6.1800 Spring 2025

Lecture #19: Distributed Transactions

getting atomicity across machines

Katrina LaCurts | lacurts@mit.edu | 6.1800 2025

6.1800 in the news

By Carl Zimmer

April 9, 2025
Leer en español

The human brain is so complex that scientific brains have a hard time making sense of it. A piece of neural tissue the size of a grain of sand might be packed with hundreds of thousands of cells linked together by miles of wiring. In 1979, Francis Crick, the Nobel-prize-winning scientist, concluded that the anatomy and activity in just a cubic millimeter of brain matter would forever exceed our understanding.

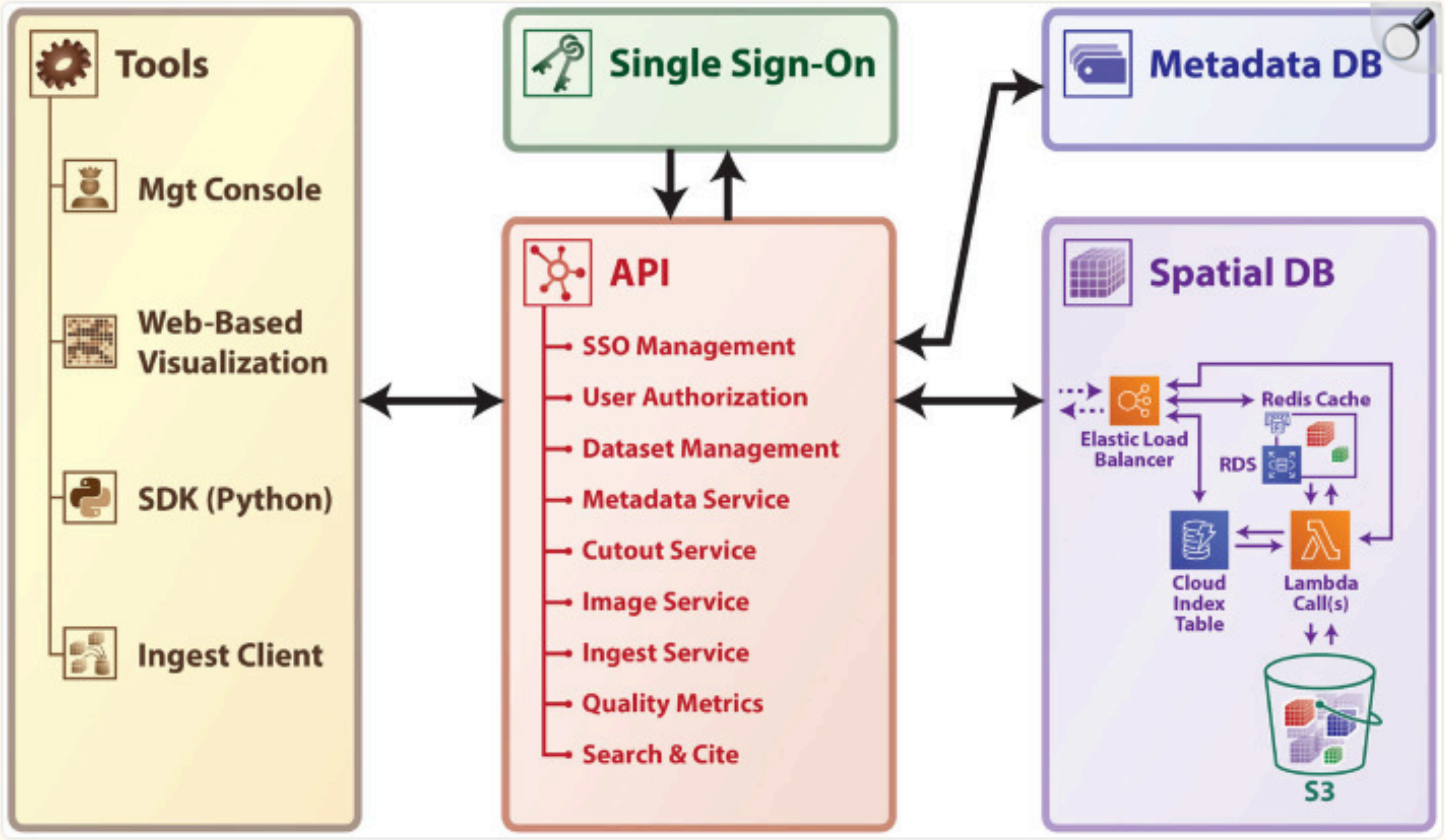
“It is no use asking for the impossible,” Dr. Crick wrote.

Forty-six years later, a team of more than 100 scientists has achieved that impossible, by recording the cellular activity and mapping the structure in a cubic millimeter of a mouse’s brain — less than one percent of its full volume. In accomplishing this feat, they amassed 1.6 petabytes of data — the equivalent of 22 years of nonstop high-definition video.

https://www.nytimes.com/2025/04/09/science/neuroscience-brain-mice-map.html

6.1800 in the news

Figure 1.



Open in a new tab

A high-level schematic of BossDB platform.

The Brain Observatory Storage Service and Database (BossDB): A Cloud-Native Approach for Petascale Neuroscience Discovery

Robert Hider Jr [1,†], Dean Kleissas [1,†], Timothy Gion [1], Daniel Xenes [1], Jordan Matelsky [1], Derek Pryor [1], Luis Rodriguez [1], Erik C Johnson [1], William Gray-Roncal [1,†], Brock Wester [1,*,†]

https://pmc.ncbi.nlm.nih.gov/articles/PMC8885591/

6.1800 in the news

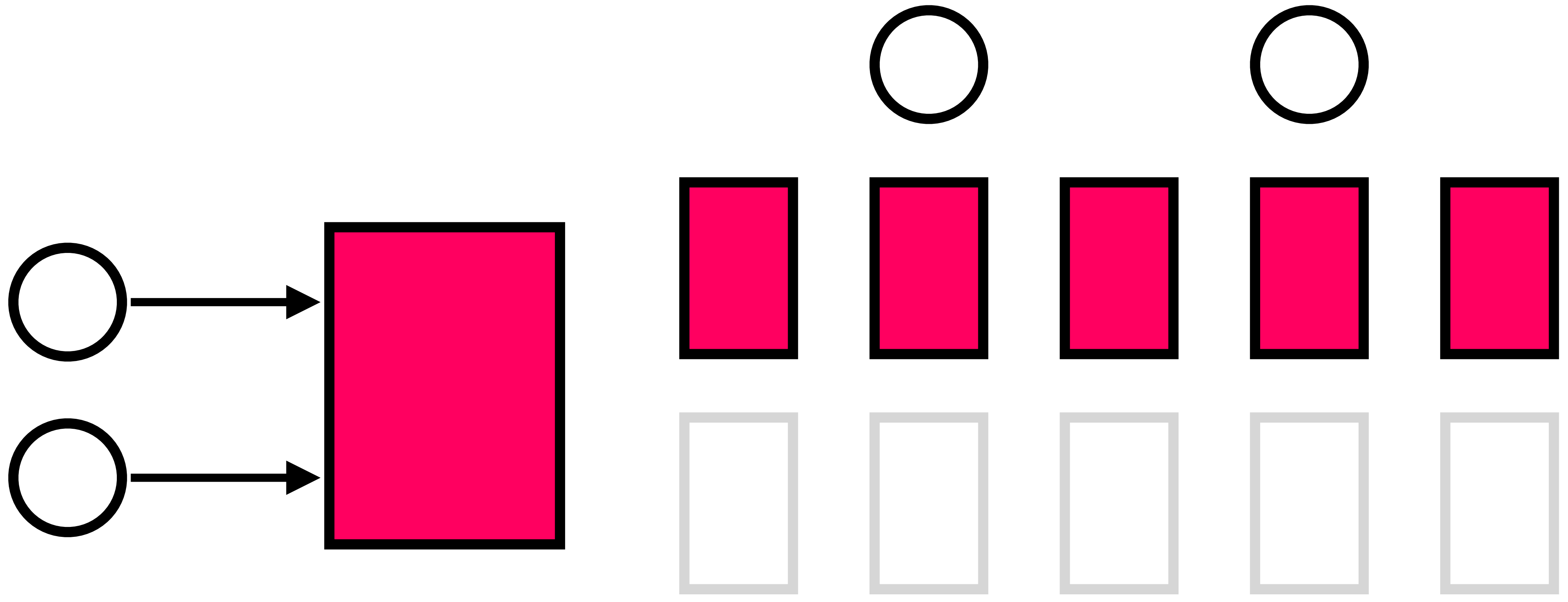
The Brain Observatory Storage Service and Database (BossDB): A Cloud-Native Approach for Petascale Neuroscience Discovery

Robert Hider Jr[1,†], Dean Kleissas[1,†], Timothy Gion[1], Daniel Xenes[1], Jordan Matelsky[1], Derek Pryor[1], Luis Rodriguez[1], Erik C Johnson[1], William Gray-Roncal[1,†], Brock Wester[1,*,†]

The spatial database is the foundation of BossDB, and uses the strengths of the cloud to efficiently store and index massive multi-dimensional image and annotation datasets (i.e., multi-channel 3D image volumes). A core concept is our managed storage hierarchy, which automatically migrates data between affordable, durable object storage (i.e., Amazon Simple Storage Service or S3) and an in-memory data store (i.e., Redis), which operates as read and write cache database for faster IO performance with a tradeoff of higher cost. The BossDB cache manages a lookup index to determine the fastest way to return data to the user, taking advantage of data stored in the hierarchy. While this requires the use of provisioned (non-serverless) resources, this allows for storage of large volumes at a low cost, while providing low latency to commonly accessed regions. We utilize AWS Lambda to perform parallel IO operations between the object store layer and memory cache layer and DynamoDB for indexing. These serverless technologies allow BossDB to rapidly and automatically scale resources during periods of heavy operation without incurring additional costs while idle.

https://pmc.ncbi.nlm.nih.gov/articles/PMC8885591/

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

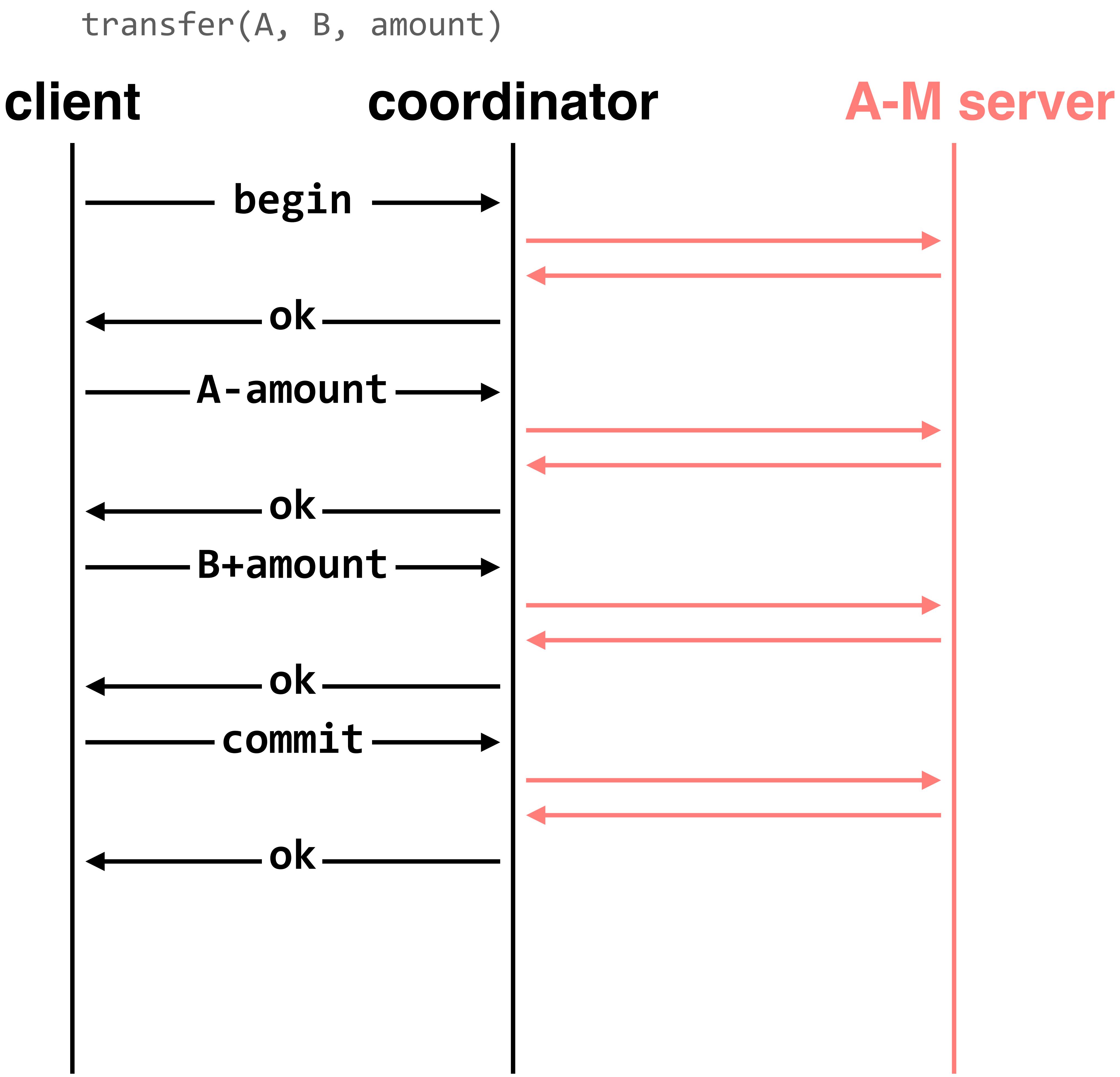
our job in lecture is to understand how a system *implements* these two abstractions. how do our systems guarantee atomicity? how do they guarantee isolation?

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

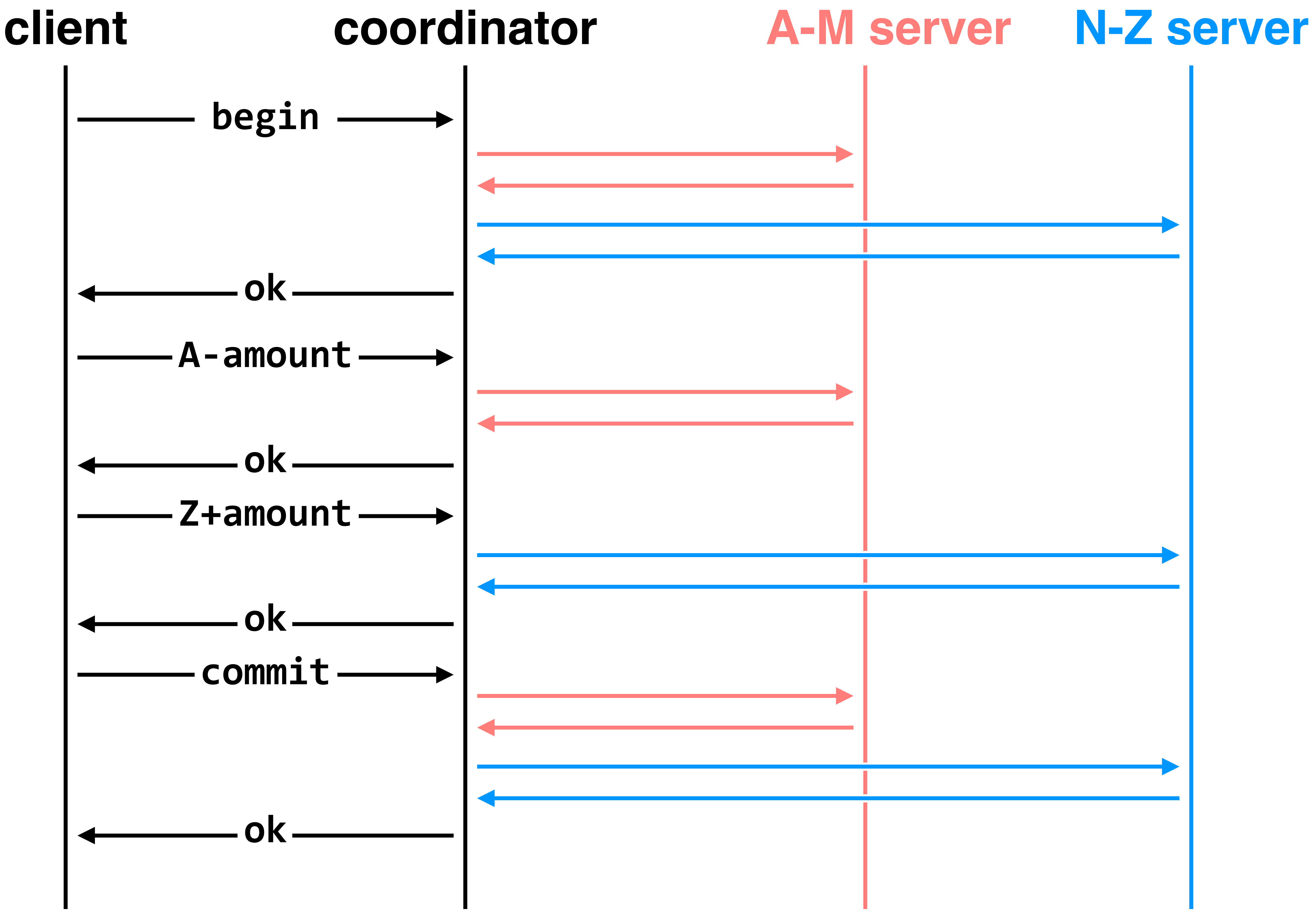
isolation: provided by **two-phase locking**

transactions across multiple machines (no failures yet)

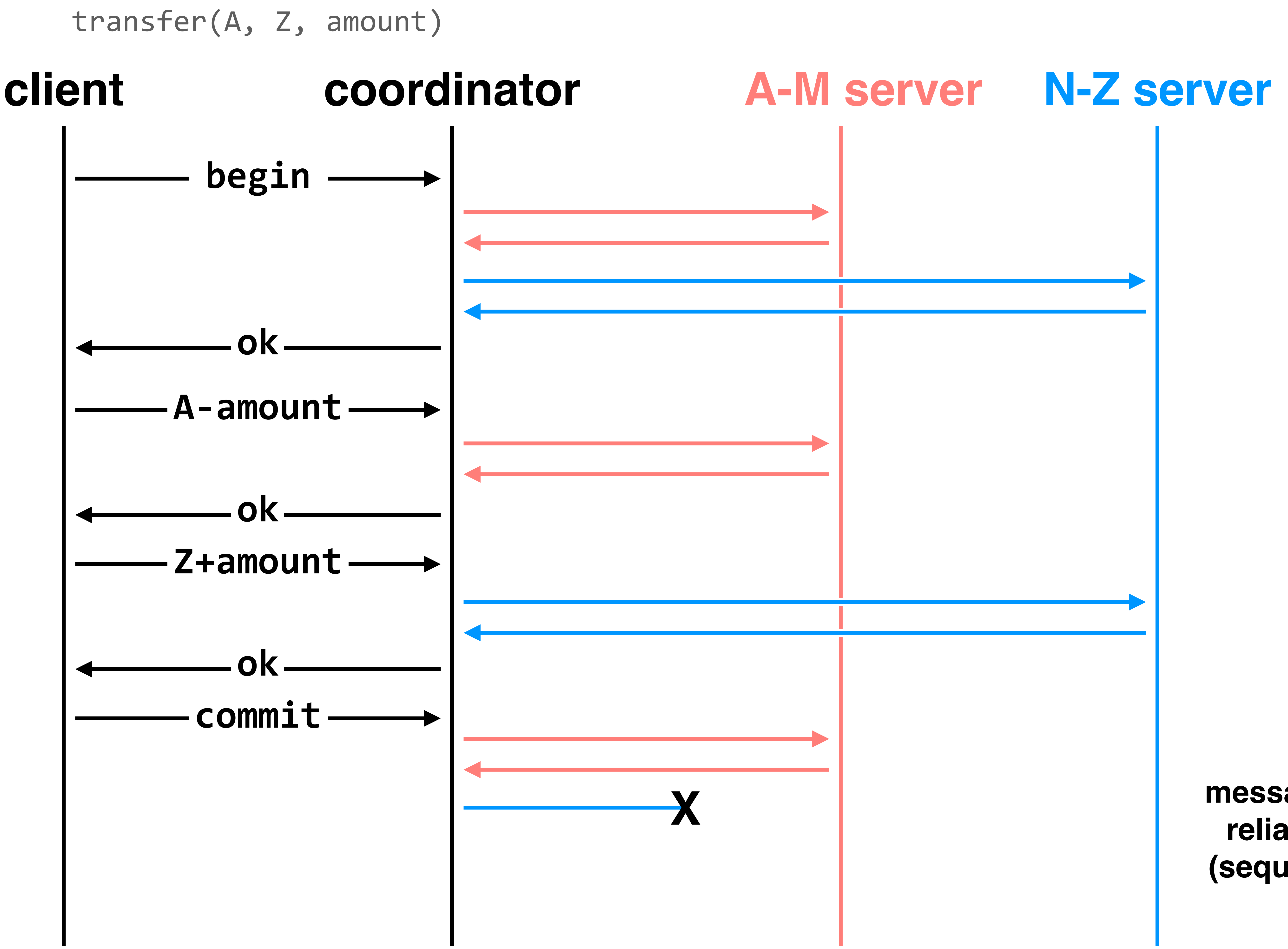


transactions across multiple machines (no failures yet)

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



transactions across multiple machines (now with failures)



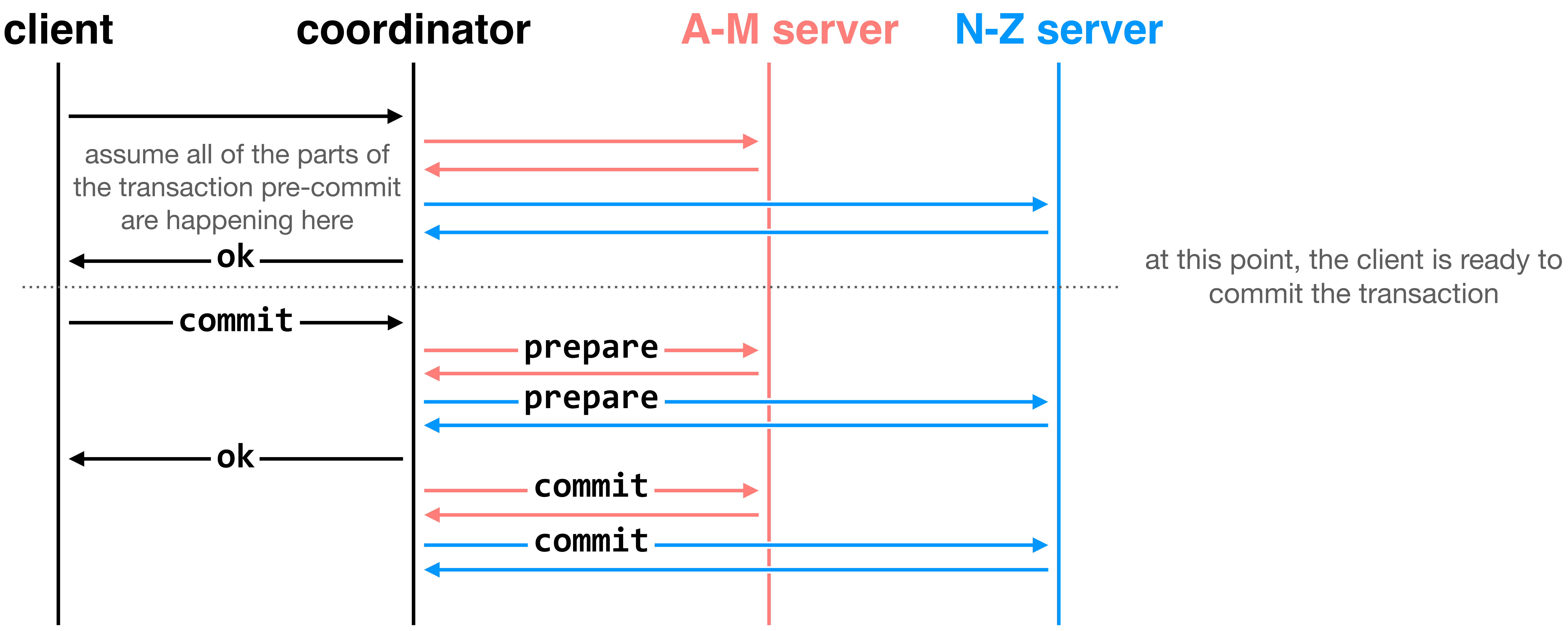
goal: develop a protocol that can provide **multi-site atomicity** in the face of all sorts of failures

(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)

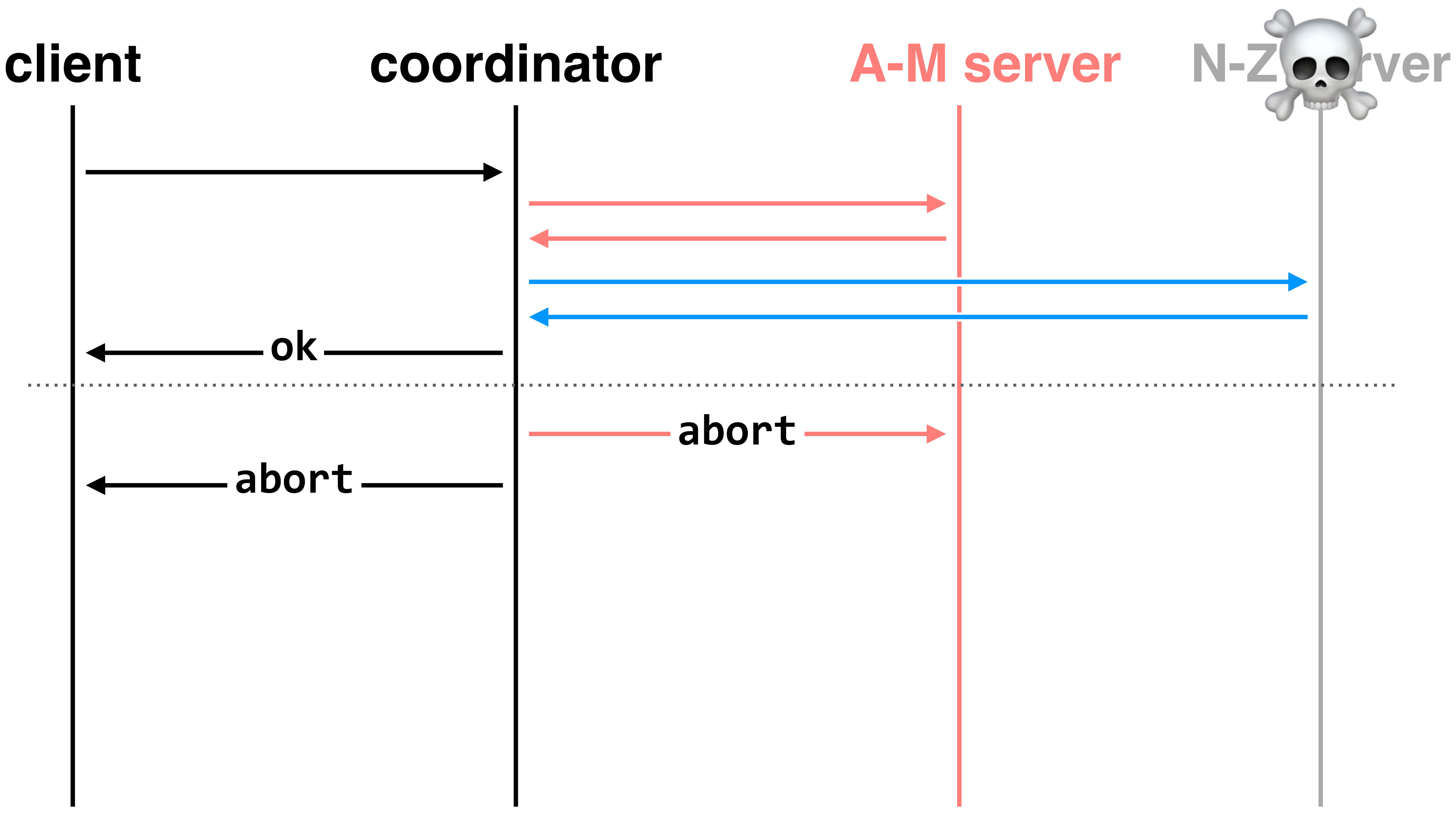
two-phase commit: nodes agree that they're ready to commit before committing



to understand why this protocol provides atomicity, we'll start by examining how it behaves under a variety of different types of failures

we will eventually understand why it requires two phases

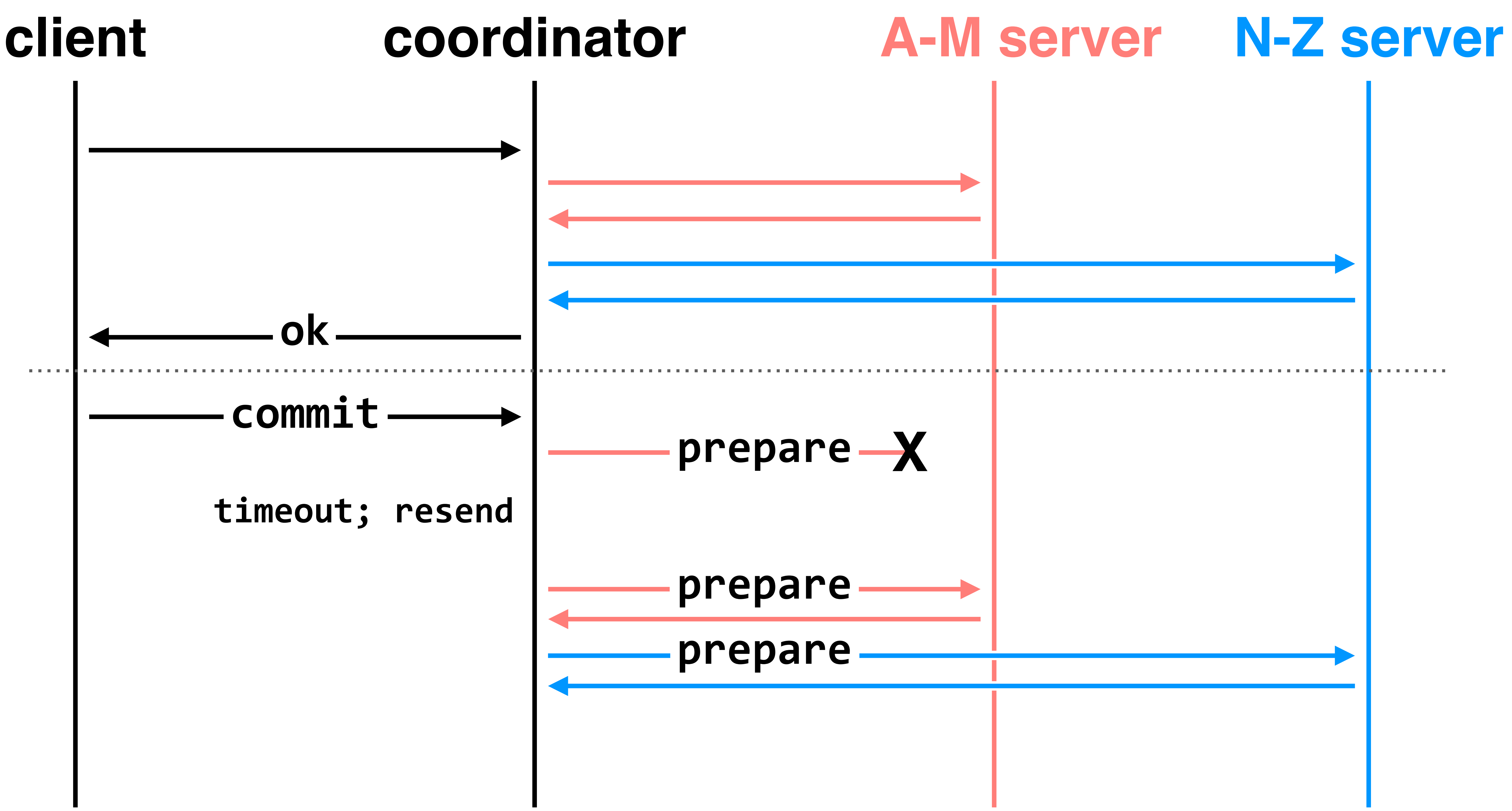
two-phase commit: nodes agree that they're ready to commit before committing



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

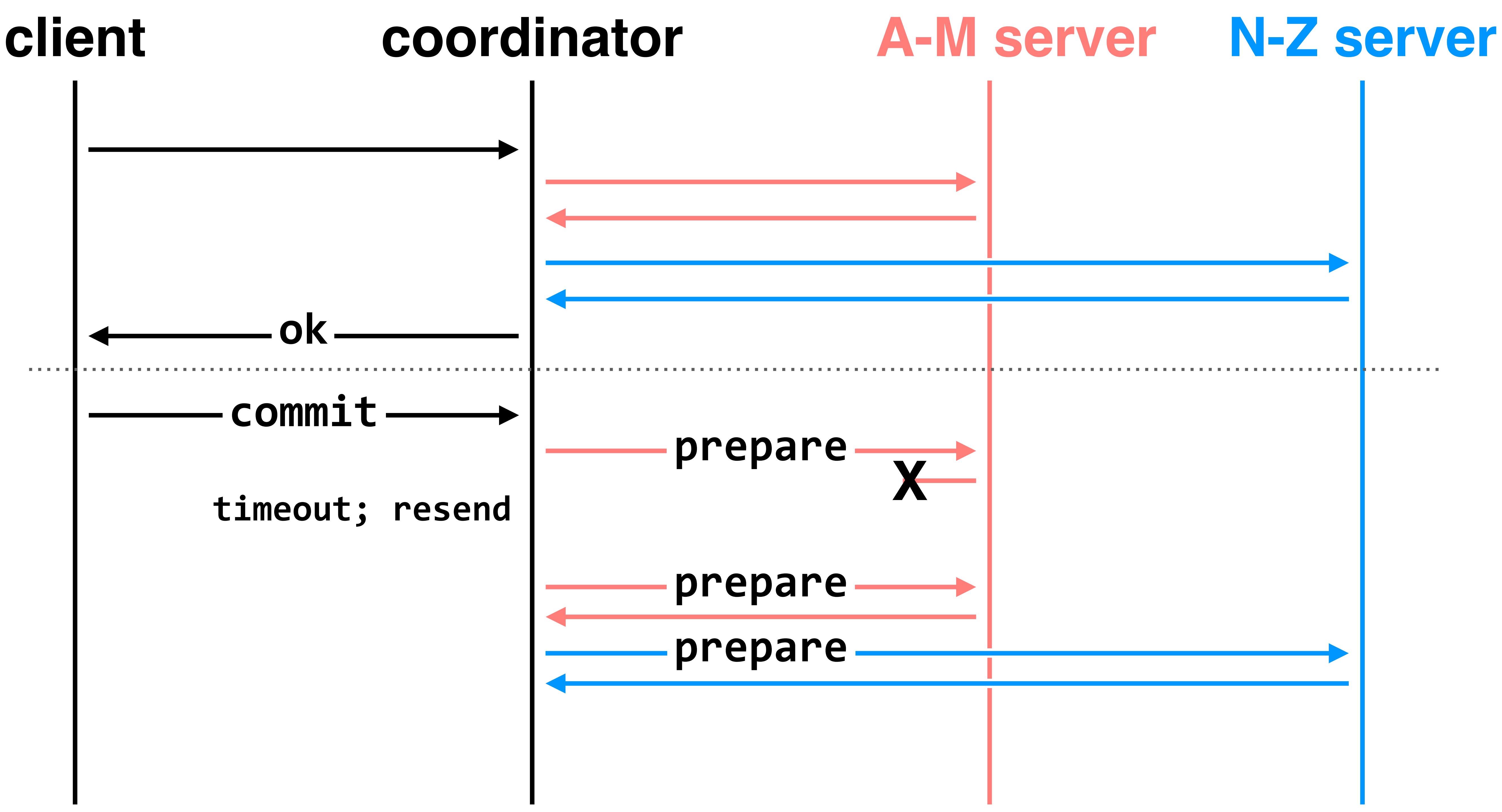
two-phase commit: nodes agree that they're ready to commit before committing



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

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

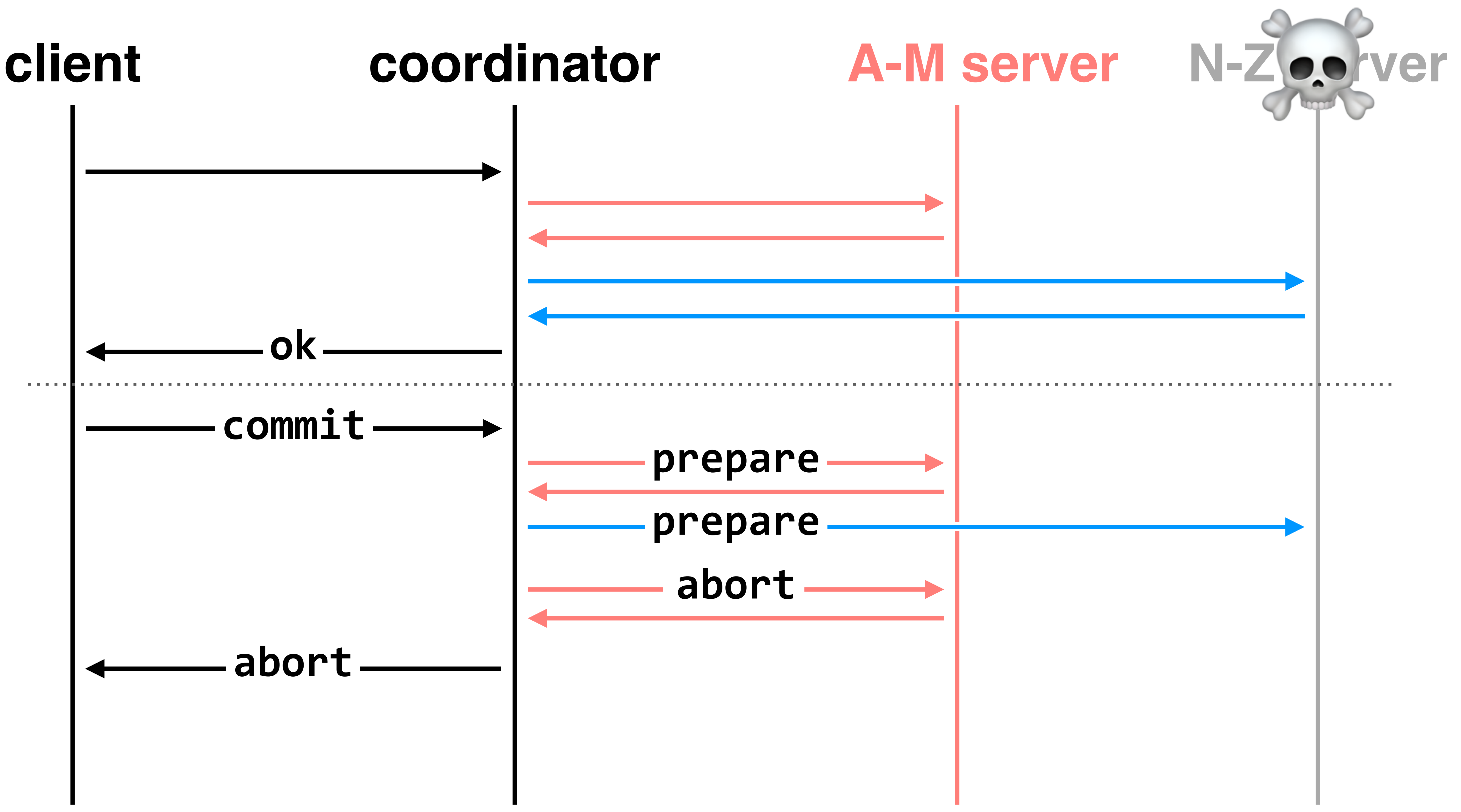


thanks to sequence numbers, A-M will ACK the second prepare message but not reprocess it

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

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

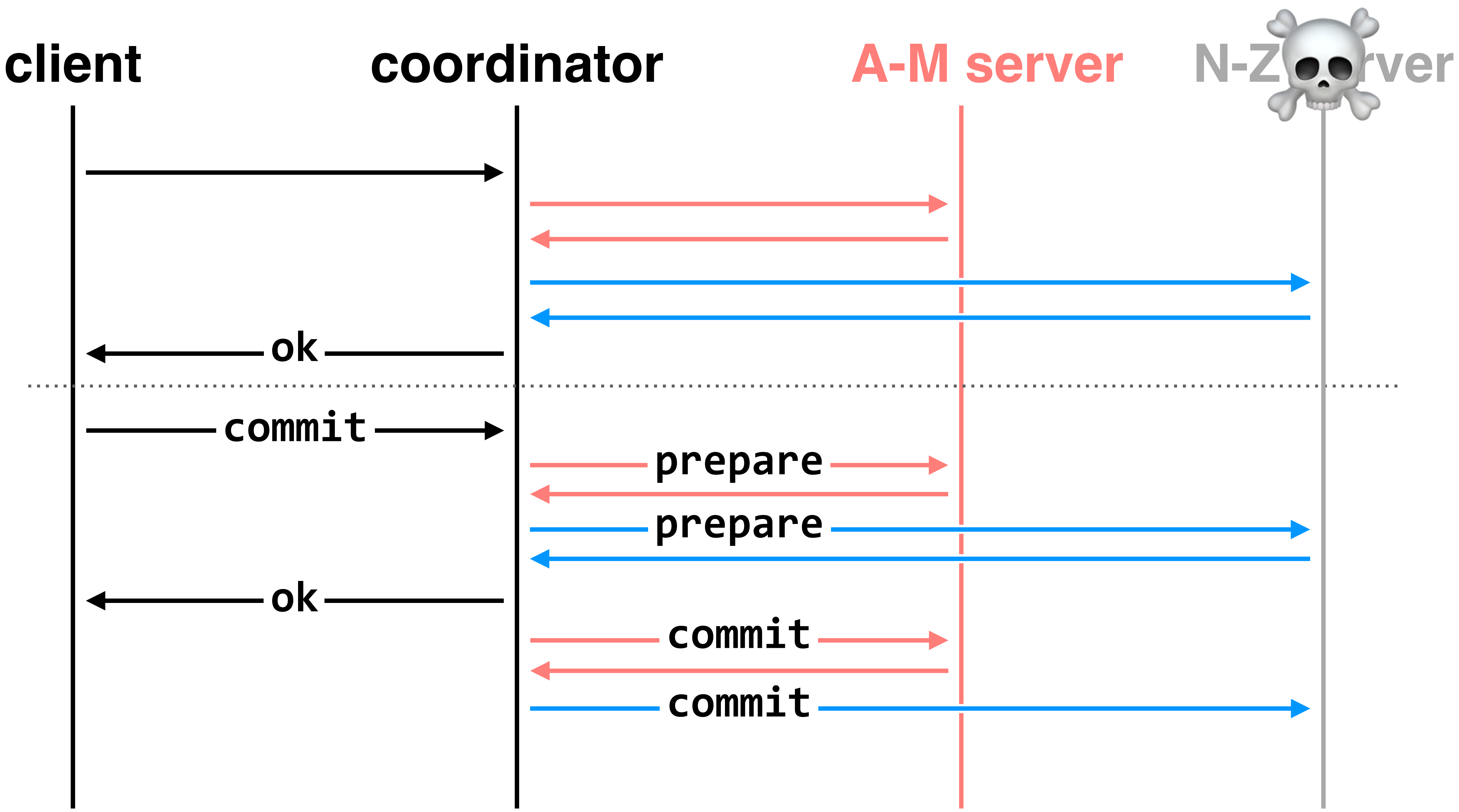


message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction

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

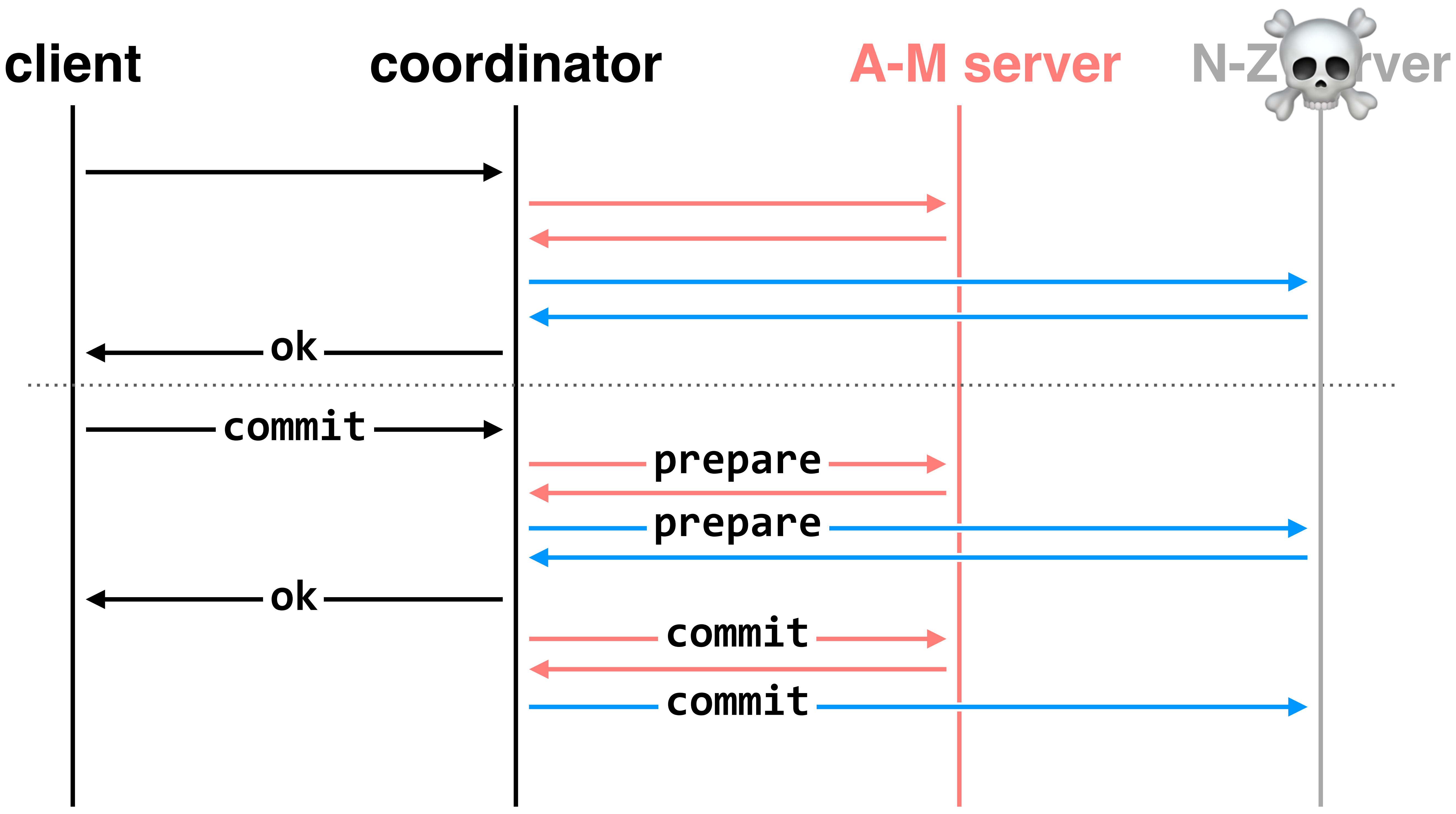
worker failure before prepare phase: coordinator can safely abort transaction

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



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

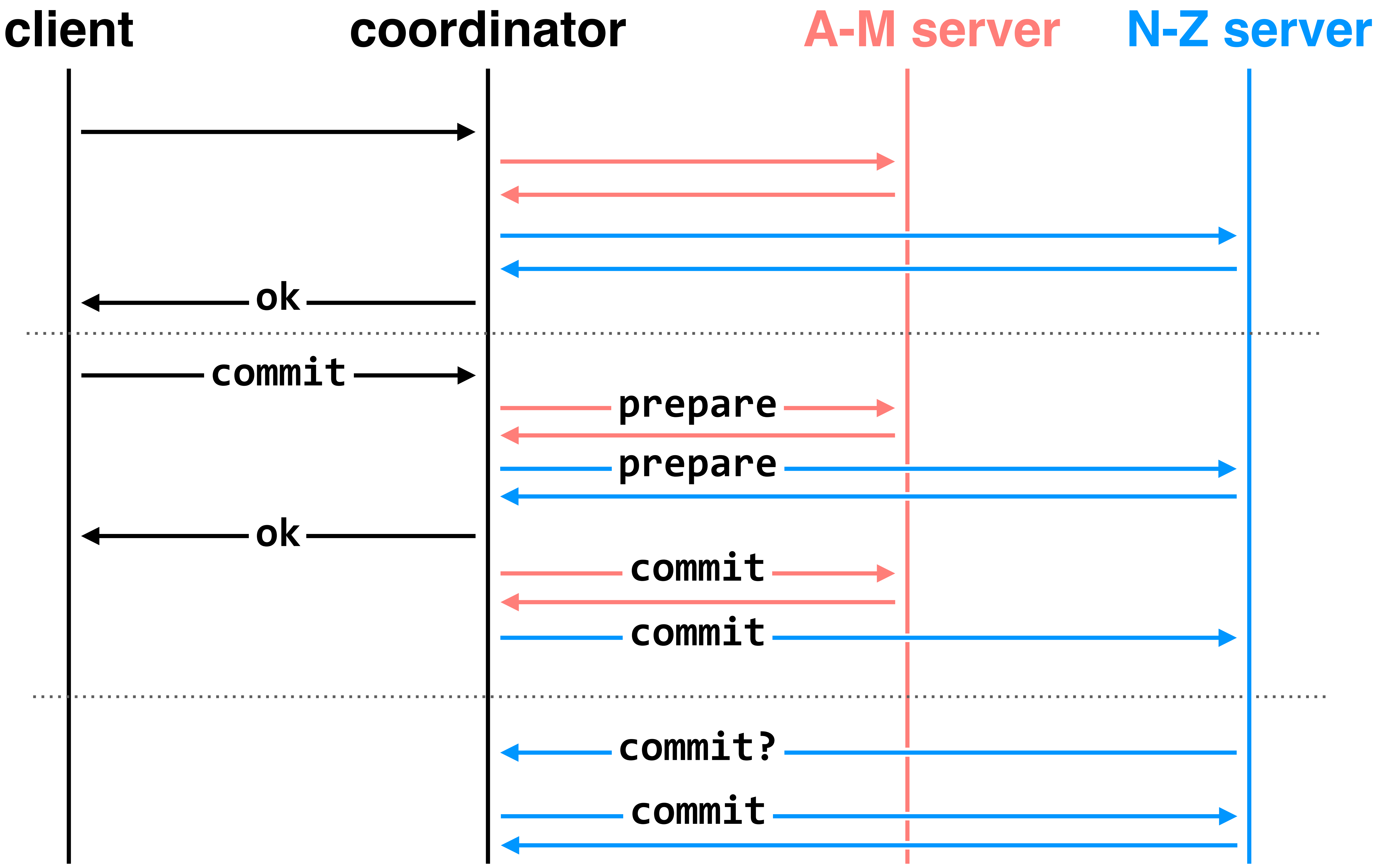
worker failure before prepare phase: coordinator can safely abort transaction

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

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

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



question: why does the **N-Z server** need to ask the coordinator whether it's okay to commit this transaction (i.e., why can't it just automatically commit after recovering and seeing the **PREPARE** record)?

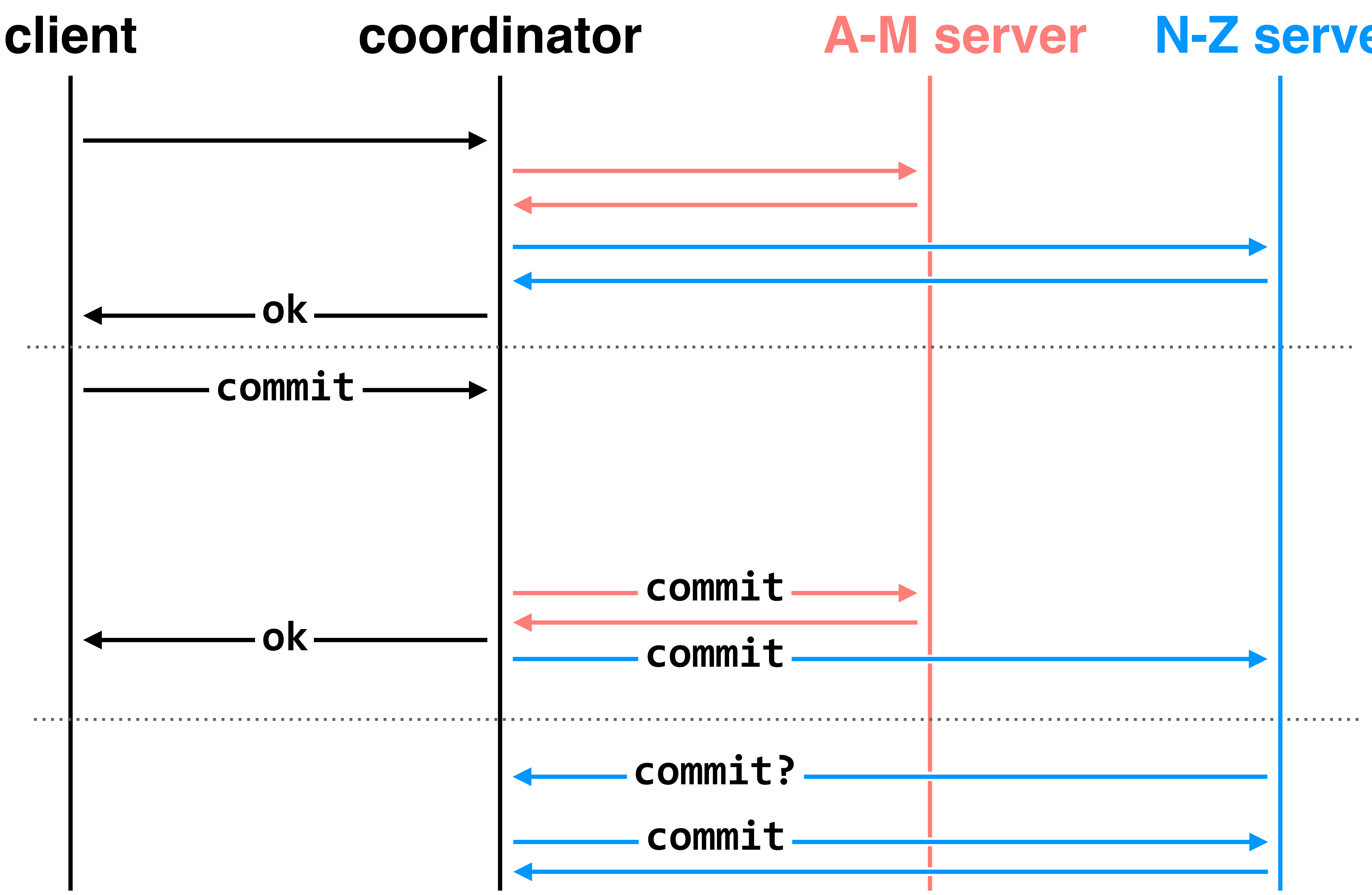
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction

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

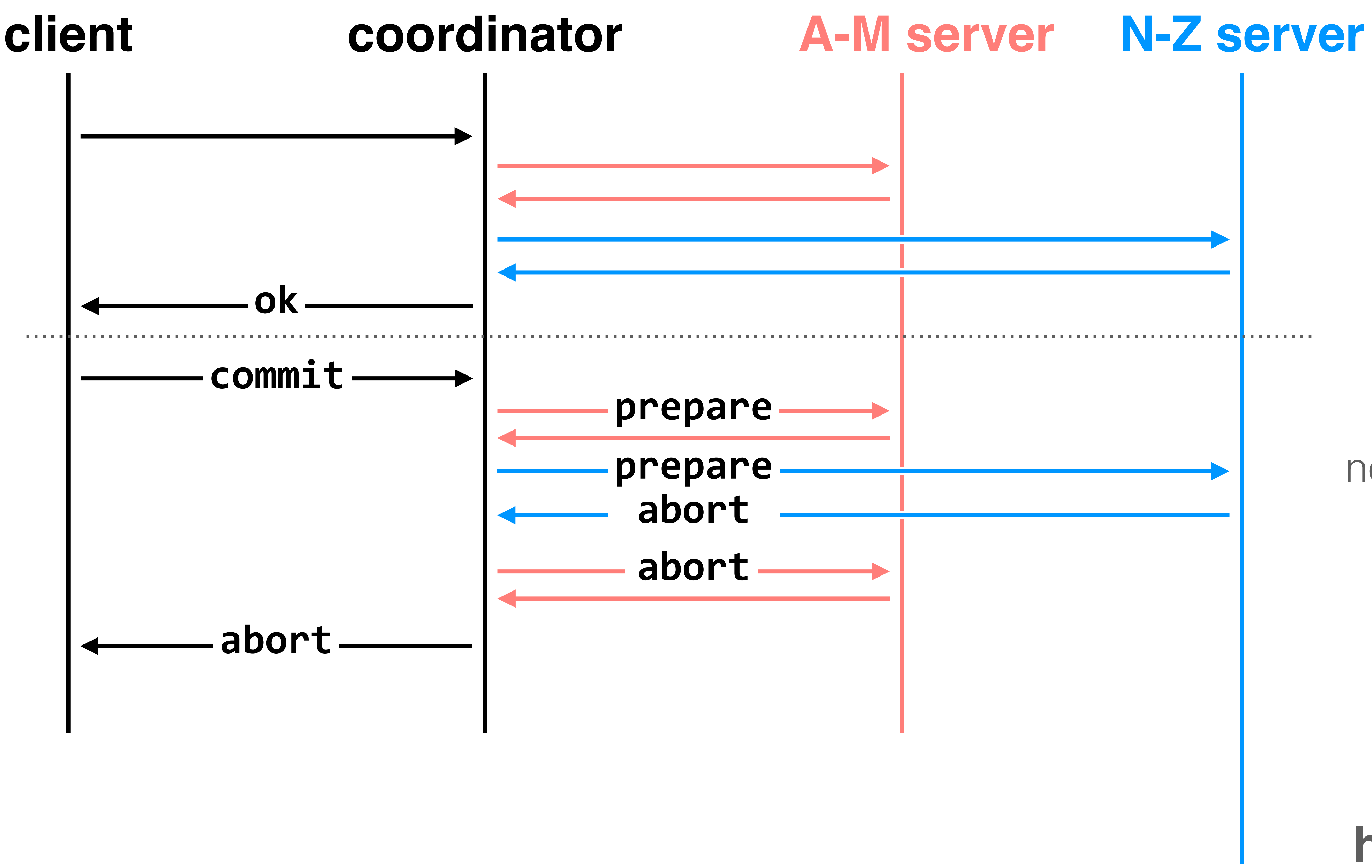


for instance, suppose we get rid of the prepare phase, and as long as one server commits, we force any that fail after that point to recover into a committed state?

broader question: why do we need two phases at all?

we've waited until this point to ask this question because it's helpful to understand how 2PC deals with failures first

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



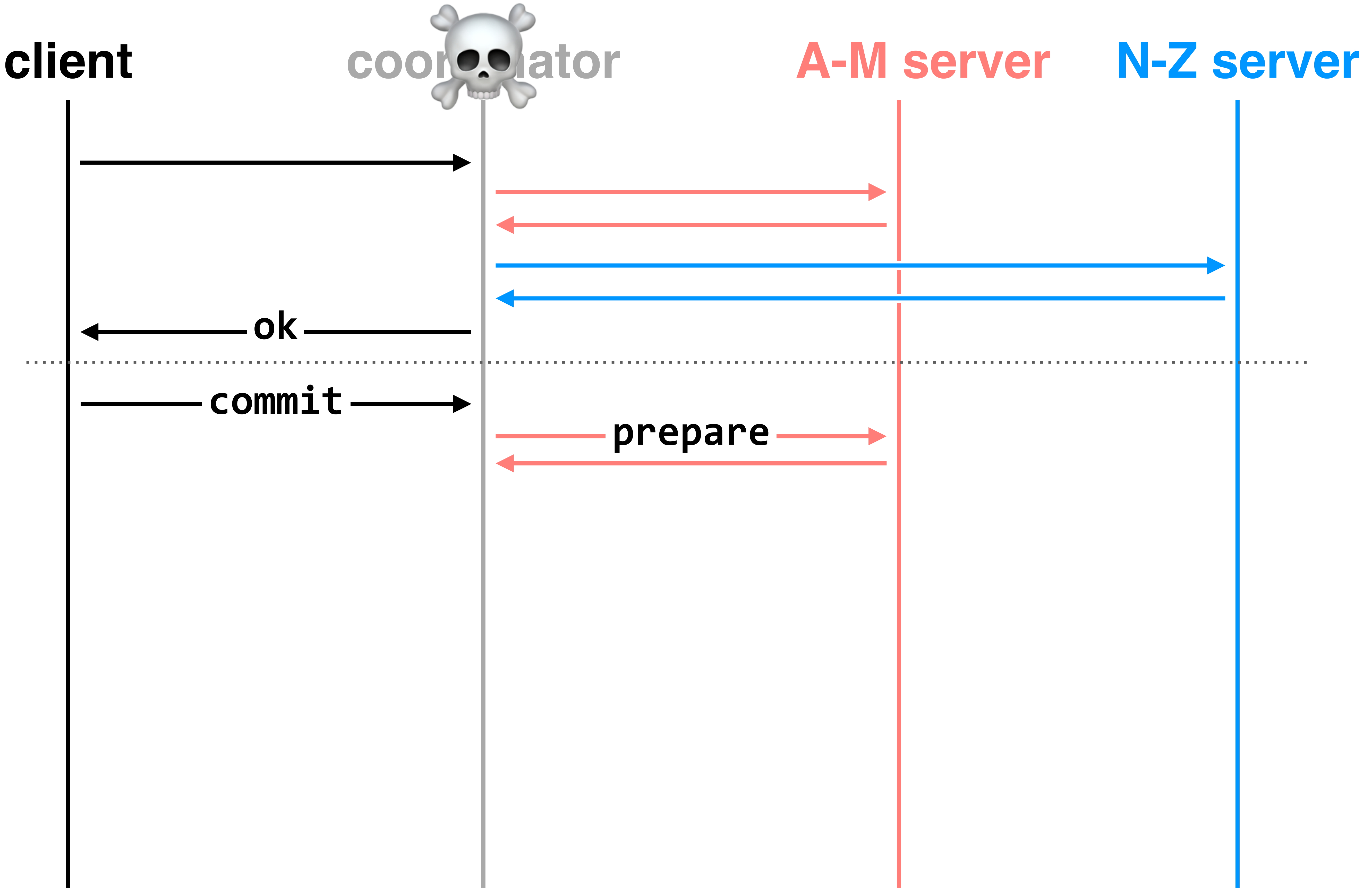
notice that the N-Z server did not fail here, but still aborted the transaction

the prepare phase of 2PC gives servers the chance to abort the transaction even if they haven't failed entirely (e.g., in the case of data corruption, local resource constraints, etc.)

broader question: why do we need two phases at all?

we've waited until this point to ask this question because it's helpful to understand how 2PC deals with failures first

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



now it's time to deal with coordinator failures

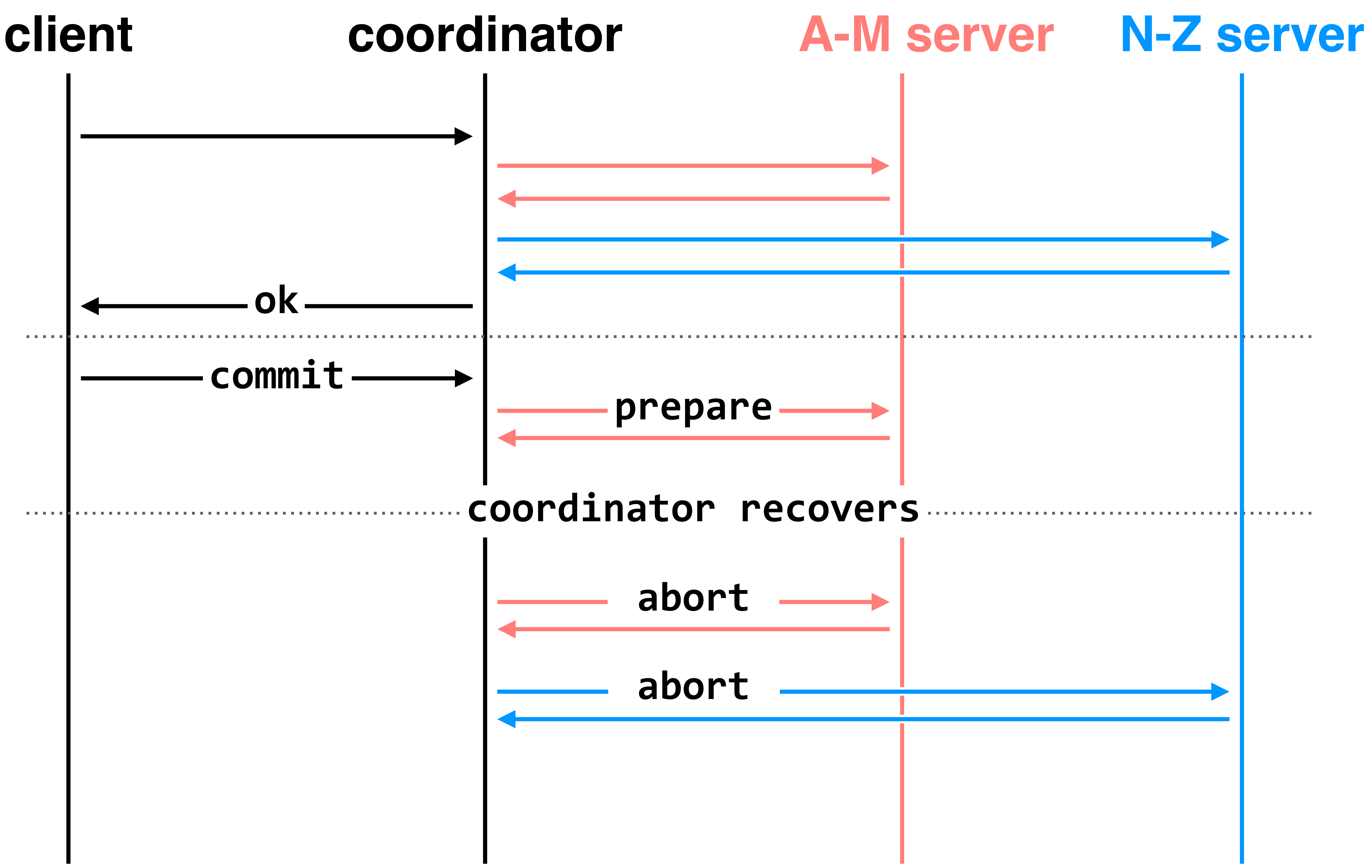
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction

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



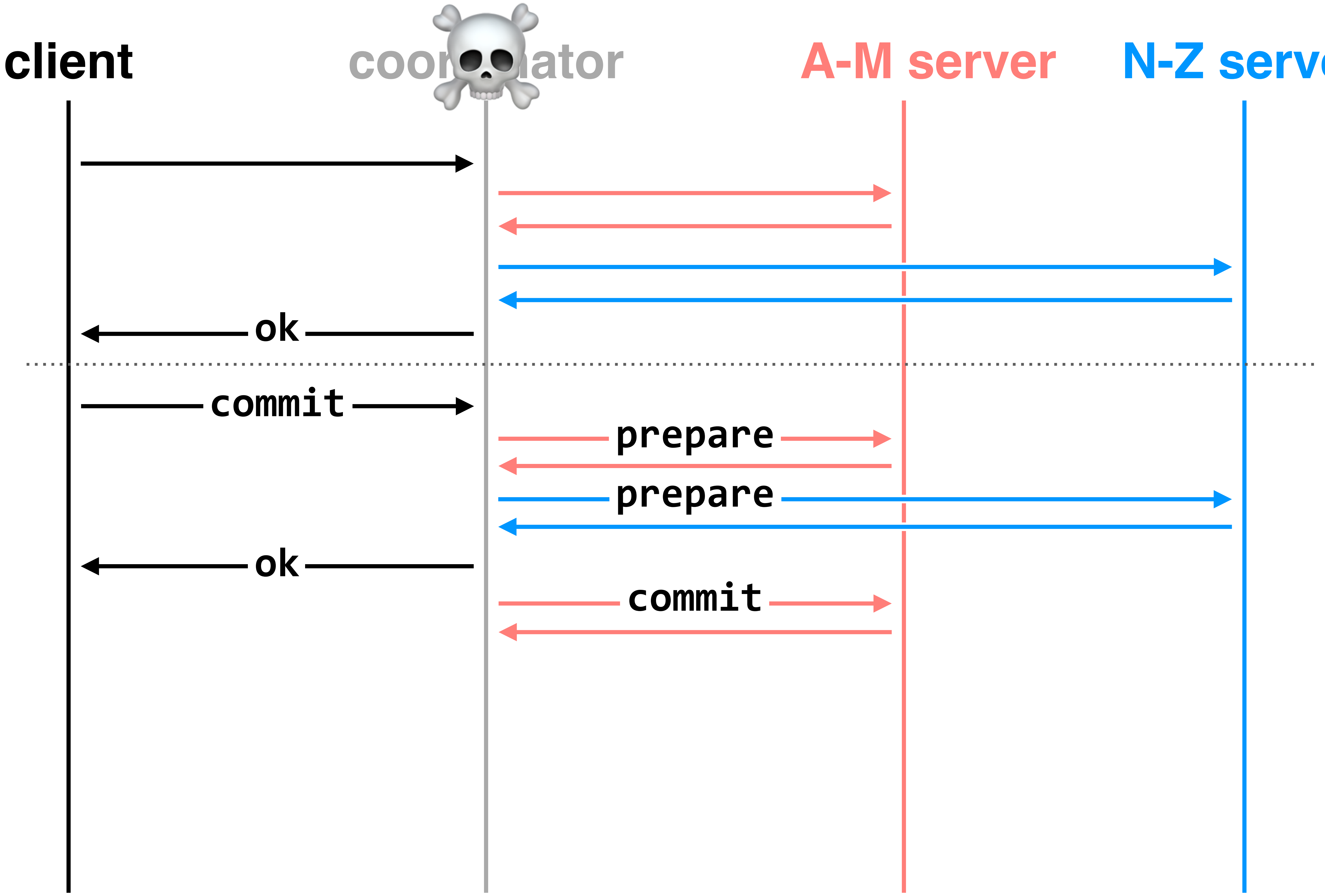
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction

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



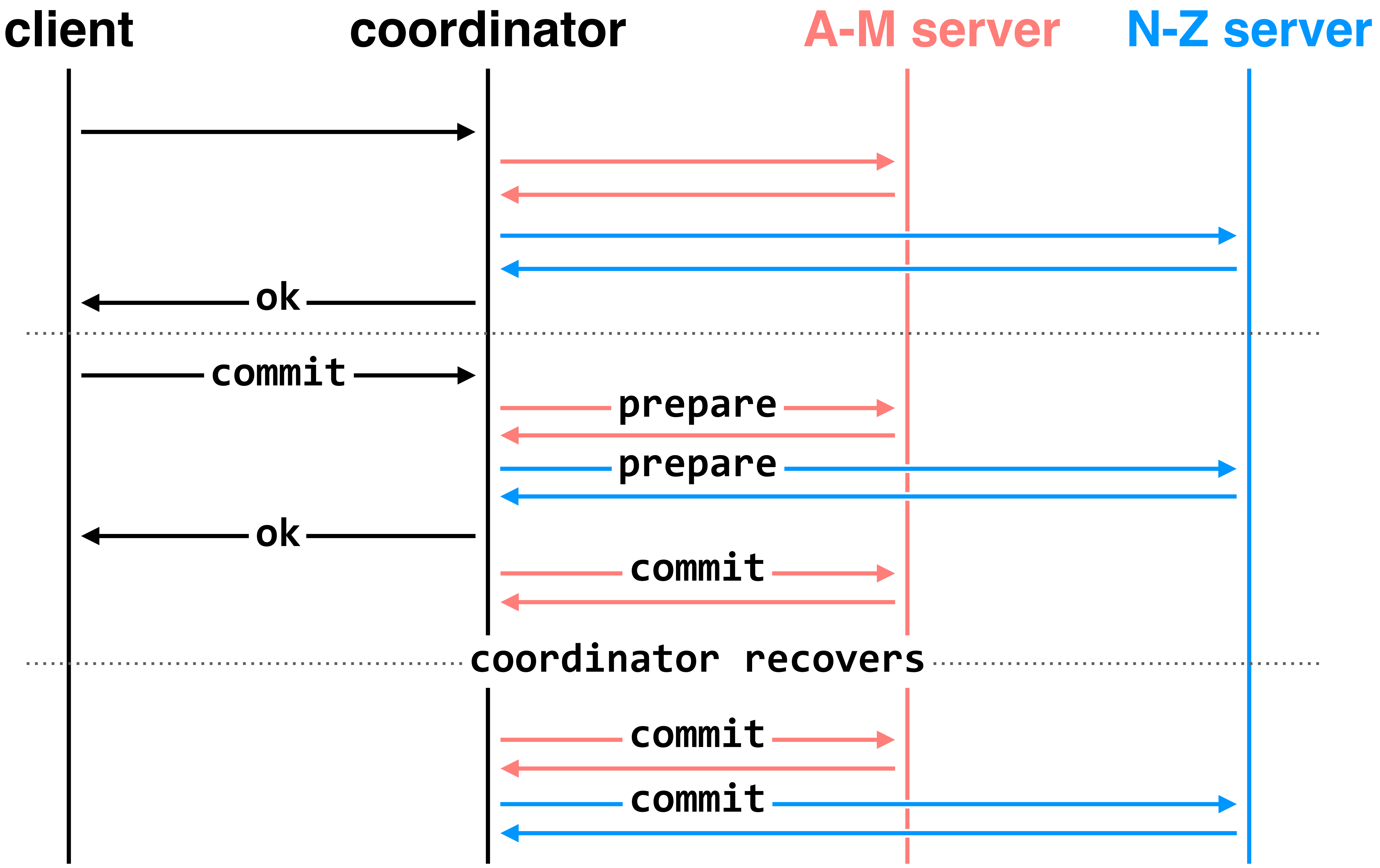
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction

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



performance issue: notice that if the coordinator fails during the prepare phase, it will **block** the transaction from progressing

there is also much more latency here than we would experience if we were running transactions on a single machine

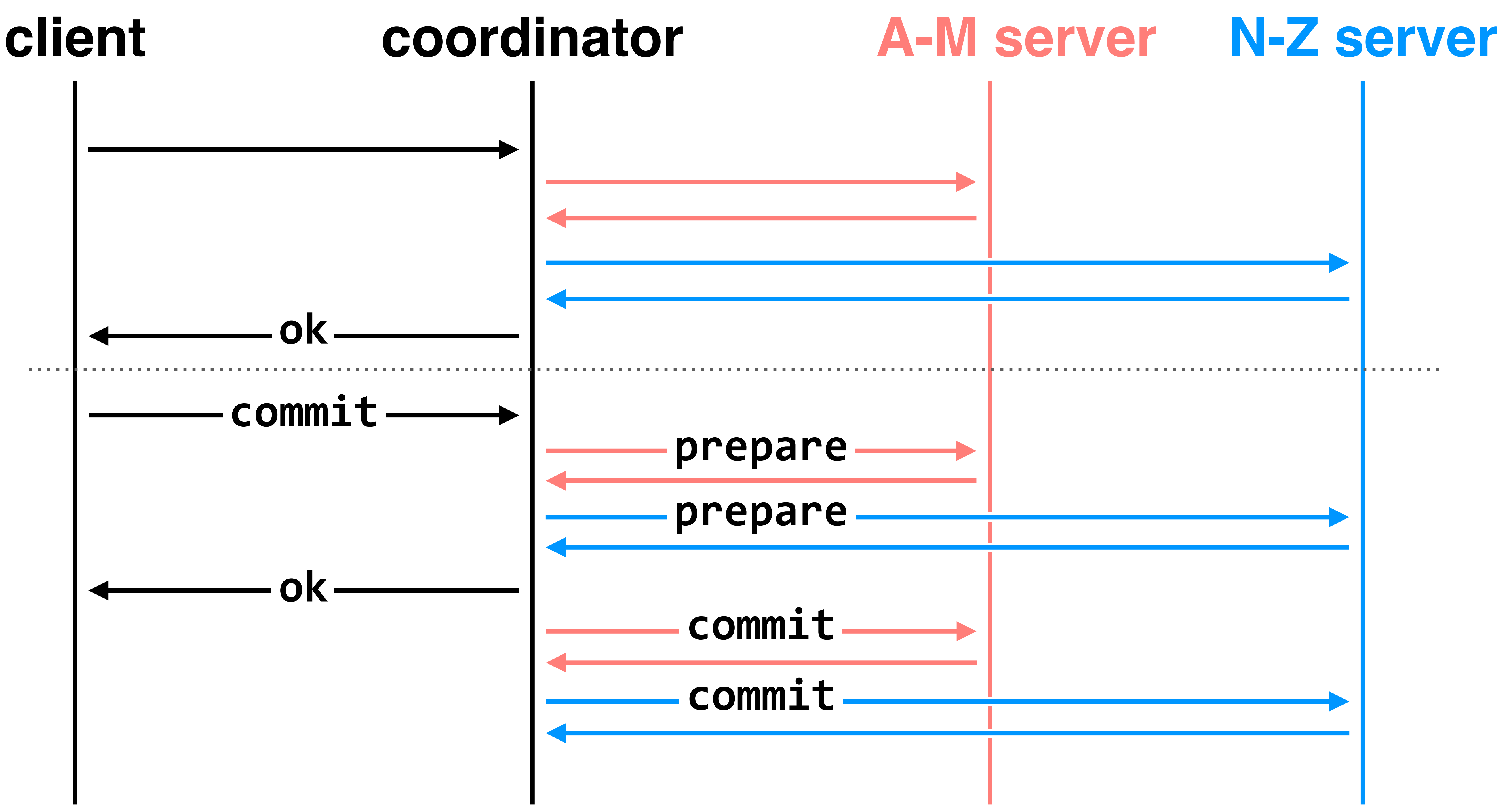
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction

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

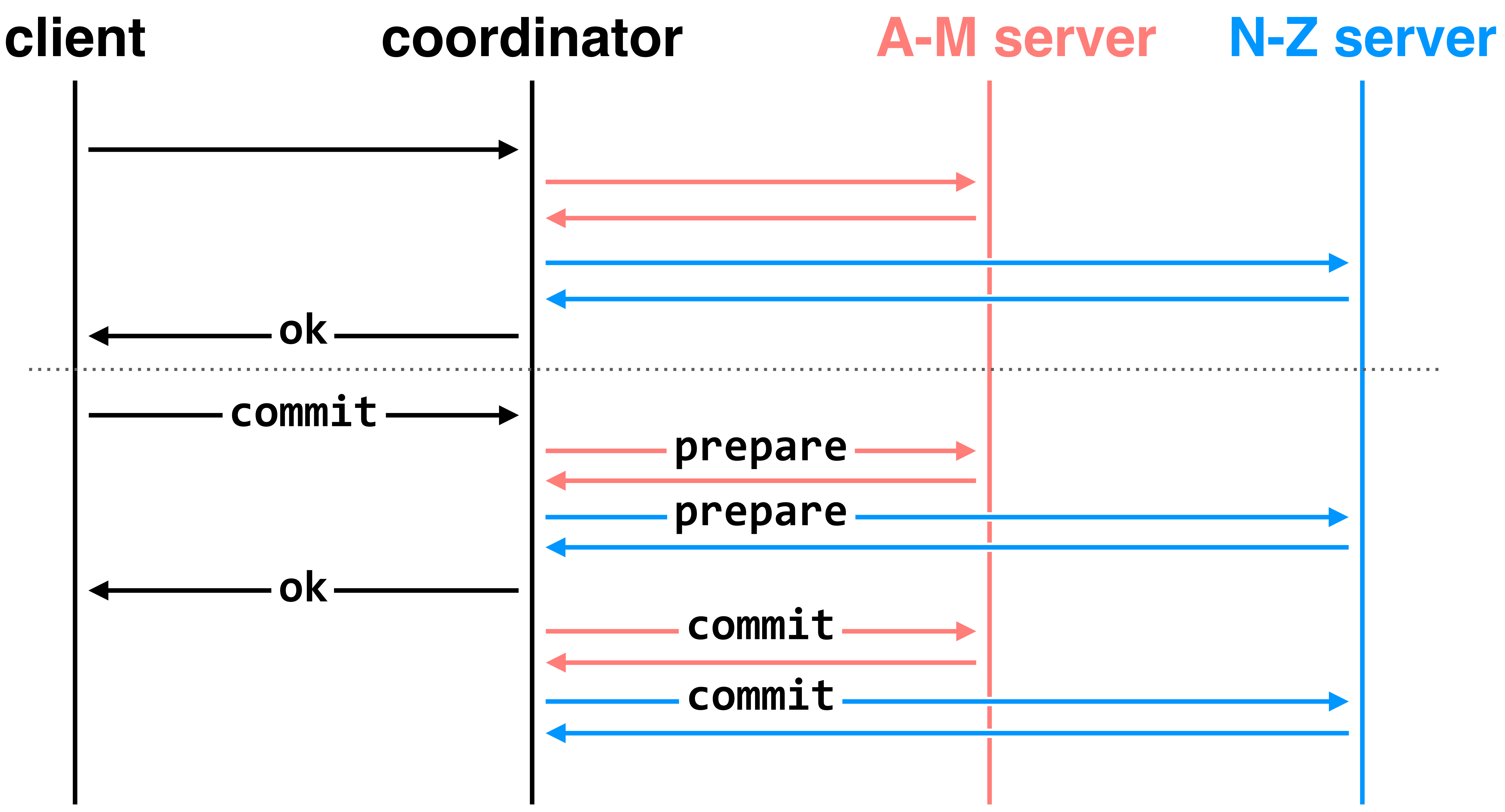
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction

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



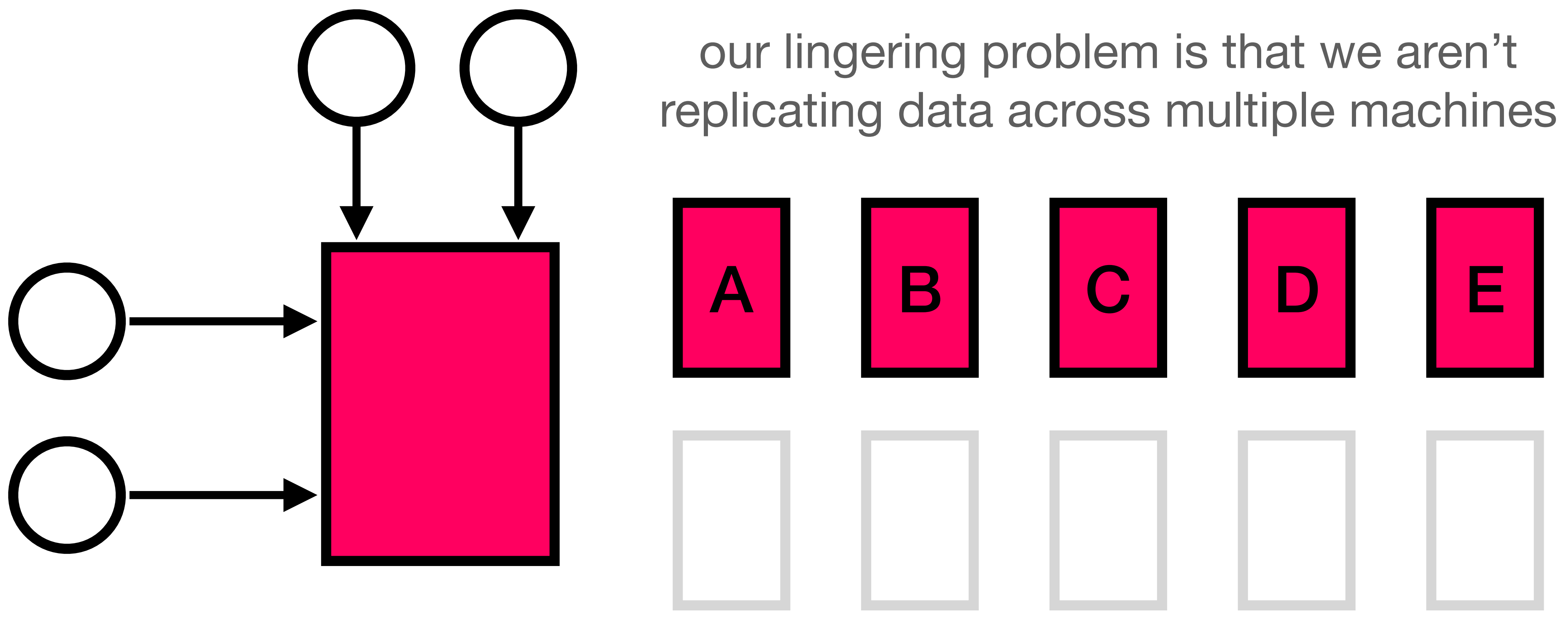
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction

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; **two-phase commit** gives us multi-site atomicity

* shadow copies *are* used in some systems

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

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**.

there are also performance issues in two-phase commit (e.g., the fact that the coordinator can block transactions from progressing if it fails), but we won't deal with those problems in this class