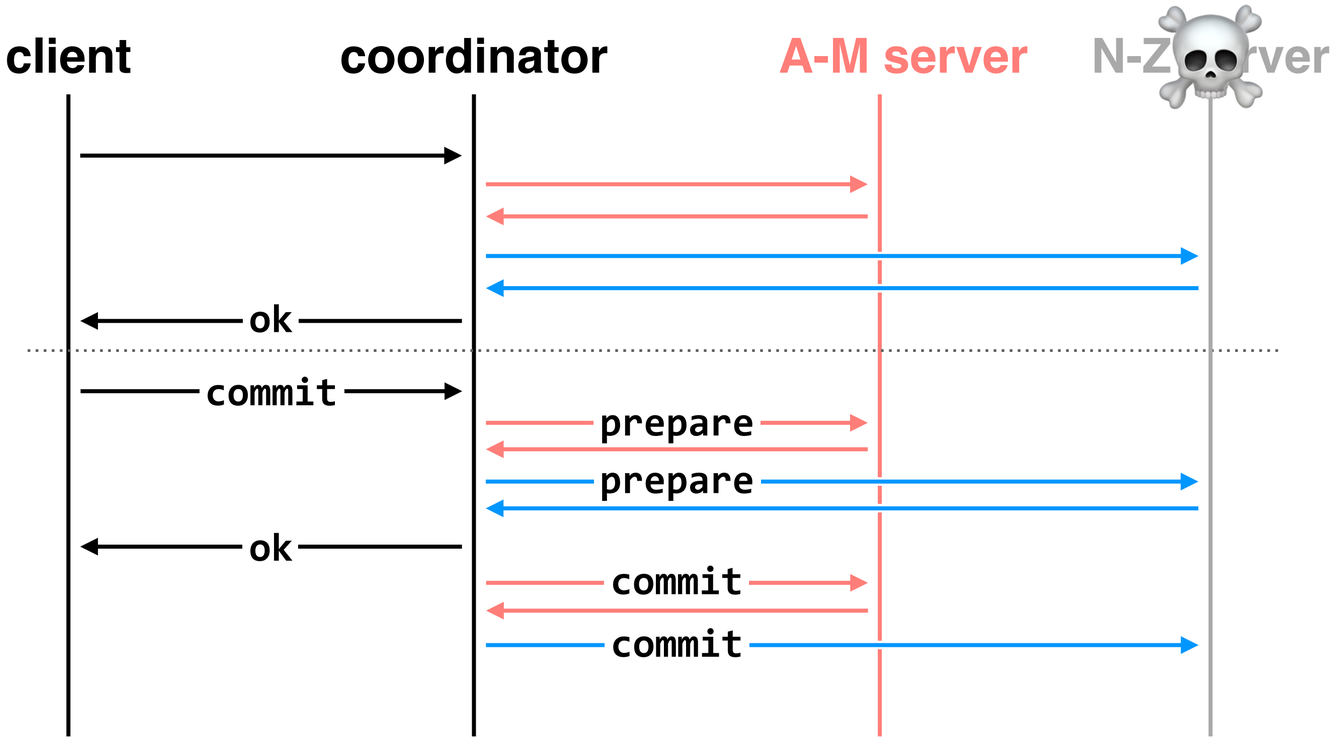


message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction without additional communication to workers

worker failure during prepare phase: coordinator can safely abort transaction, will send explicit abort messages to live workers

two-phase commit: nodes agree that they're ready to commit before committing



message loss at any stage: handled by reliable transport; coordinator will time out and resend message

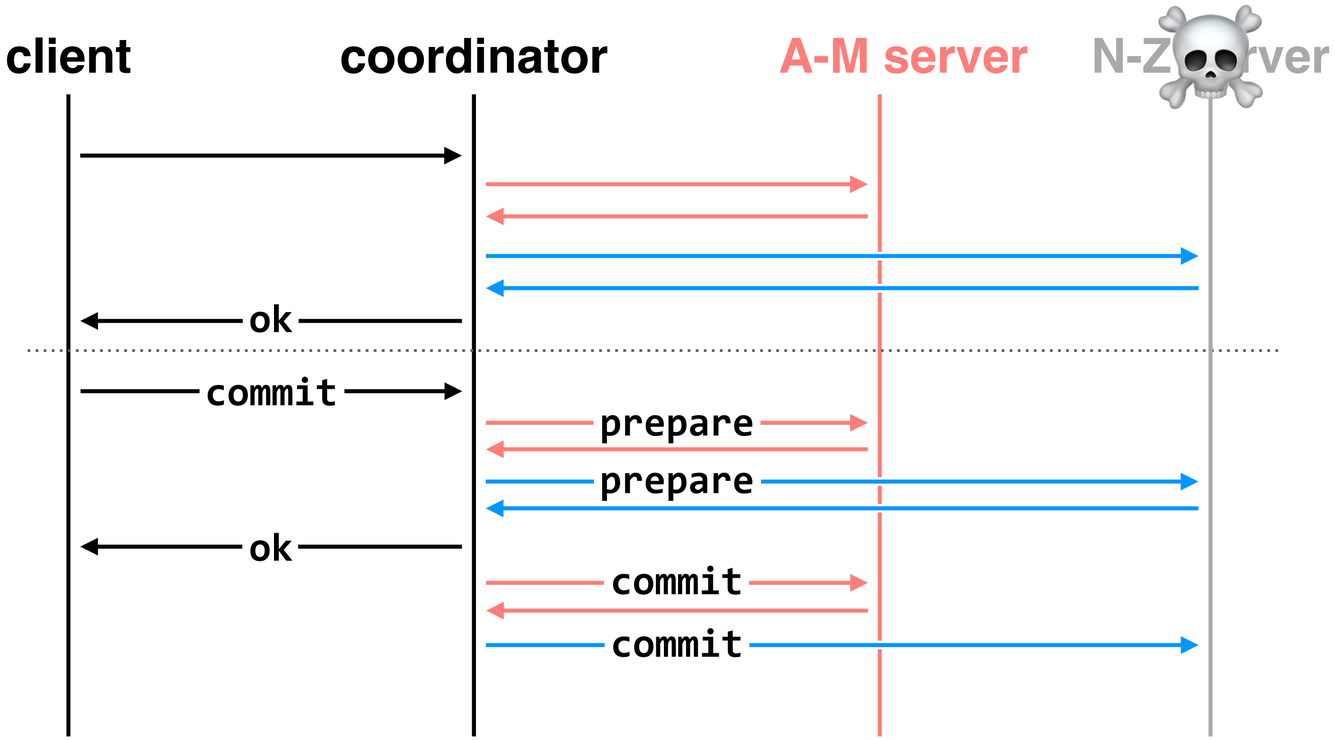
worker failure before prepare phase: coordinator can safely abort transaction without additional communication to workers

worker failure during prepare phase: coordinator can safely abort transaction, will send explicit abort messages to live workers

worker failure during commit phase

if workers fail after the commit point, we **cannot abort** the transaction. workers must be able to recover into a prepared state, and then commit

two-phase commit: nodes agree that they're ready to commit before committing



workers write **PREPARE** records once prepared. the recovery process — reading through the log — will indicate which transactions are prepared but not committed

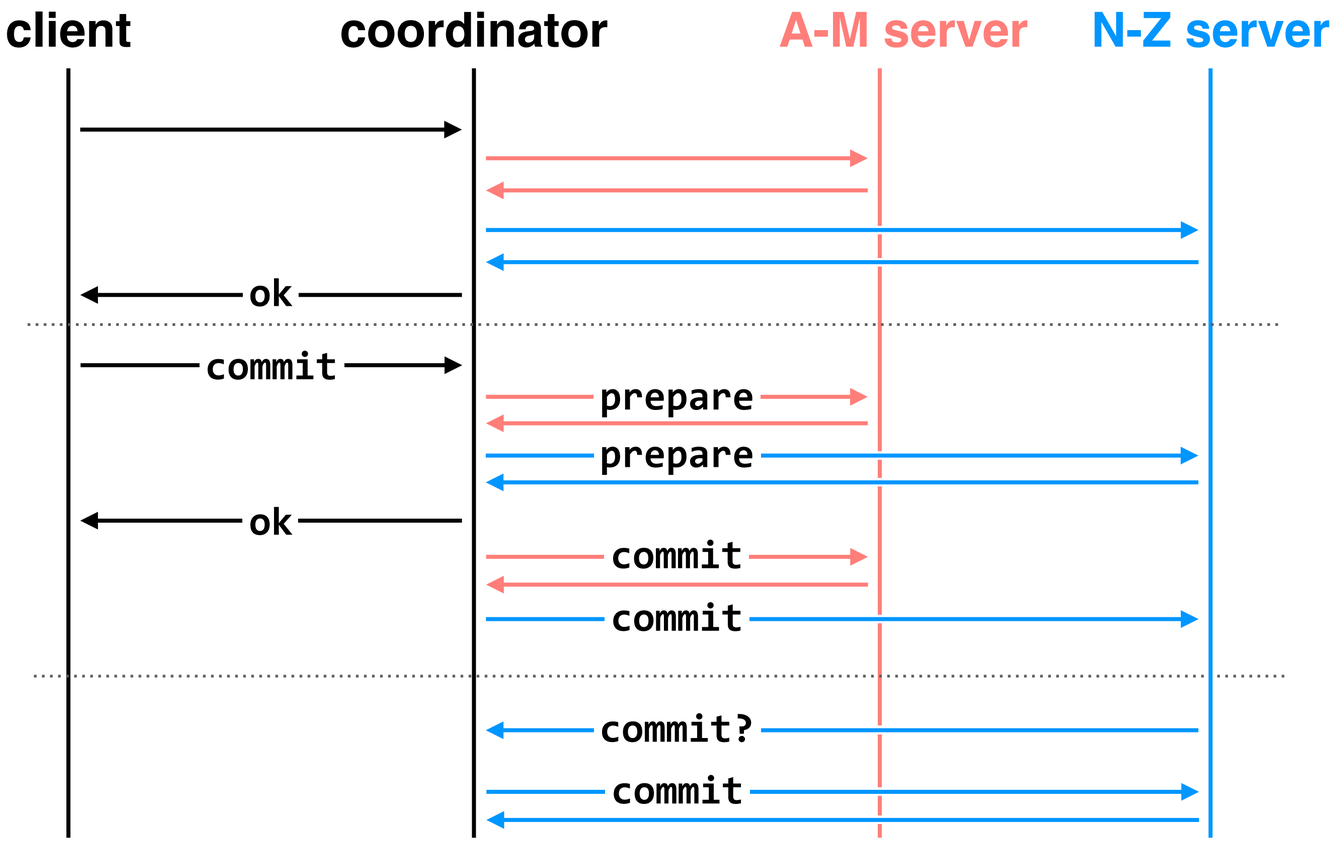
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction without additional communication to workers

worker failure during prepare phase: coordinator can safely abort transaction, will send explicit abort messages to live workers

worker failure during commit phase: coordinator *cannot* abort the transaction

two-phase commit: nodes agree that they're ready to commit before committing



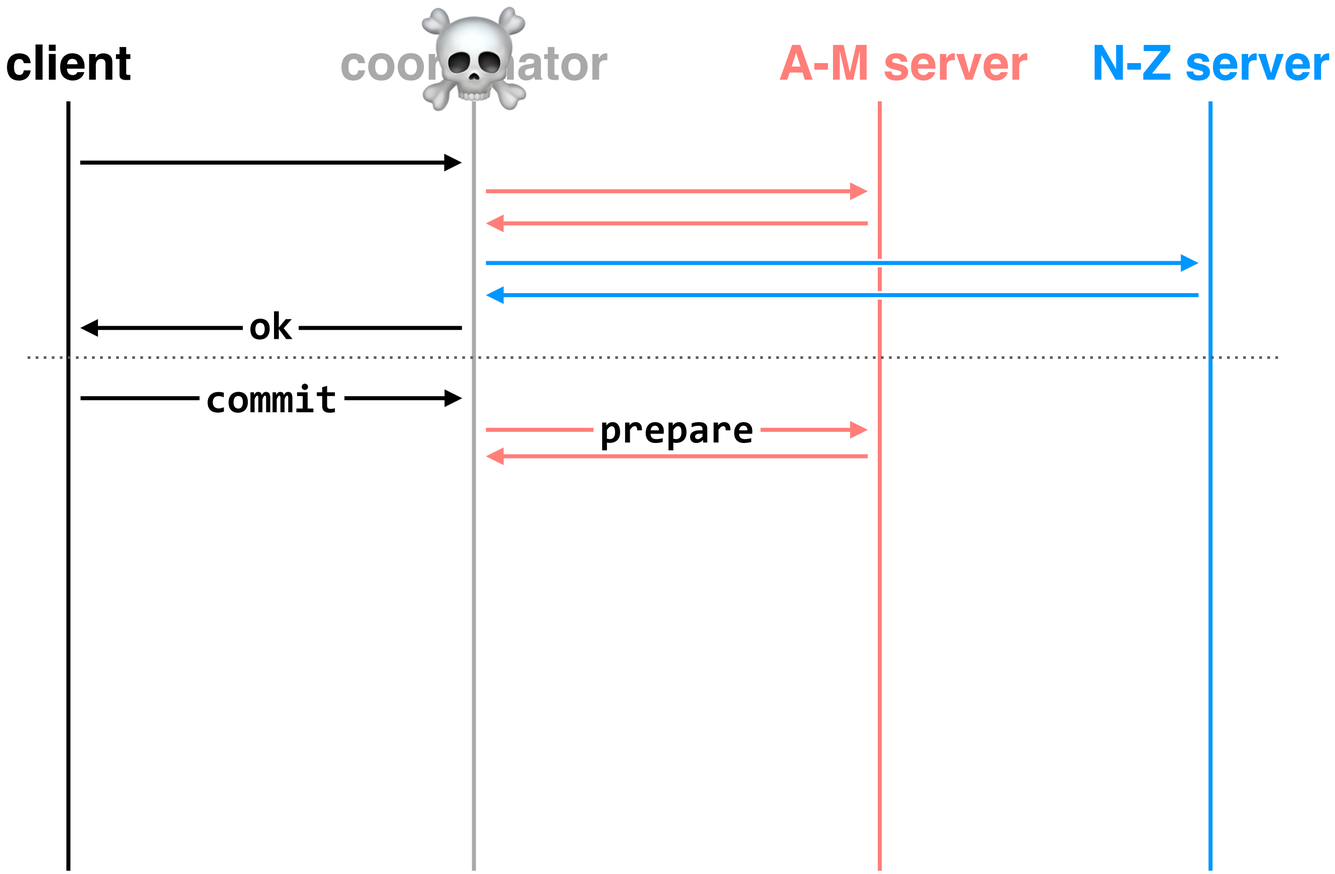
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction without additional communication to workers

worker failure during prepare phase: coordinator can safely abort transaction, will send explicit abort messages to live workers

worker failure during commit phase: coordinator *cannot* abort the transaction; prepared workers must commit the transaction during recovery

two-phase commit: nodes agree that they're ready to commit before committing



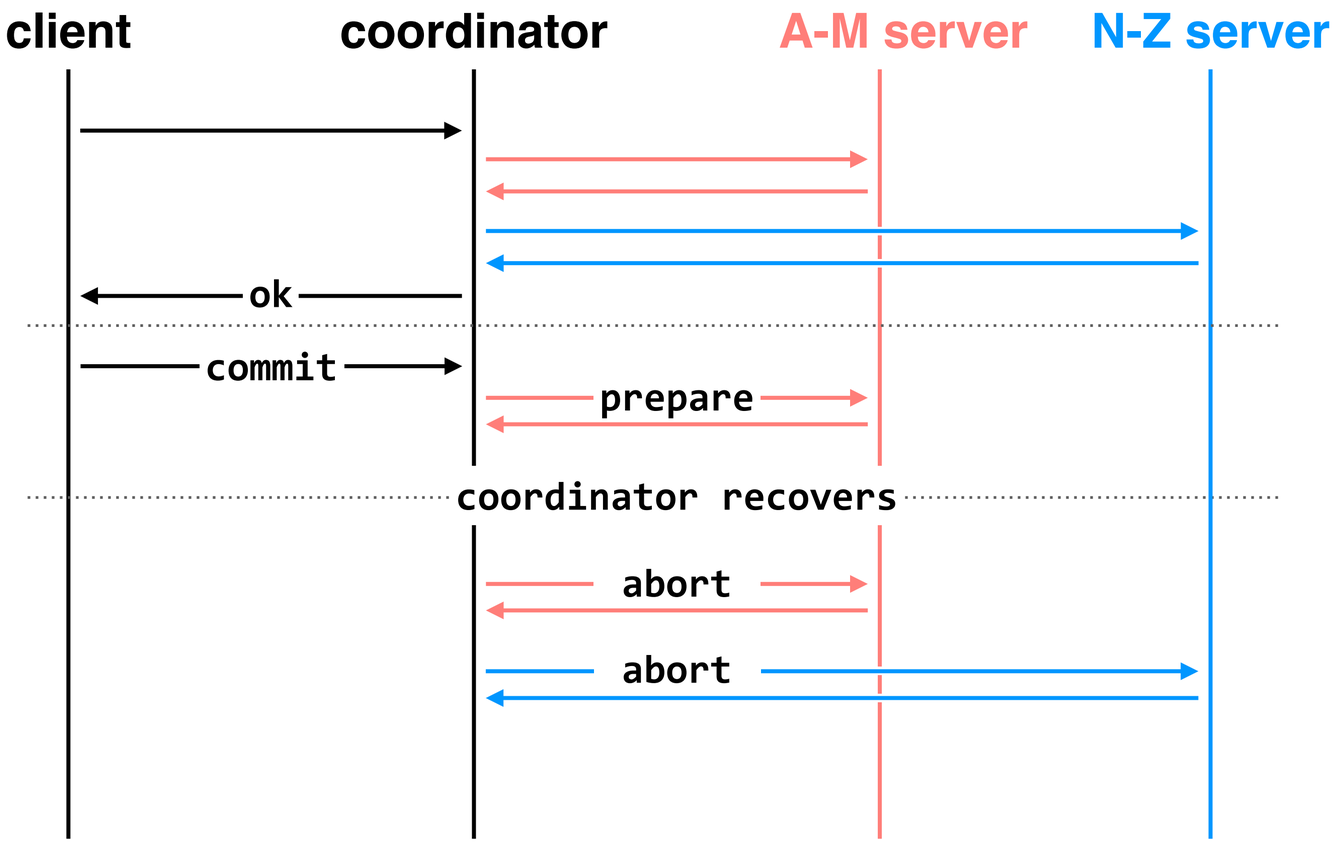
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction without additional communication to workers

worker failure during prepare phase: coordinator can safely abort transaction, will send explicit abort messages to live workers

worker failure during commit phase: coordinator *cannot* abort the transaction; prepared workers must commit the transaction during recovery

two-phase commit: nodes agree that they're ready to commit before committing



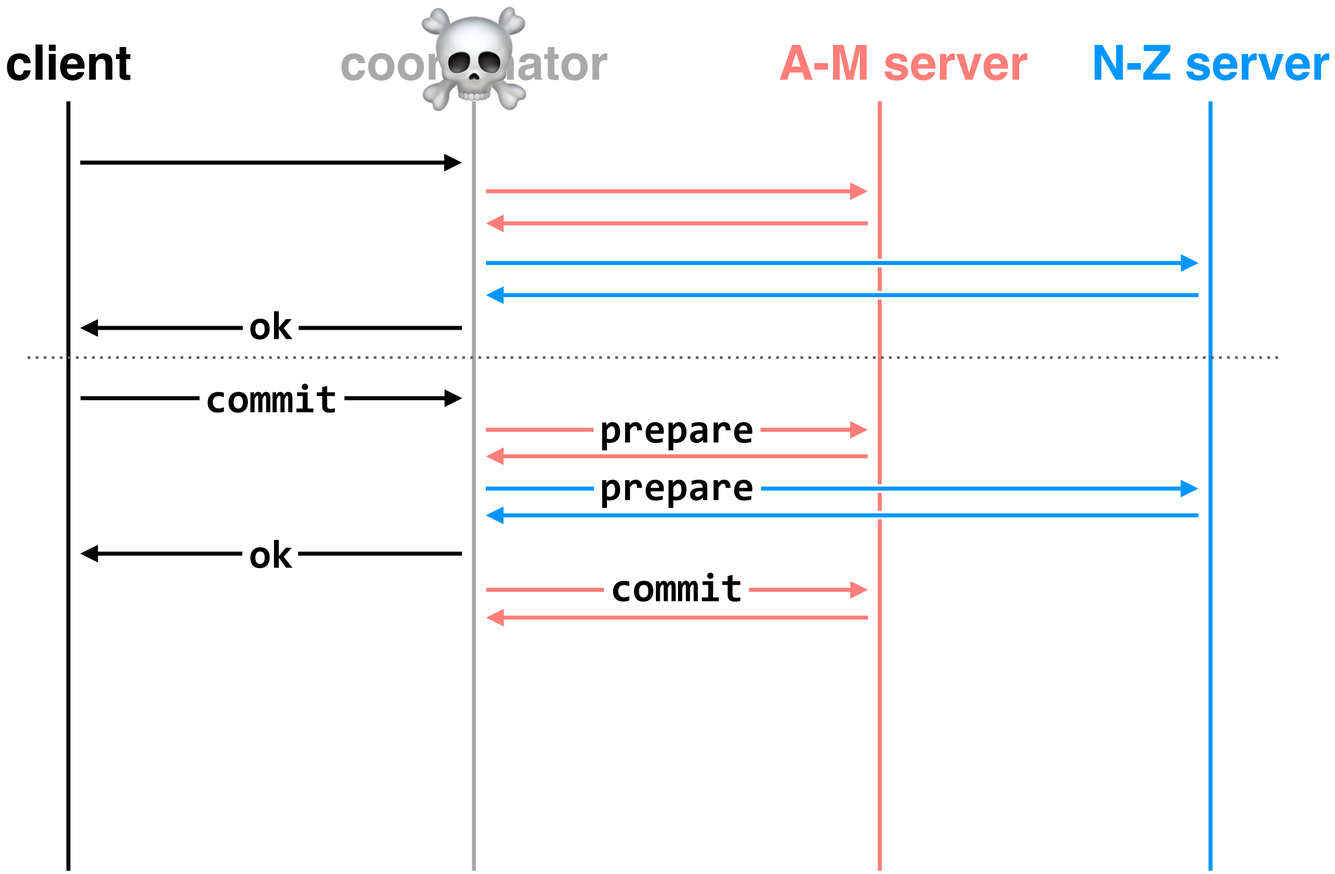
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction without additional communication to workers

worker failure or coordinator failure during prepare phase: coordinator can safely abort transaction, will send explicit abort messages to live workers

worker failure during commit phase: coordinator *cannot* abort the transaction; prepared workers must commit the transaction during recovery

two-phase commit: nodes agree that they're ready to commit before committing



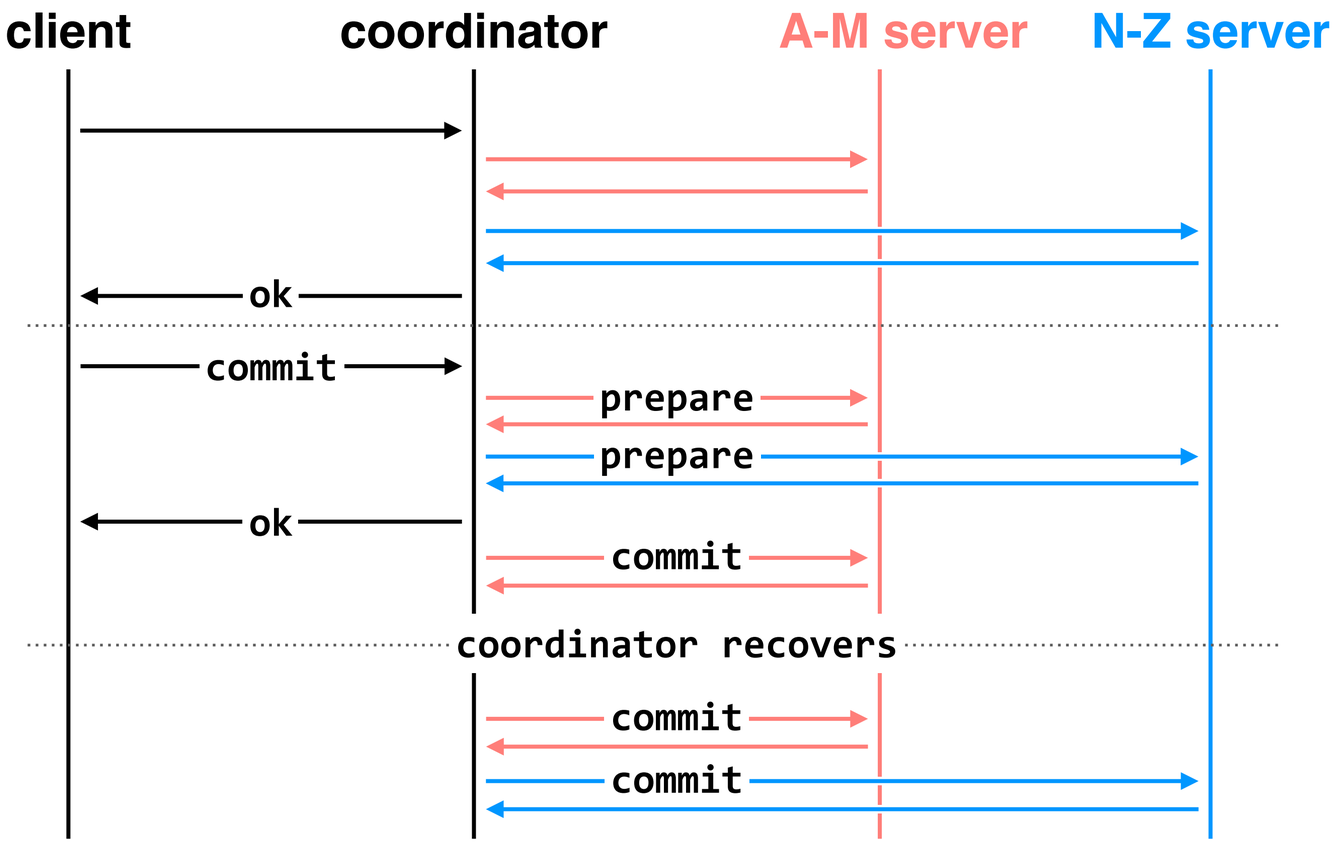
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction without additional communication to workers

worker failure or coordinator failure during prepare phase: coordinator can safely abort transaction, will send explicit abort messages to live workers

worker failure during commit phase: coordinator *cannot* abort the transaction; prepared workers must commit the transaction during recovery

two-phase commit: nodes agree that they're ready to commit before committing



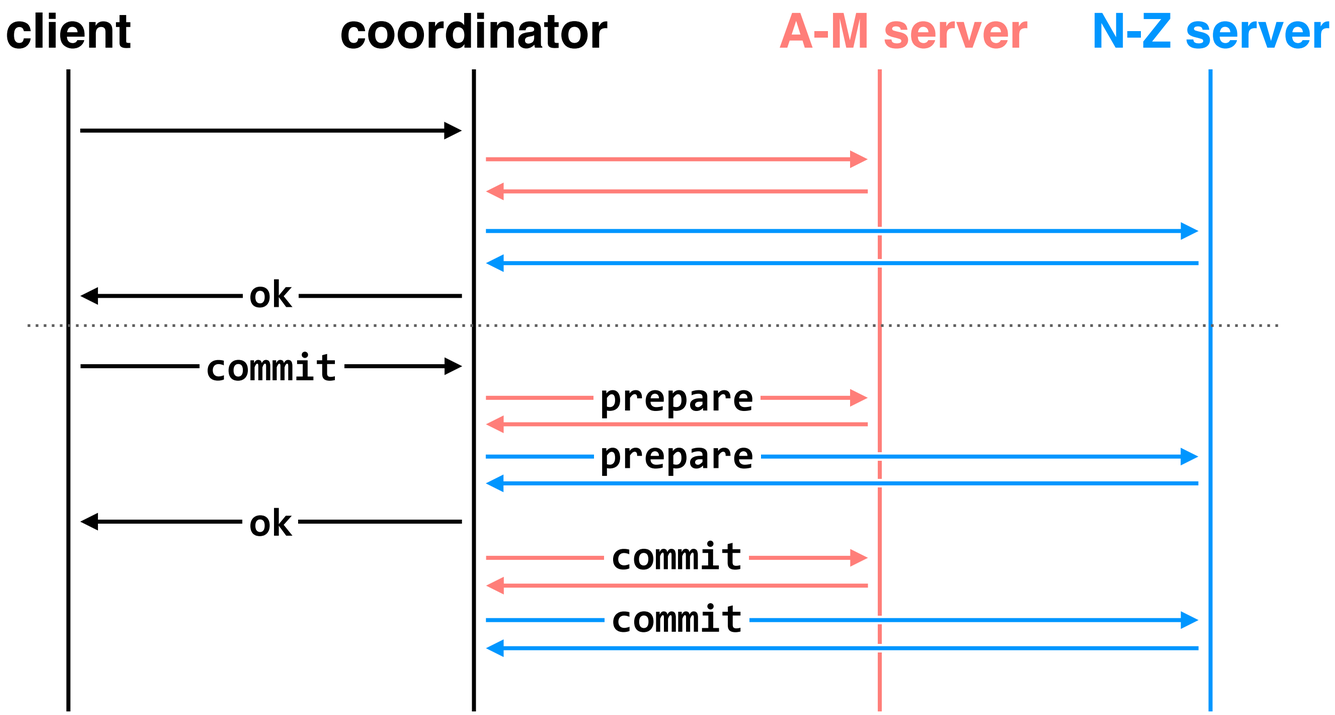
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction without additional communication to workers

worker failure or coordinator failure during prepare phase: coordinator can safely abort transaction, will send explicit abort messages to live workers

worker failure or coordinator failure during commit phase: coordinator *cannot* abort the transaction; machines must commit the transaction during recovery

two-phase commit: nodes agree that they're ready to commit before committing



problem: in our example, when workers fail, some of the data (e.g., accounts A-M) is completely unavailable

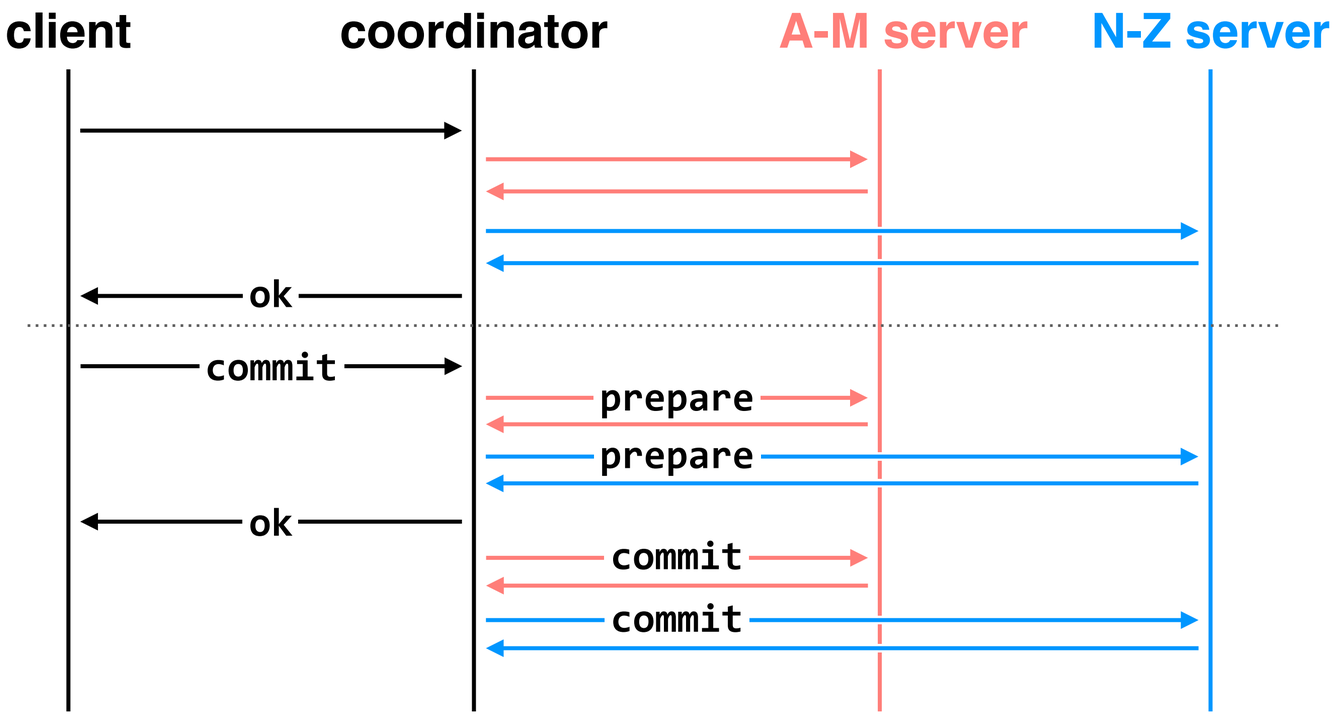
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction without additional communication to workers

worker failure or coordinator failure during prepare phase: coordinator can safely abort transaction, will send explicit abort messages to live workers

worker failure or coordinator failure during commit phase: coordinator *cannot* abort the transaction; machines must commit the transaction during recovery

two-phase commit: nodes agree that they're ready to commit before committing



message loss at any stage: handled by reliable transport; coordinator will time out and resend message

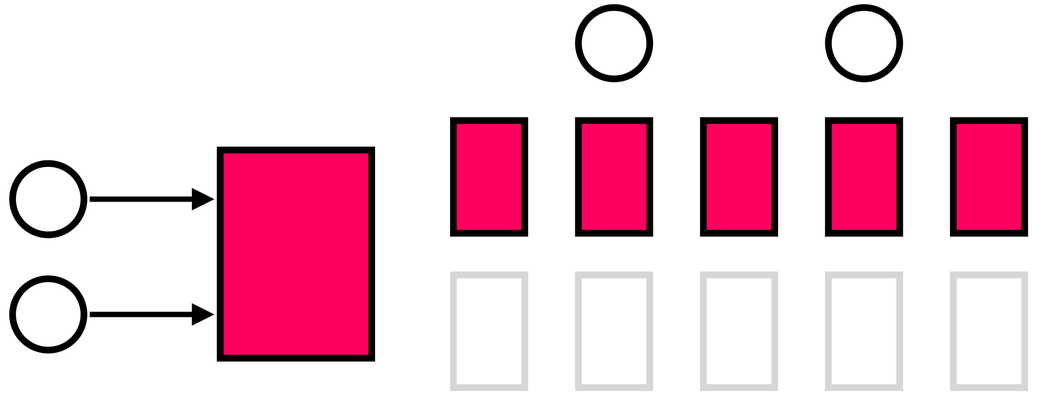
worker failure before prepare phase: coordinator can safely abort transaction without additional communication to workers

worker failure or coordinator failure during prepare phase: coordinator can safely abort transaction, will send explicit abort messages to live workers

worker failure or coordinator failure during commit phase: coordinator *cannot* abort the transaction; machines must commit the transaction during recovery

solution: replicate data. but to address this problem, we need to worry about keeping multiple copies of the same piece of data **consistent**, and what type of consistency we even want

our goal is to build **reliable systems from unreliable components**. we want to build systems that serve many clients, store a lot of data, perform well, all while keeping availability high



transactions — which provide **atomicity** and **isolation** — make it easier for us to reason about failures

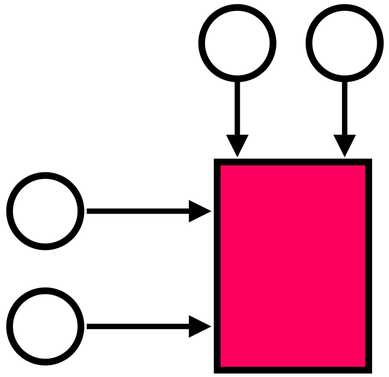
our job in lecture is to understand how a system *implements* these two abstractions.
how do our systems guarantee atomicity? how do they guarantee isolation?

atomicity: provided by **logging**, which gives better performance than shadow copies* at the cost of some added complexity

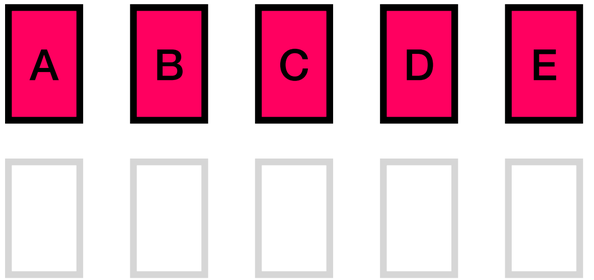
* shadow copies *are* used in some systems

isolation: provided by **two-phase locking**

our goal is to build **reliable systems from unreliable components**. we want to build systems that serve many clients, store a lot of data, perform well, all while keeping availability high



our lingering problem is that we aren't replicating data across multiple machines



transactions — which provide **atomicity** and **isolation** — make it easier for us to reason about failures

our job in lecture is to understand how a system *implements* these two abstractions.
how do our systems guarantee atomicity? how do they guarantee isolation?

atomicity: provided by **logging**, which gives better performance than shadow copies* at the cost of some added complexity; **two-phase commit** gives us multi-site atomicity

isolation: provided by **two-phase locking**

* shadow copies *are* used in some systems

two-phase commit allows us to achieve **multi-site atomicity**; transactions remain atomic even when they require communication with multiple machines.

two-phase commit is often abbreviated 2PC. two-phase locking (last week's topic) is often abbreviated 2PL. they are not the same!

in two-phase commit, failures prior to the commit point can be aborted. failures after the commit point cannot; machines must commit the transaction in recovery

our remaining issue deals with availability and replication: we will replicate data across sites to improve availability, but must deal with keeping multiple copies of the data **consistent**.