6.1800 Spring 2024

Lecture #19: Distributed Transactions

getting atomicity across machines

Katrina LaCurts | lacurts@mit.edu | 6.1800 2024

in-network resource management

recall this slide from Lecture 12…

type of management	what does this type of management allow a switch to do	example protocols	how the protocol works	pros/cons?
Queue Management	signal congestion, potentially before queues are full	`DropTail`	drop packets when the queue is full	simple, but queues get full (among other problems)
		`RED, ECN`	drop or mark packets before the queue is full	can keep queues from filling up, but complicated
Delay-based Scheduling	prioritize latency-sensitive traffic	`Priority Queueing`	serve some queues before others	prioritized queues can starve out the others
Bandwidth-based Scheduling	enforce (weighted) fairness among different types of traffic	`Round-robin`	try to give each type of traffic an equal share of bandwidth	can't handle variable packet sizes
		`Weighted Round-robin`	round robin, but incorporate average packet size	average packet size hard to get
		`Deficit Round-robin`	round robin, but do a better job with packet sizes	honestly pretty good

is in-network resource management a good idea on the Internet?

6.1800 in the news

ISPs can charge extra for fast gaming under FCC's Internet rules, critics say

FCC plan rejected request to ban what agency calls "positive" discrimination.

JON BRODKIN - 4/16/2024, 4:38 PM

Some net neutrality proponents are worried that soon-to-be-approved Federal Communications Commission rules will allow harmful fast lanes because the plan doesn't explicitly ban "positive" discrimination.

FCC Chairwoman Jessica Rosenworcel's proposed rules for Internet service providers would prohibit blocking, throttling, and paid prioritization. The rules mirror the ones imposed by the FCC during the Obama era and repealed during Trump's presidency. But some advocates are criticizing a decision to let Internet service providers speed up certain types of applications as long as application providers don't have to pay for special treatment.

Almost certainly, the millions of Americans who celebrated the 2015 Open Internet Order and fought the 2017 repeal think that net neutrality protections ban fast lanes and slow lanes.

The draft order takes a different approach.

The no-throttling rule that the FCC proposed in October explicitly prohibited ISPs from slowing down apps and classes of apps; it was silent on whether the rule also applies to speeding up.

Given the mobile ISPs' public statements about their plans for 5G fast lanes, public interest groups, startups, and members of Congress asked the FCC to clarify in the Order that the no-throttling rule also prohibits ISPs from speeding up apps and classes of apps.

The draft order did not do that.

https://arstechnica.com/tech-policy/2024/04/isps-can-charge-extra-for-fast-gaming-under-fccs-internet-rules-critics-say/

https://cyberlaw.stanford.edu/blog/2024/04/harmful-5g-fast-lanes-are-coming-fcc-needs-stop-them

6.1800 in the news

5G Network Slicing

As stated in a recent ex parte[3], network slicing is not a service but is a core component of the 5G standard being deployed in networks around the world; slicing provides network operators with additional network management capabilities to better match wireless services to the network resources that they require. This will enable providers to better meet the needs of particular business applications and consumer preferences than they could over a best-efforts network that generally treats all traffic the same.

https://www.fcc.gov/ecfs/document/1041519473041135/1

6.1800 in the news

Not all 5G slices are harmful.

Just to be clear, net neutrality proponents are not asking the FCC to ban network slicing. There's lots of ways for ISPs to use slices for things that are not normal internet service such as a dedicated slice for a farming operation using remote controlled tractors, slices for telemetry data and oversight of autonomous cars, or providing a slice for a stadium's video system at a crowded game.

There are good reasons to isolate this kind of traffic, and it can be done without reducing user choice or tilting the online playing field. Under the FCC's draft order, such services would be so-called enterprise service offerings, to which the Open Internet protections don't apply.

https://cyberlaw.stanford.edu/blog/2024/04/harmful-5g-fast-lanes-are-coming-fcc-needs-stop-them

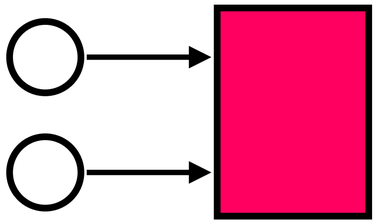
6.1800 in the news

Just a short 17 months ago in its 2022 Broadband Labeling Order[6], which took effect April 10, 2024, the Commission agreed with AT&T and correctly determined that requiring packet loss reporting was inadvisable and not useful for consumers. The Commission accurately noted that OMB in 2016 concluded that packet loss would not be required for mobile broadband, and the Commission also noted that consumers have little to no understanding of packet loss.

The Commission has greatly underestimated the burdens of the collection and reporting requirements currently pending adoption. These requirements may require AT&T to: develop new systems and software; collect, analyze, and verify vast amounts of new data; train thousands of employees and contractors; potentially install new equipment in dozens of vehicles used for drive testing; potentially add thousands of miles to existing drive test routes; and implement numerous other costly initiatives.

https://www.fcc.gov/ecfs/document/1041519473041135/1

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

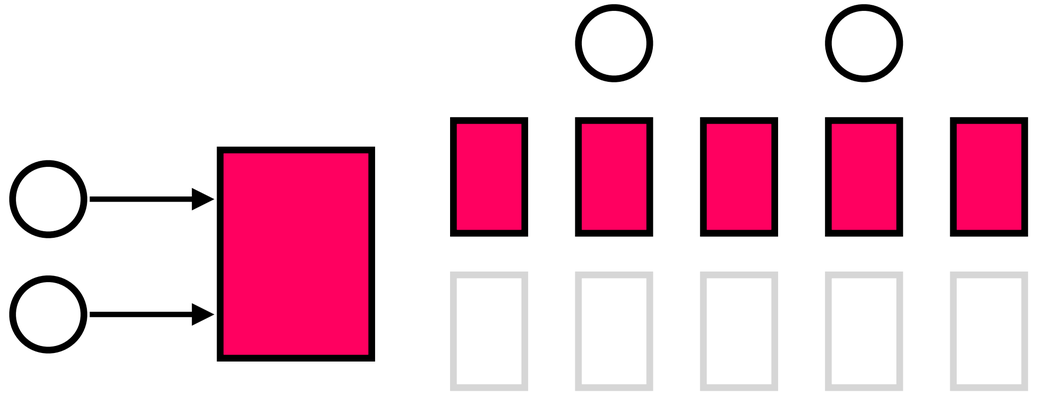
our job in lecture is to understand how a system *implements* these two abstractions.
how do our systems guarantee atomicity? how do they guarantee isolation?

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)

```
transfer(A, B, amount)
```

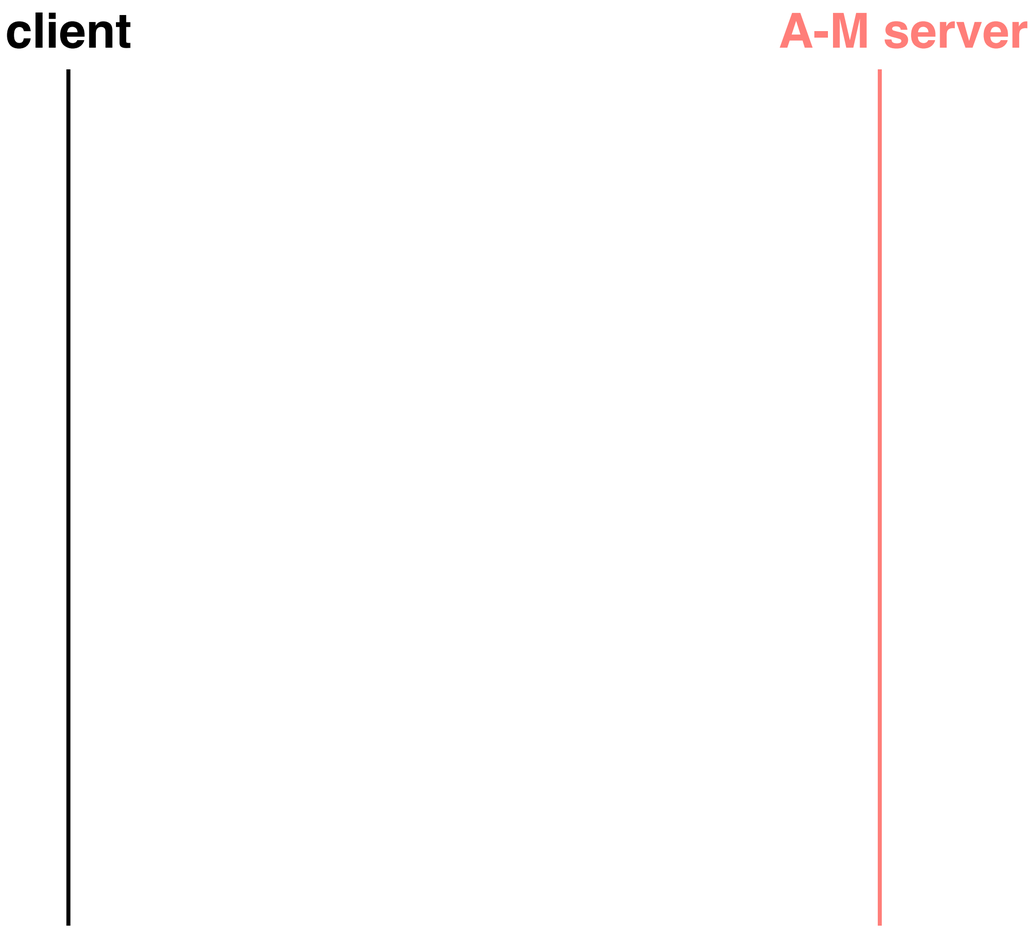
transactions across multiple machines (no failures yet)

```
transfer(A, B, amount)
```

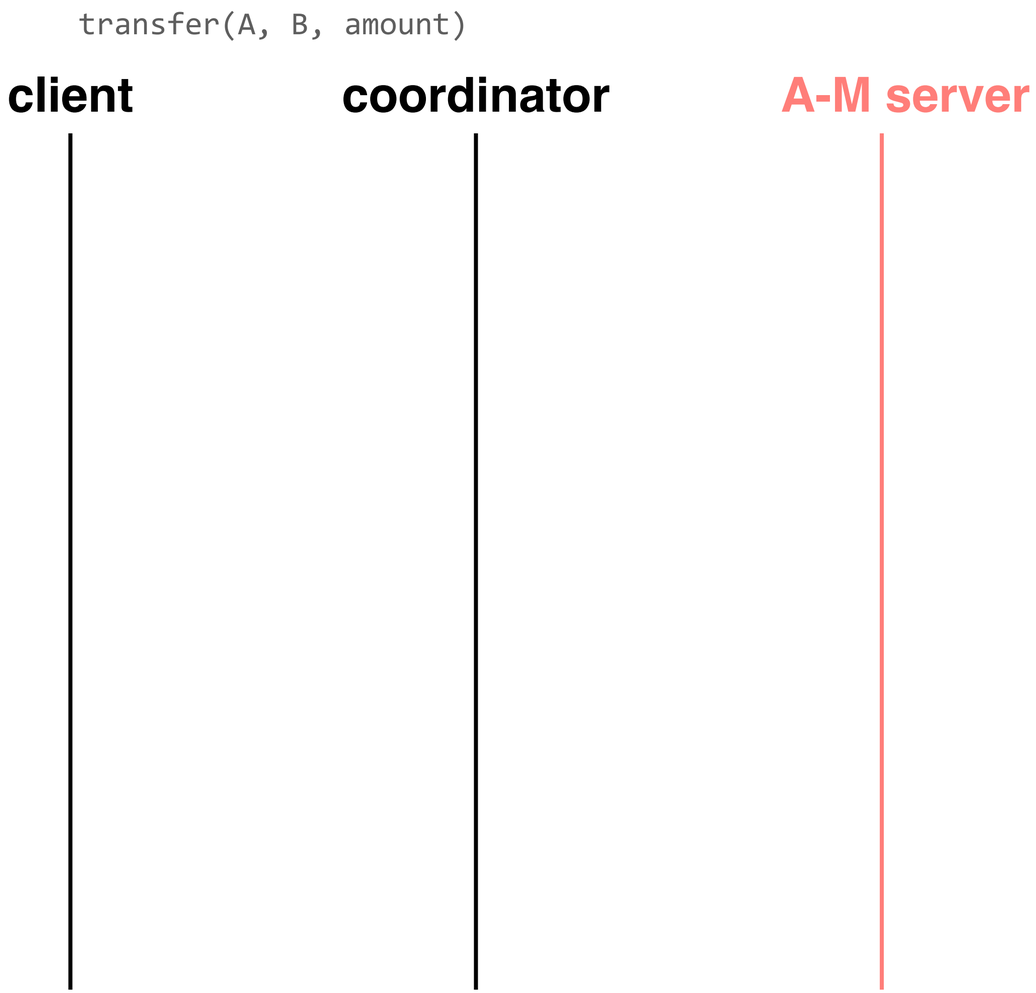
client

transactions across multiple machines (no failures yet)

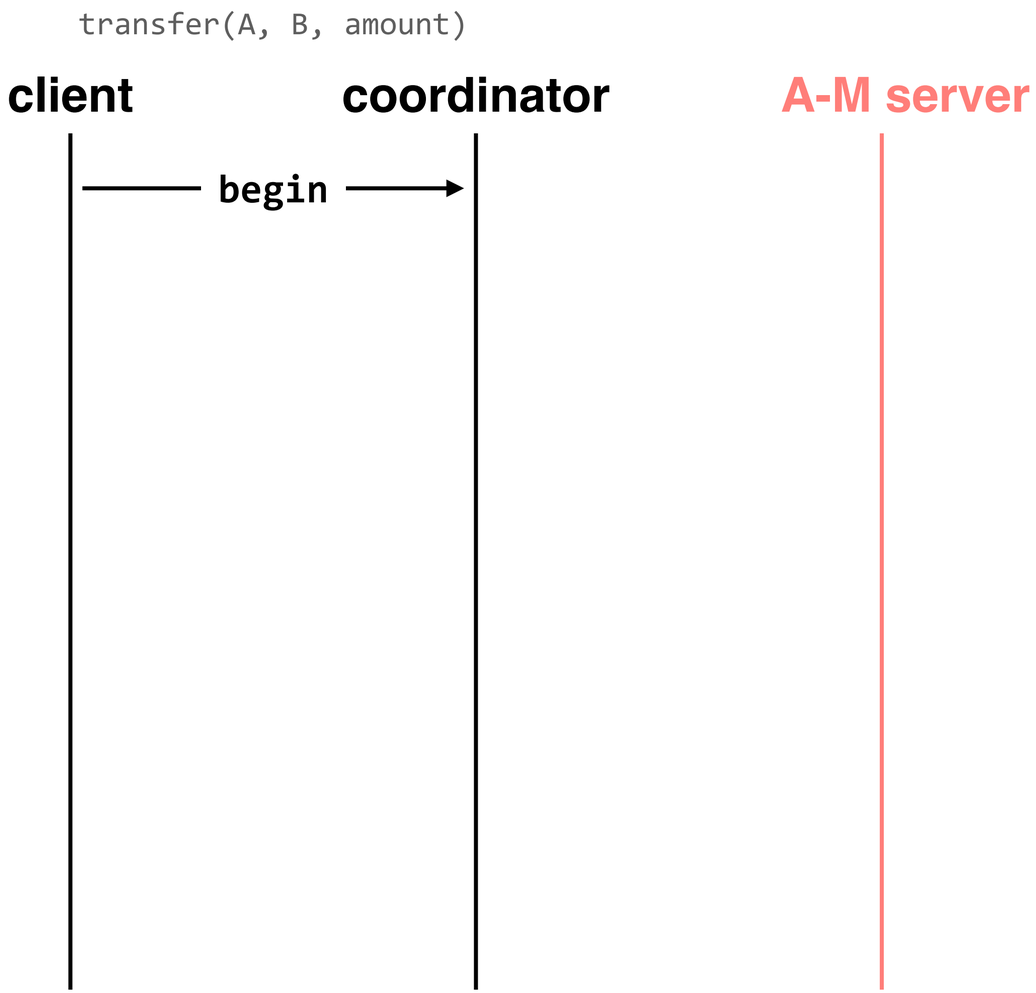
```
transfer(A, B, amount)
```



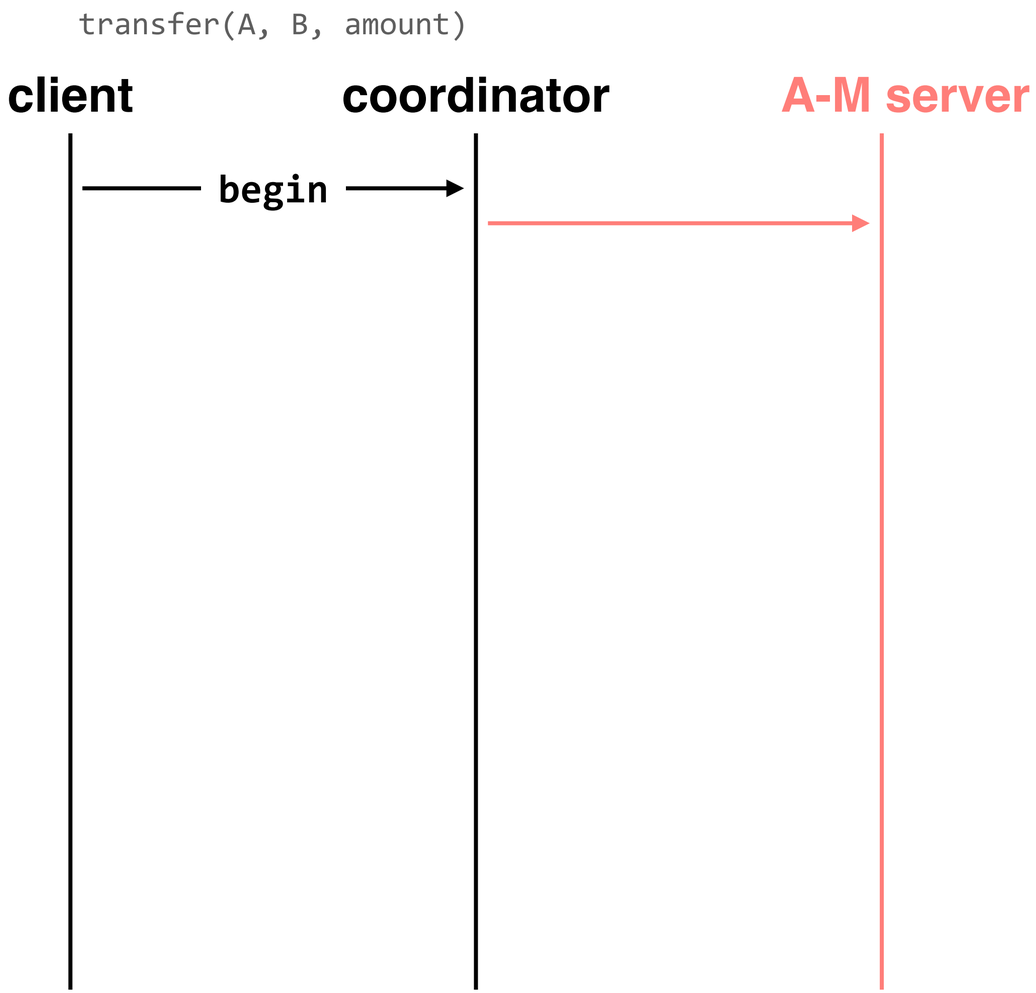
transactions across multiple machines (no failures yet)



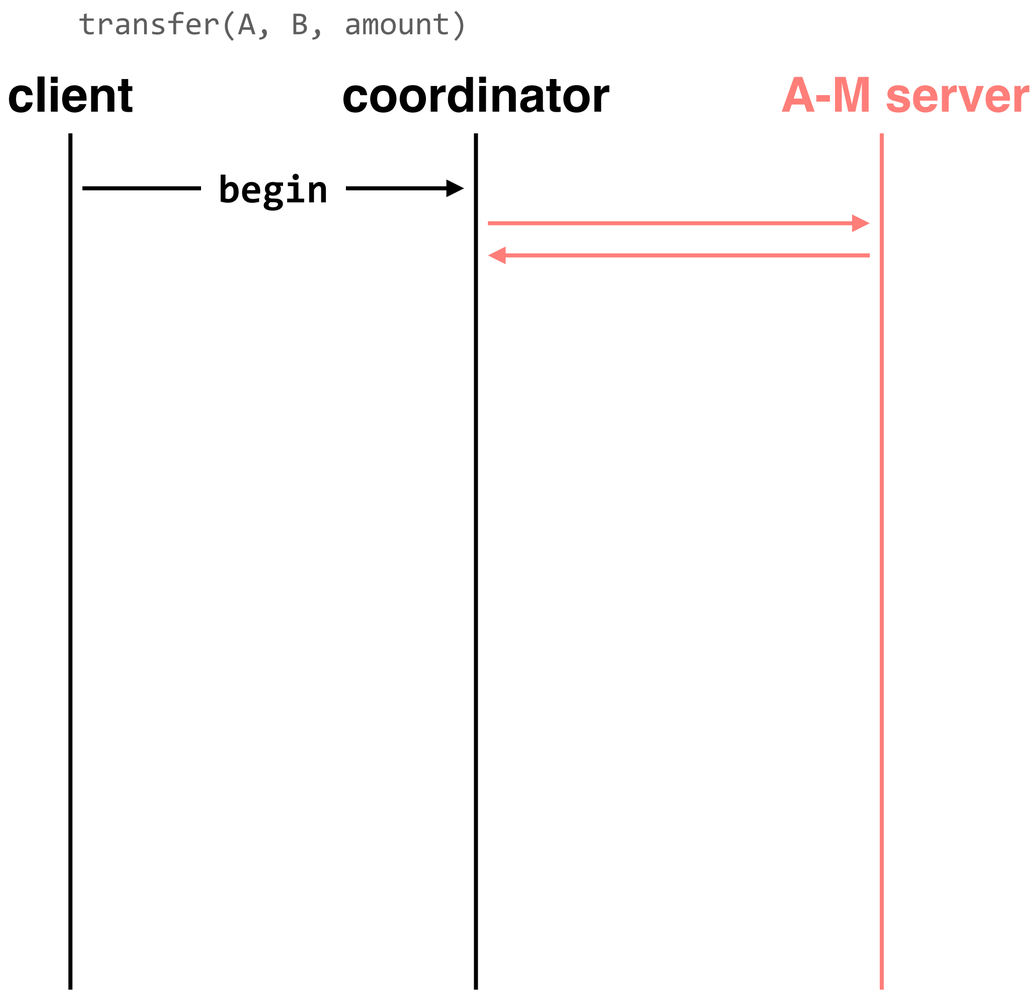
transactions across multiple machines (no failures yet)



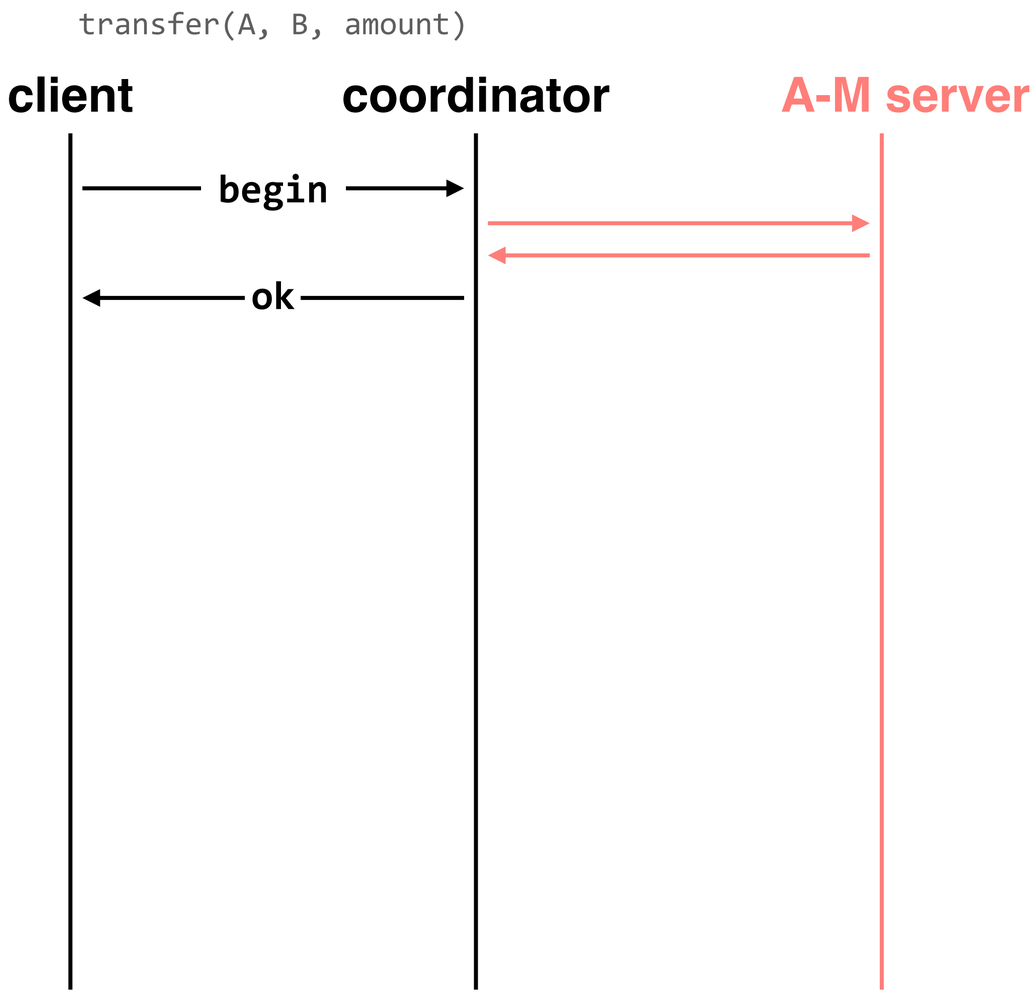
transactions across multiple machines (no failures yet)



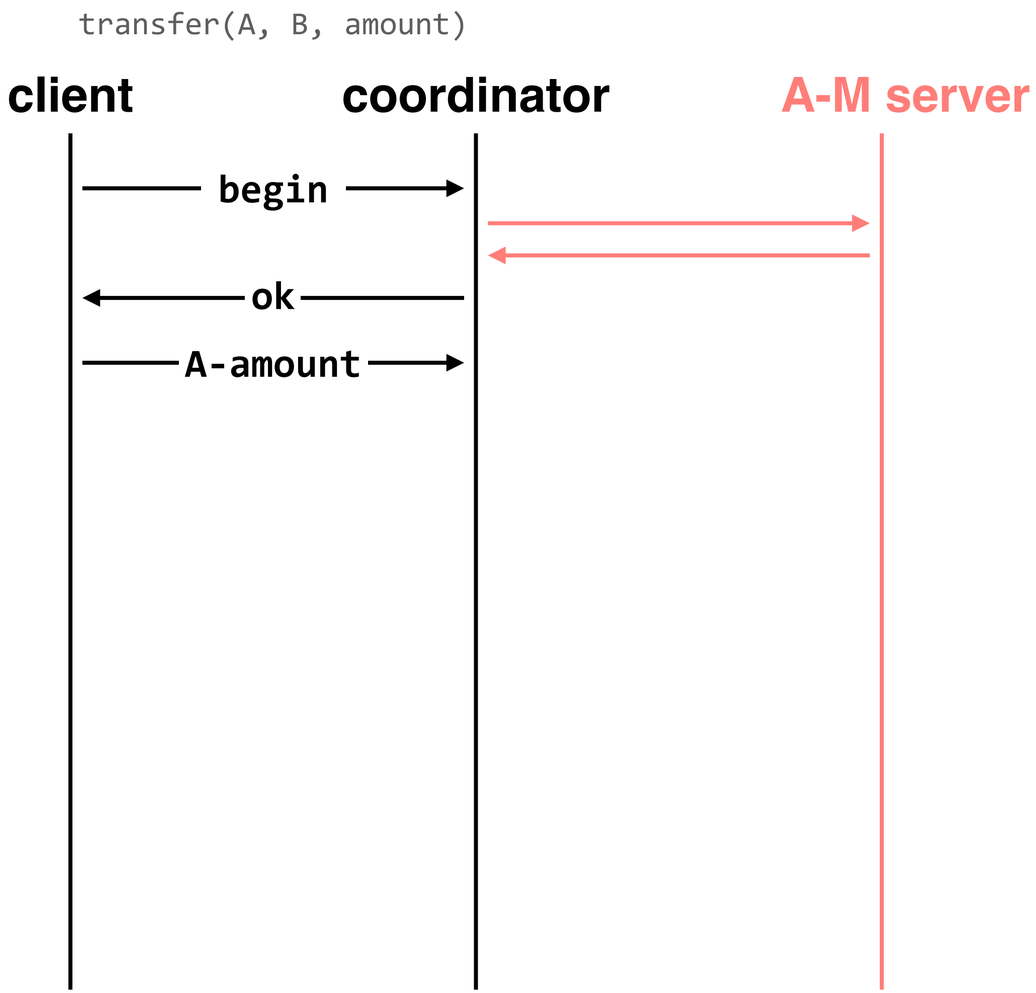
transactions across multiple machines (no failures yet)



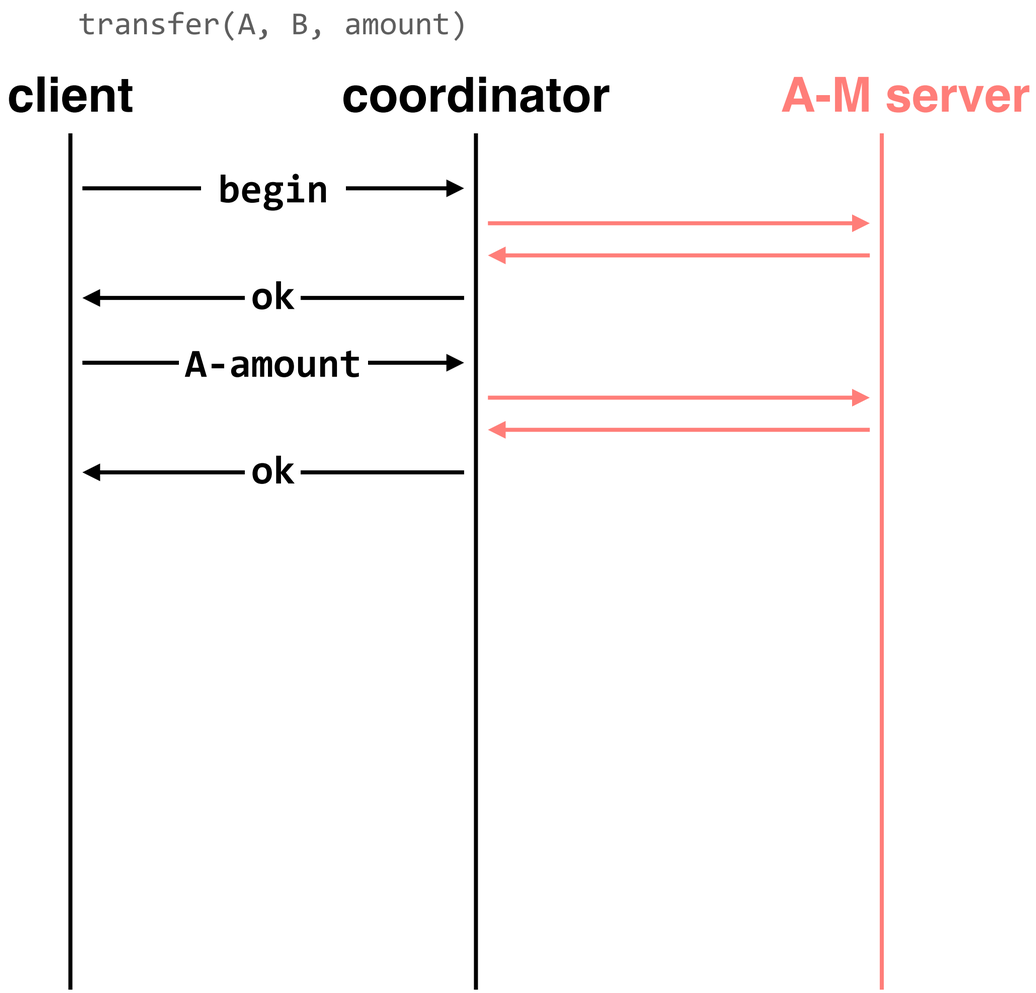
transactions across multiple machines (no failures yet)



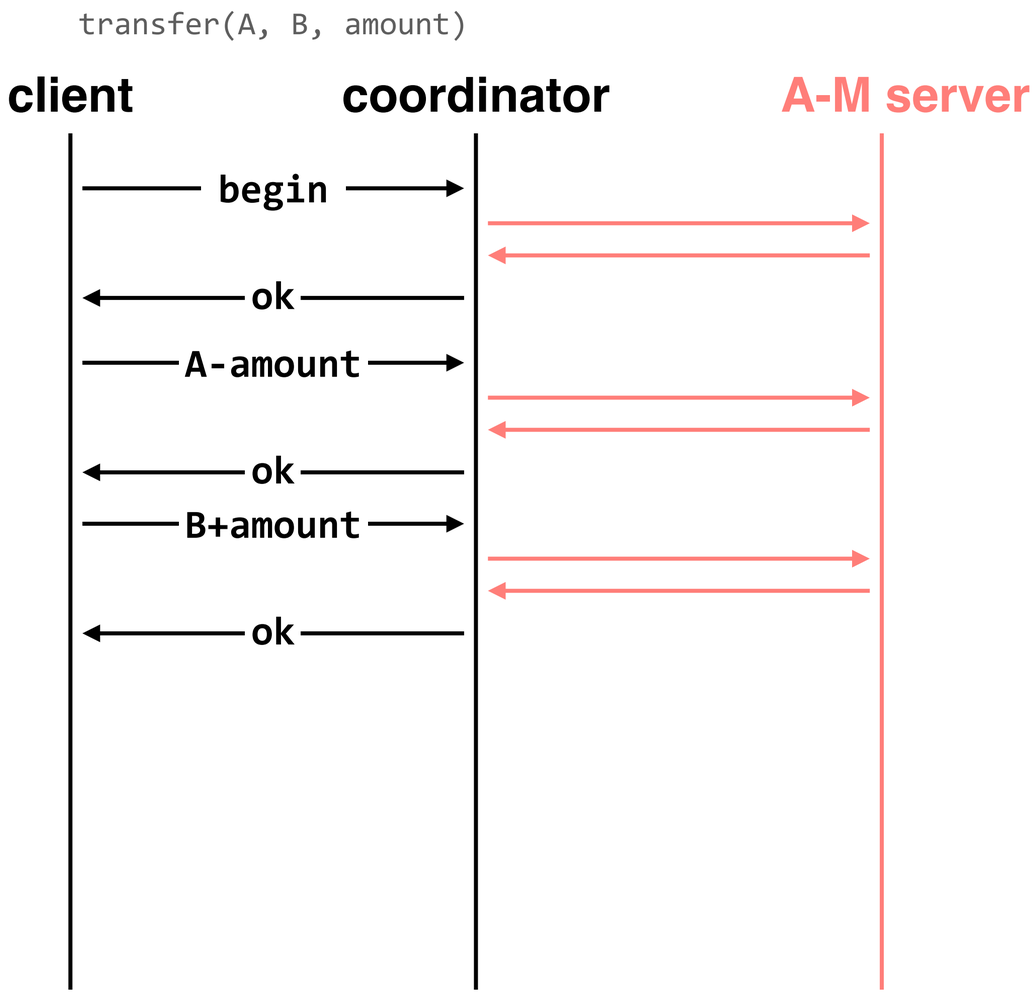
transactions across multiple machines (no failures yet)



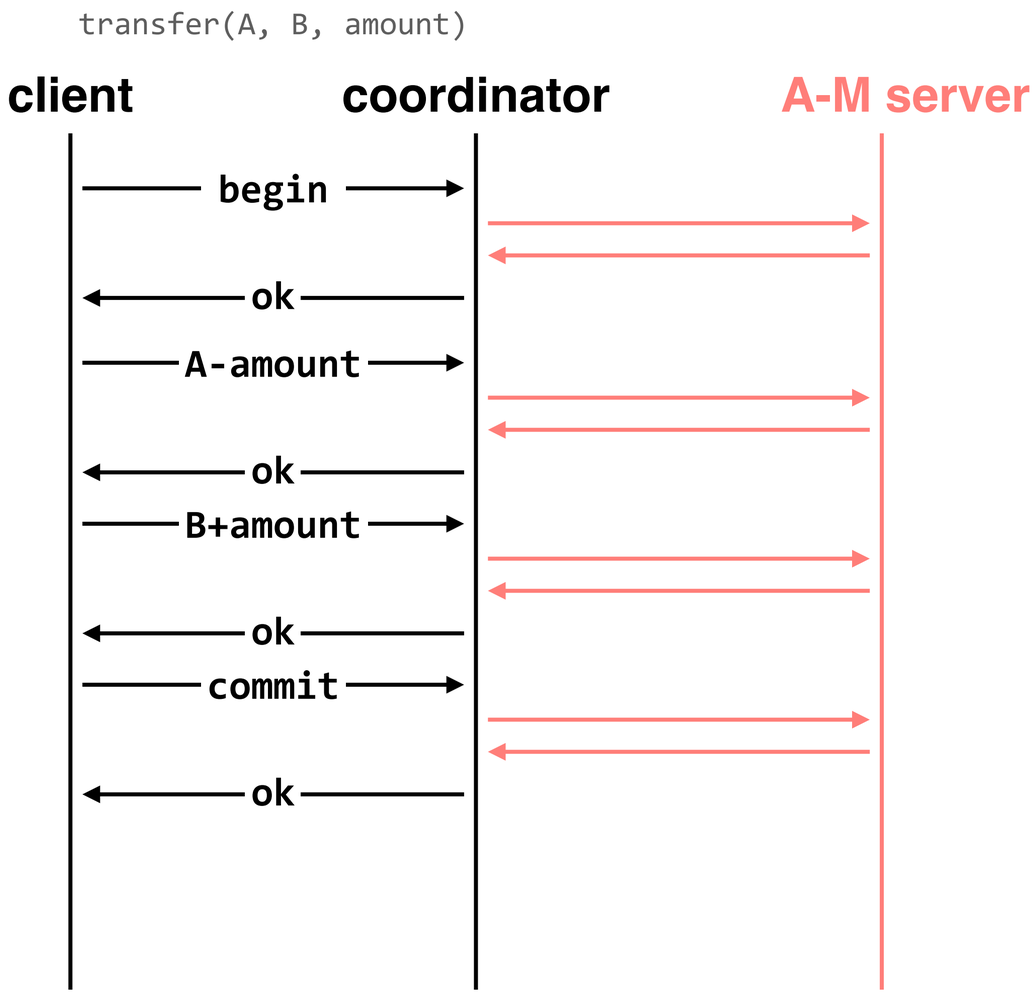
transactions across multiple machines (no failures yet)



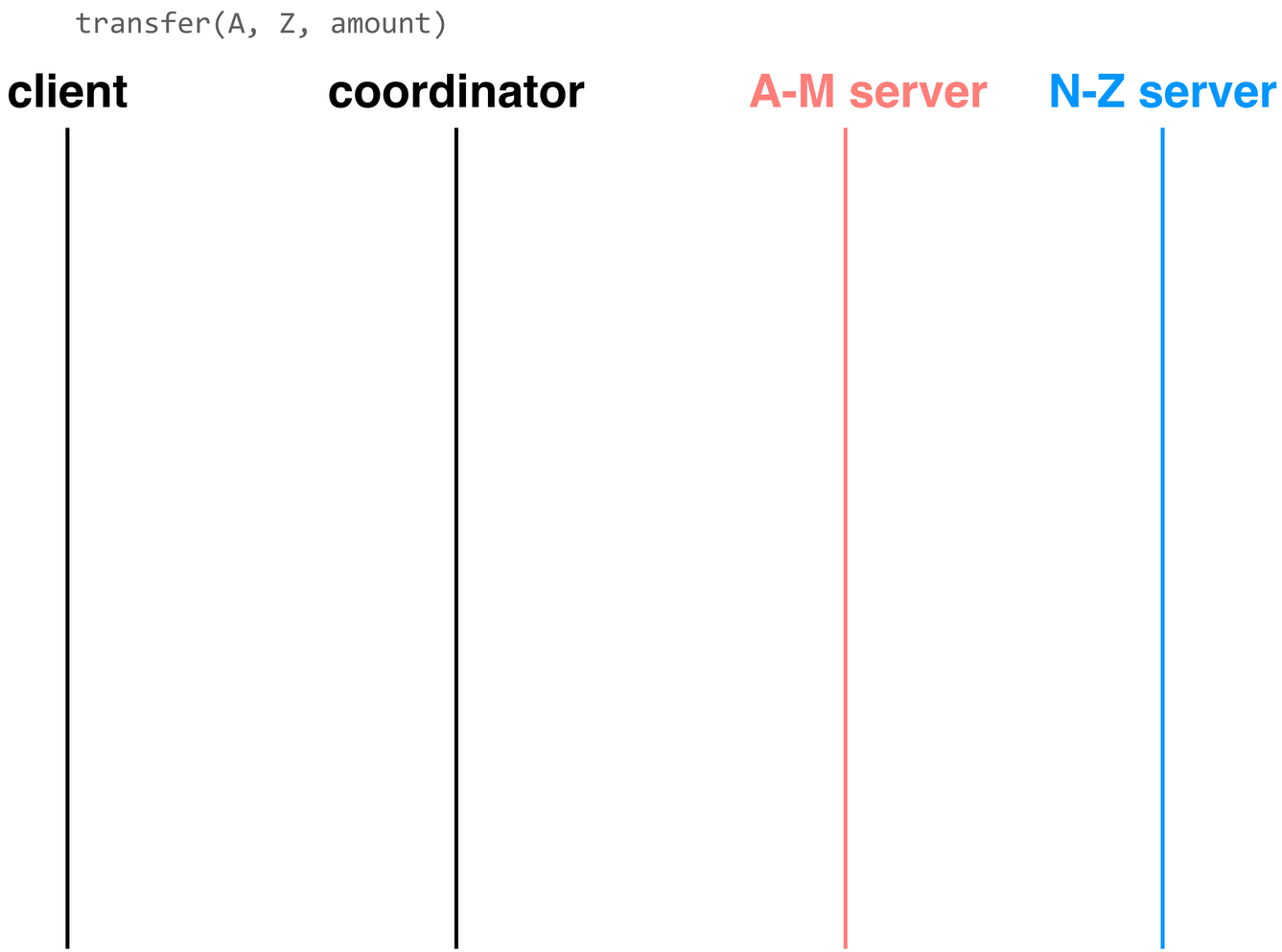
transactions across multiple machines (no failures yet)



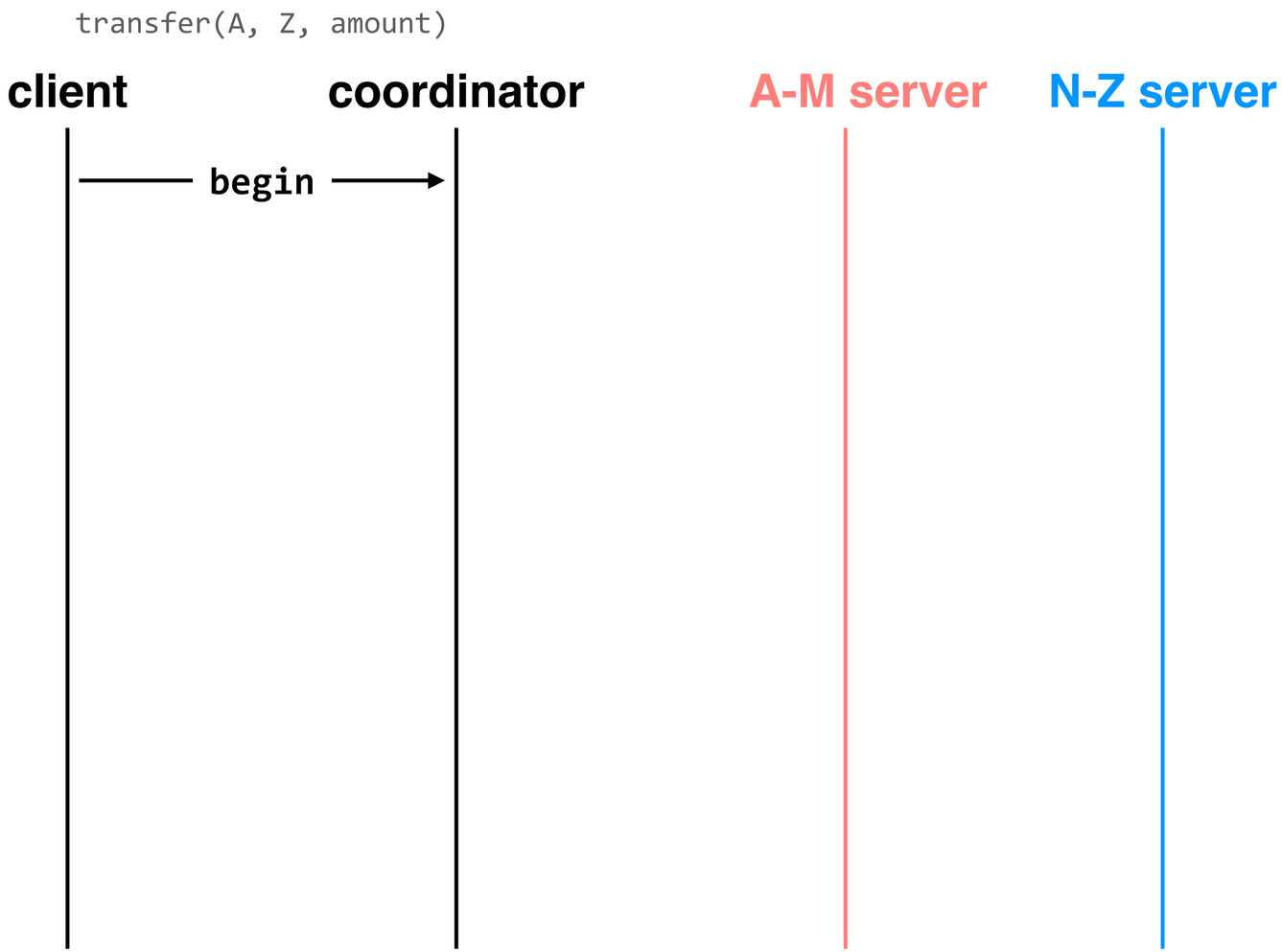
transactions across multiple machines (no failures yet)



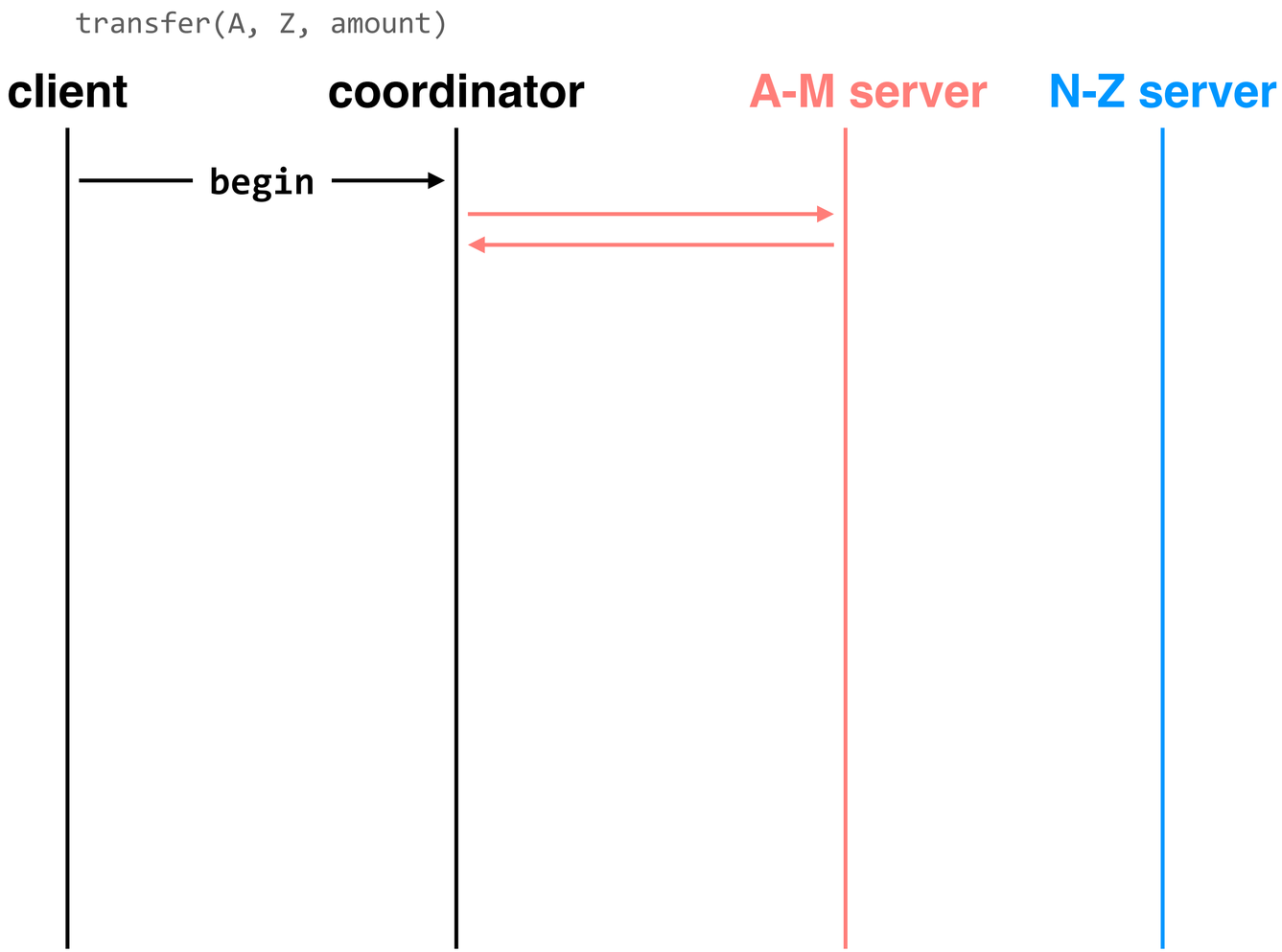
transactions across multiple machines (no failures yet)



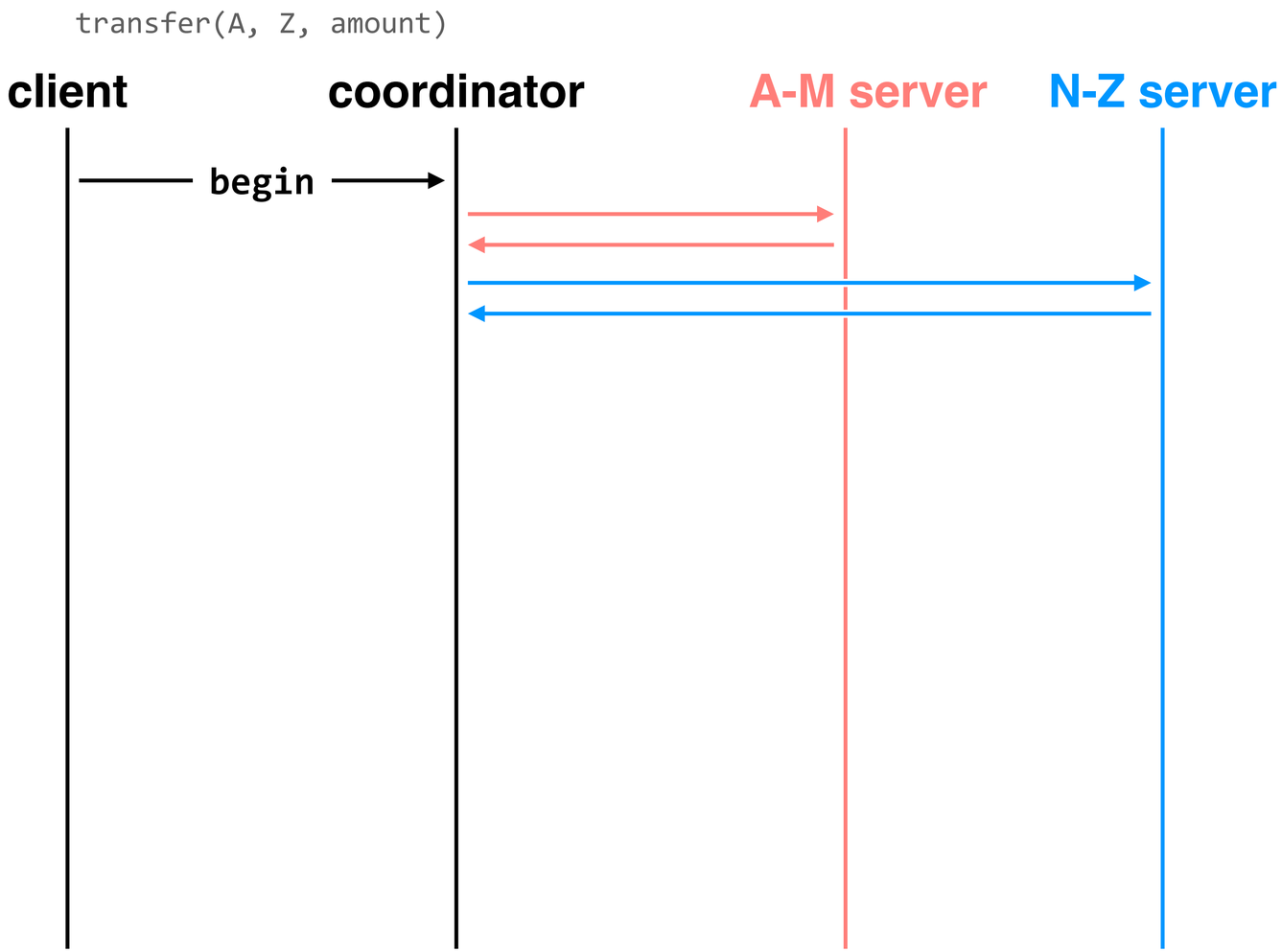
transactions across multiple machines (no failures yet)



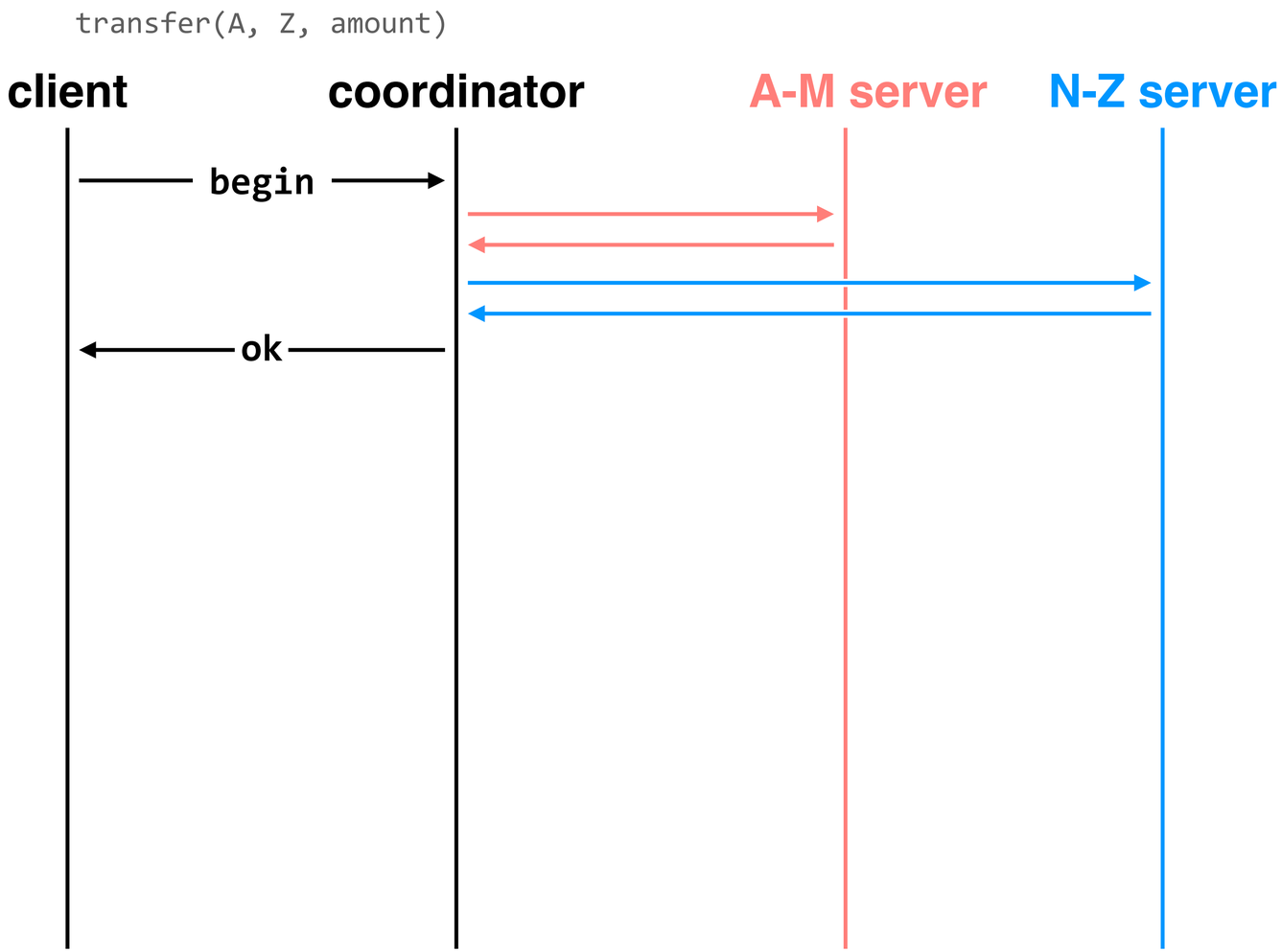
transactions across multiple machines (no failures yet)



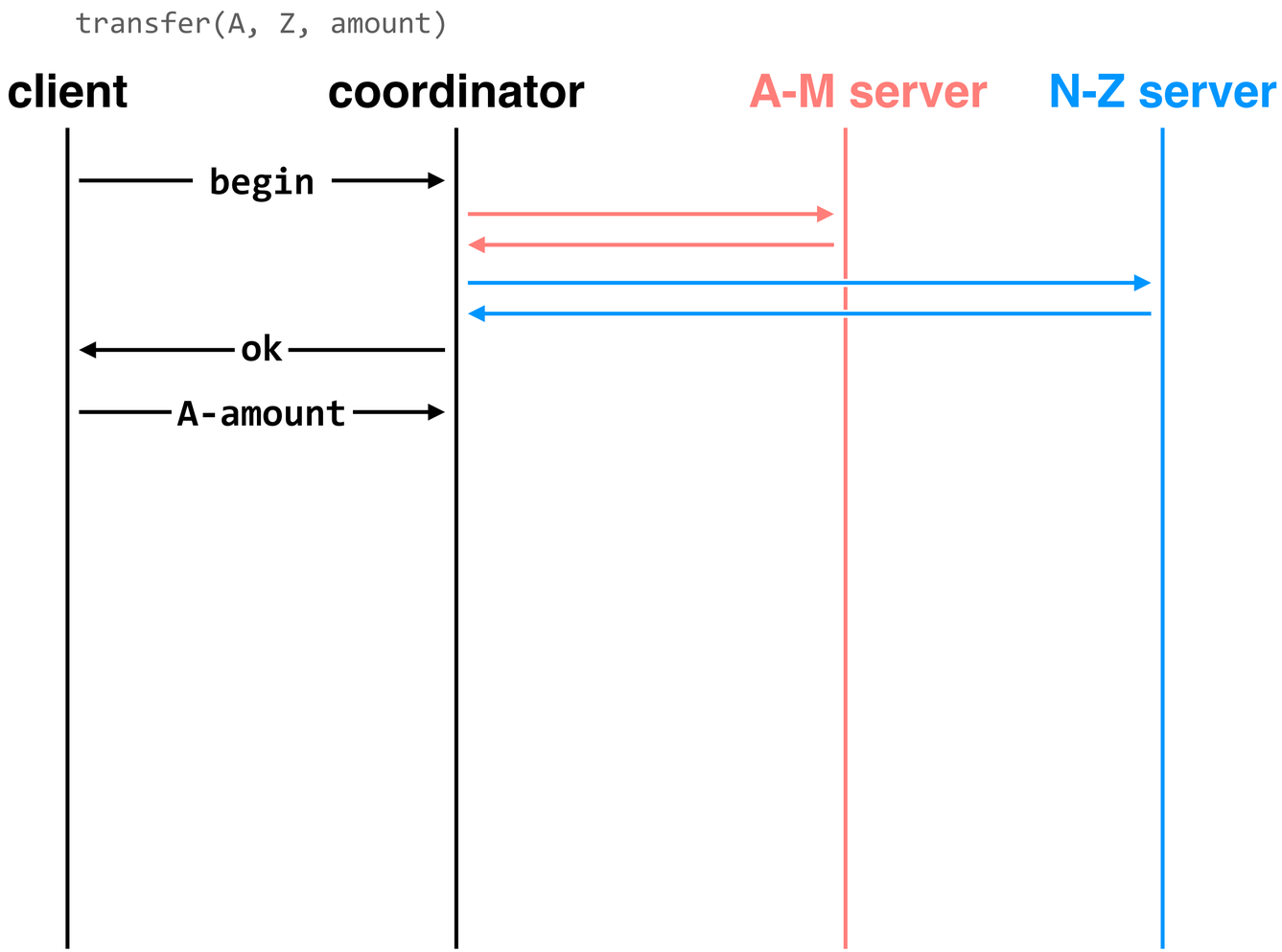
transactions across multiple machines (no failures yet)



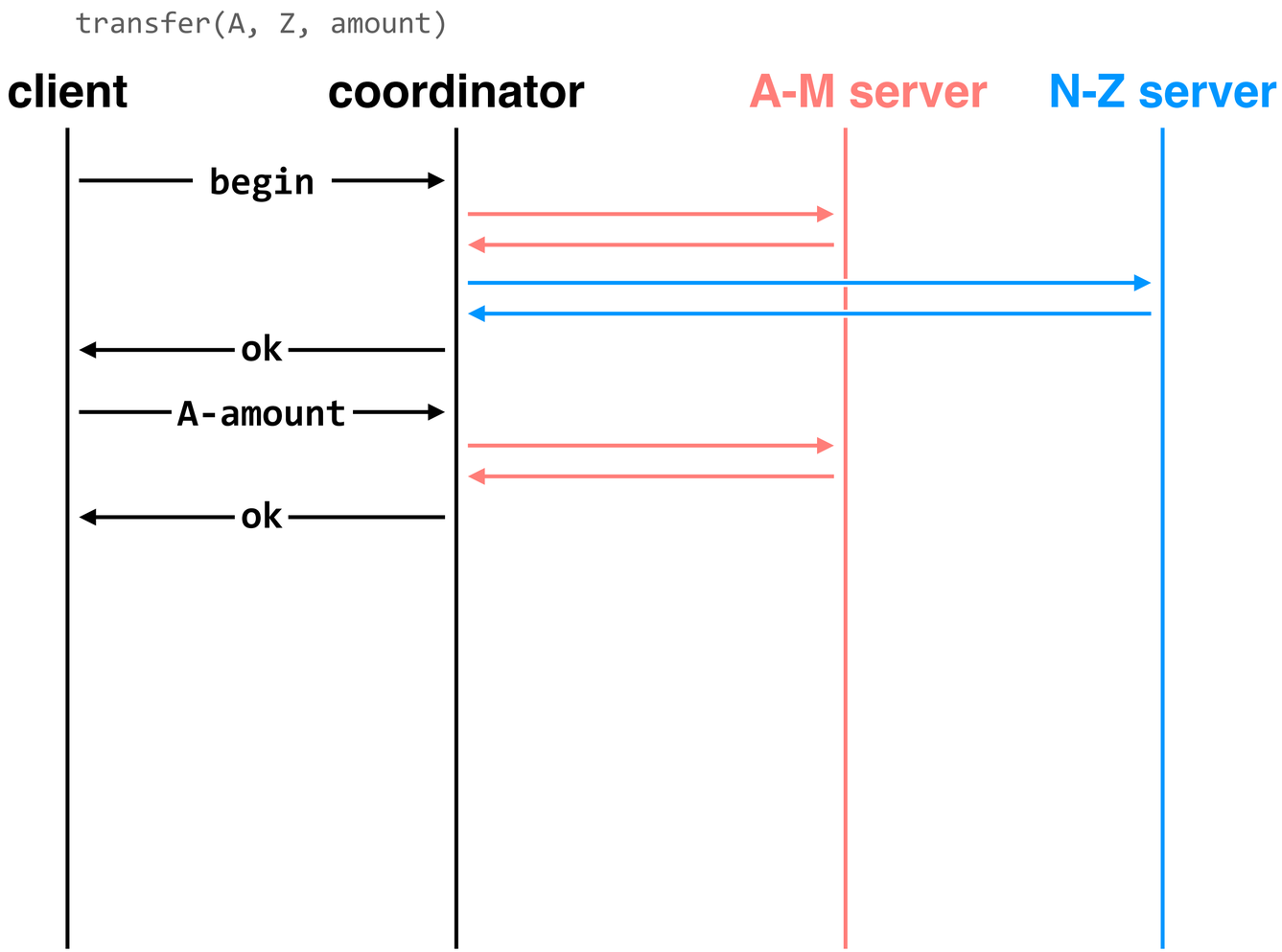
transactions across multiple machines (no failures yet)



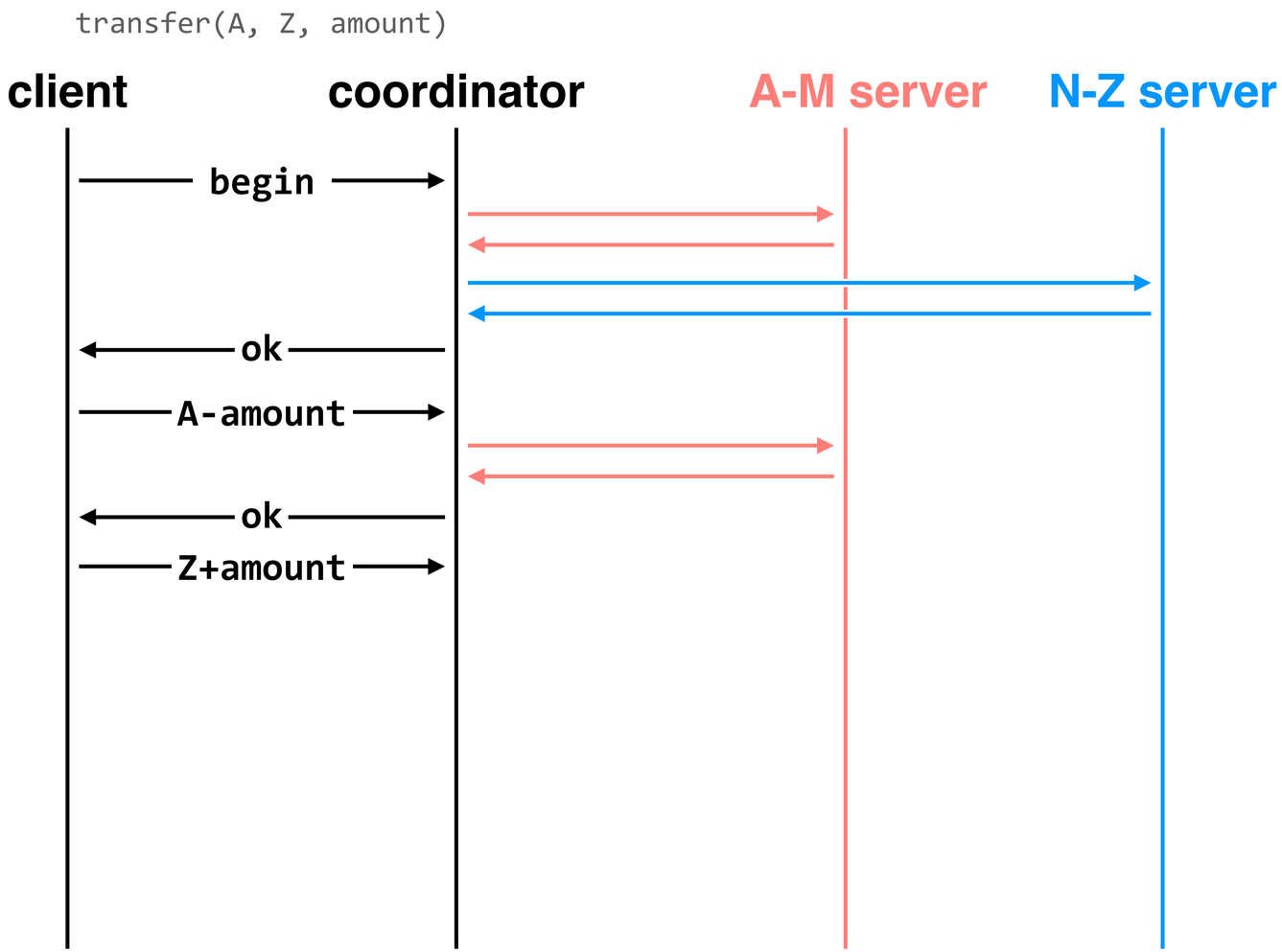
transactions across multiple machines (no failures yet)



transactions across multiple machines (no failures yet)

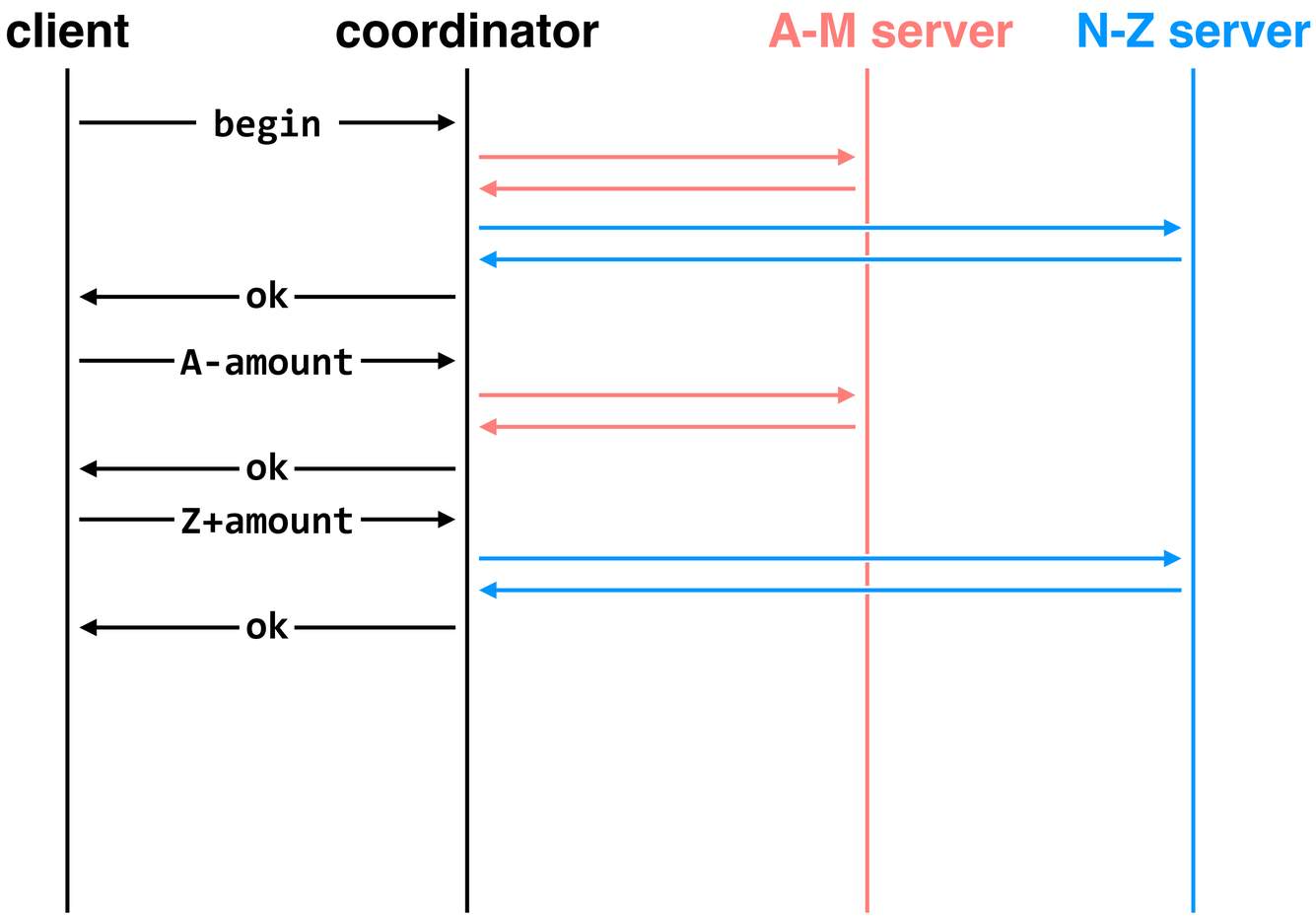


transactions across multiple machines (no failures yet)



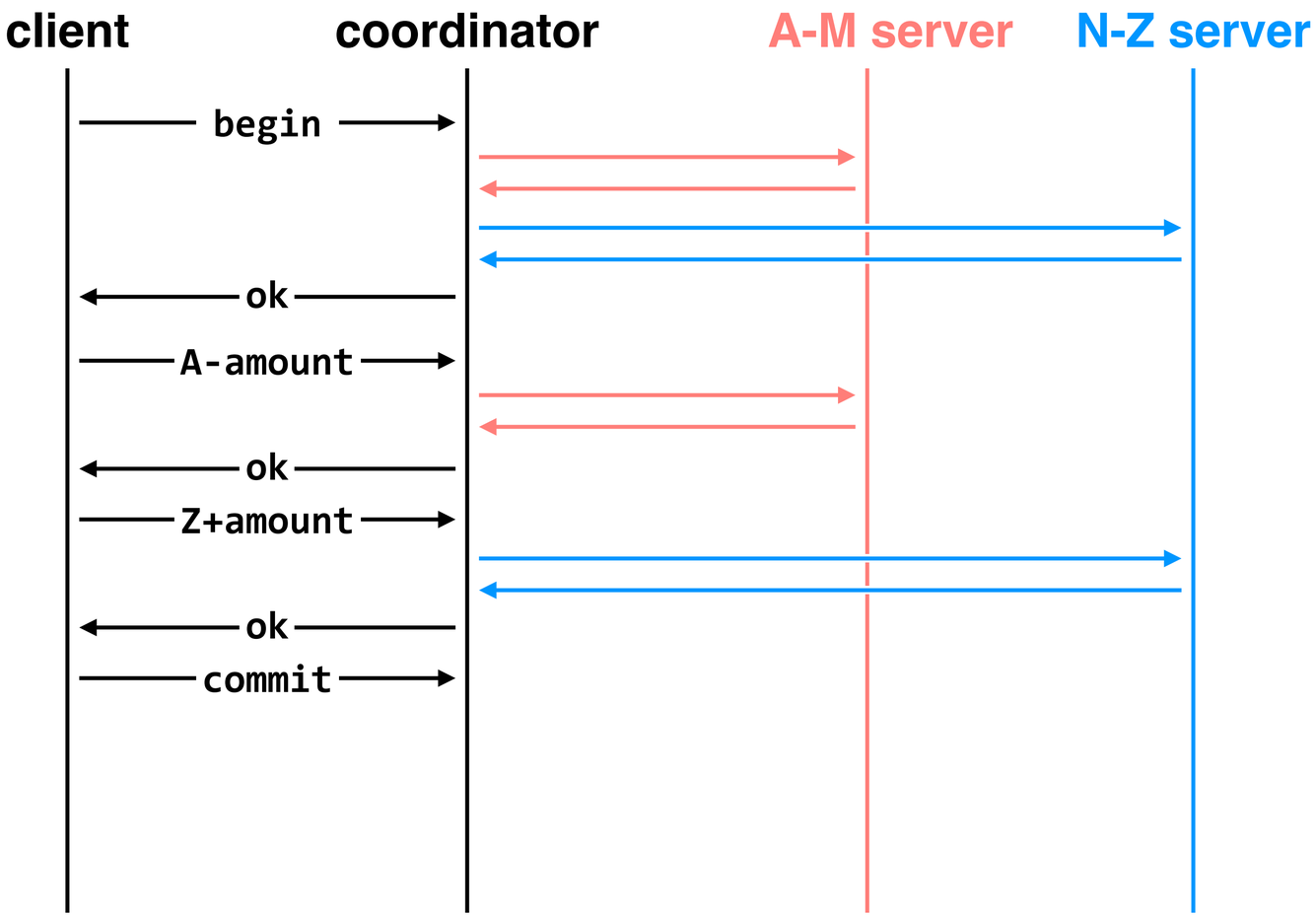
transactions across multiple machines (no failures yet)

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



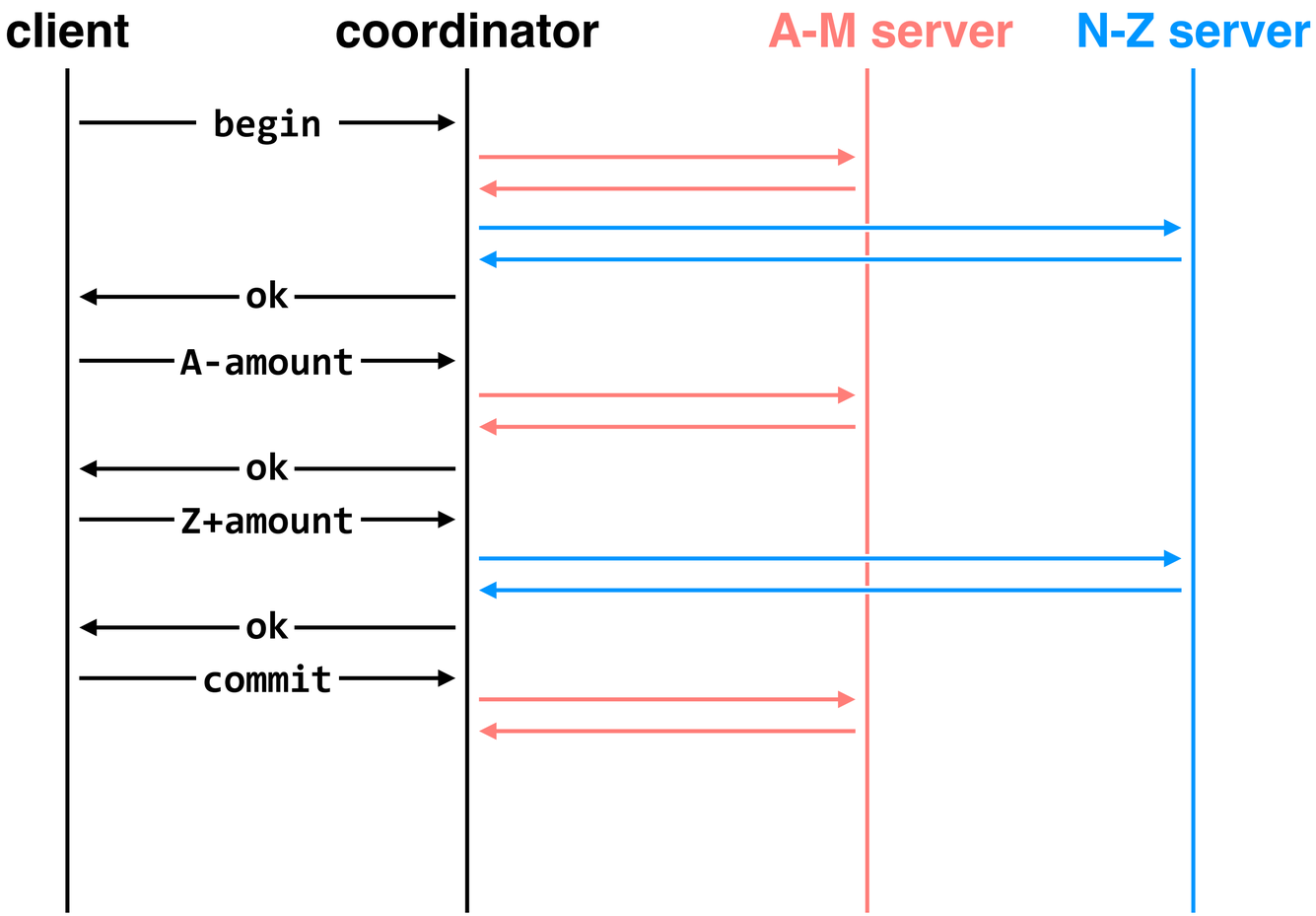
transactions across multiple machines (no failures yet)

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



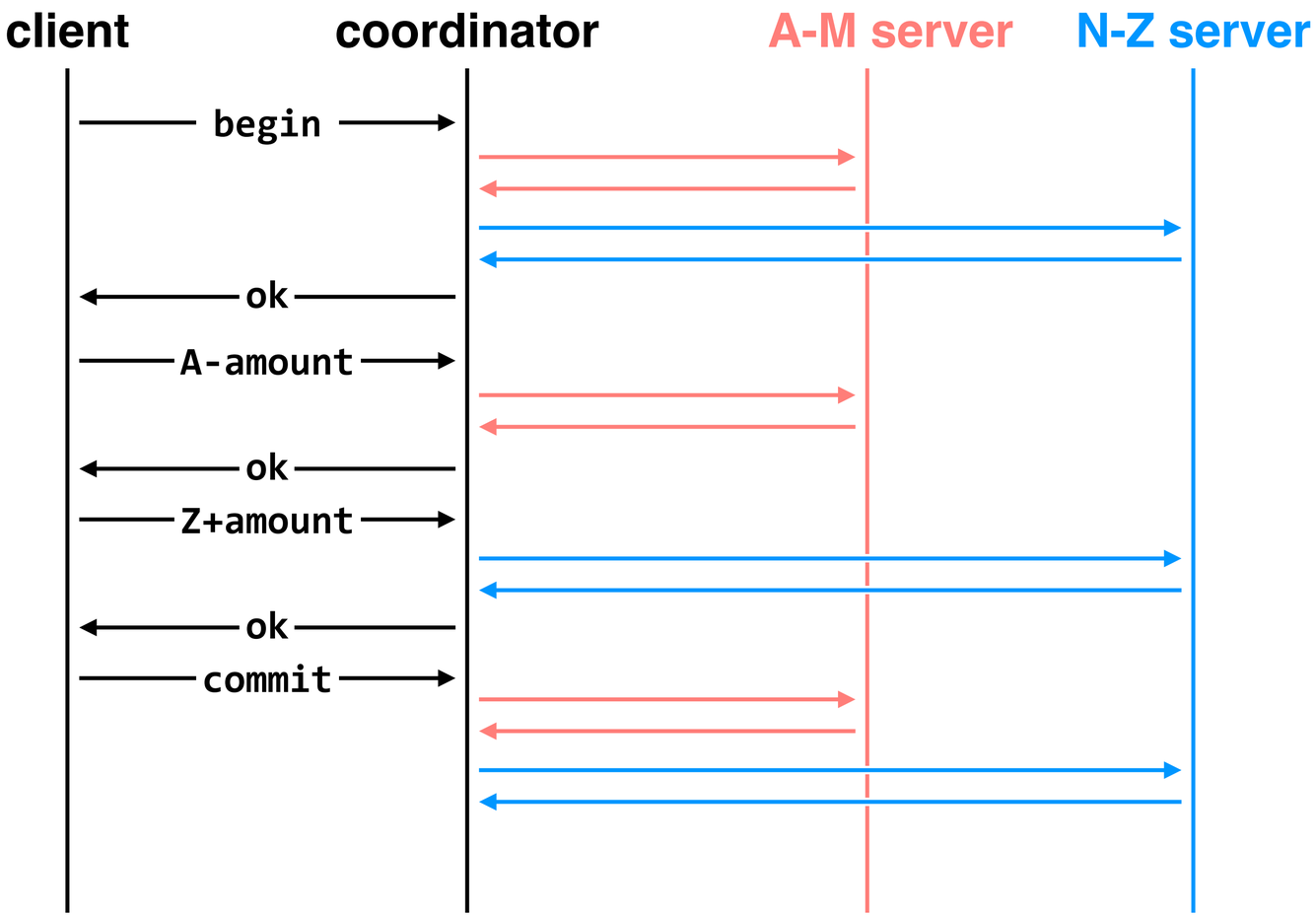
transactions across multiple machines (no failures yet)

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



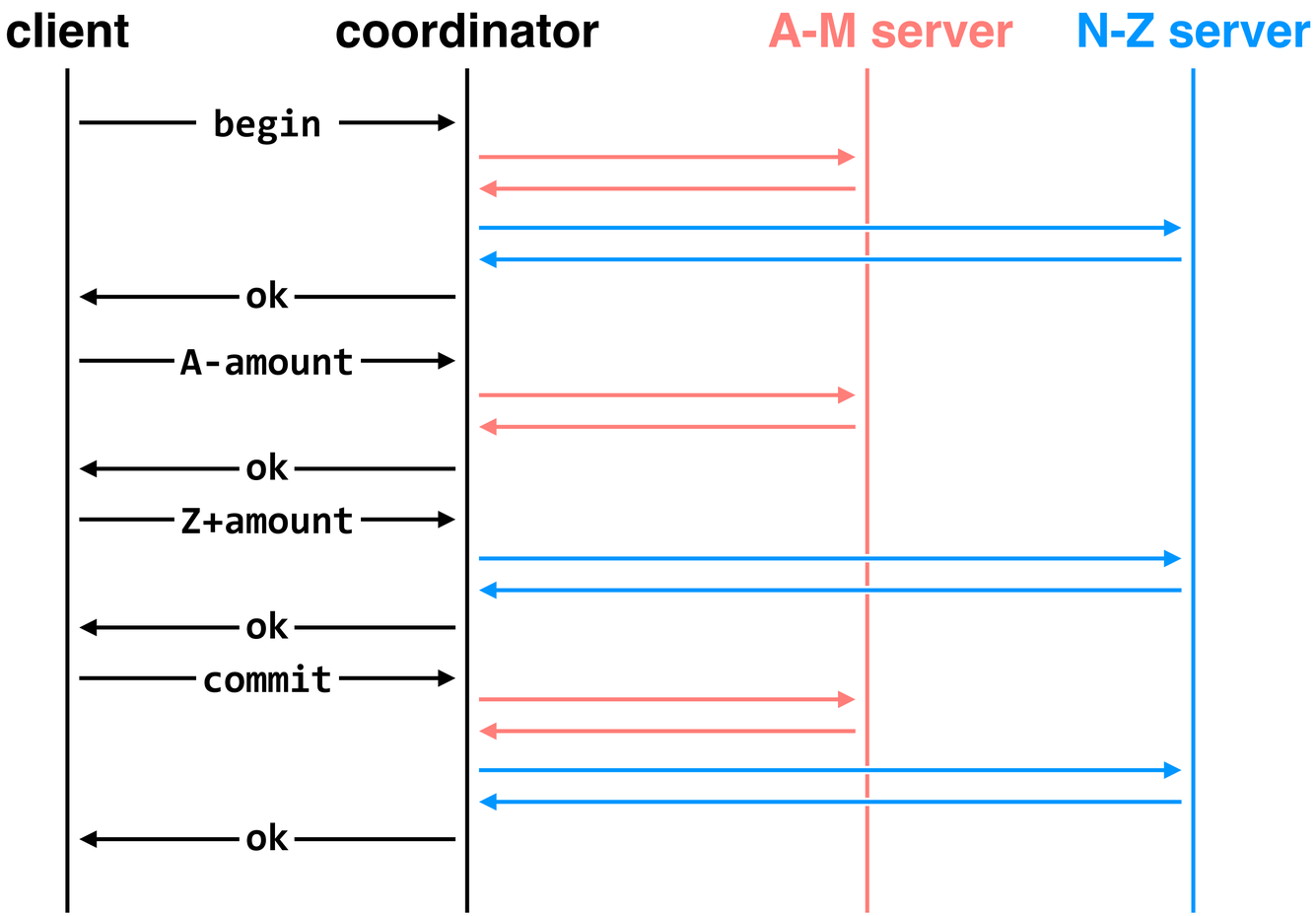
transactions across multiple machines (no failures yet)

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



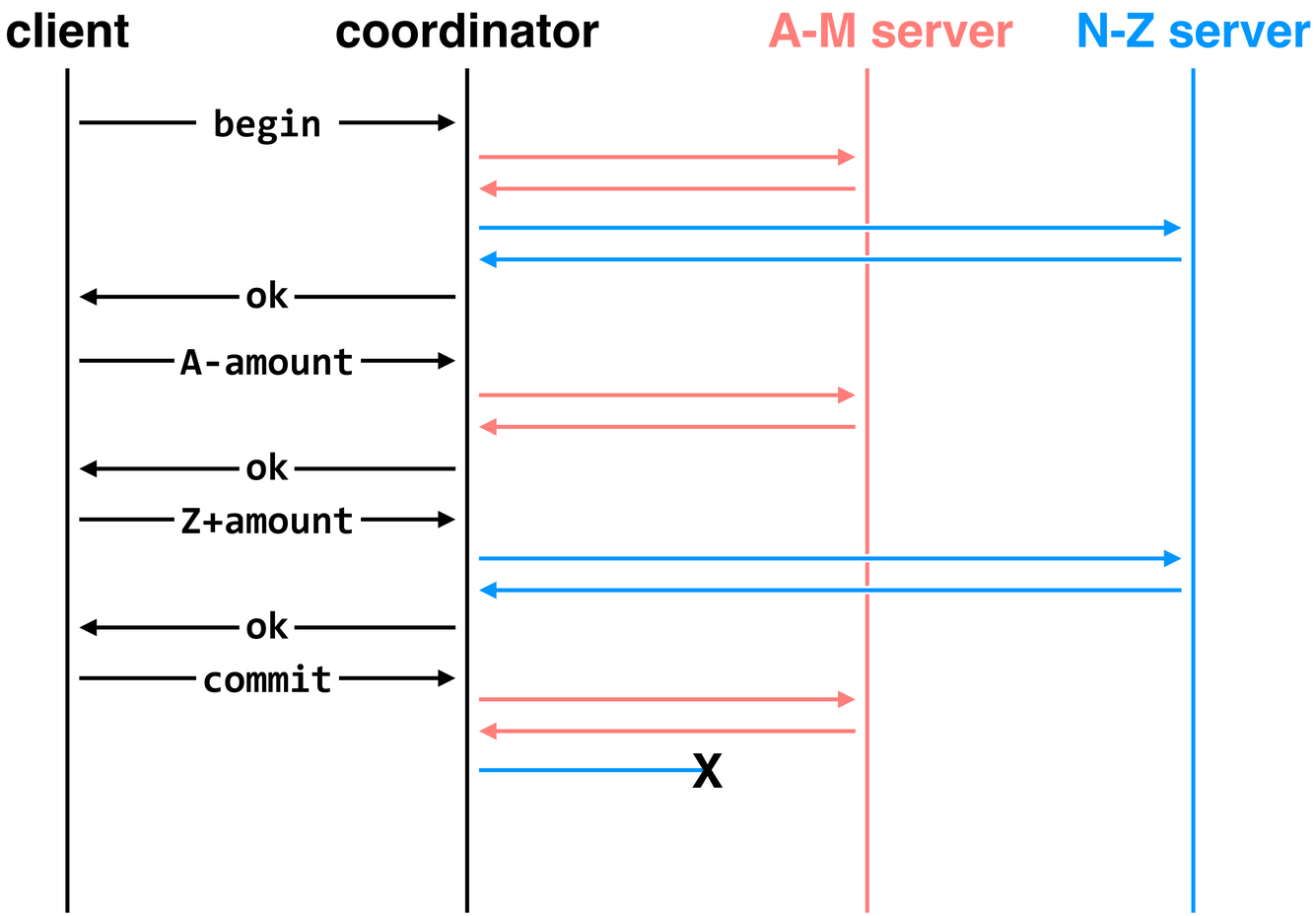
transactions across multiple machines (no failures yet)

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



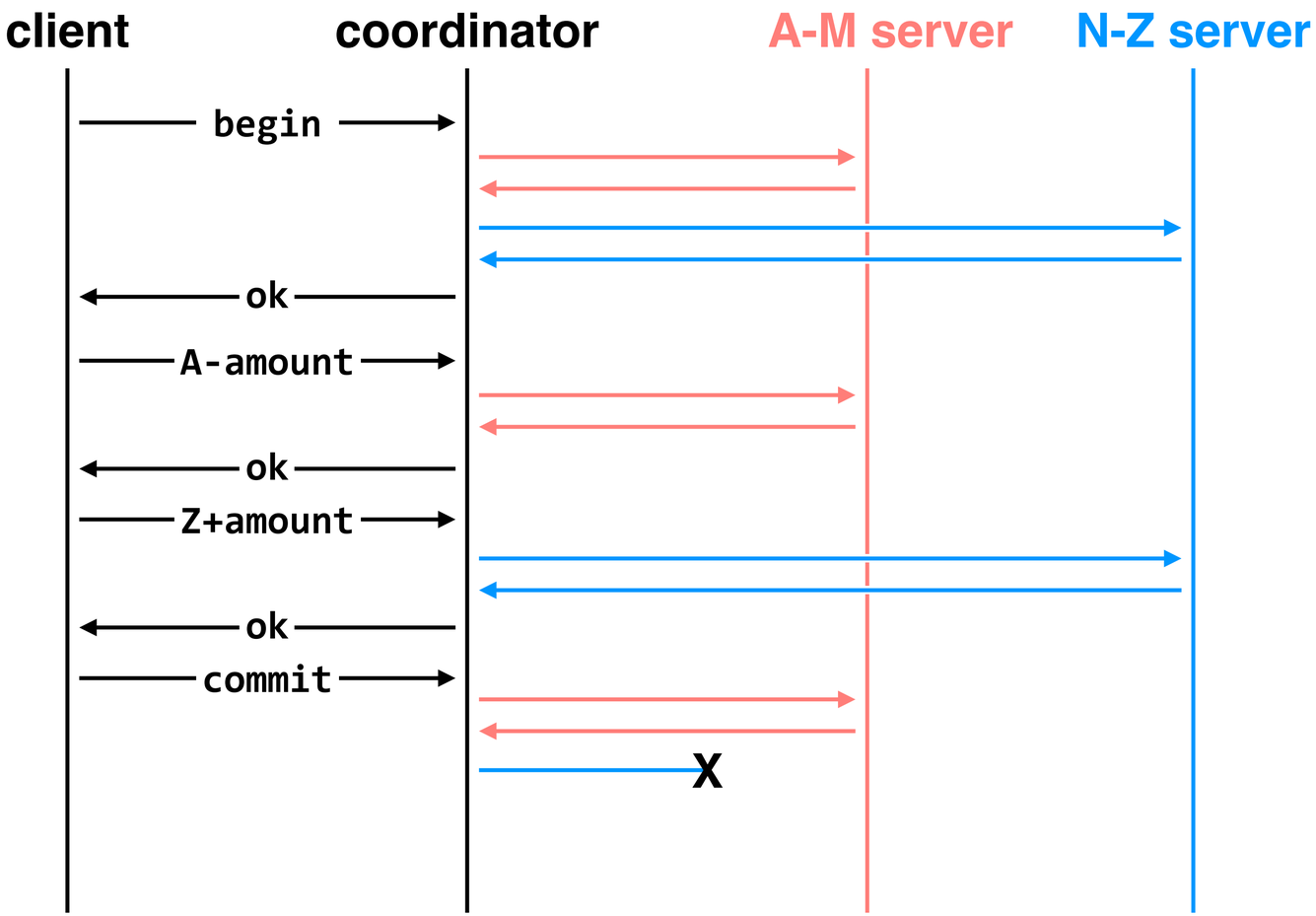
transactions across multiple machines (now with failures)

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



transactions across multiple machines (now with failures)

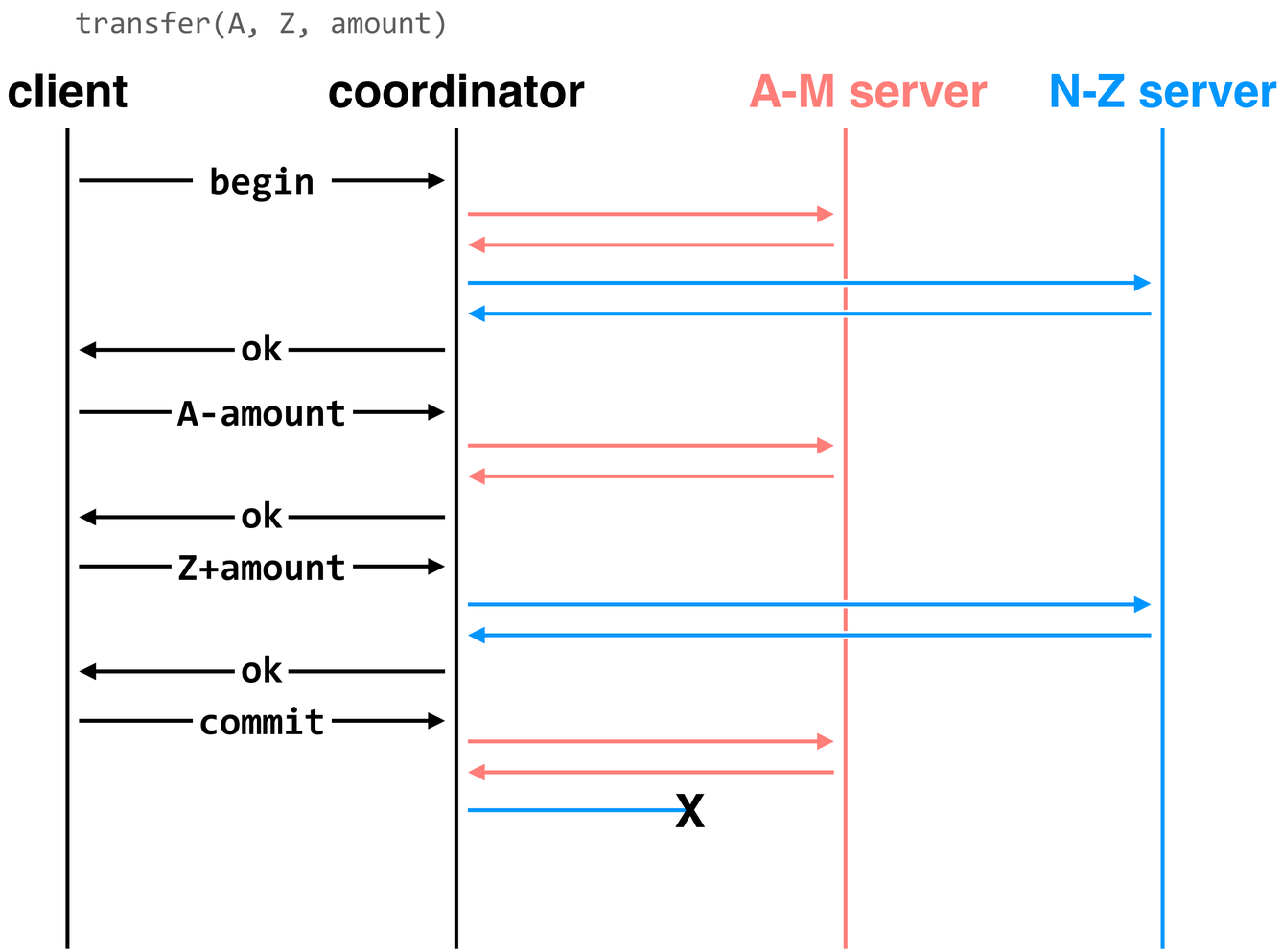
```
transfer(A, Z, amount)
```



problem: one server committed, the other did not

(we'd have a similar problem if the N-Z server crashed)

transactions across multiple machines (now with failures)

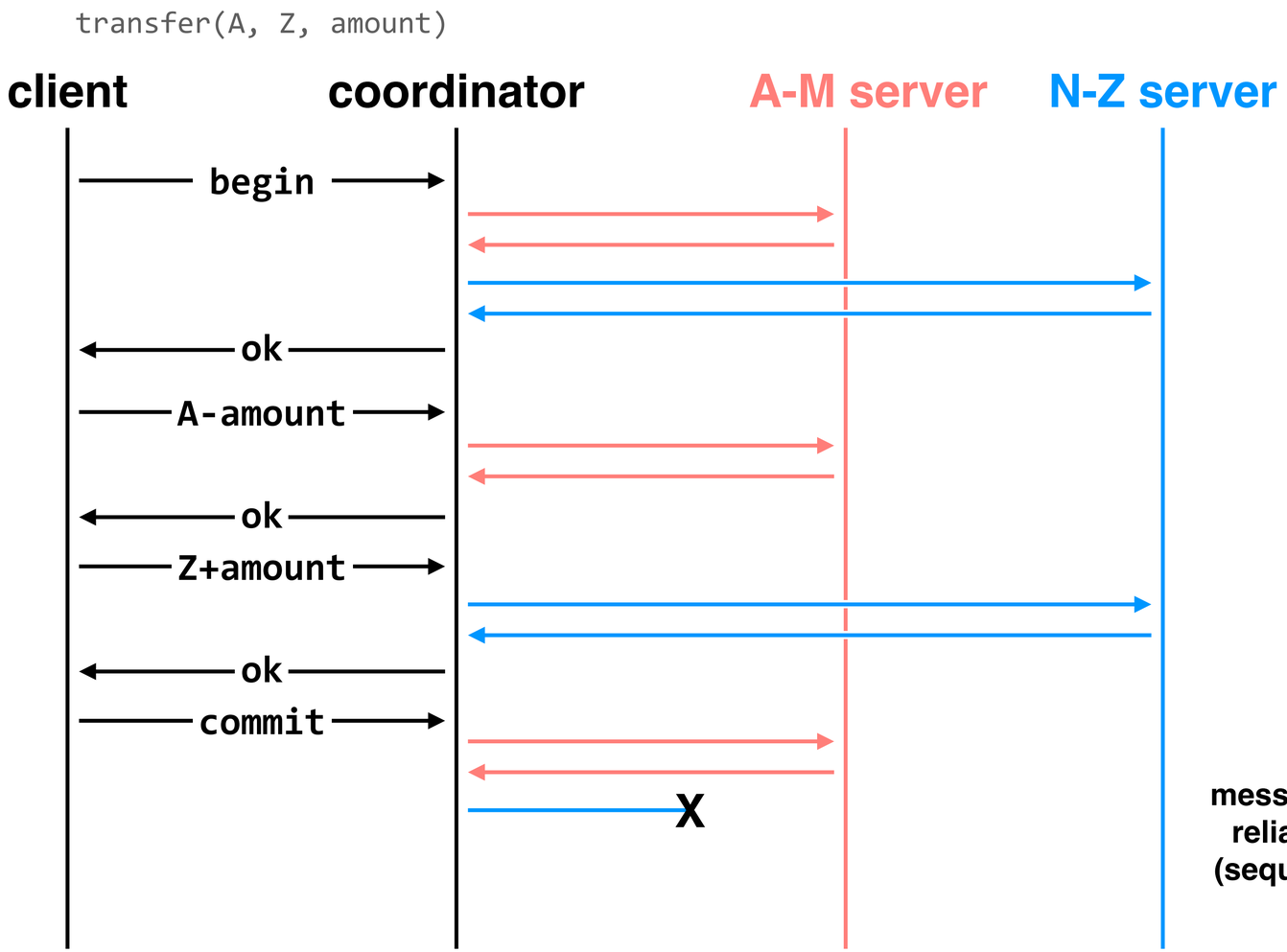


problem: one server committed, the other did not
(we'd have a similar problem if the N-Z server crashed)

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)

transactions across multiple machines (now with failures)



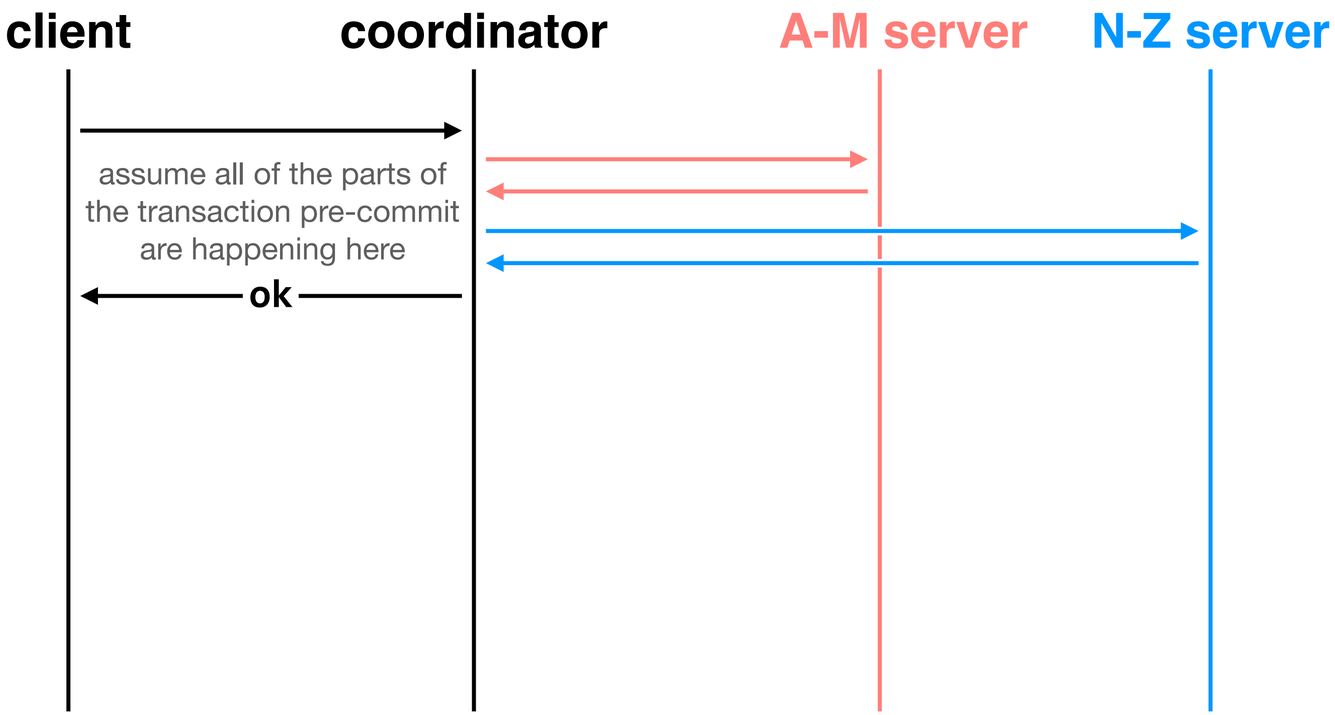
goal: develop a protocol that can provide **multi-site atomicity** in the face of all sorts of failures

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

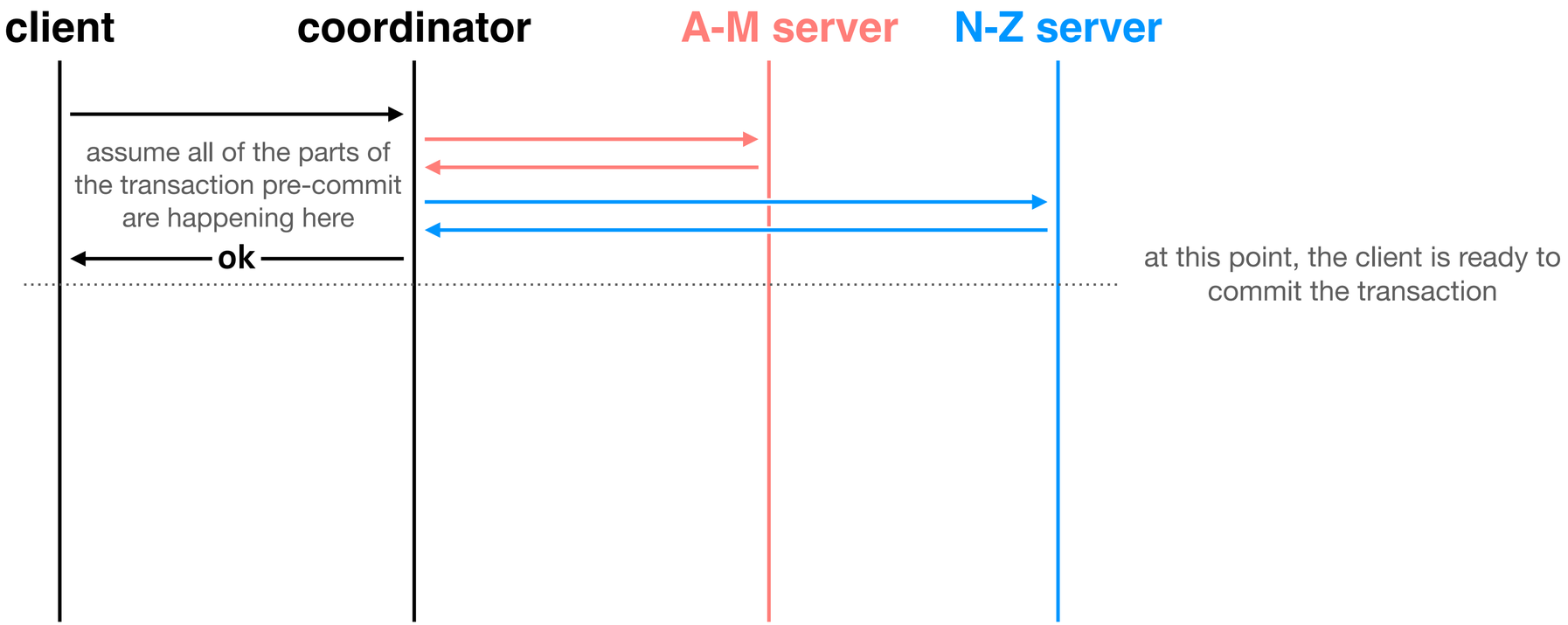
message failures solved with reliable transport protocol (sequence numbers + ACKs)

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

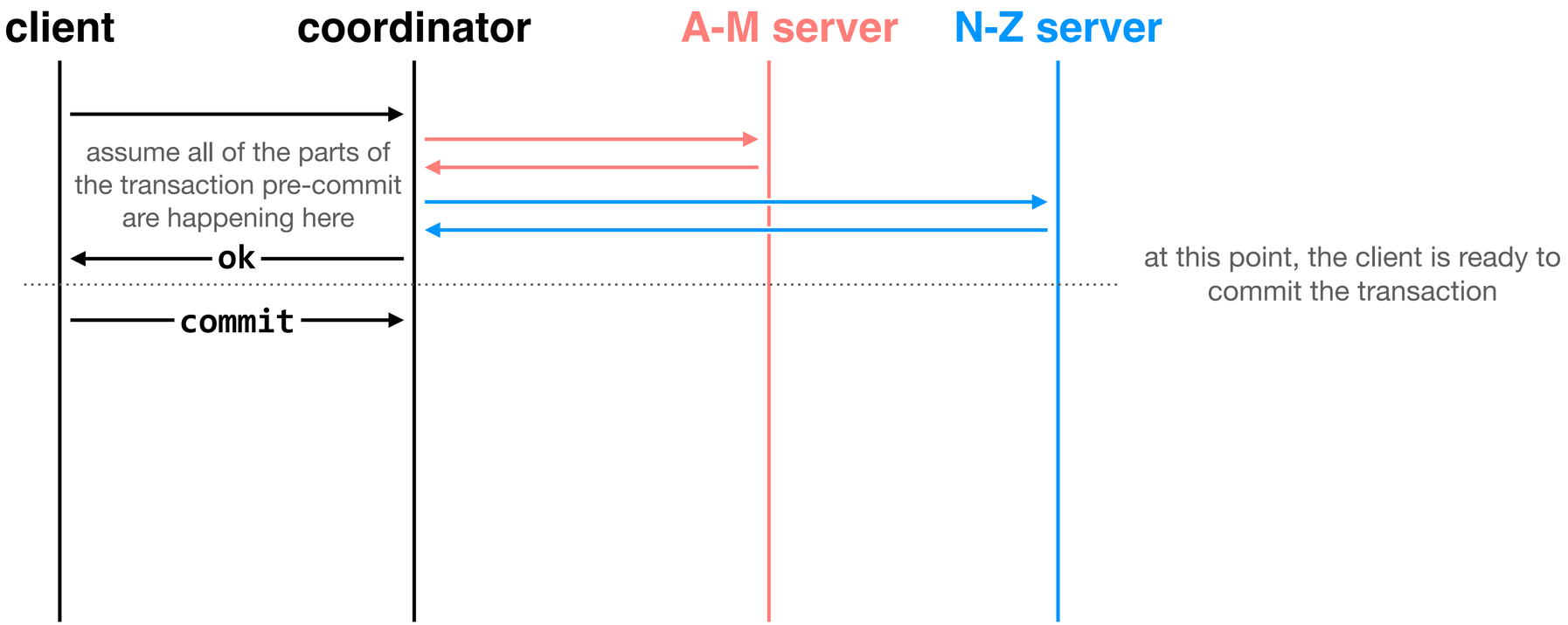
two-phase commit: nodes agree that they're ready to commit before committing



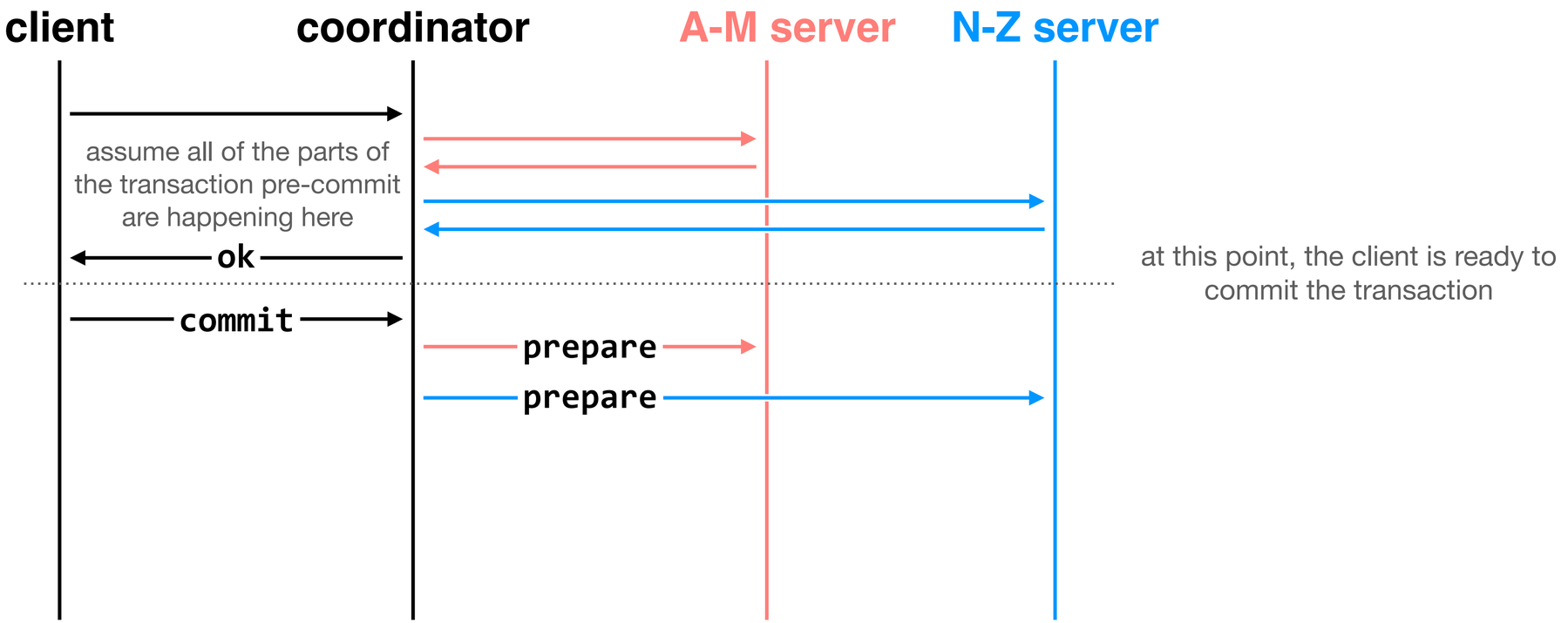
two-phase commit: nodes agree that they're ready to commit before committing



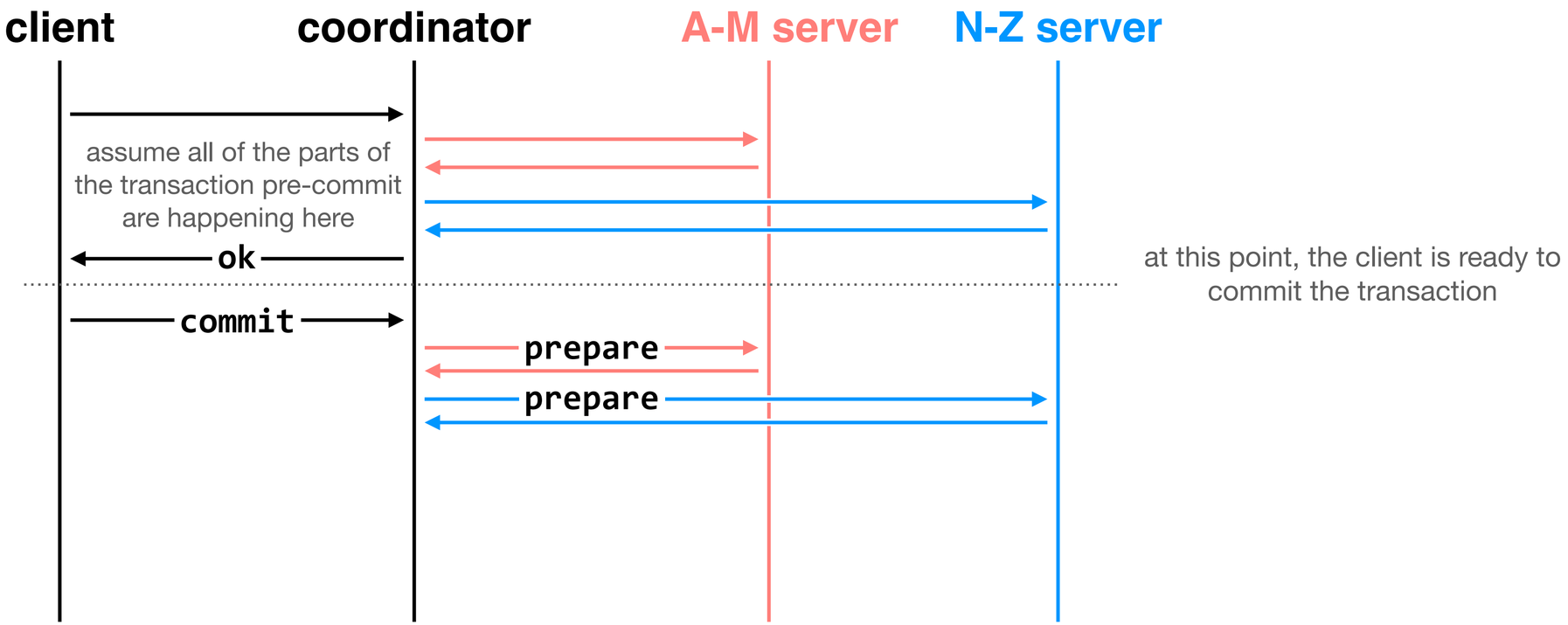
two-phase commit: nodes agree that they're ready to commit before committing



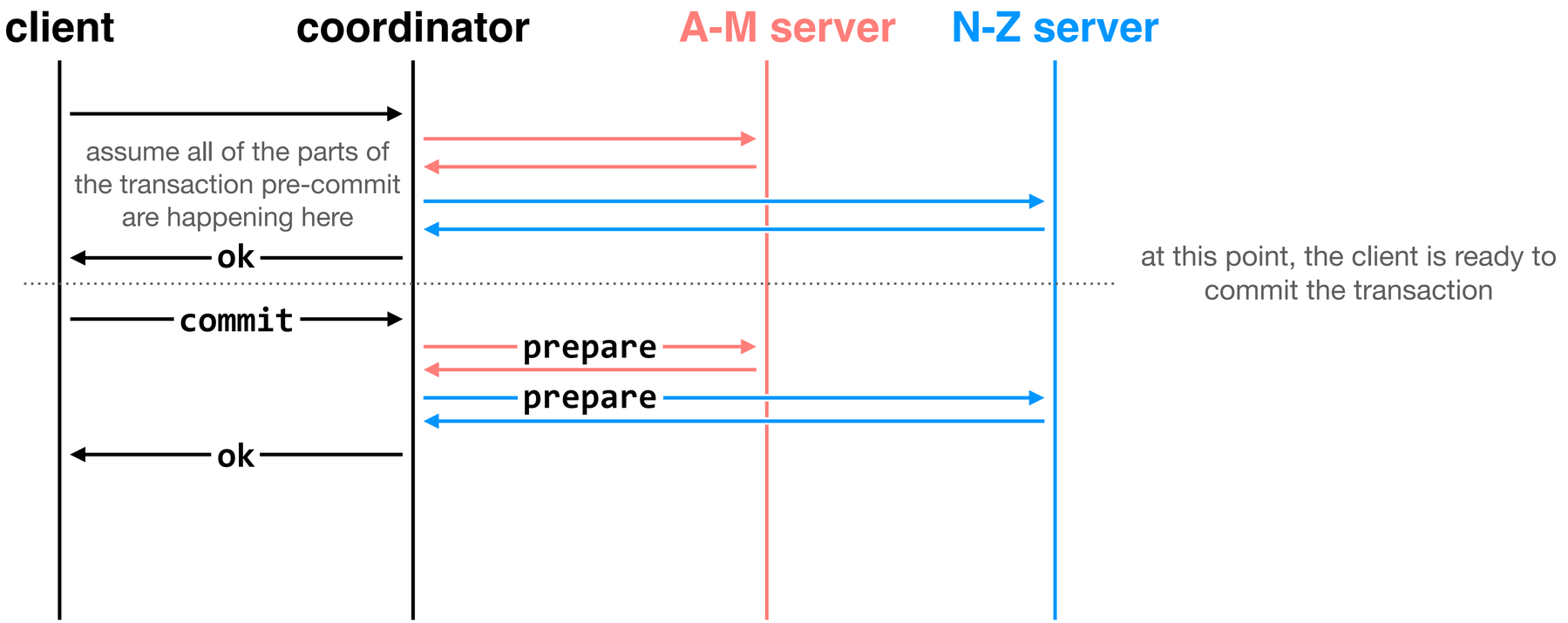
two-phase commit: nodes agree that they're ready to commit before committing



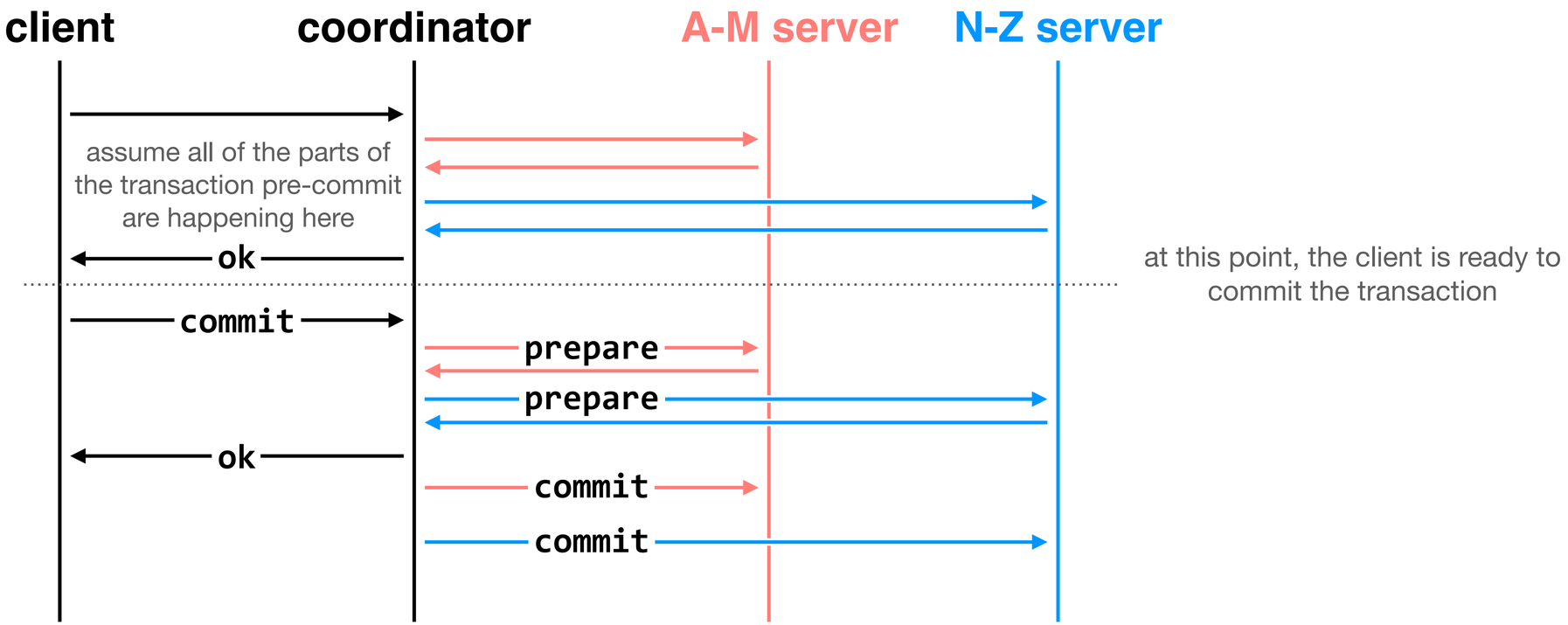
two-phase commit: nodes agree that they're ready to commit before committing



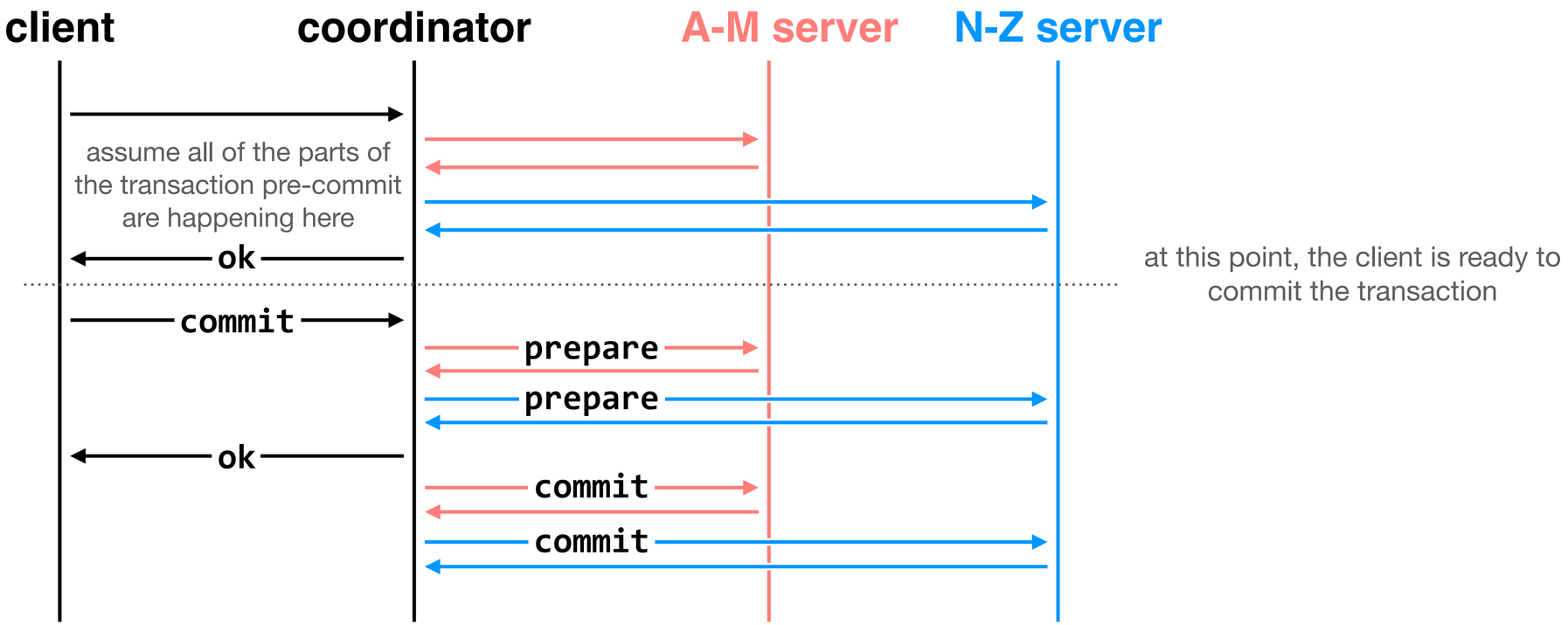
two-phase commit: nodes agree that they're ready to commit before committing



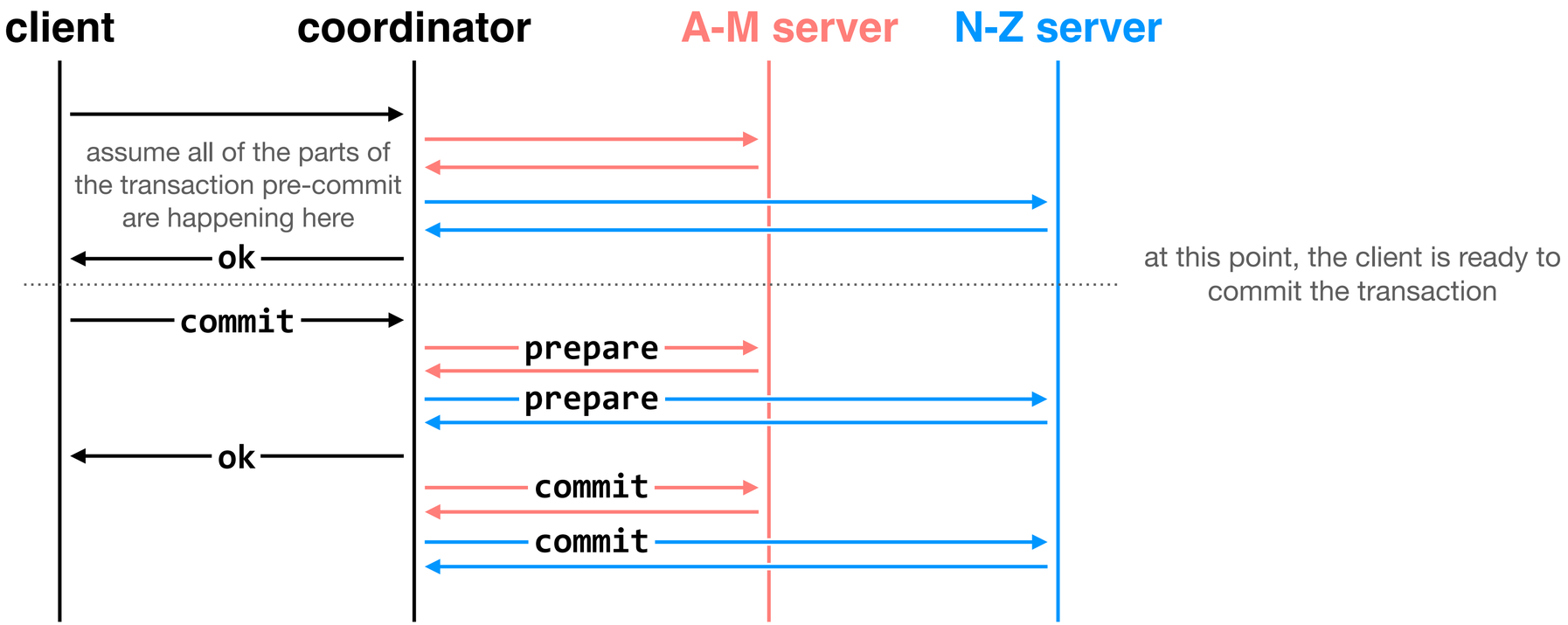
two-phase commit: nodes agree that they're ready to commit before committing



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



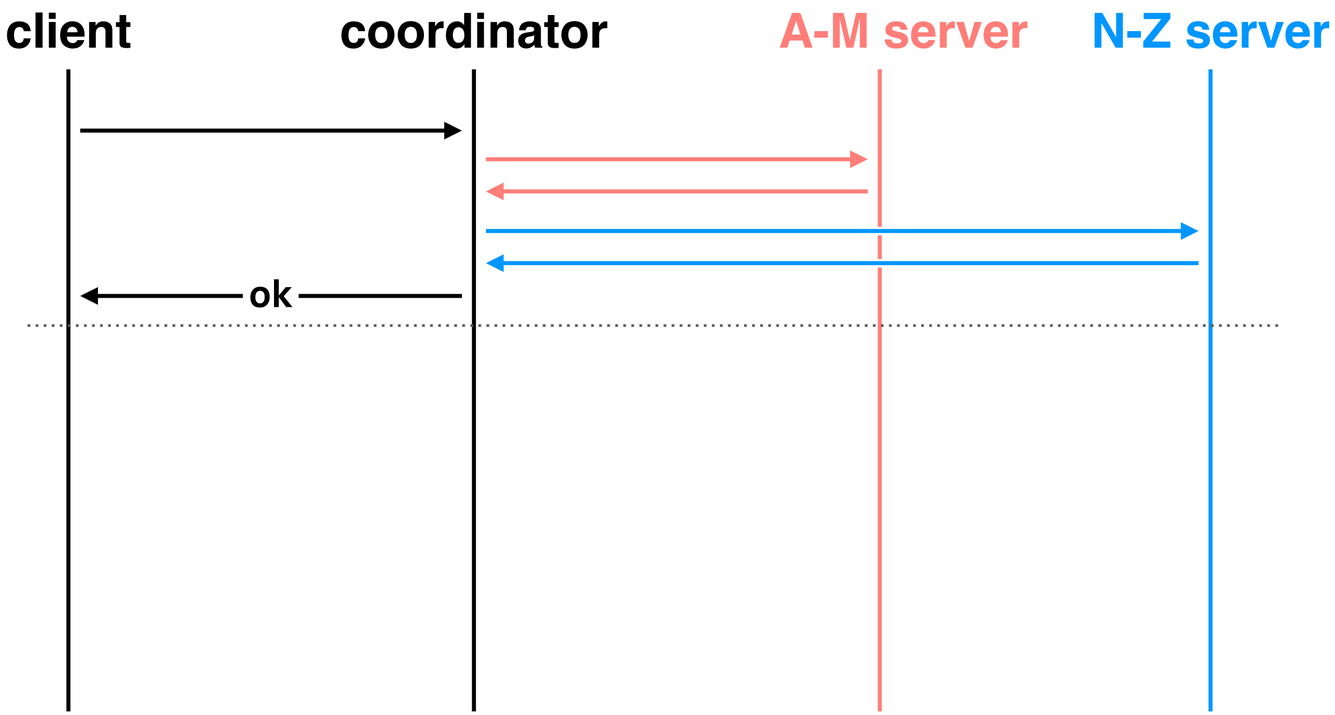
two-phase commit: nodes agree that they're ready to commit before committing



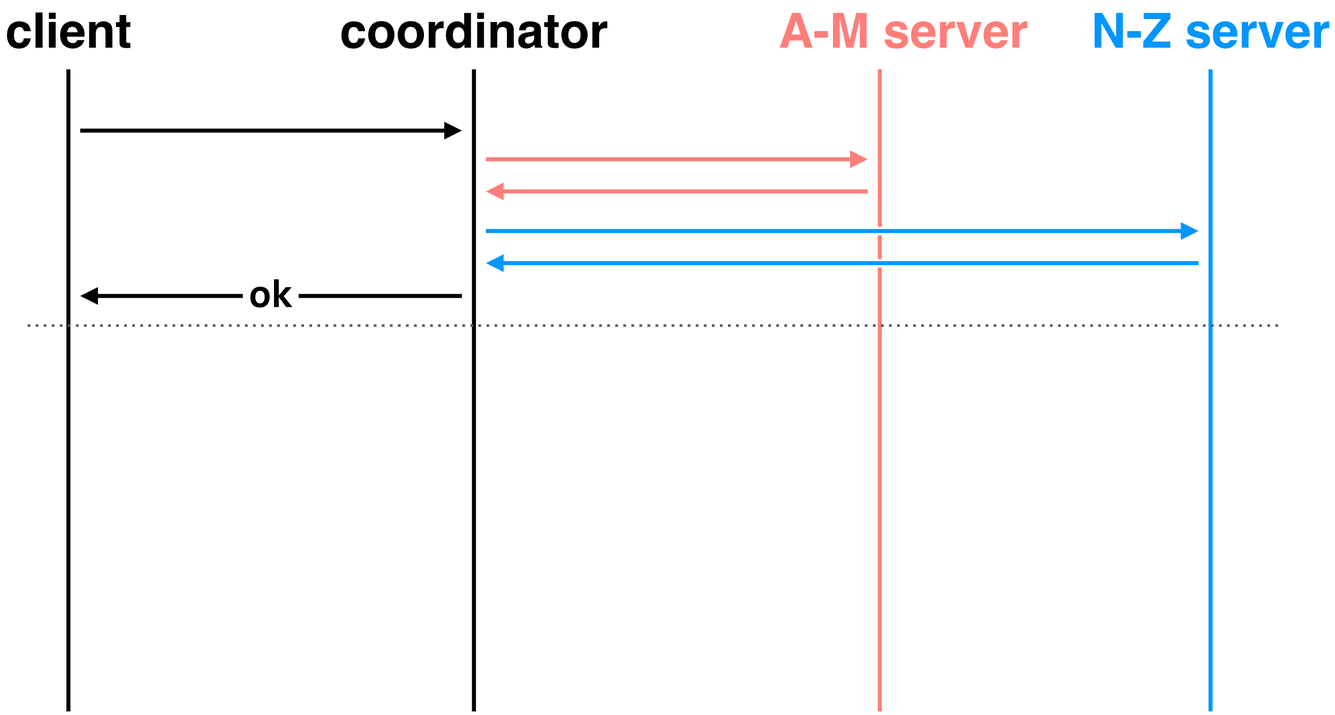
to understand why this protocol provides atomicity, we'll start by examining how it behaves under a variety of different types of failures

we will eventually understand why it requires two phases

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

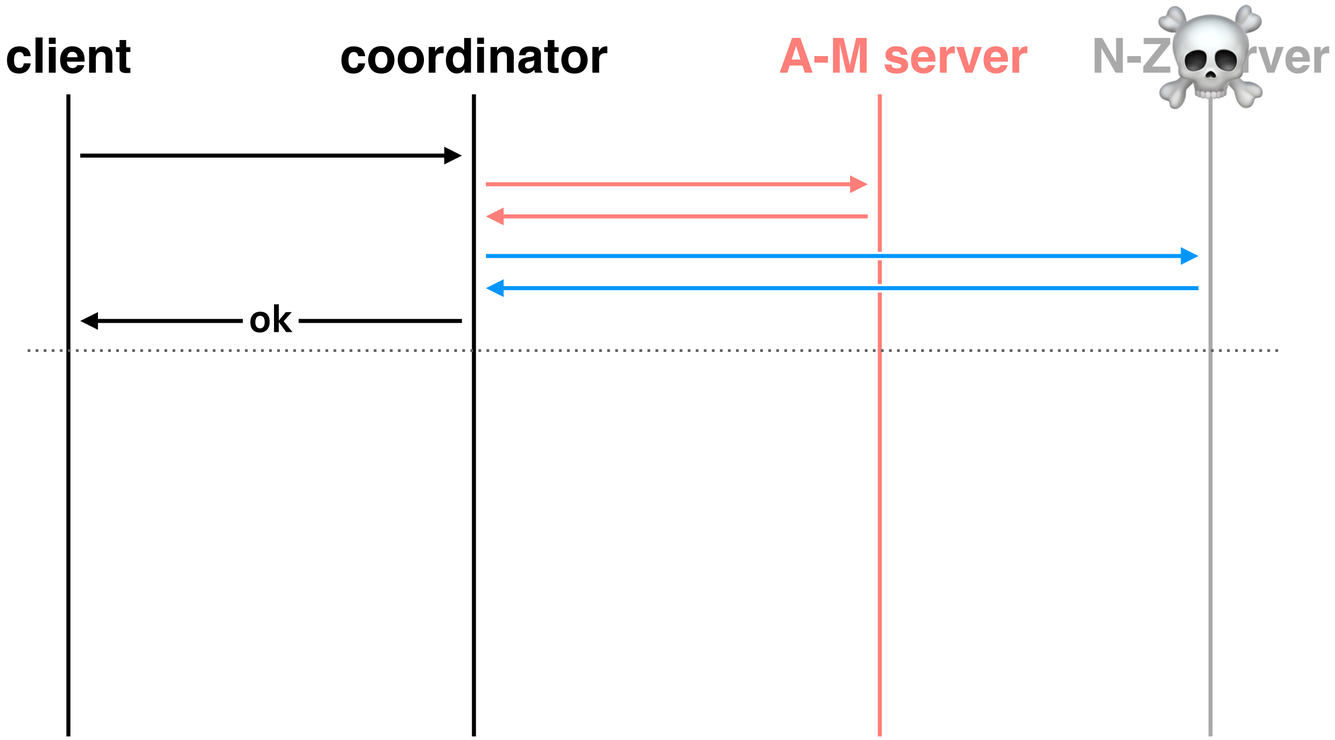


two-phase commit: nodes agree that they're ready to commit before committing



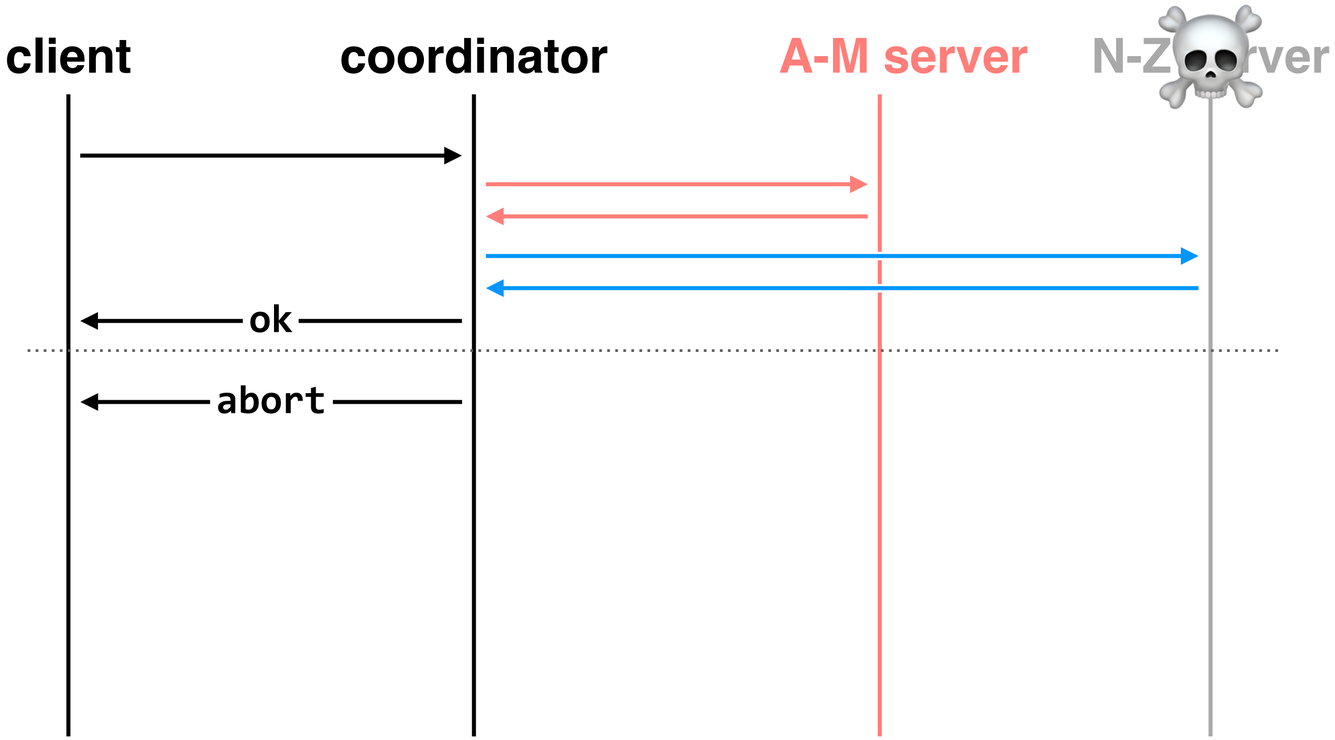
`worker failure before prepare phase`

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



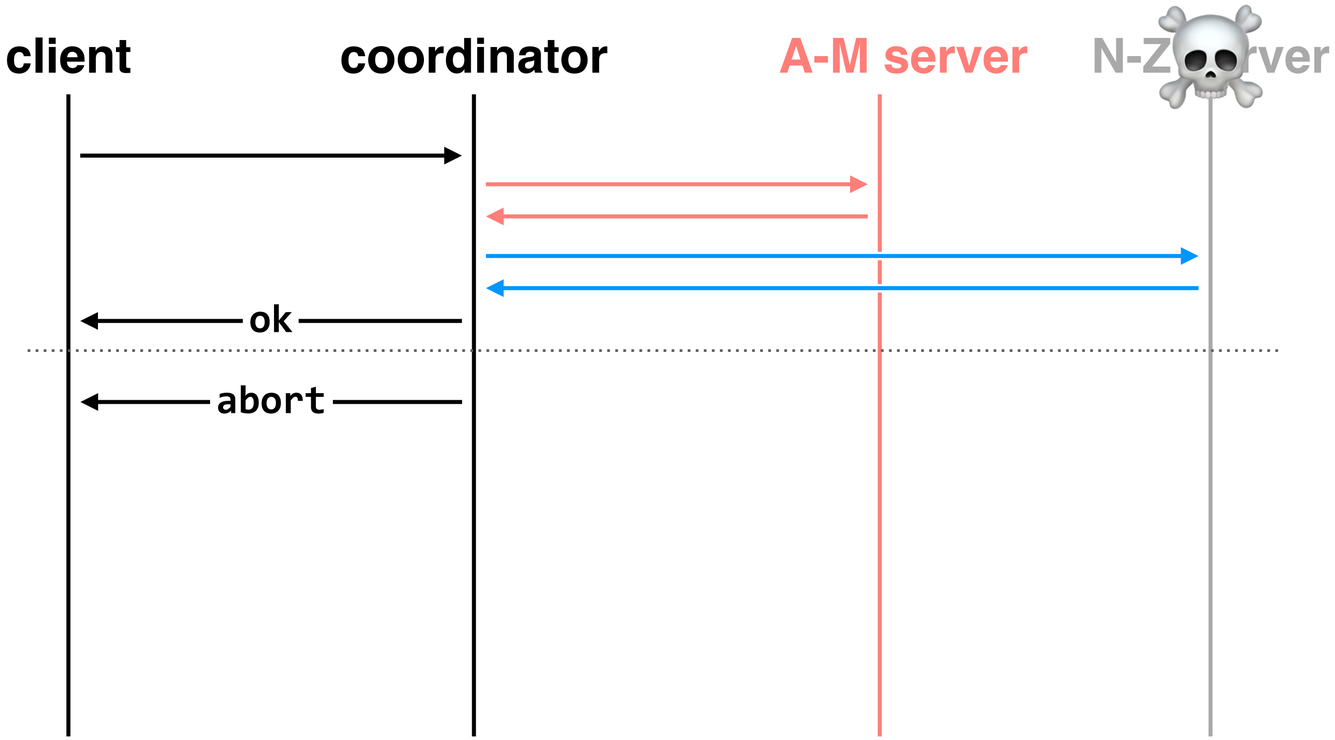
`worker failure before prepare phase`

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



worker failure before prepare phase

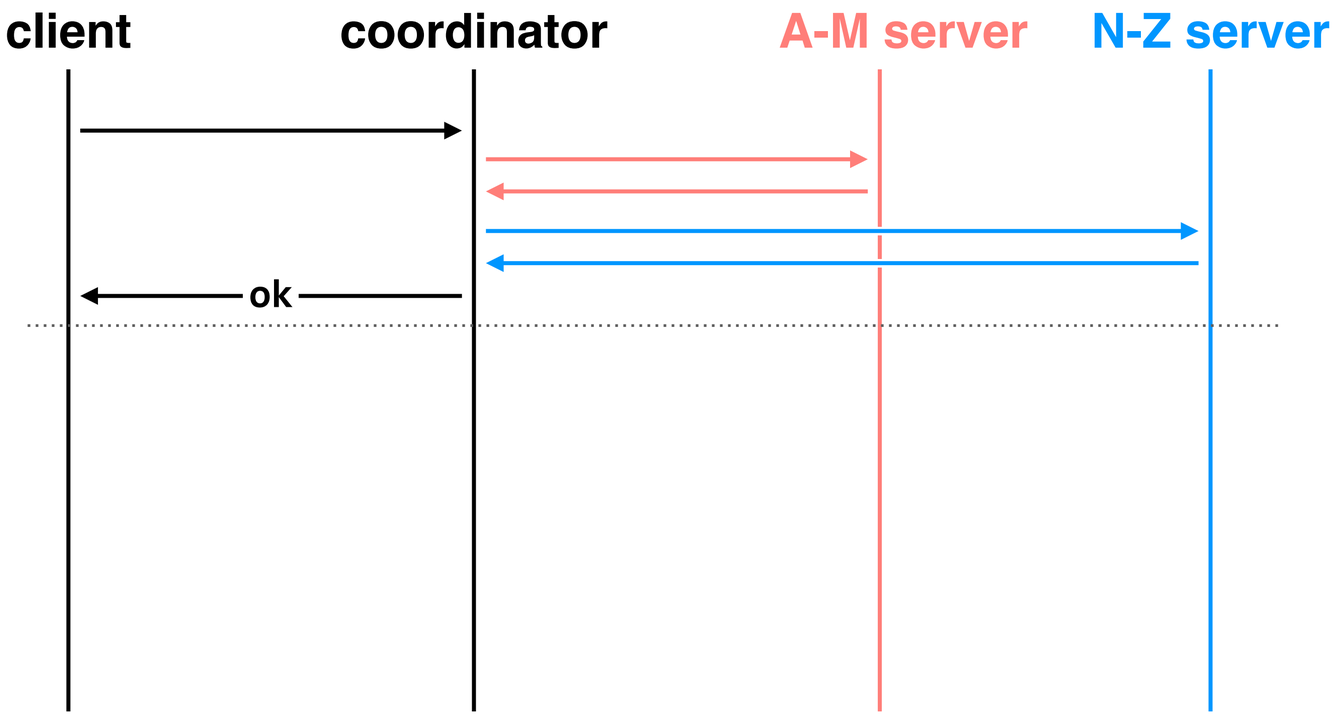
two-phase commit: nodes agree that they're ready to commit before committing



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

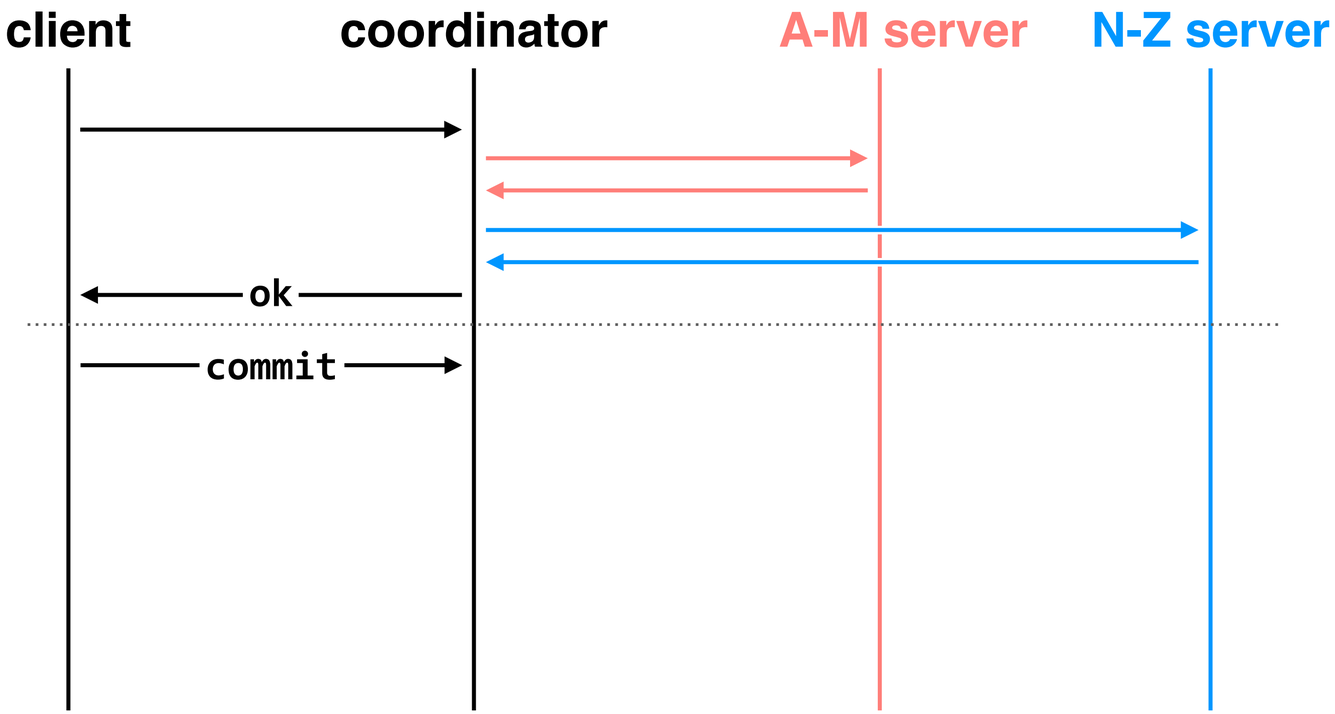
two-phase commit: nodes agree that they're ready to commit before committing



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

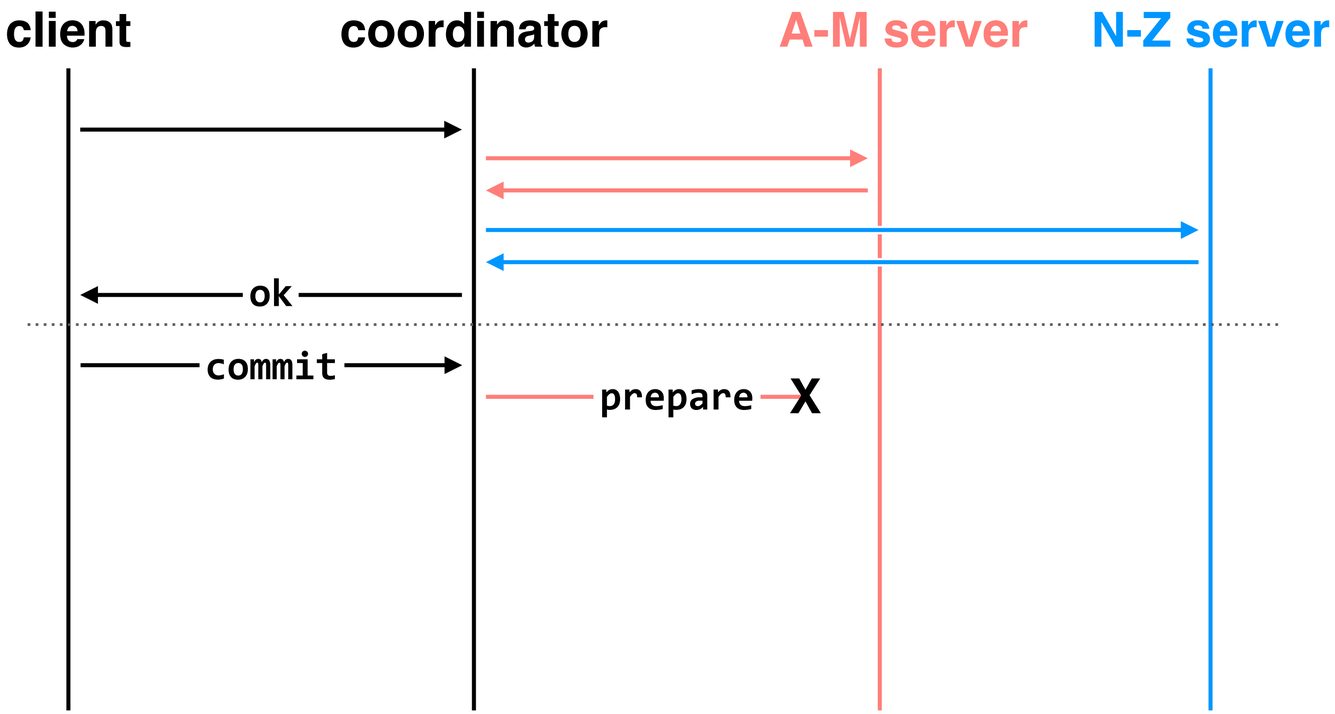
two-phase commit: nodes agree that they're ready to commit before committing



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

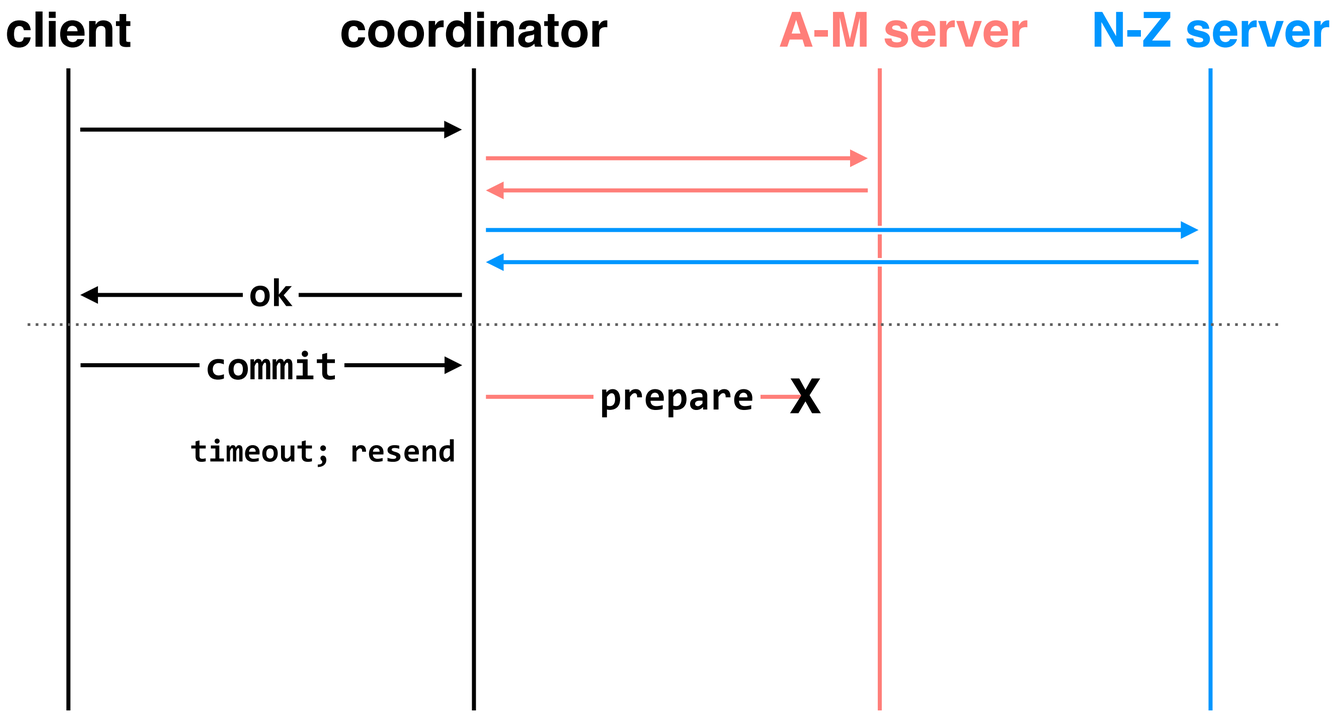
two-phase commit: nodes agree that they're ready to commit before committing



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

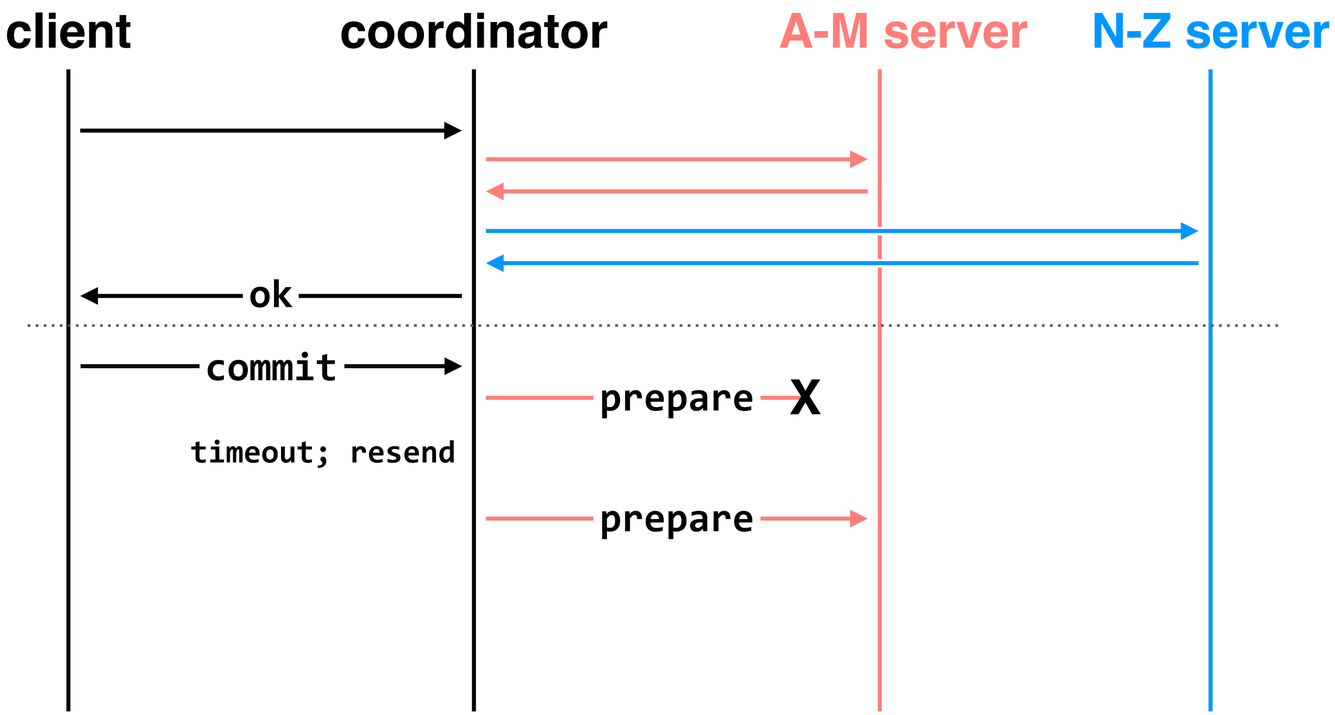
two-phase commit: nodes agree that they're ready to commit before committing



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

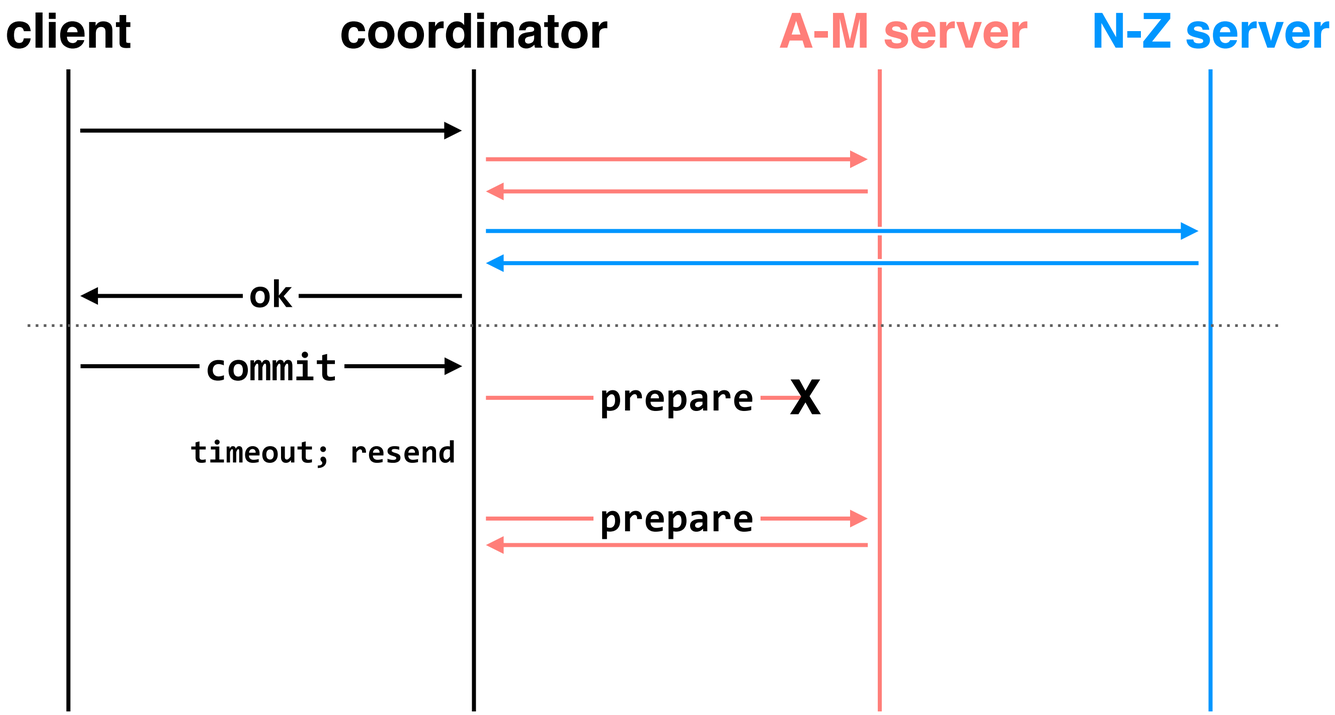
two-phase commit: nodes agree that they're ready to commit before committing



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

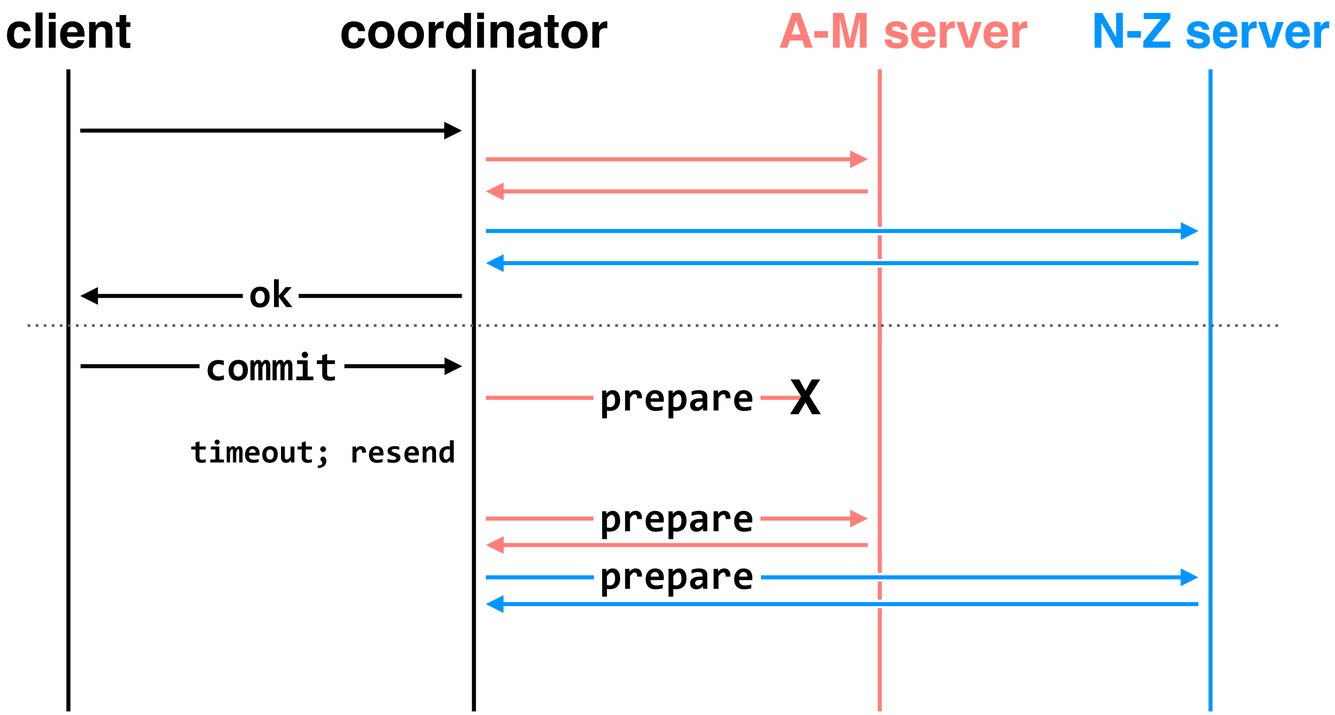
two-phase commit: nodes agree that they're ready to commit before committing



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

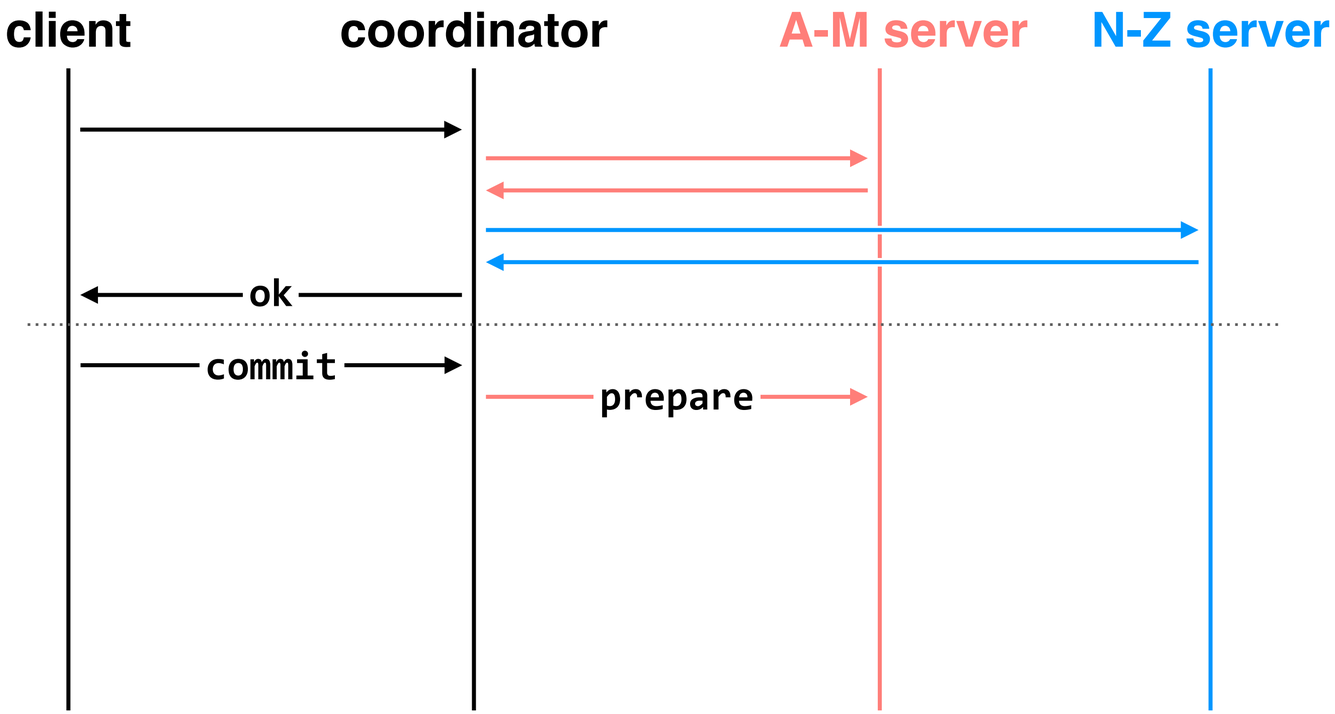
two-phase commit: nodes agree that they're ready to commit before committing



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

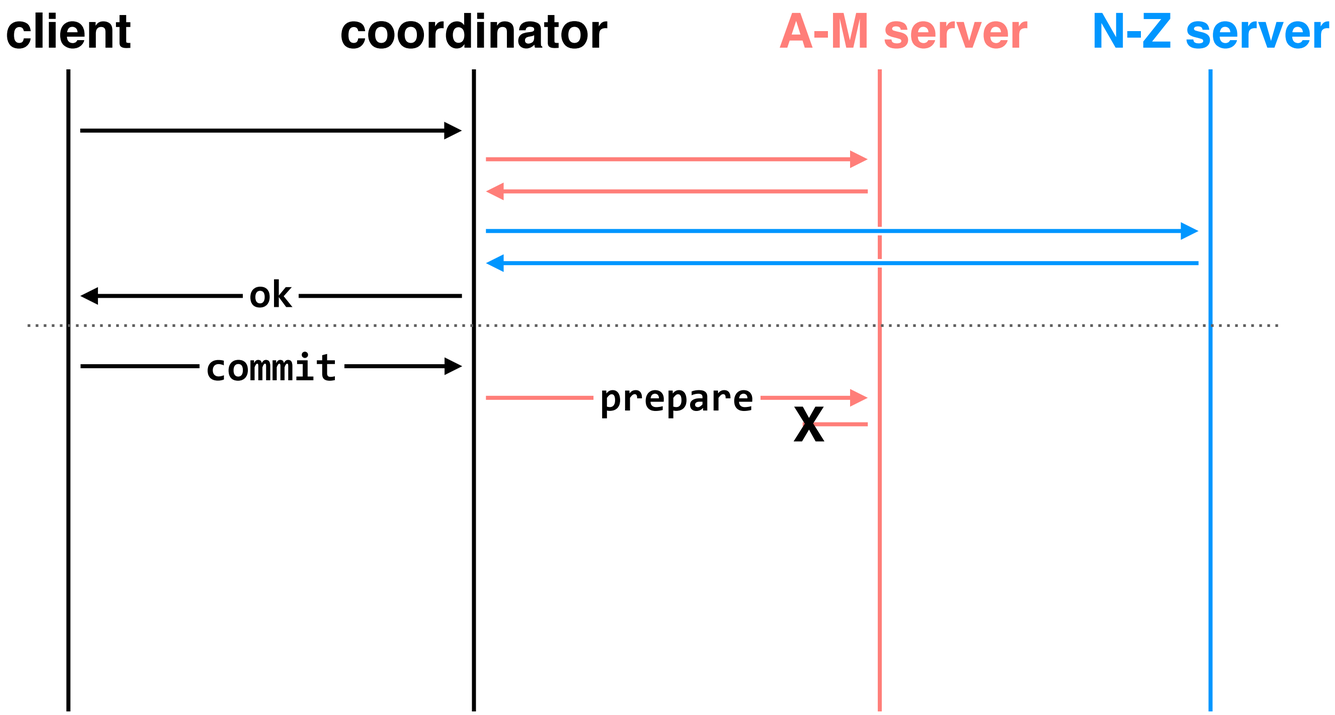
two-phase commit: nodes agree that they're ready to commit before committing



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

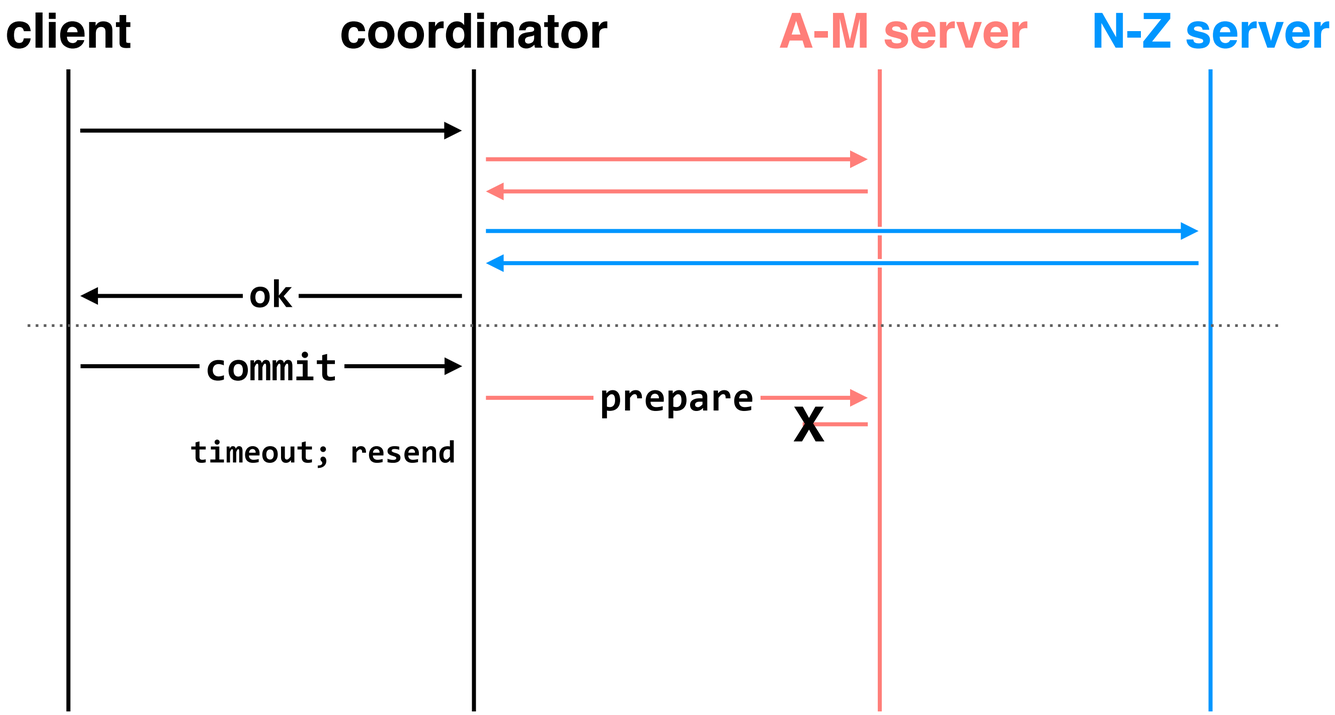
two-phase commit: nodes agree that they're ready to commit before committing



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

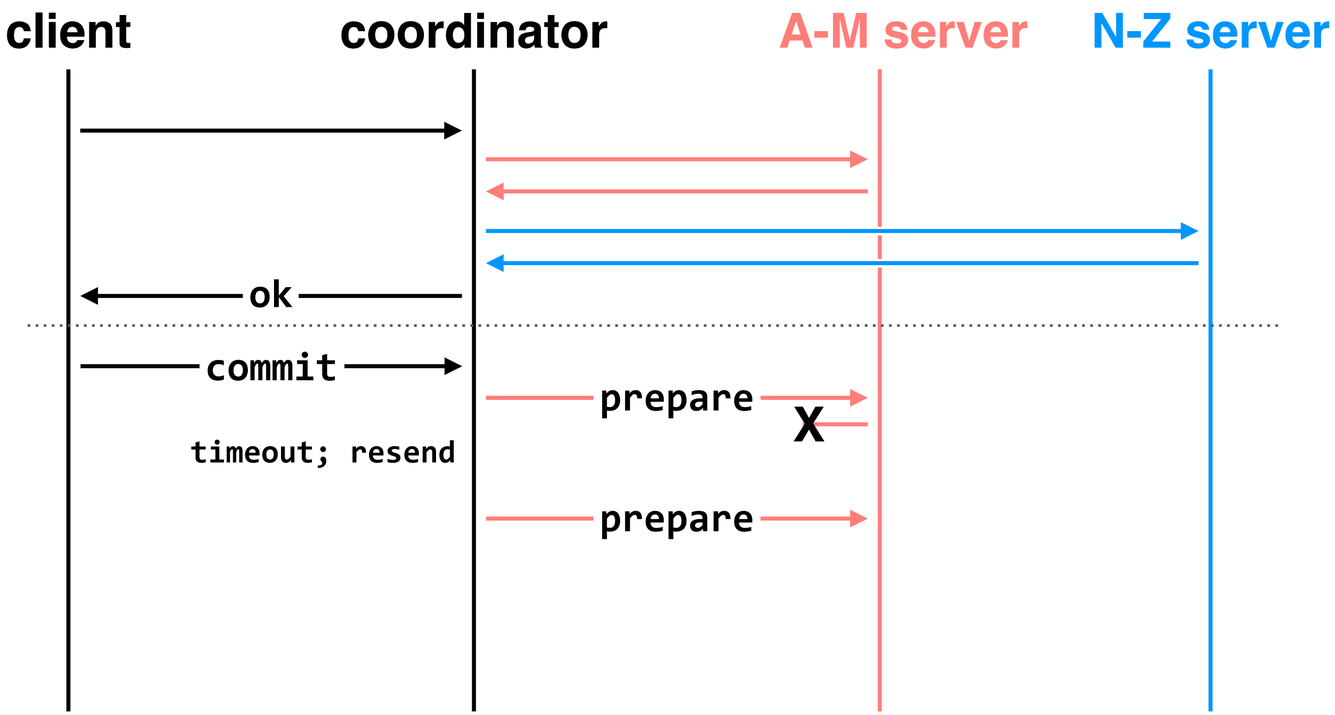
two-phase commit: nodes agree that they're ready to commit before committing



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

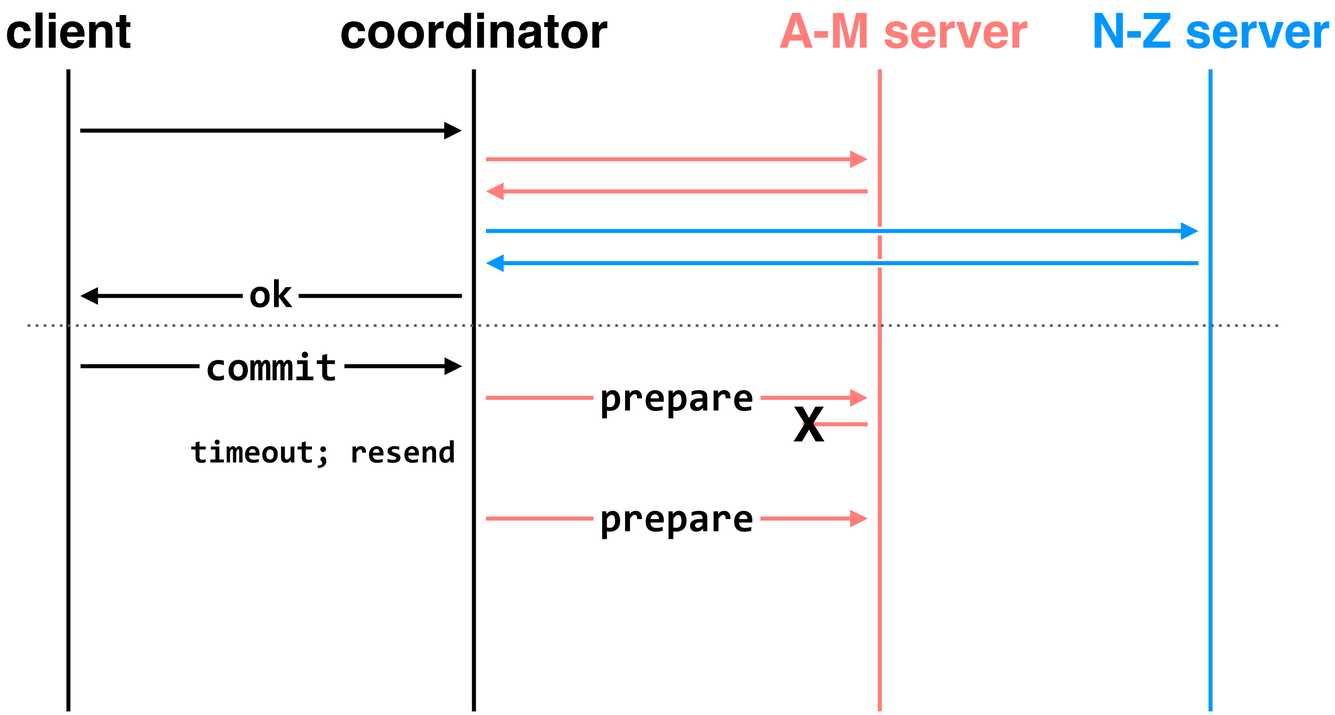
two-phase commit: nodes agree that they're ready to commit before committing



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

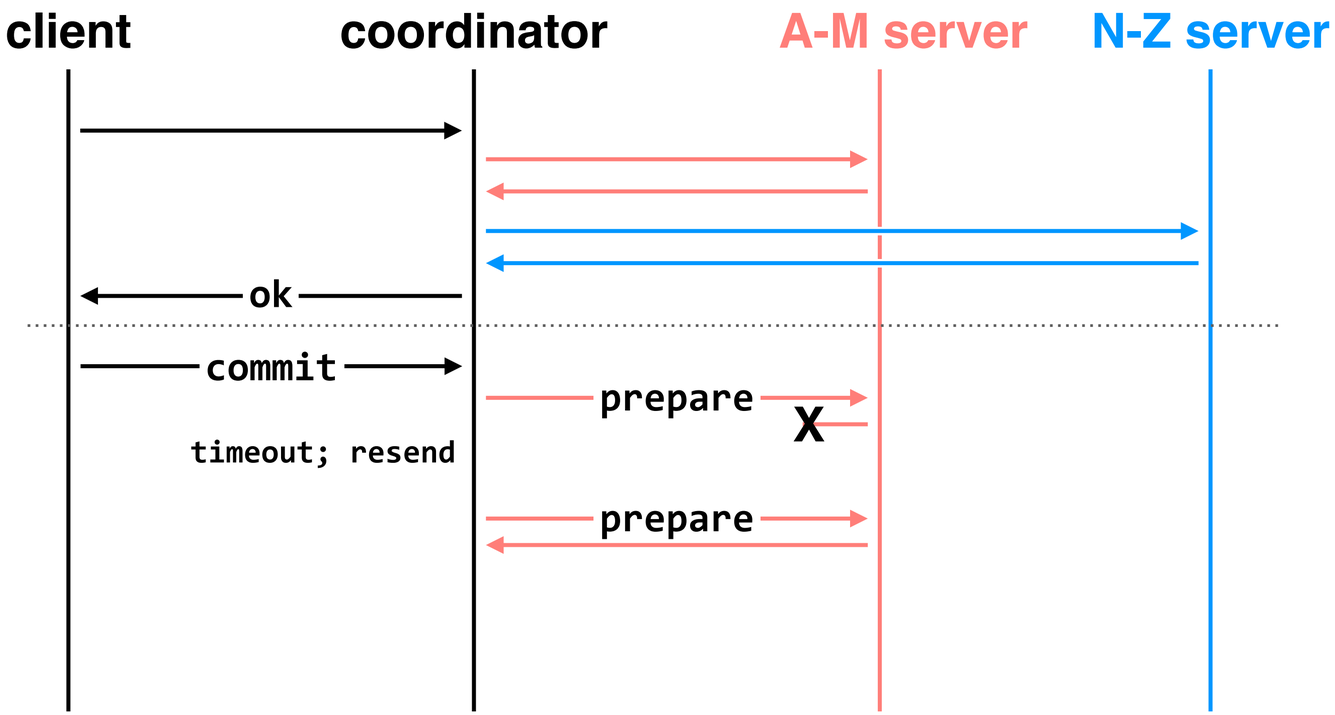


thanks to sequence numbers, A-M will ACK the second prepare message but not reprocess it

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

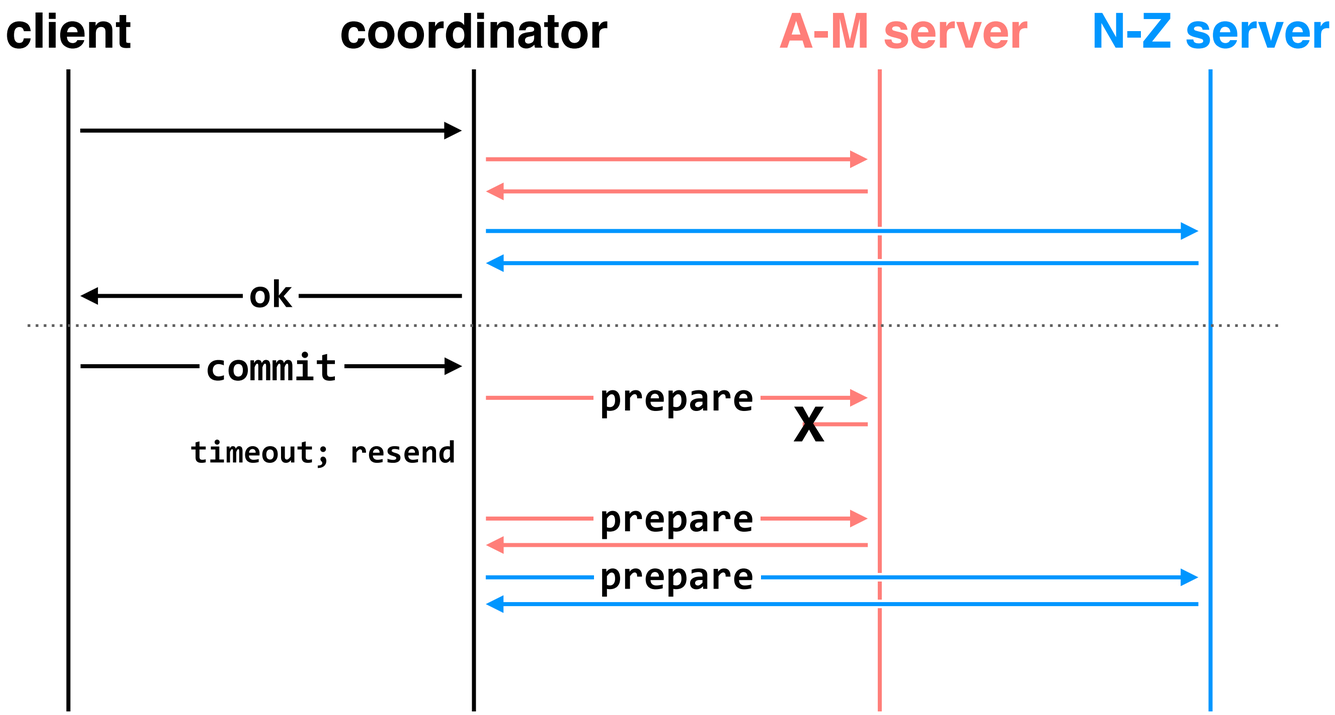


thanks to sequence numbers, A-M will ACK the second prepare message but not reprocess it

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

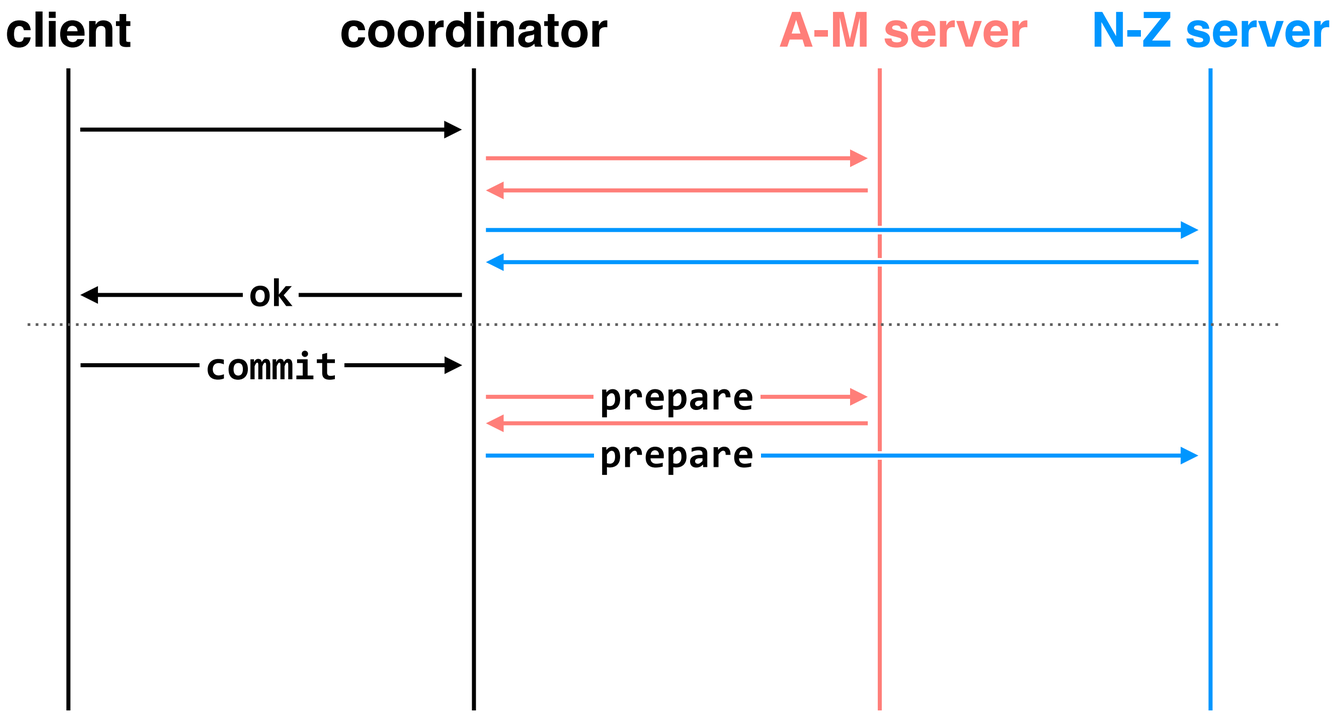


thanks to sequence numbers, A-M will ACK the second prepare message but not reprocess it

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

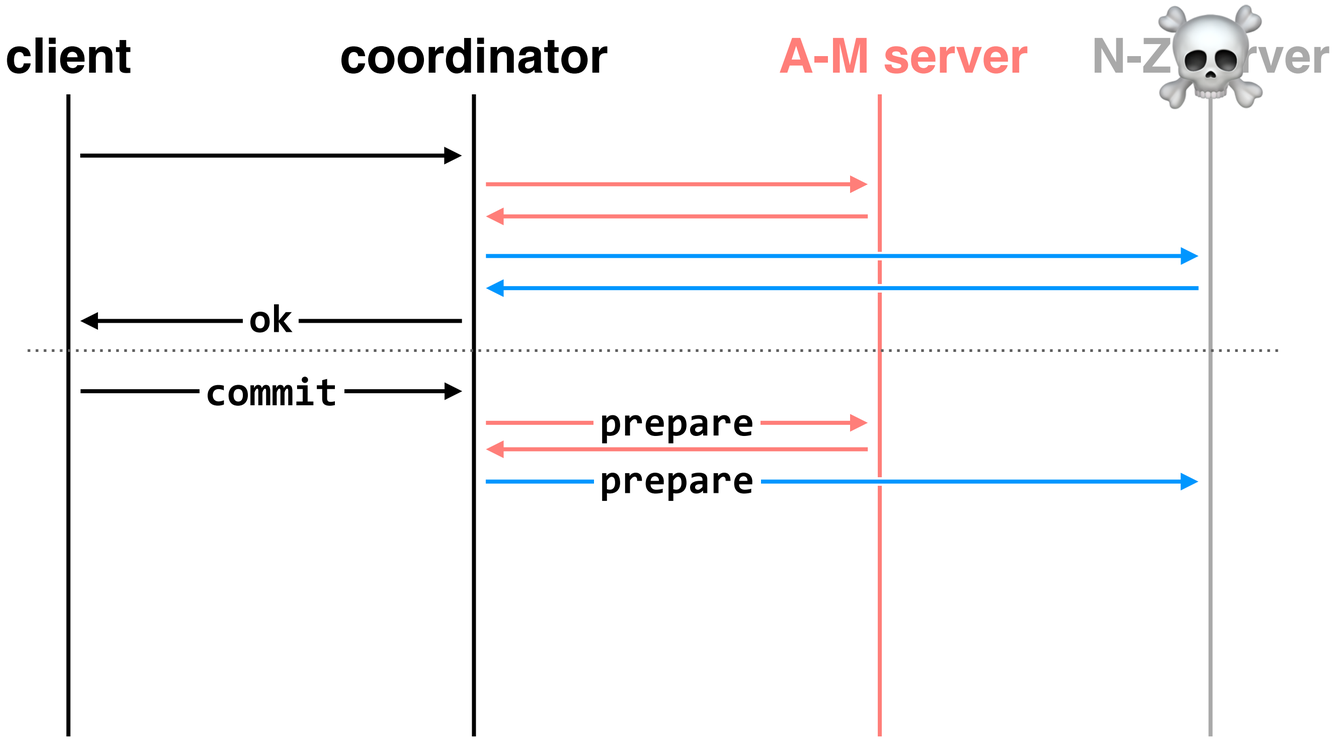


message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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

worker failure during prepare phase

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

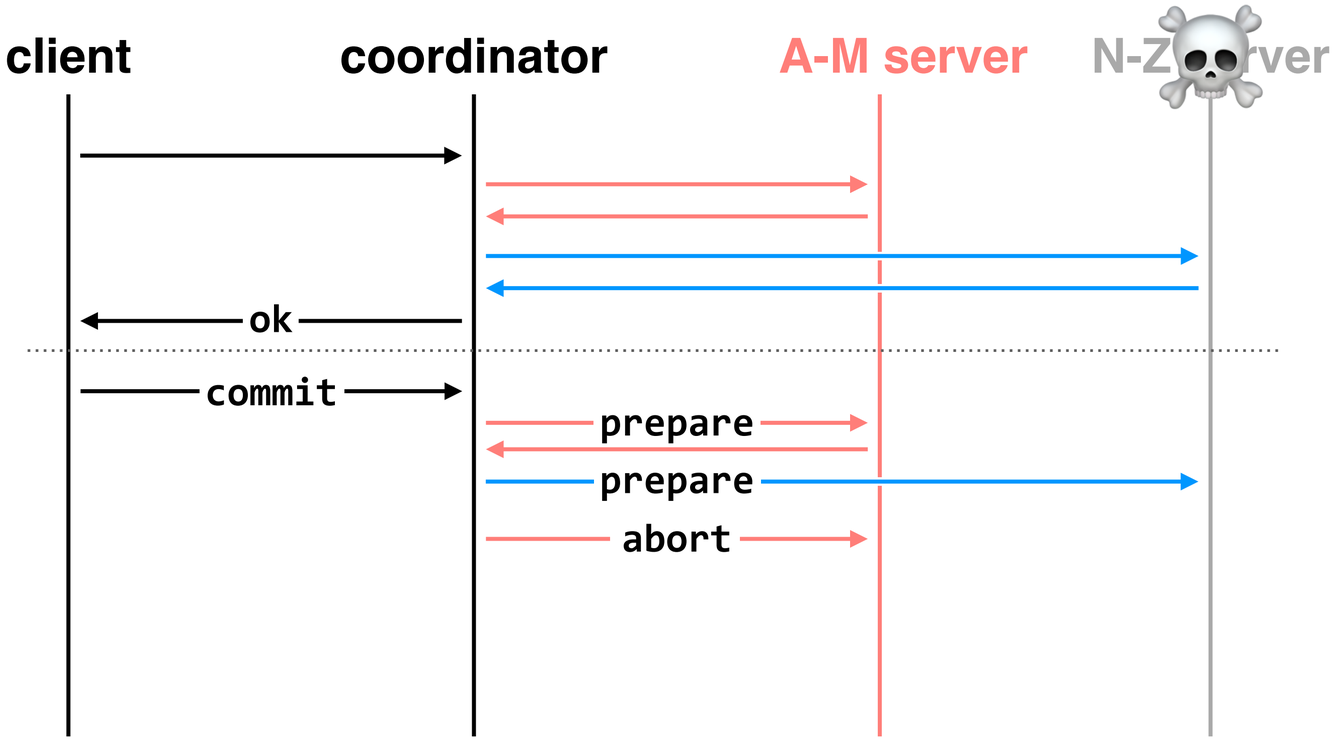


message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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

worker failure during prepare phase

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

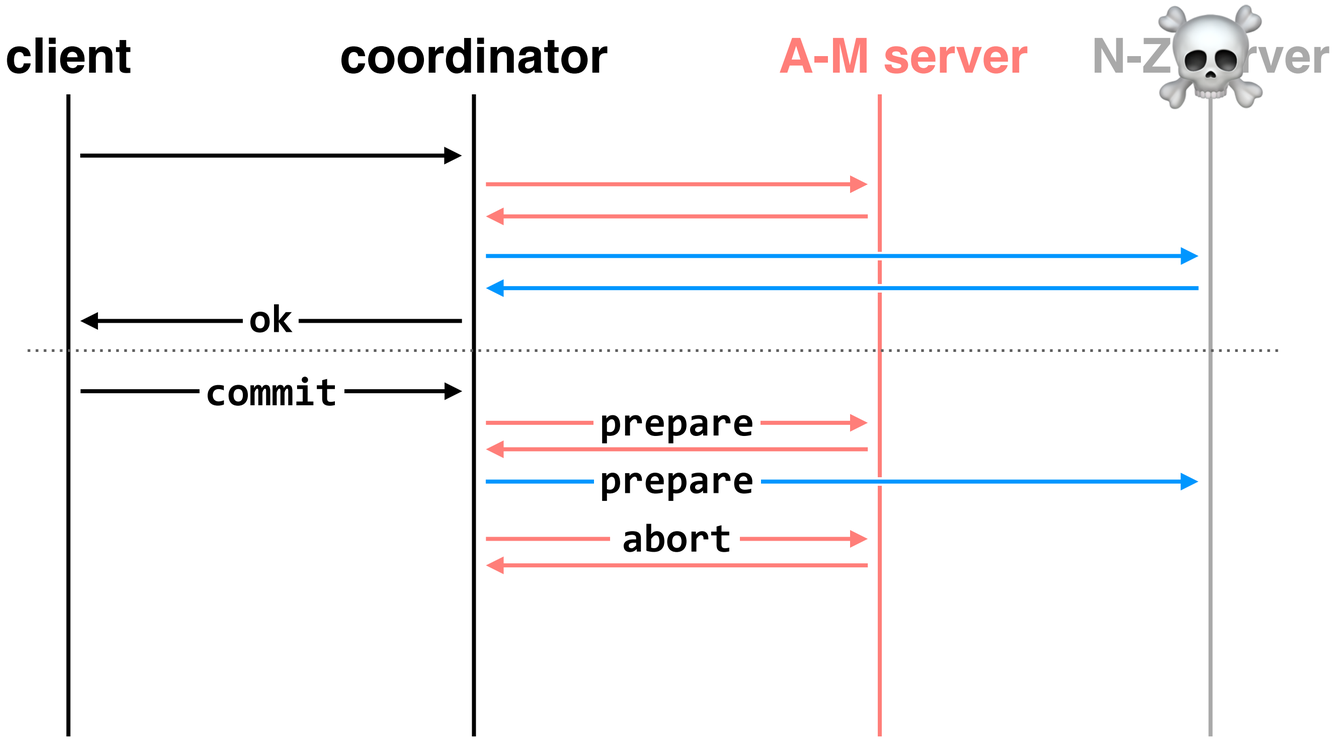


message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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

worker failure during prepare phase

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

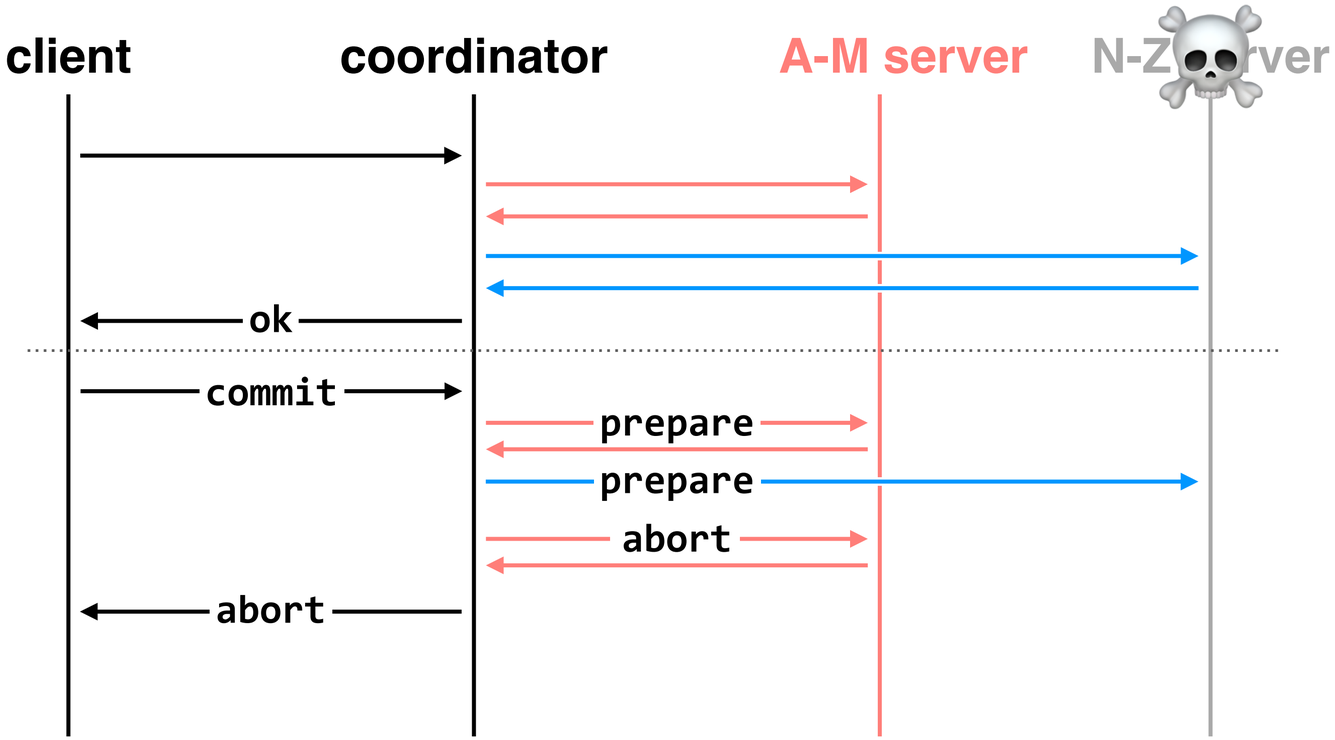


message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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

worker failure during prepare phase

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

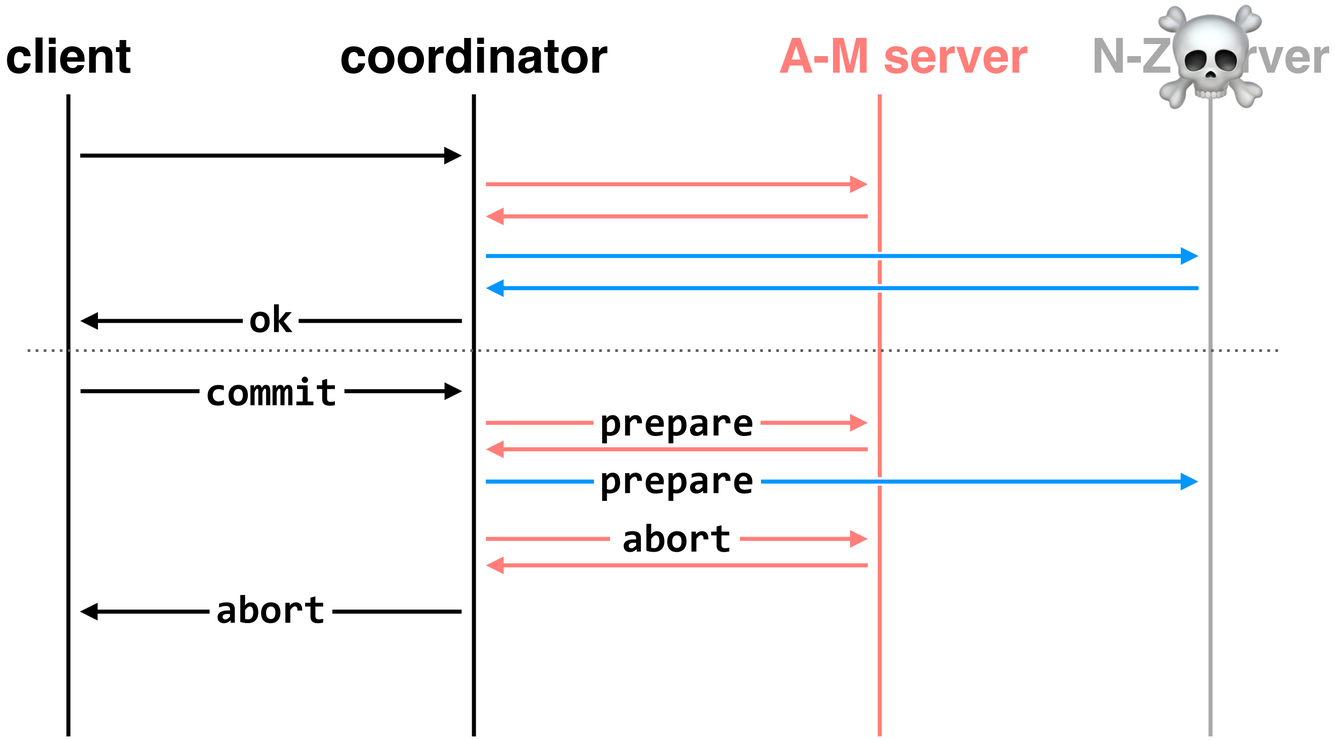


message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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

worker failure during prepare phase

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

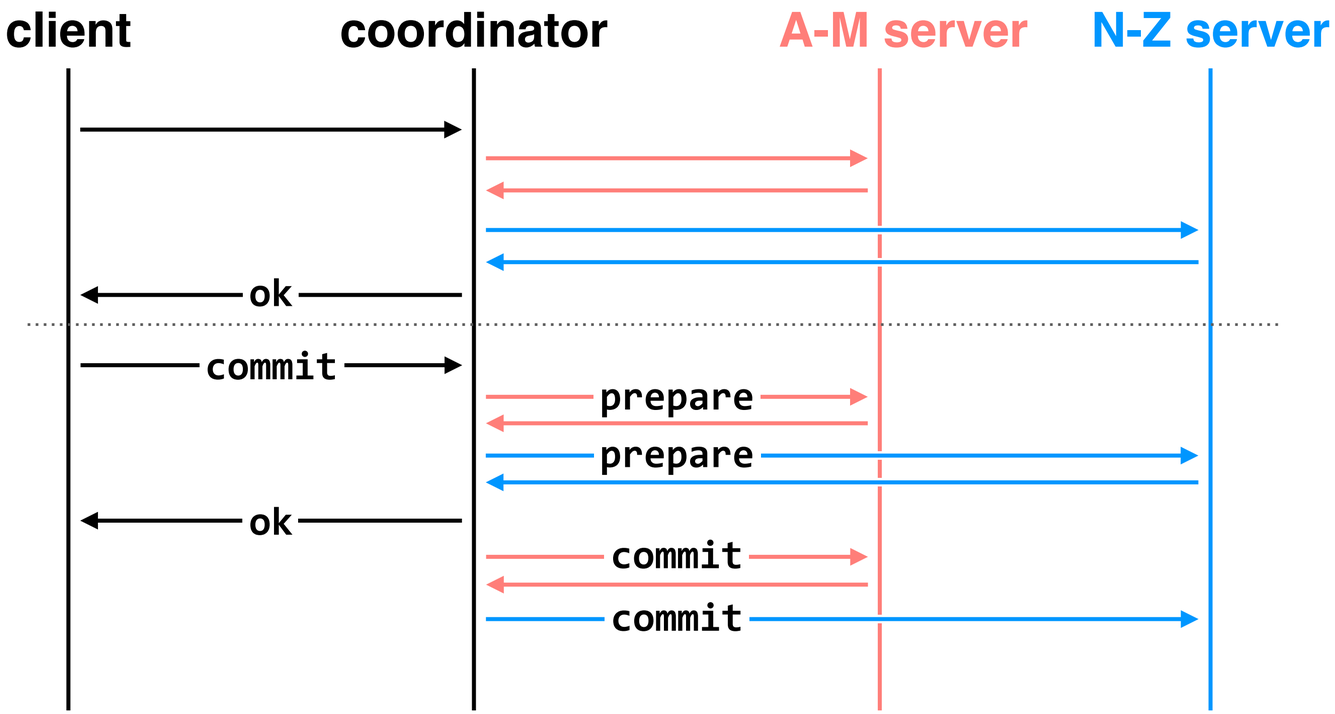


message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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



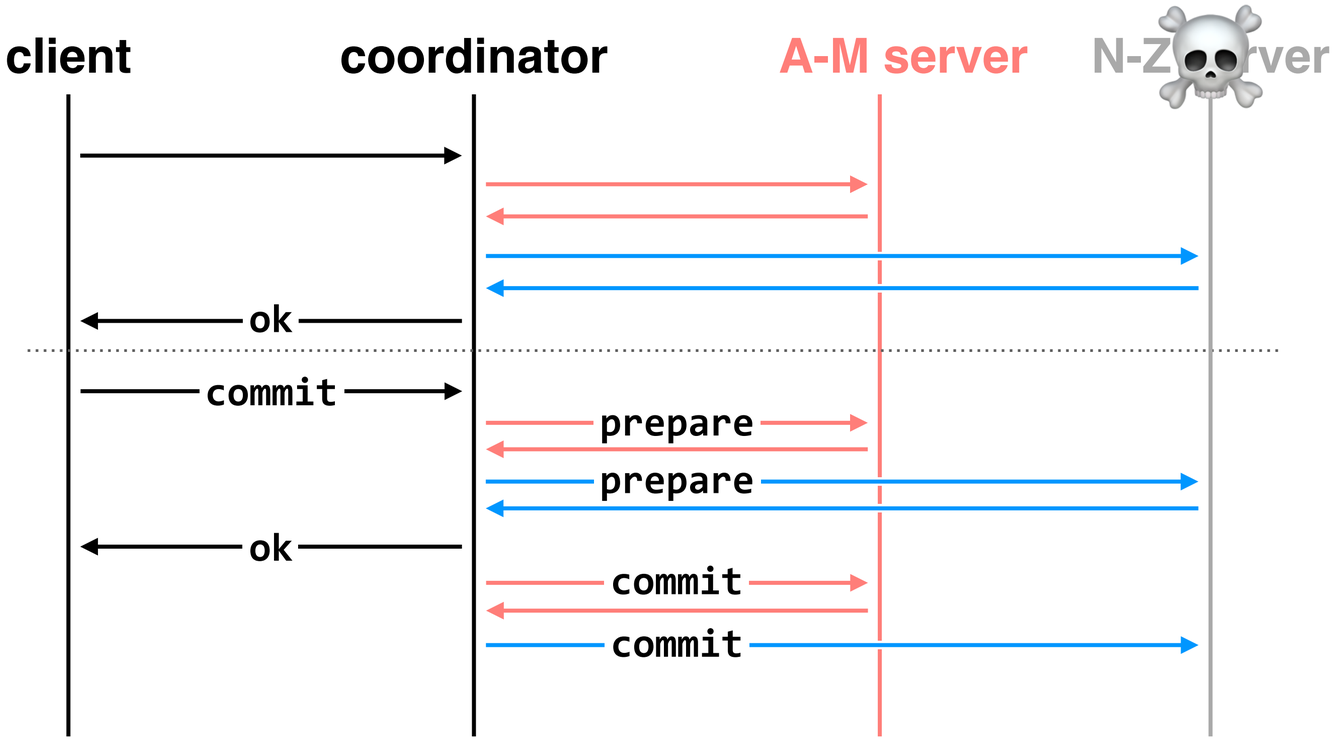
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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

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



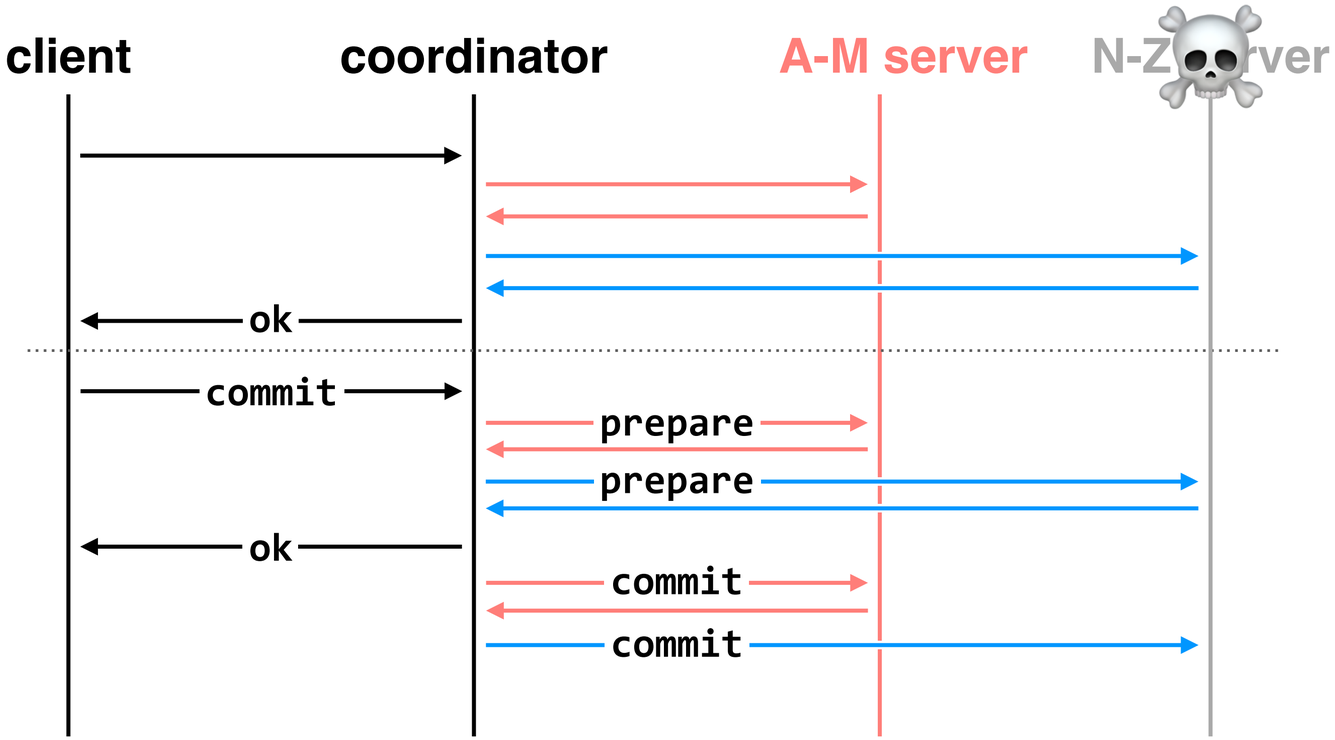
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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

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

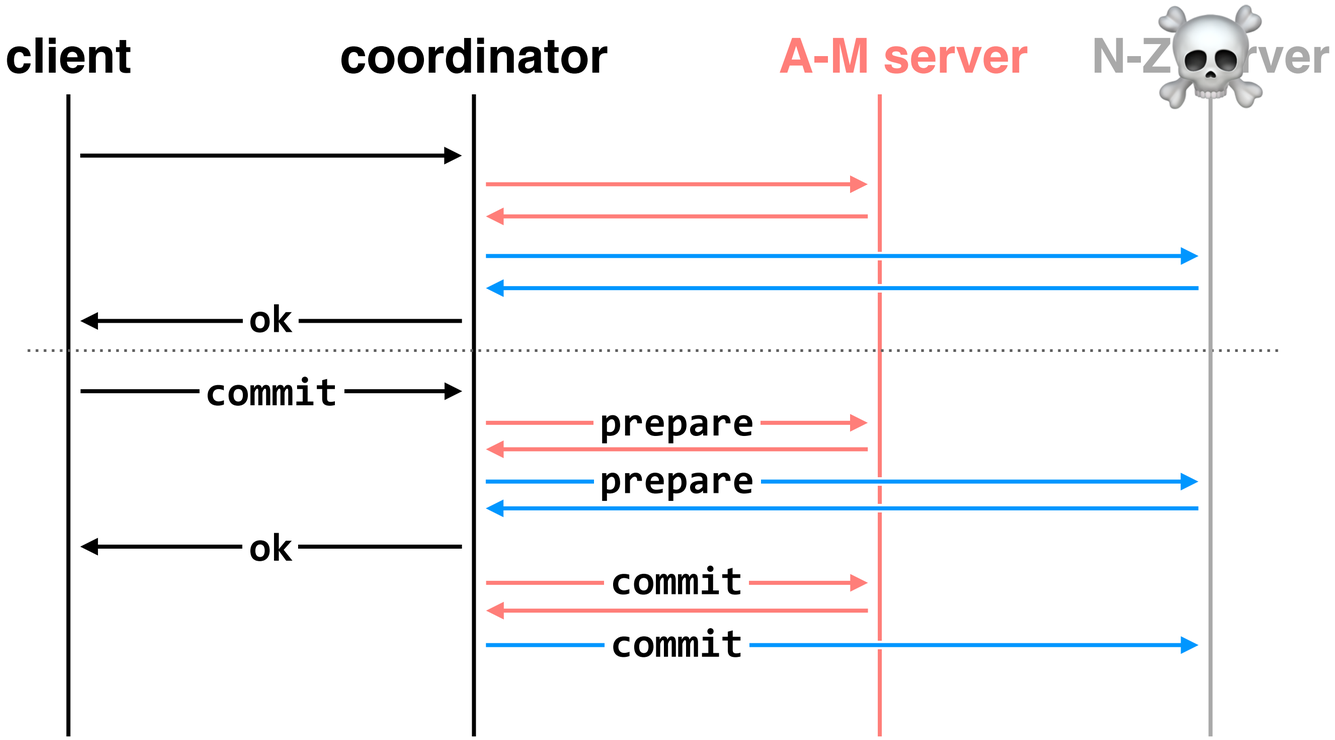
worker failure before prepare phase: coordinator can safely abort transaction without additional communication to workers

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



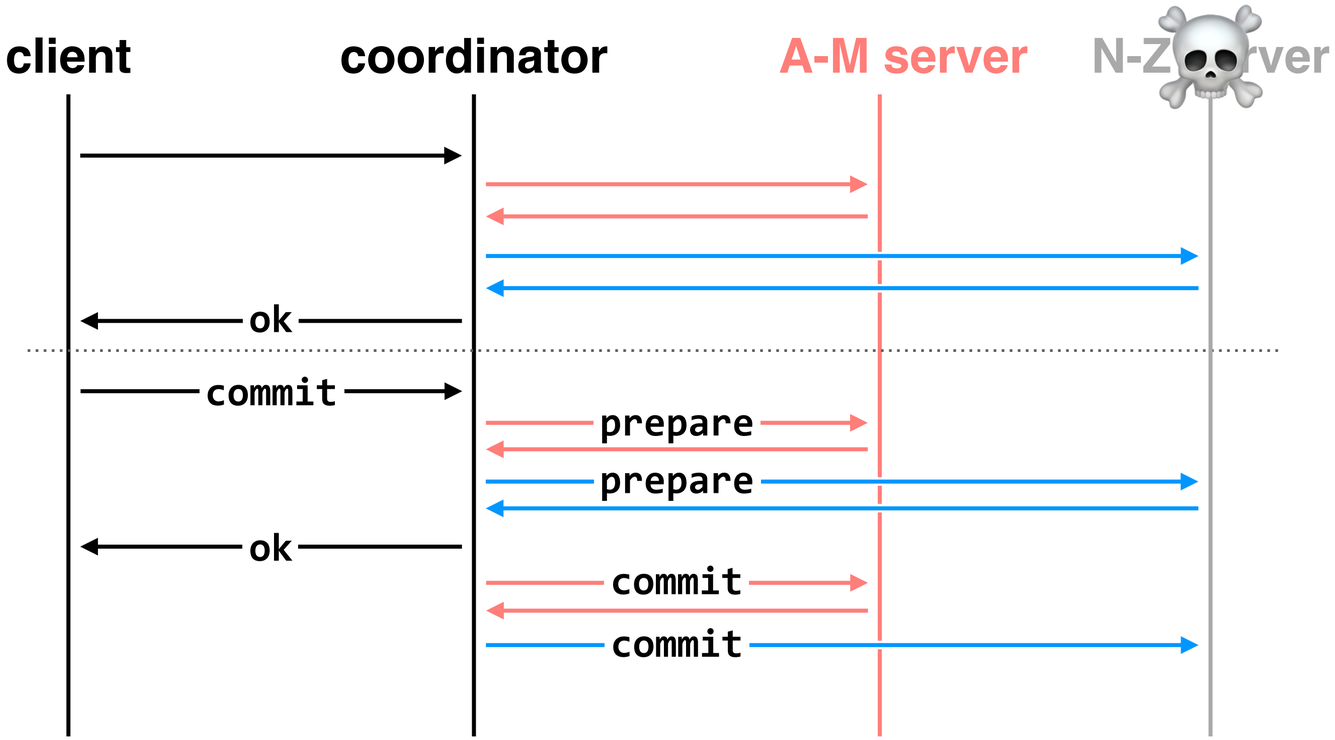
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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



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

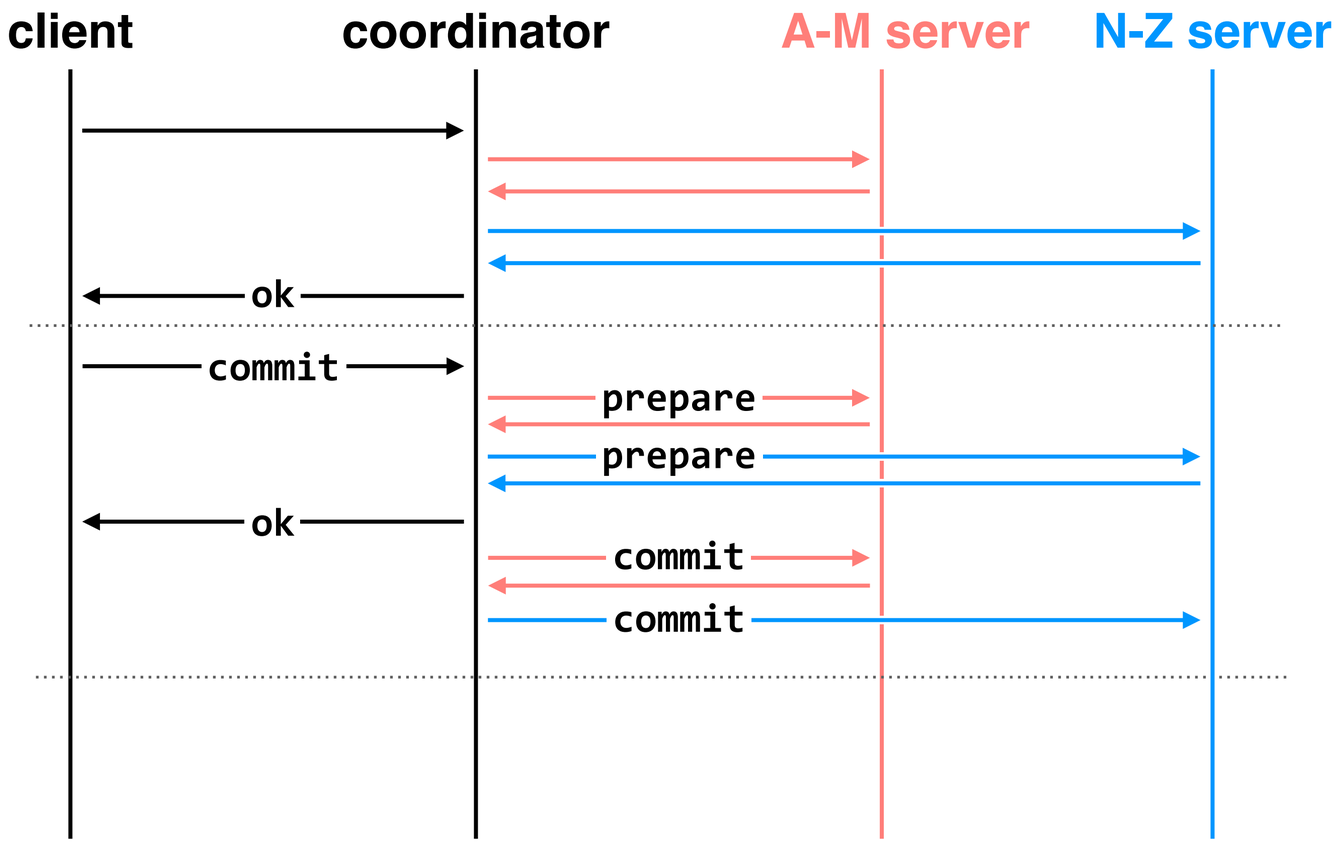
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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



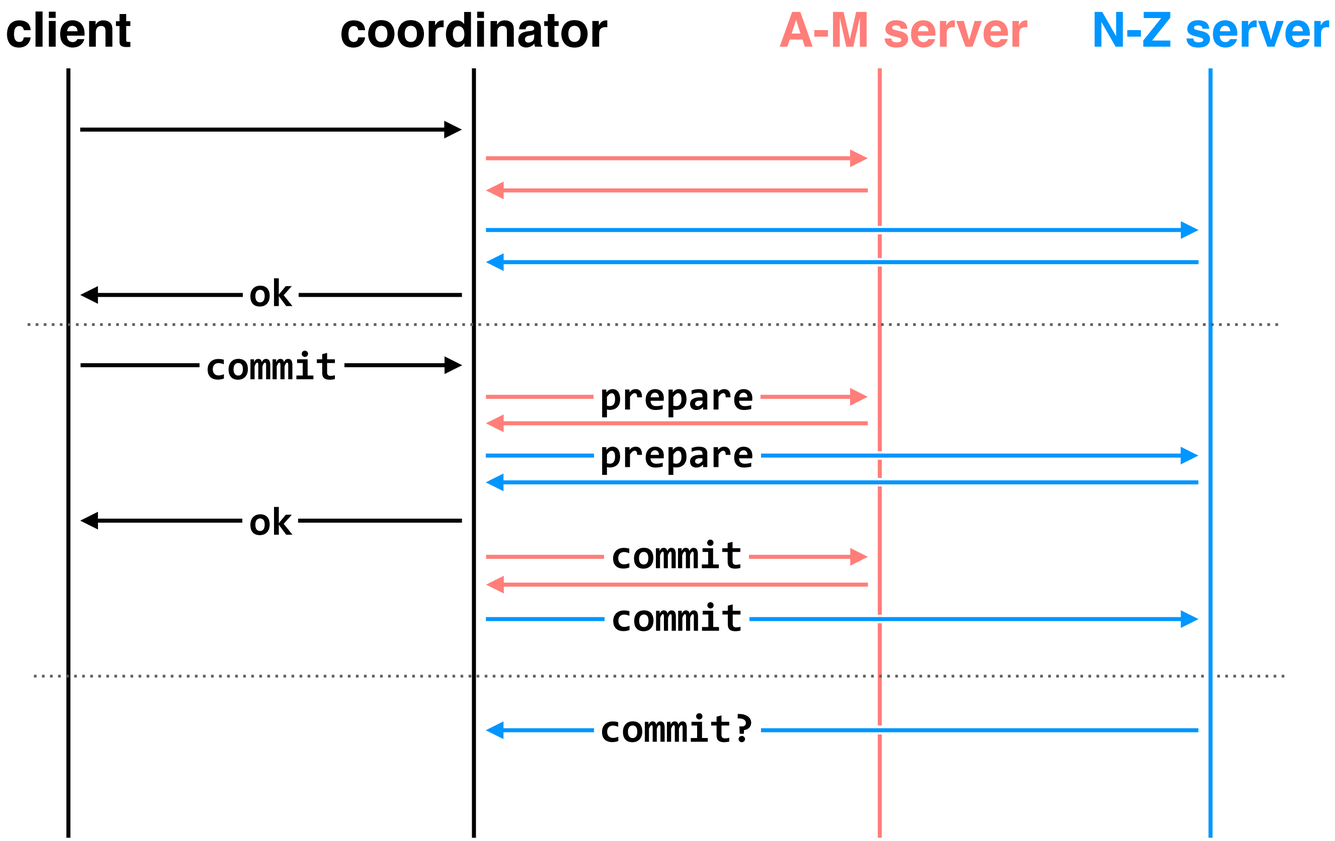
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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



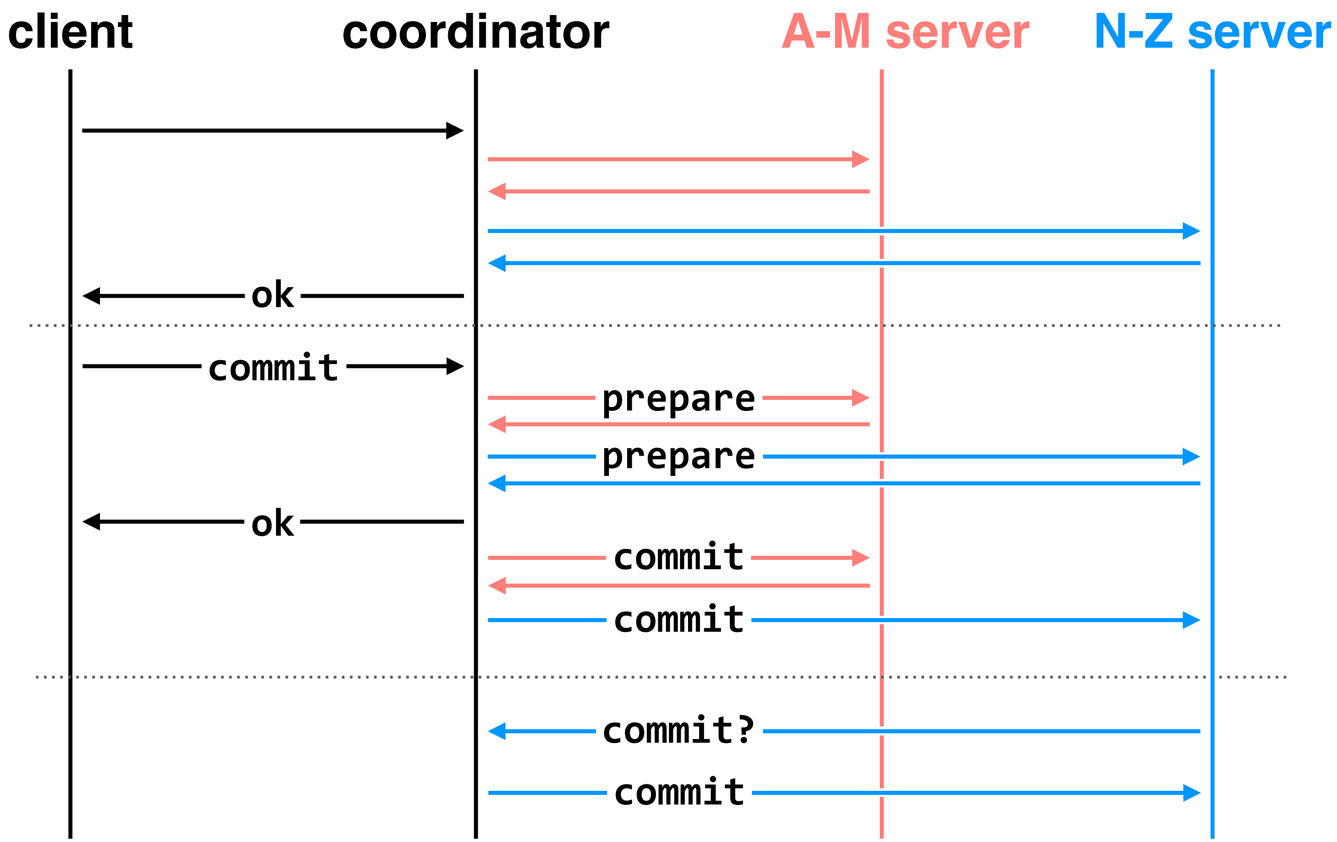
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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



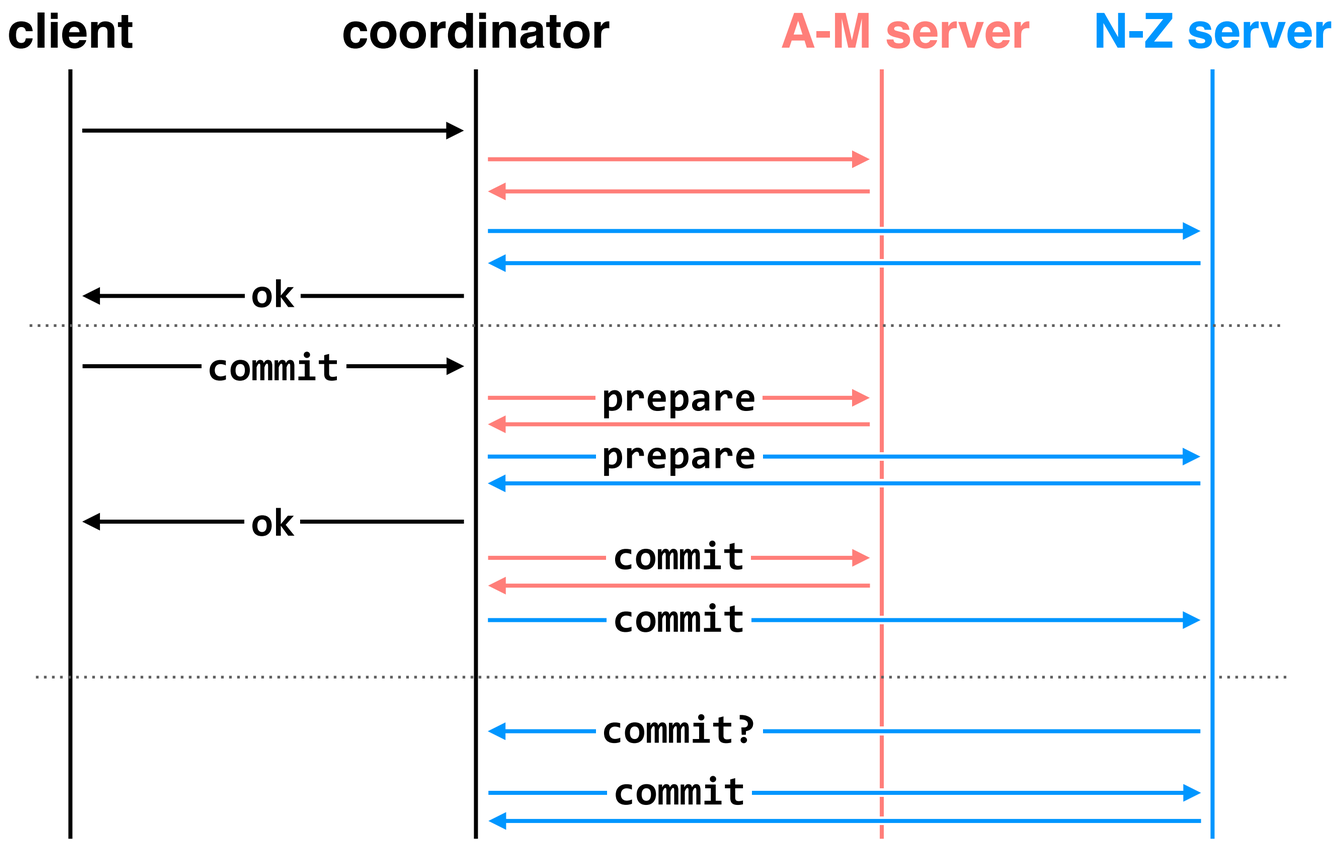
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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

worker failure 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



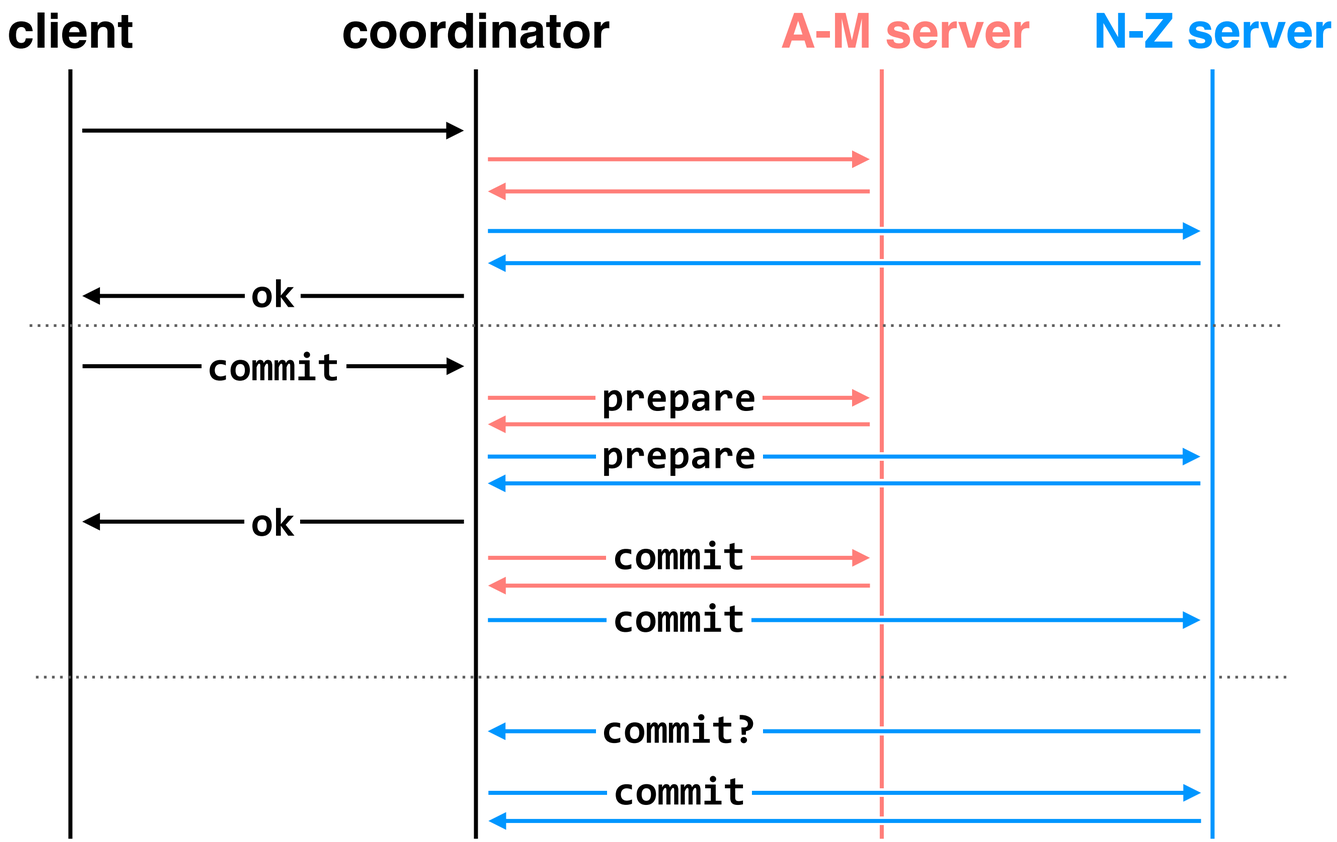
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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



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)?

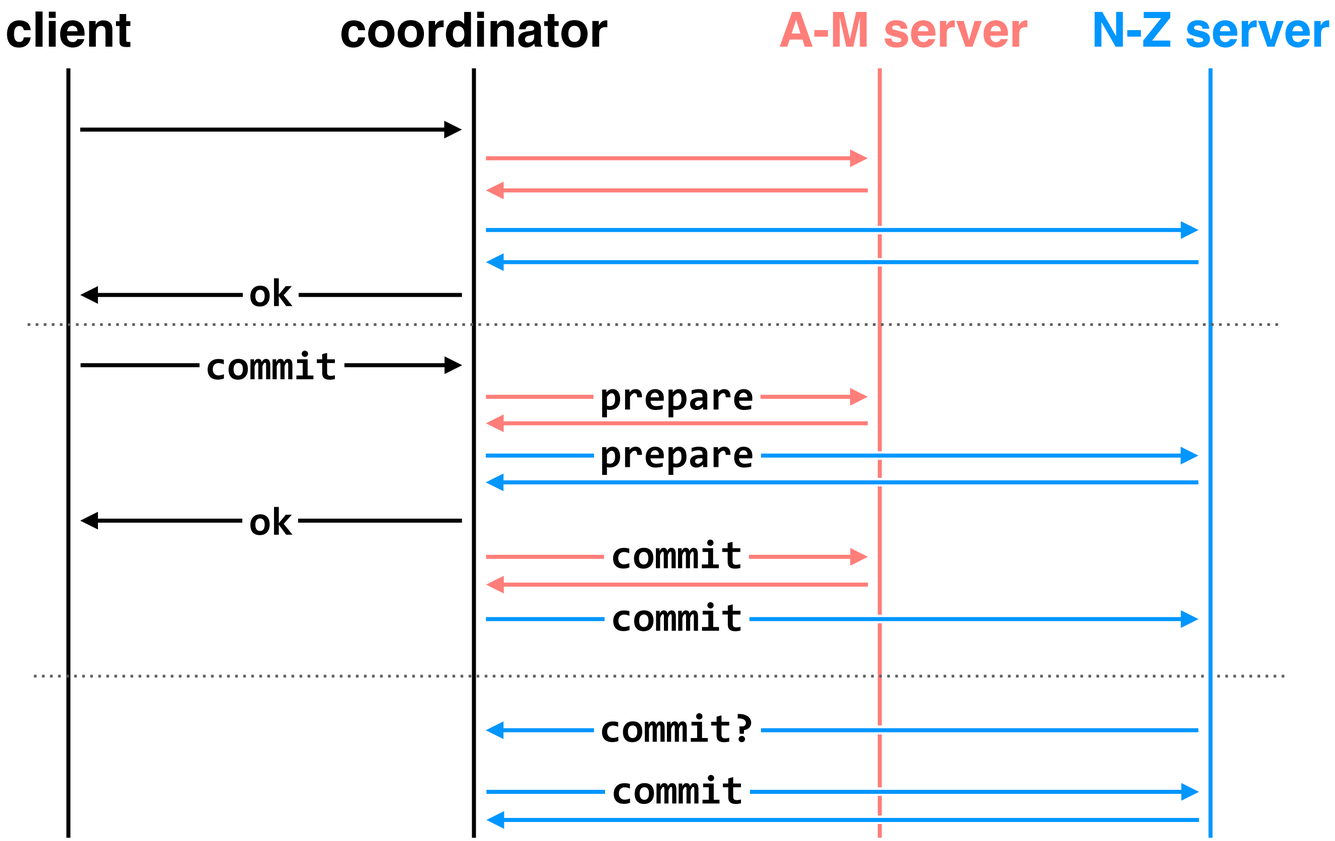
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction 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



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

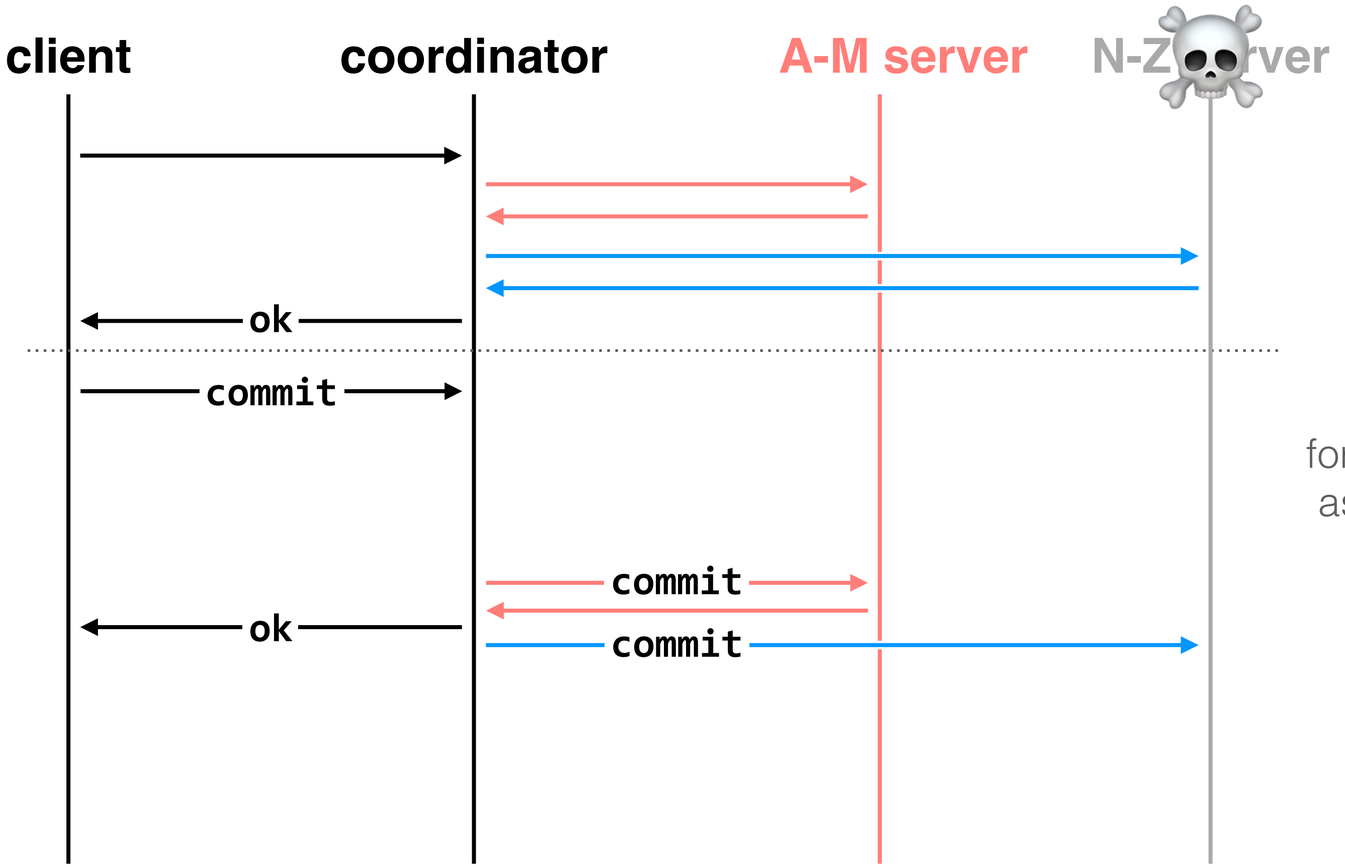
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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

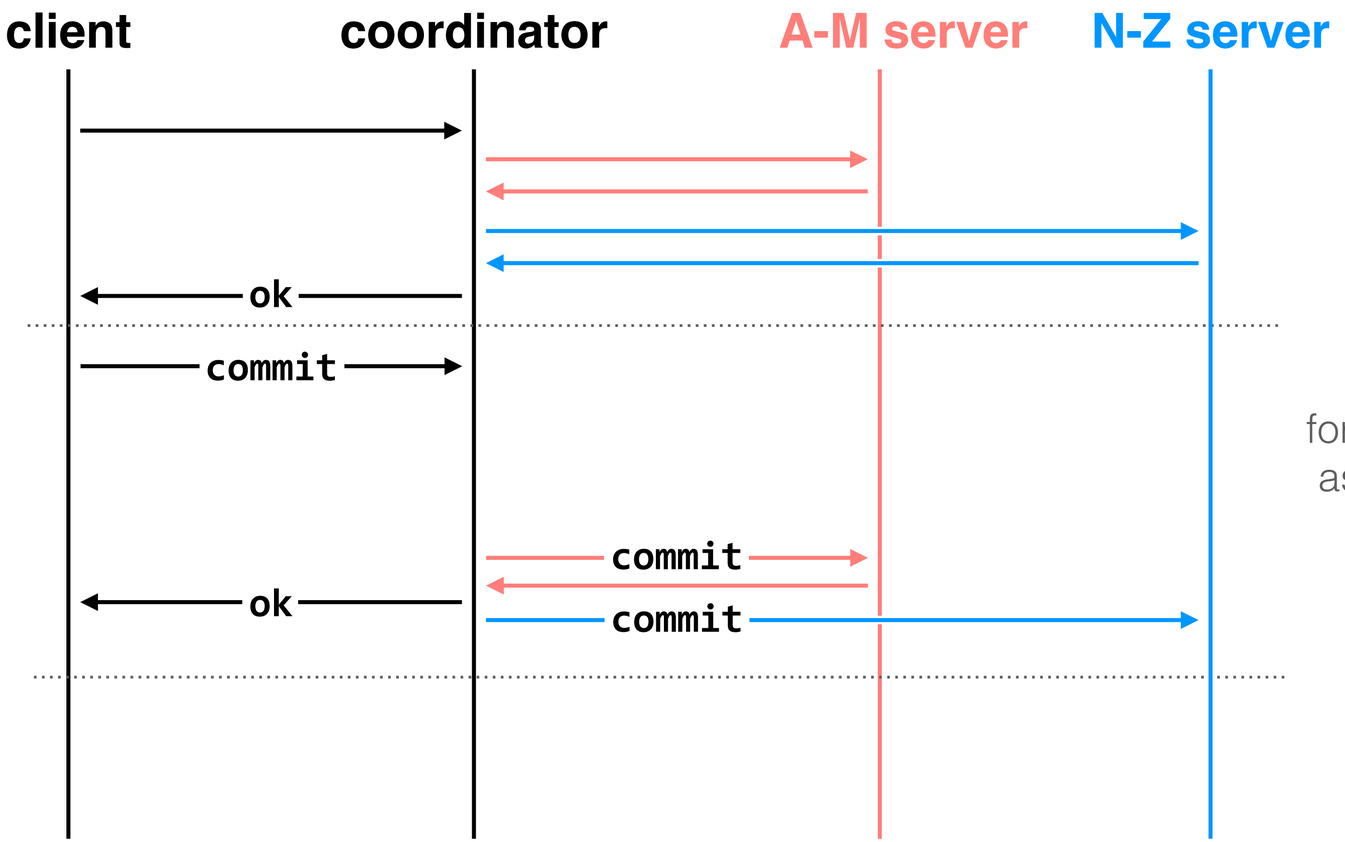


for instance, suppose we get rid of the prepare phase, and as long as one server commits, we force any that fail after that point to recover into a committed state?

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

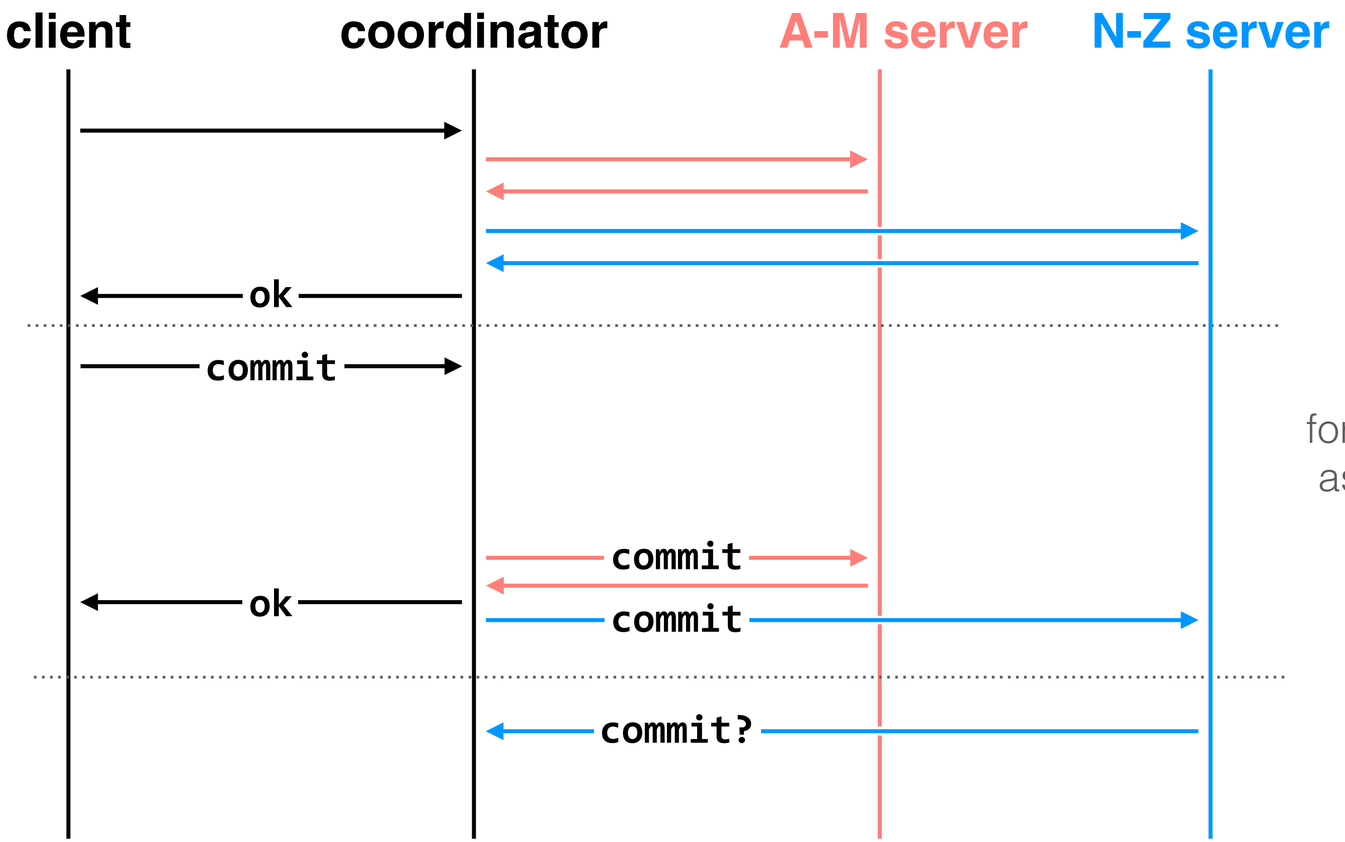


for instance, suppose we get rid of the prepare phase, and as long as one server commits, we force any that fail after that point to recover into a committed state?

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

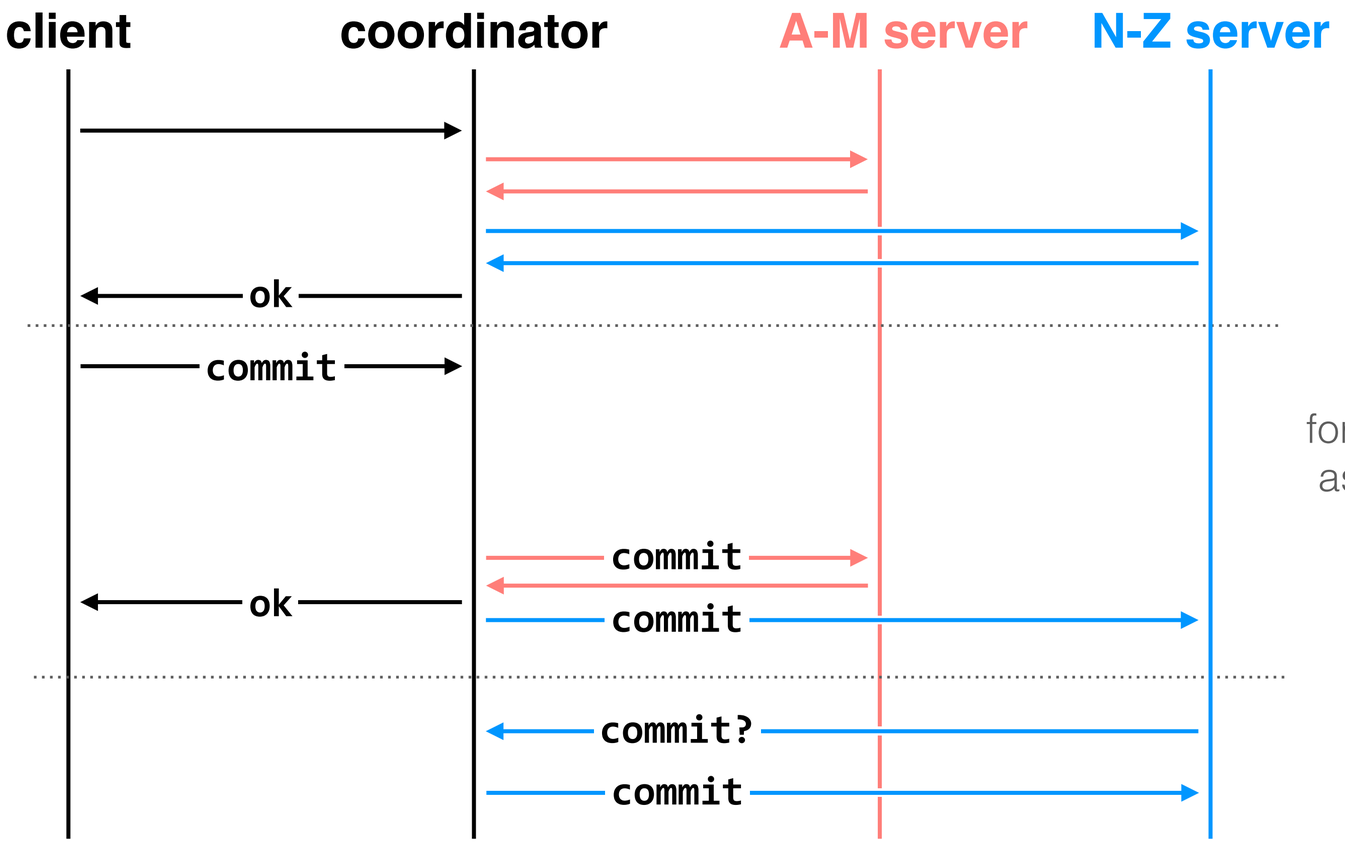


for instance, suppose we get rid of the prepare phase, and as long as one server commits, we force any that fail after that point to recover into a committed state?

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

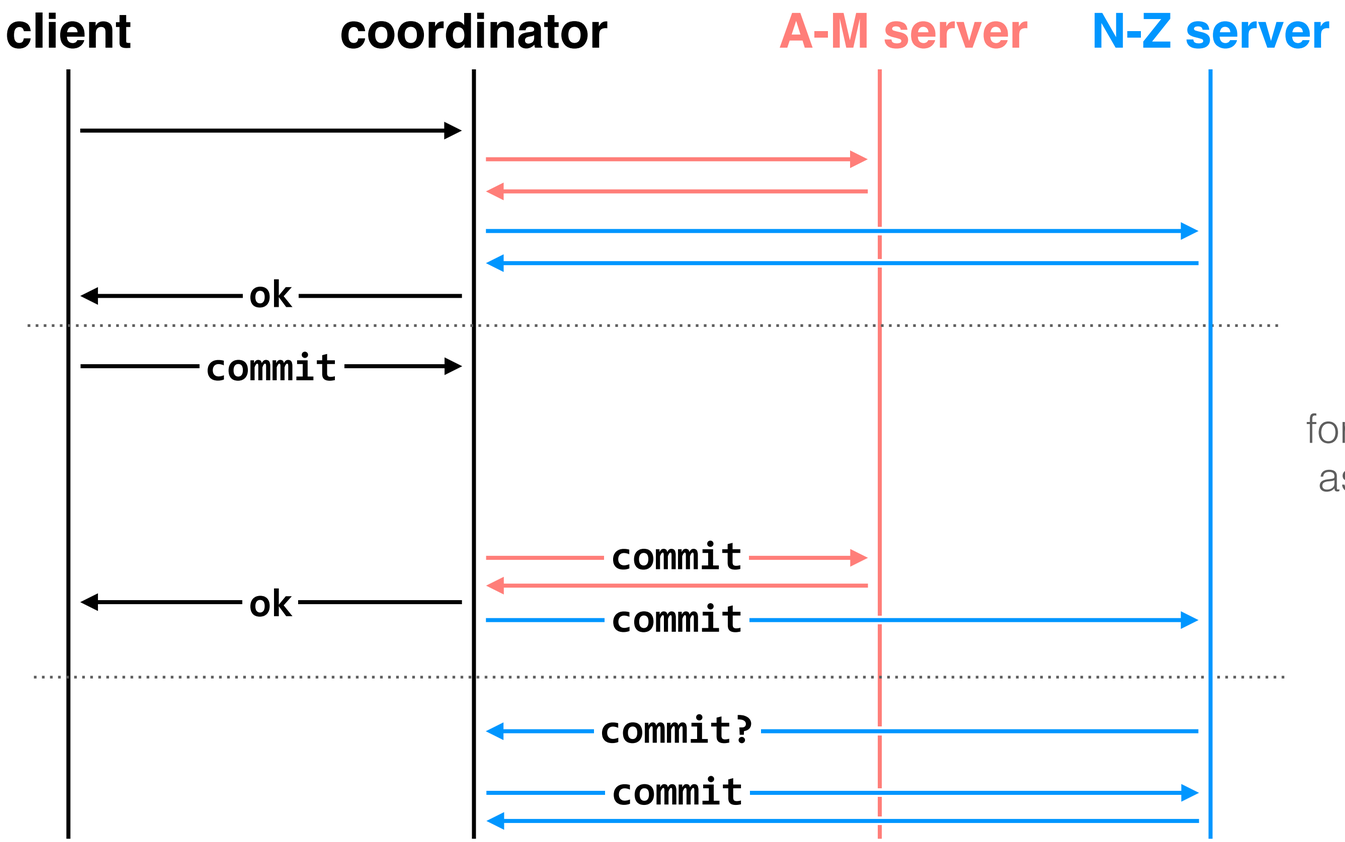


for instance, suppose we get rid of the prepare phase, and as long as one server commits, we force any that fail after that point to recover into a committed state?

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

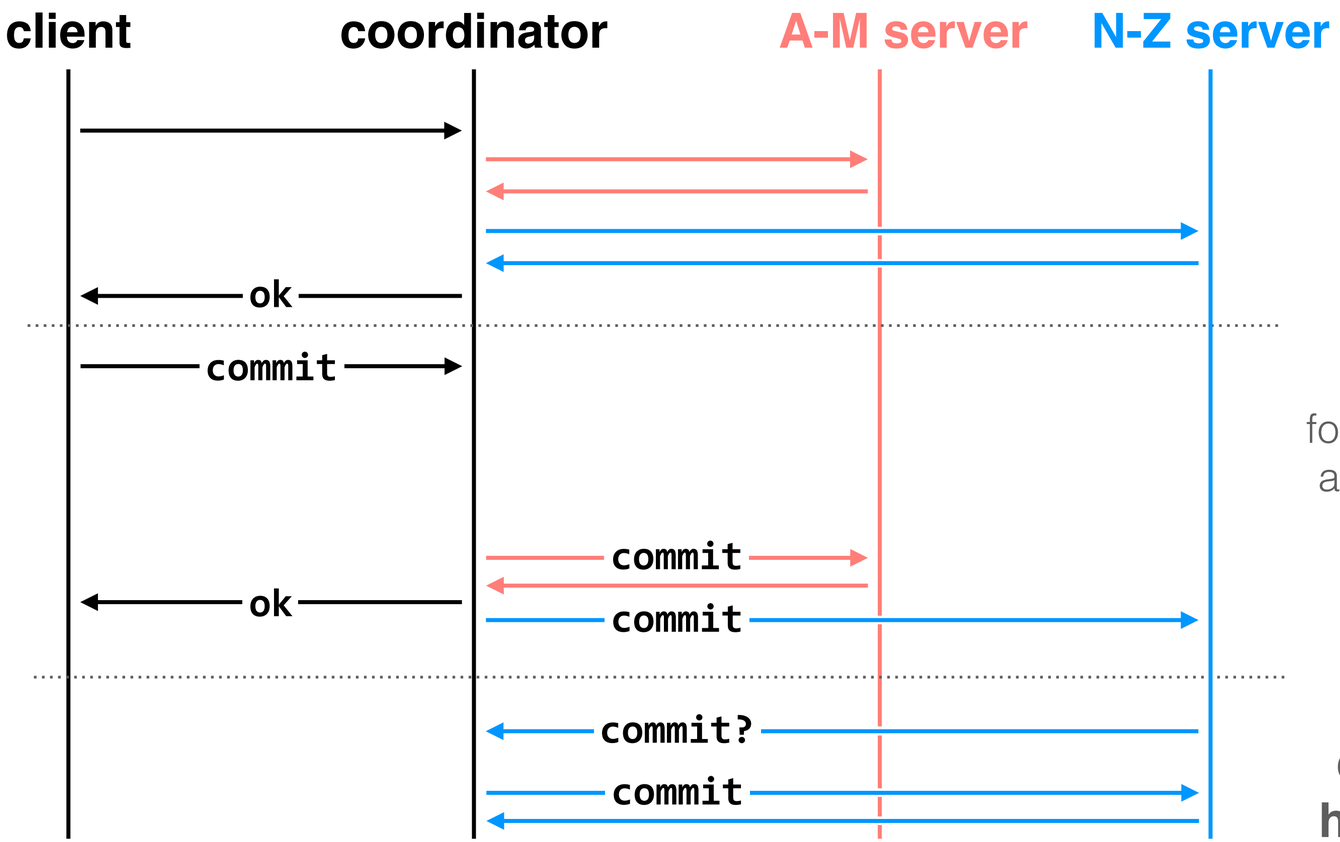


for instance, suppose we get rid of the prepare phase, and as long as one server commits, we force any that fail after that point to recover into a committed state?

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



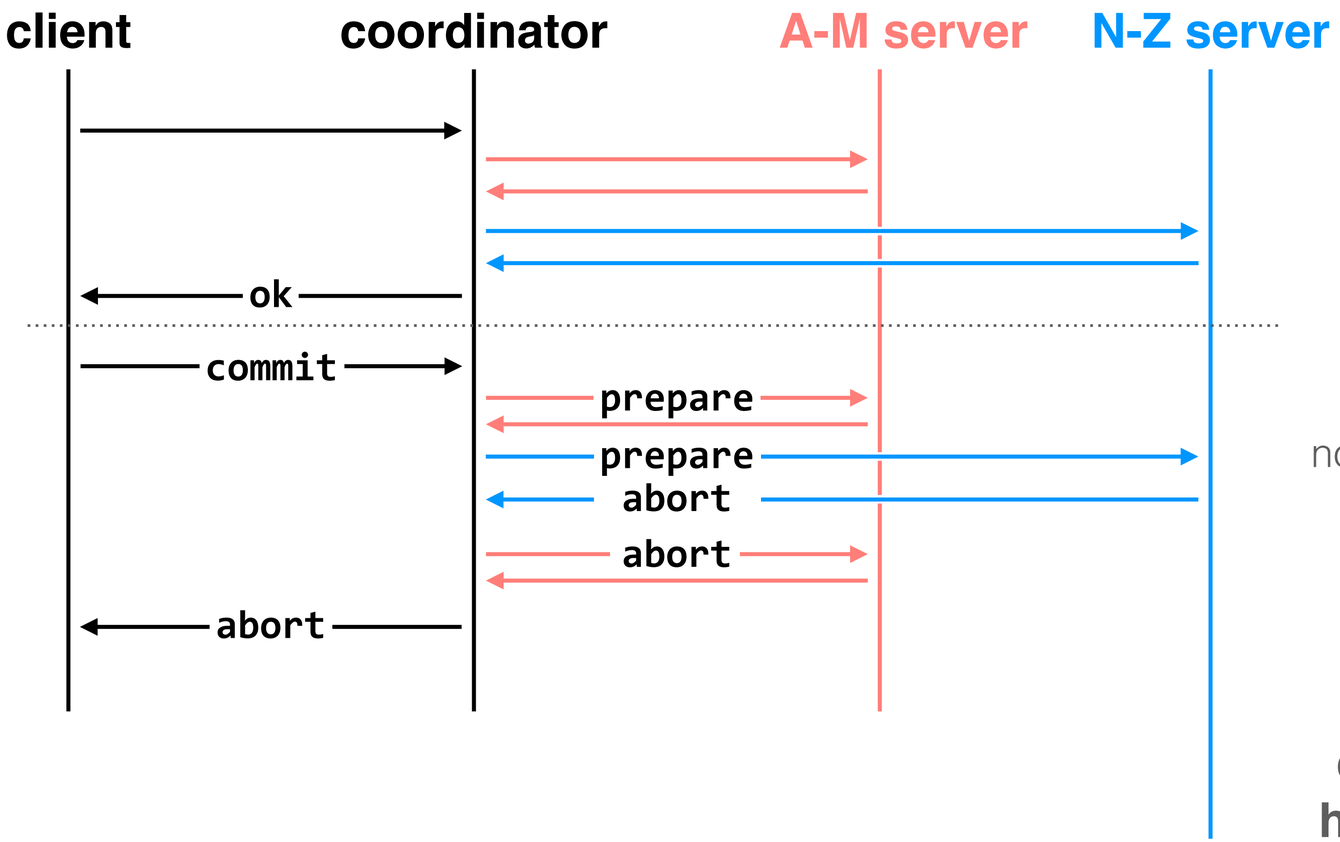
for instance, suppose we get rid of the prepare phase, and as long as one server commits, we force any that fail after that point to recover into a committed state?

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



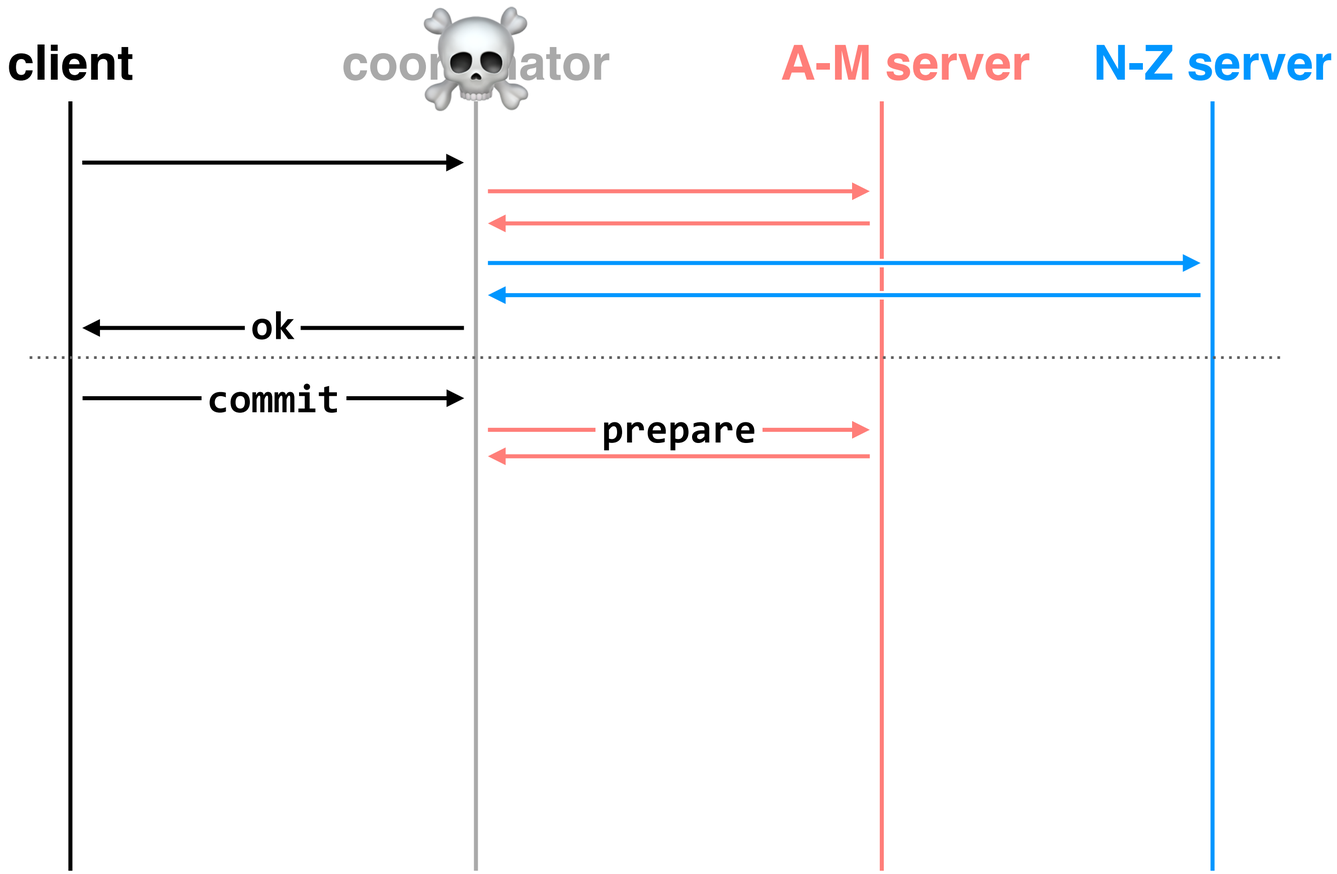
notice that the N-Z server did not fail here, but still aborted the transaction

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

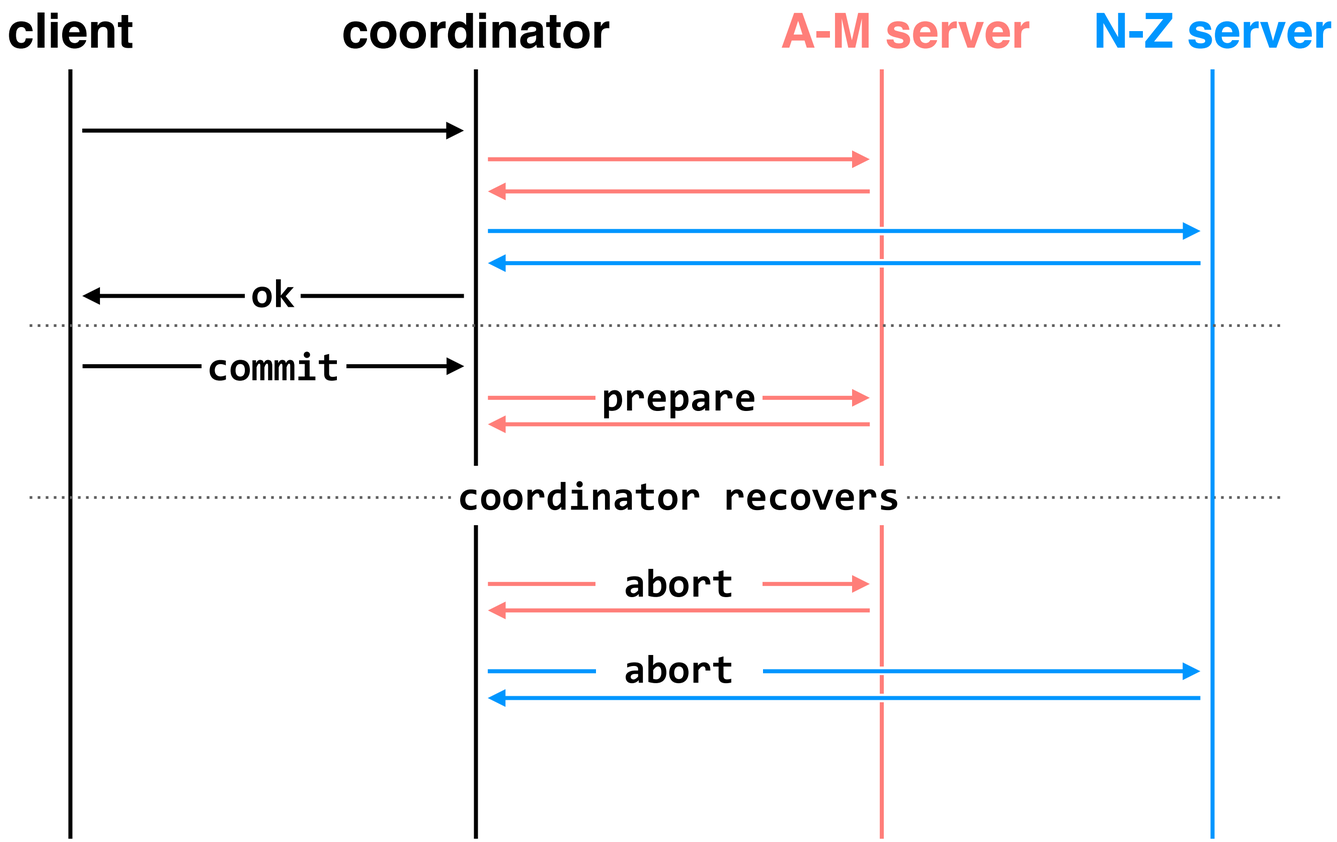
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction 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



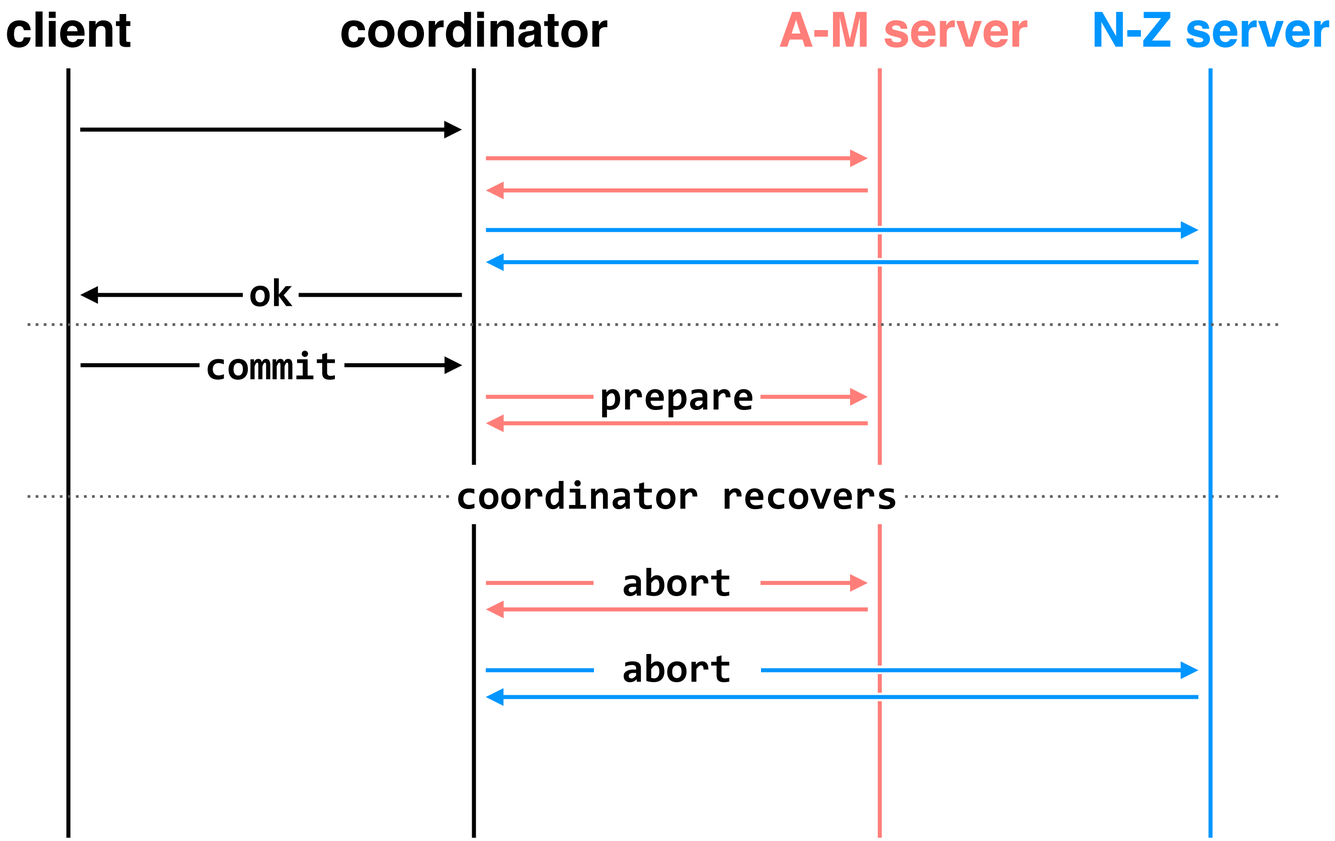
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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



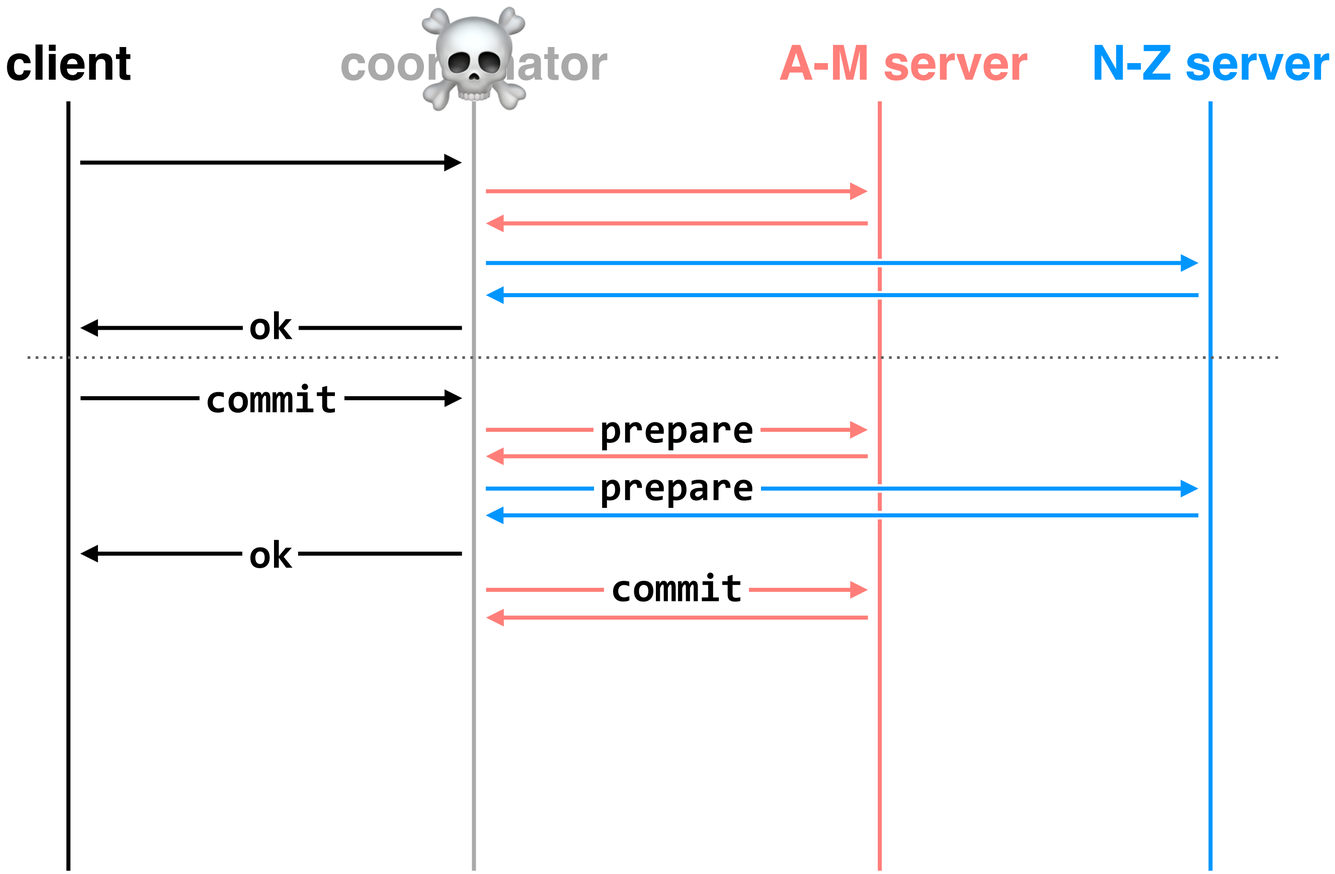
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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



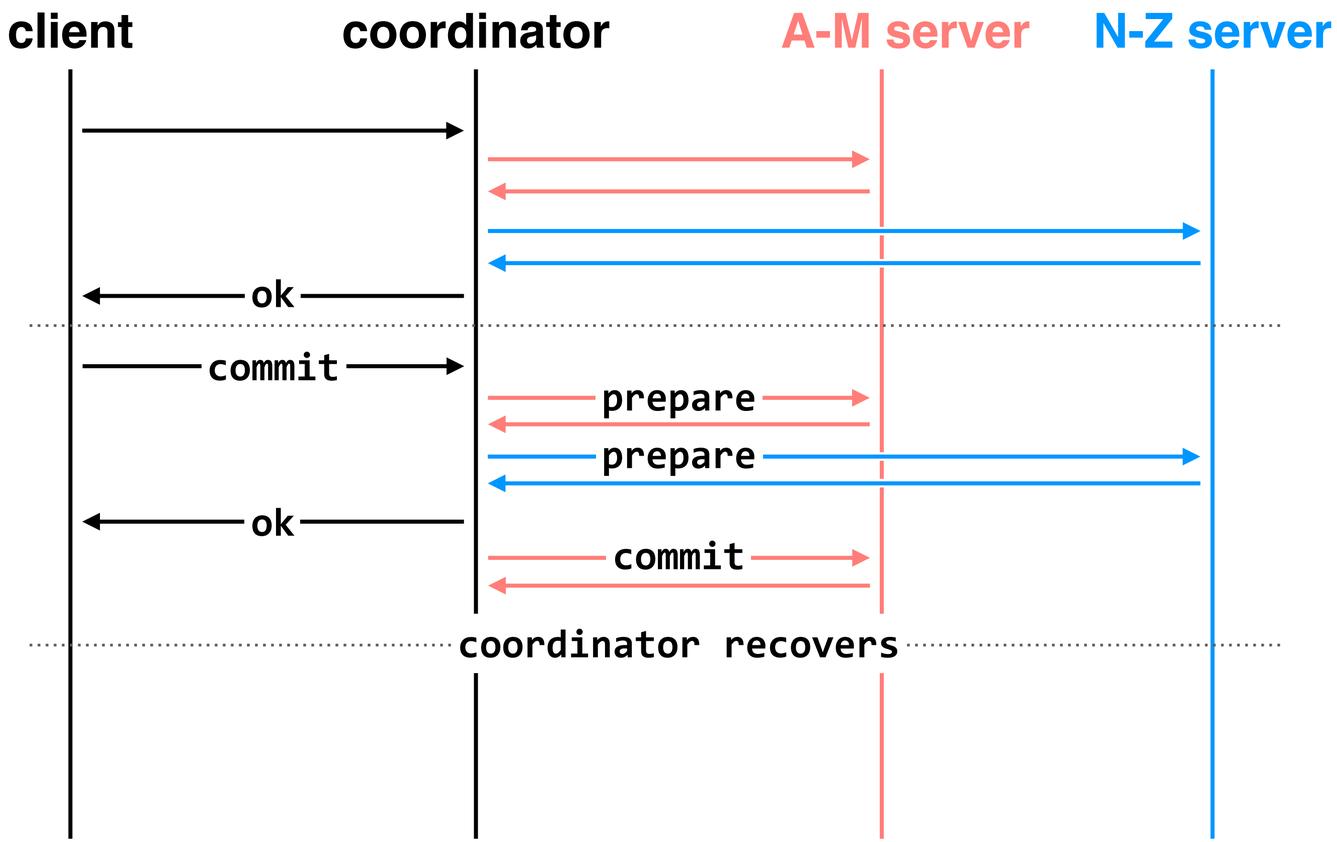
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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



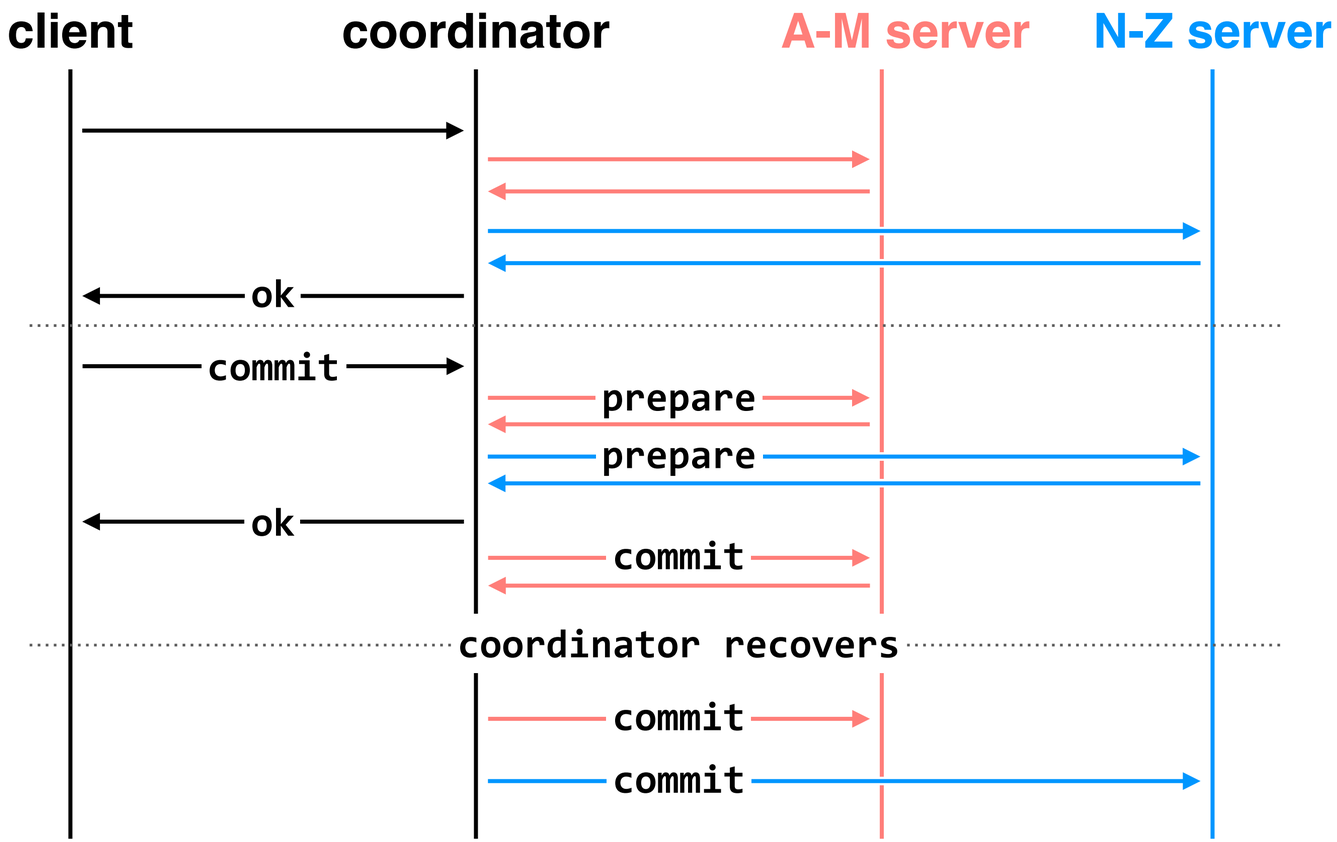
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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 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



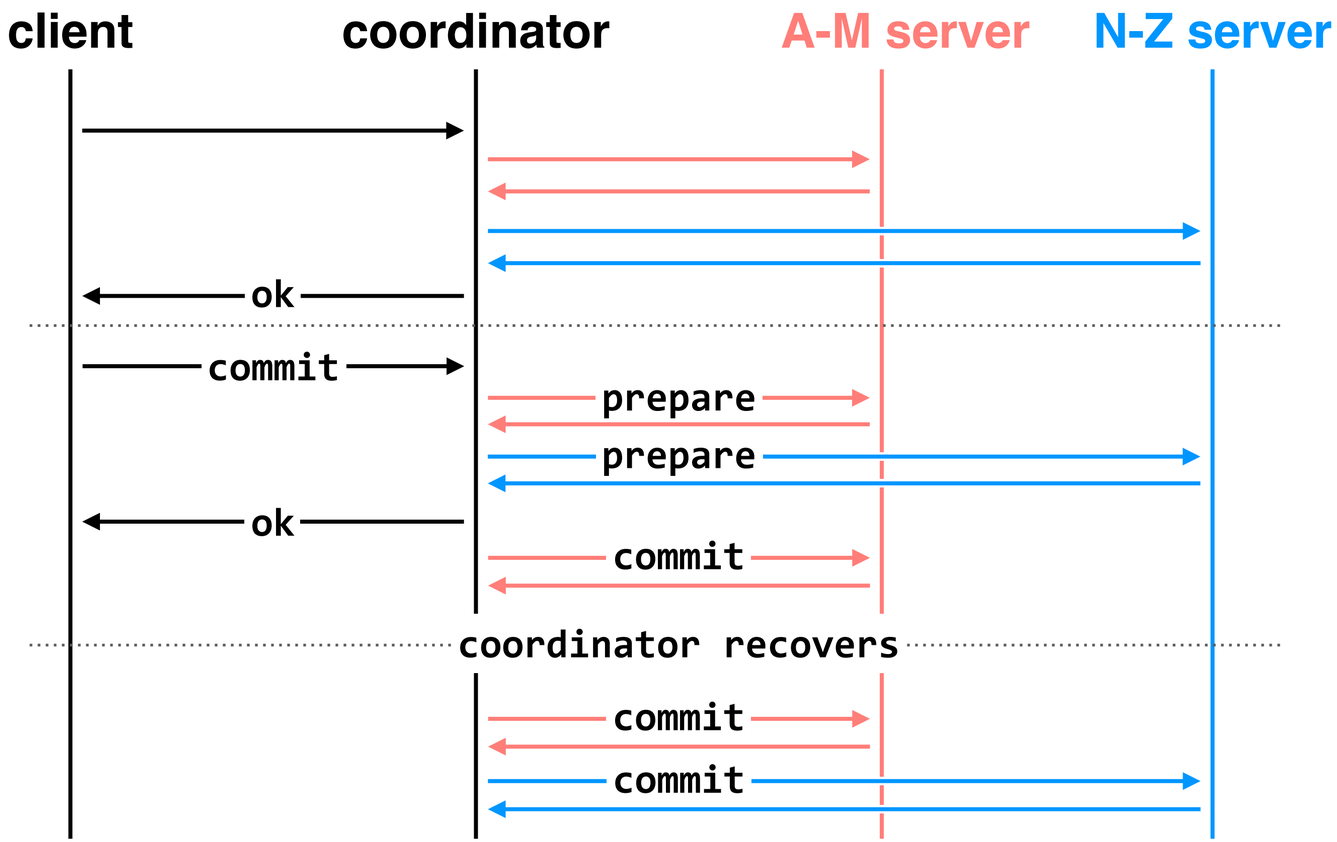
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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



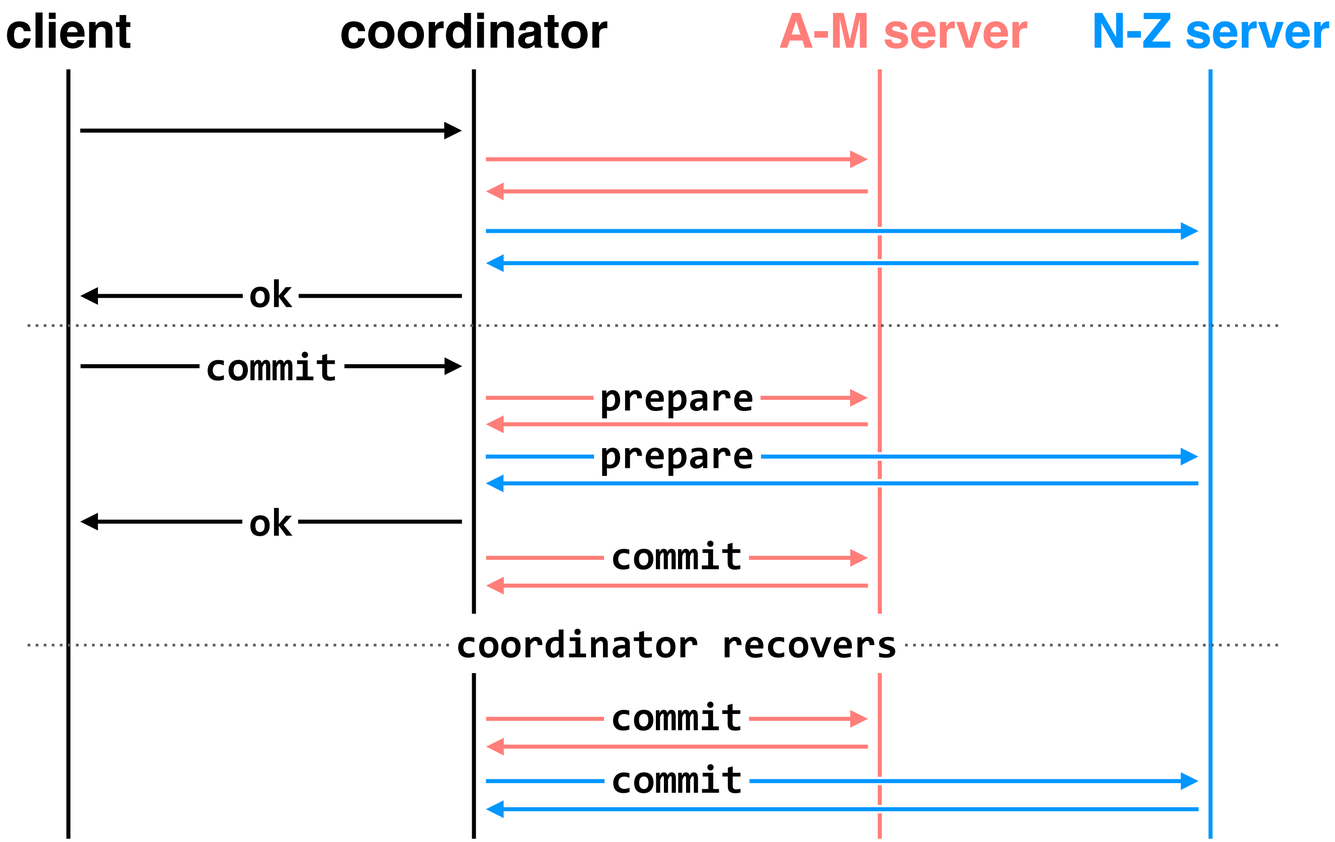
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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



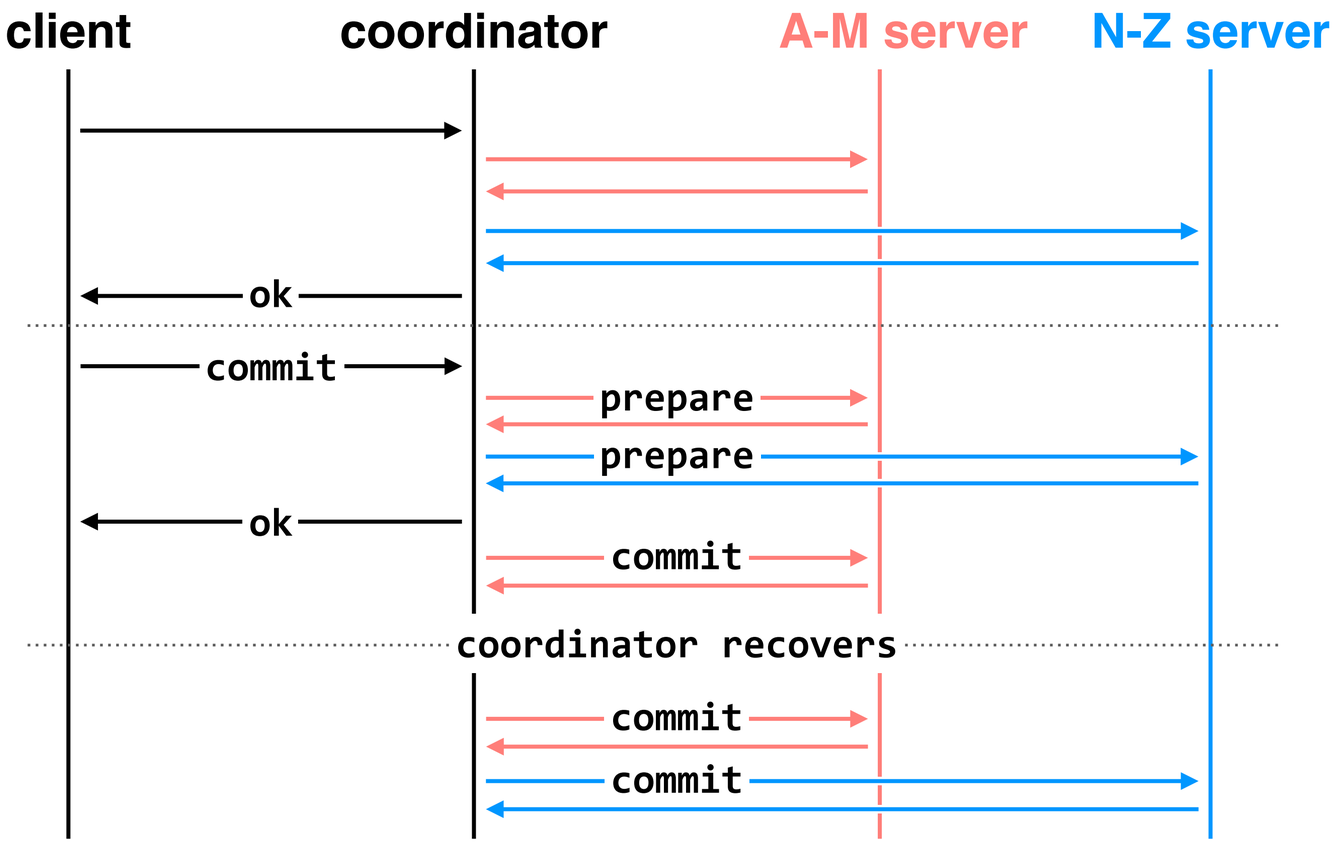
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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



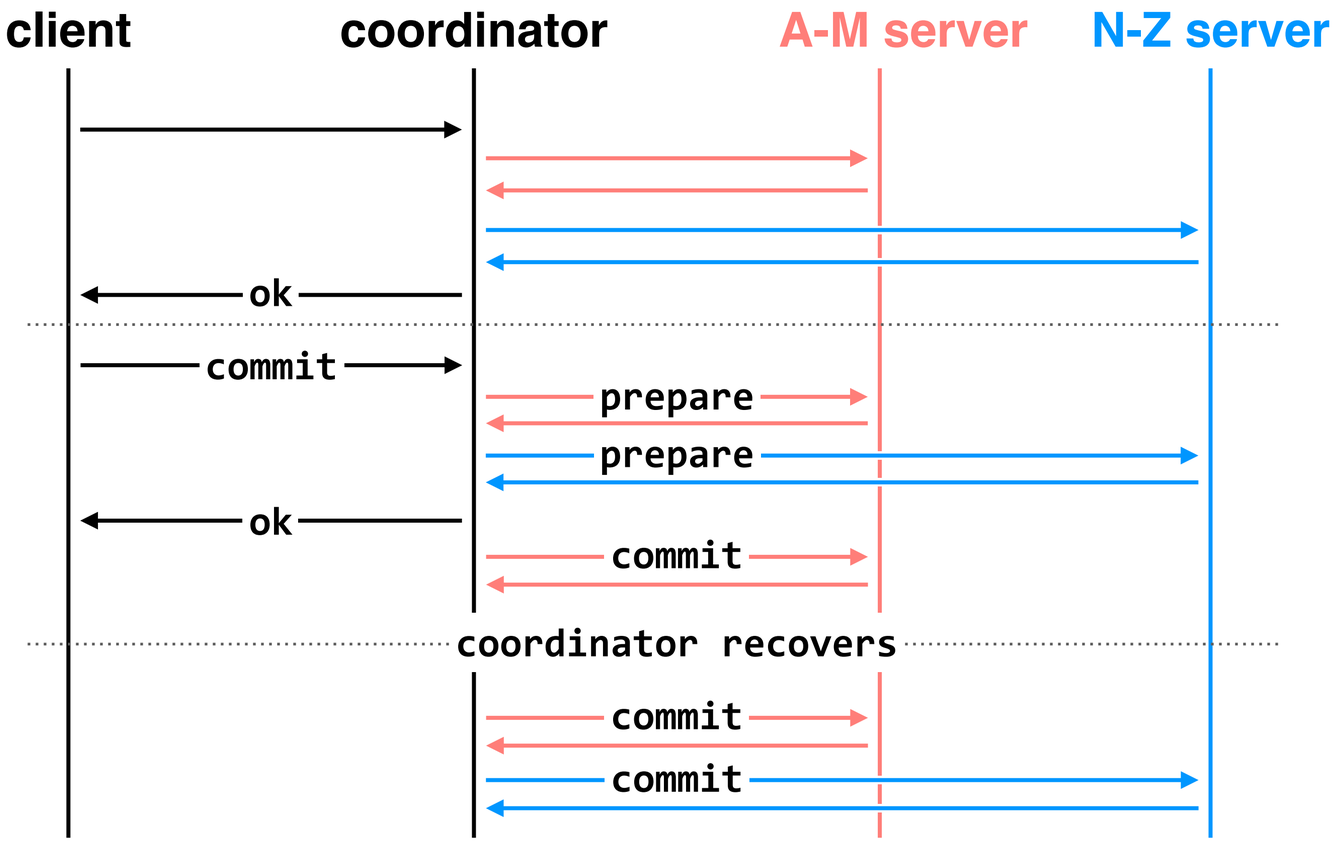
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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



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

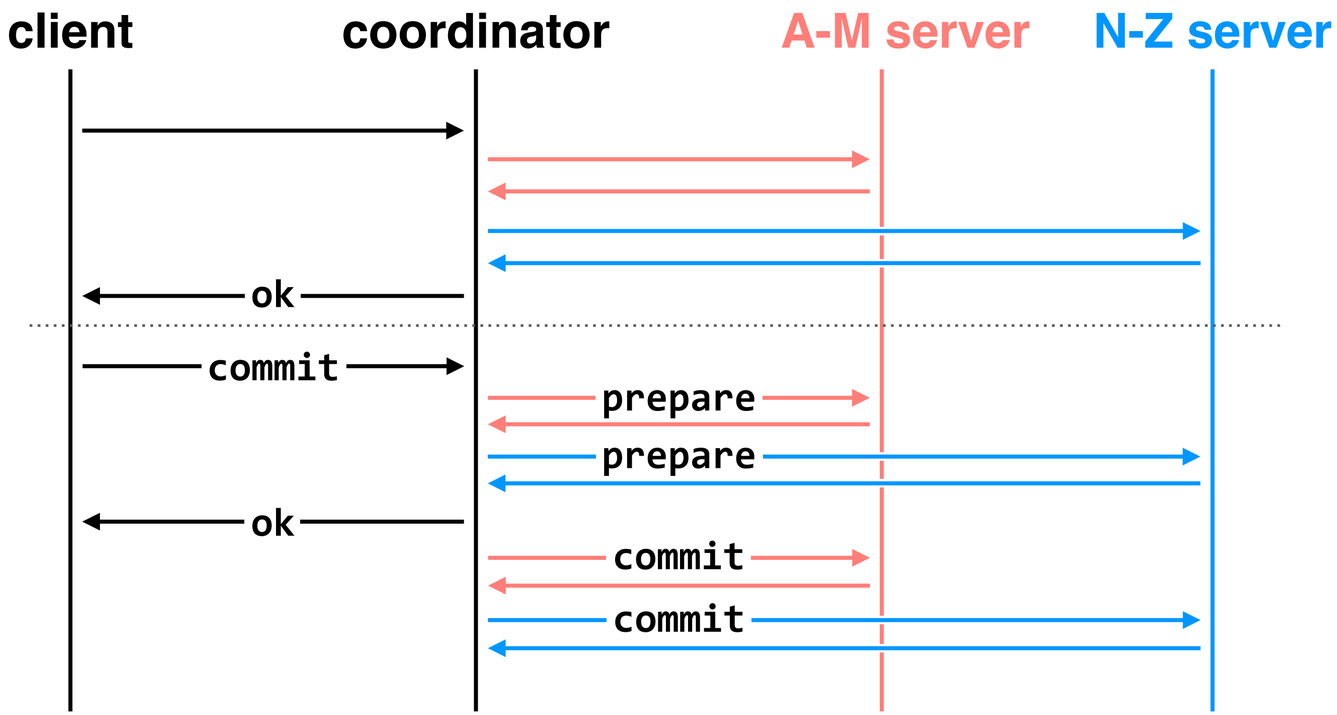
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

worker failure before prepare phase: coordinator can safely abort transaction 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



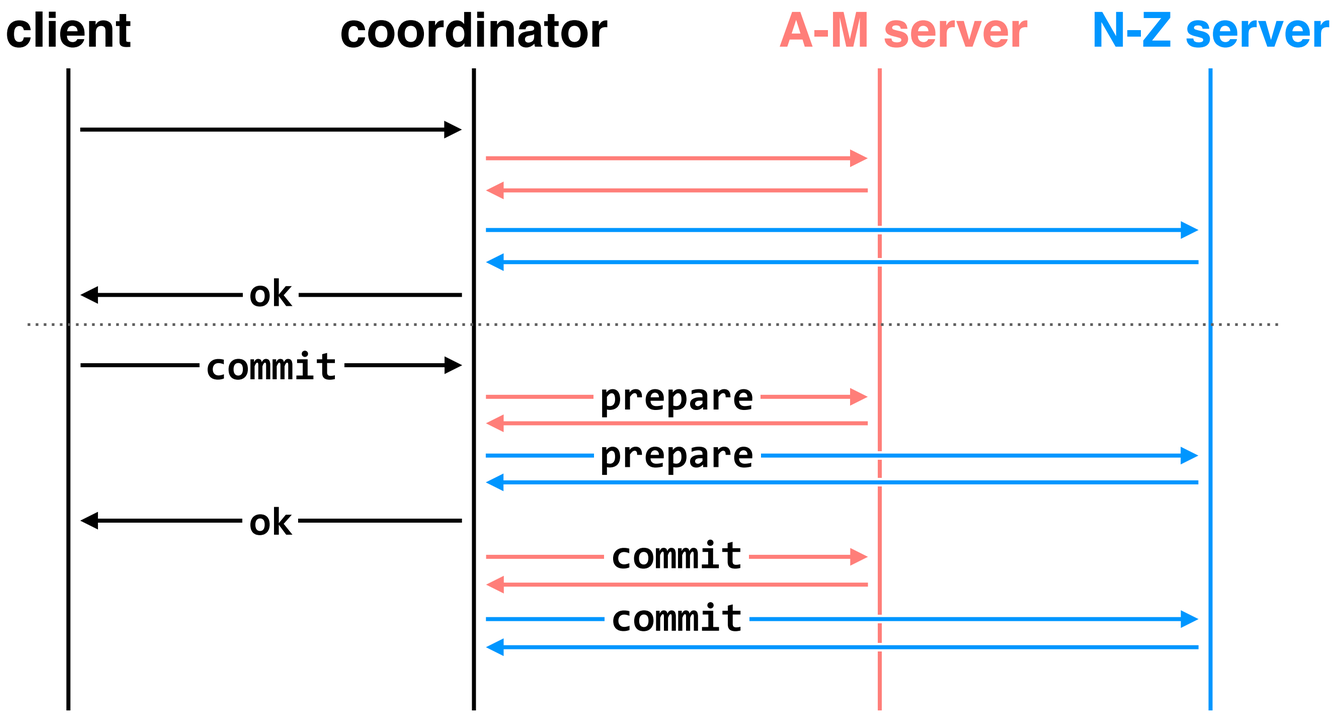
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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

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

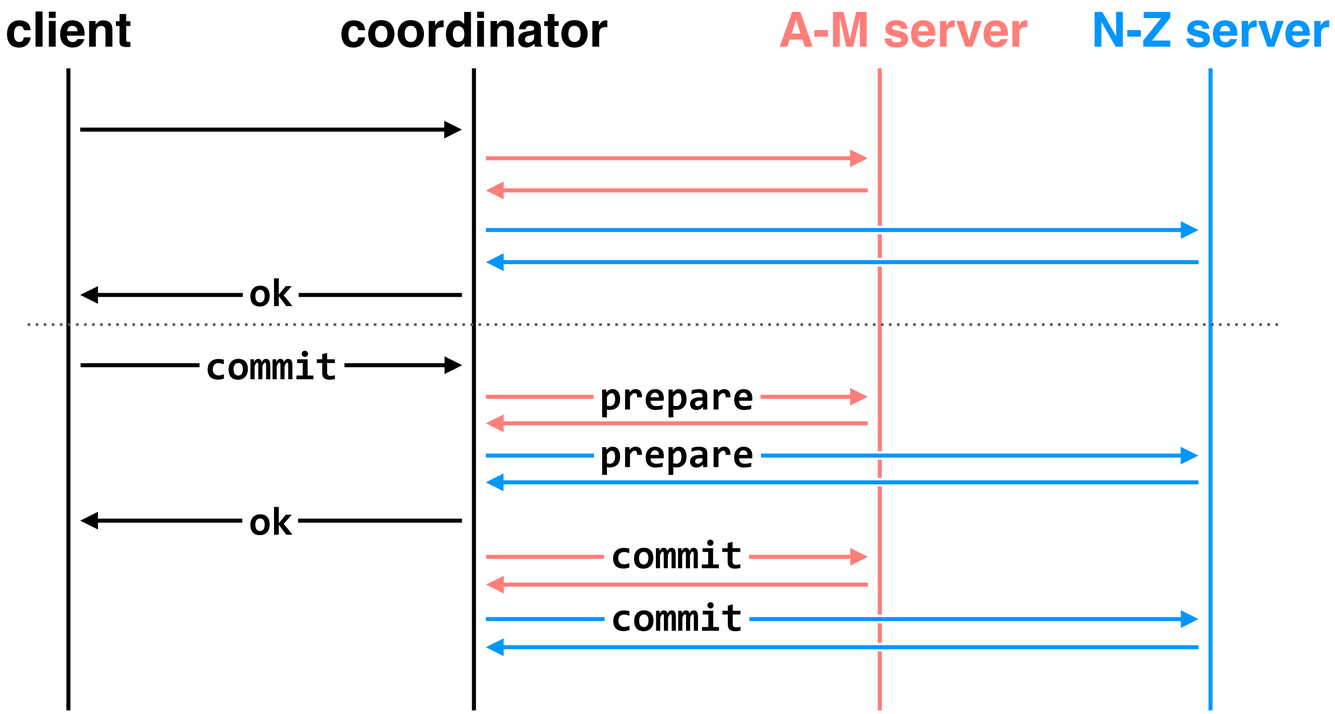
message loss at any stage: handled by reliable transport; coordinator will time out and resend message

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

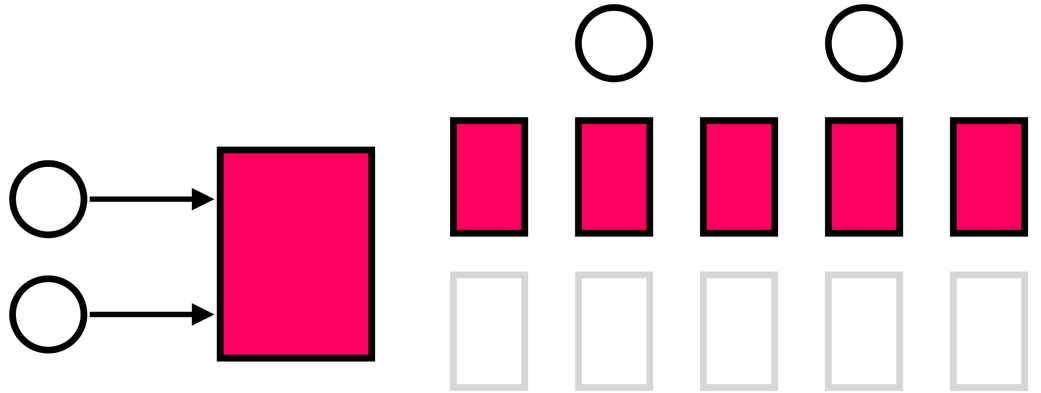
worker failure before prepare phase: coordinator can safely abort transaction without additional communication to workers

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

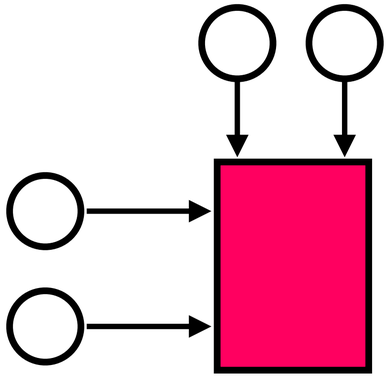
our job in lecture is to understand how a system *implements* these two abstractions.
how do our systems guarantee atomicity? how do they guarantee isolation?

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

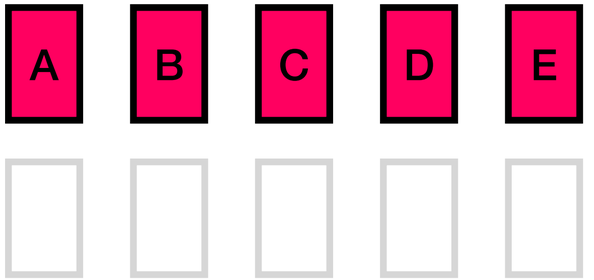
* shadow copies *are* used in some systems

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**.

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