The basics of file systems

Consensus in a distributed system

Today we are going to see how to implement a reliable distributed FSM.

Reminder

The algorithm of a proposer:

1. Choose an epoch number n that was not used earlier.
2. Send $\mathrm{prepare}(n)$ to all acceptors. It is a request to ignore requests with the epoch number $< n$.
3. Wait for $\mathrm{promise}(n, m_i, v_i)$ from a majority of acceptors and **remember a value v with the highest epoch number**. If all replies have $v_i = nil$, the proposer may propose its value.
4. Send $\mathrm{propose}(n, v)$ to all acceptors **that have replied with a promise**.
5. Wait for $\mathrm{accept}()$ from a majority of acceptors.
6. If timed out, go to #1.

The algorithm of an acceptor:

1. Upon receiving $\mathrm{prepare}(n)$, remember the epoch n and ignore all requests with lower epoch numbers. Reply with $\mathrm{promise}(n, m, v)$ where (m, v) is the last accepted value or (nil, nil) if no value has been accepted yet.
   Note: $\mathrm{promise}(n, m, v)$ has $n > m$.
2. Upon receiving $\mathrm{propose}(k, w)$ with k that is greater than the last remembered epoch, accept the value (k, w) and reply $\mathrm{accept}(k, w)$.

Reminder

The algorithm of a proposer:

1. Choose an epoch number n that was not used earlier.
2. Send $\mathrm{prepare}(n)$ to all acceptors. It is a request to ignore requests with the epoch number $< n$.
3. Wait for $\mathrm{promise}(n, m_i, v_i)$ from a majority of acceptors and **remember a value v with the highest epoch number**. If all replies have $v_i = nil$, the proposer may propose its value.
4. Send $\mathrm{propose}(n, v)$ to all acceptors **that have replied with a promise**.
5. Wait for $\mathrm{accept}()$ from a majority of acceptors.
6. If timed out, go to #1.

The algorithm of an acceptor:

1. Upon receiving $\mathrm{prepare}(n)$, remember the epoch n and ignore all requests with lower epoch numbers. Reply with $\mathrm{promise}(n, m, v)$ where (m, v) is the last accepted value or (nil, nil) if no value has been accepted yet.
   Note: $\mathrm{promise}(n, m, v)$ has $n > m$.
2. Upon receiving $\mathrm{propose}(k, w)$ with k that is greater than the last remembered epoch, accept the value (k, w) and reply $\mathrm{accept}(k, w)$.

Improvement: upon receiving a $\mathrm{propose}(k, w)$ with a stale epoch number k, an acceptor can reply with a NACK (Negative ACKnowledgement) to the proposer. PAXOS works fine if $\mathrm{propose}(k, w)$ is ignored and the proposer needs to time out. However, a NACK enables the proposer to restart much sooner.

Replicated FSM

From now on we will assume that every process plays all 3 PAXOS roles (proposer, acceptor and learner). Moreover, assume there is only one process proposing new values.

The N-th step of a replicated journal proceeds this way:

1. choose a new epoch number and send prepare(),
2. wait for the other processes to reply with promise()s,
3. send propose() with the value of step N,
4. wait for the other processes to accept() it,
5. send commit() to the other processes to persist the new length of the journal,
6. wait for other processes to confirm the commit() and reply to a client that started the step N.

Replicated FSM

From now on we will assume that every process plays all 3 PAXOS roles (proposer, acceptor and learner). Moreover, assume there is only one process proposing new values.

The N-th step of a replicated journal proceeds this way:

1. choose a new epoch number and send prepare(),
2. wait for the other processes to reply with promise()s,
3. send propose() with the value of step N,
4. wait for the other processes to accept() it,
5. send commit() to the other processes to persist the new length of the journal,
6. wait for other processes to confirm the commit() and reply to a client that started the step N.

Questions:

1. Suppose a process lags behind the cluster (suppose it had network problems for a while). How does it catch up?
2. How many `fsync()`s does each step need?

Replicated FSM

From now on we will assume that every process plays all 3 PAXOS roles (proposer, acceptor and learner). Moreover, assume there is only one process proposing new values.

The N-th step of a replicated journal proceeds this way:

1. choose a new epoch number and send prepare(),
2. wait for the other processes to reply with promise()s,
3. send propose() with the value of step N,
4. wait for the other processes to accept() it,
5. send commit() to the other processes to persist the new length of the journal,
6. wait for other processes to confirm the commit() and reply to a client that started the step N.

Questions:

1. Suppose a process lags behind the cluster (suppose it had network problems for a while). How does it catch up?
   - Every prepare() and promise() from a process P includes the number of the last step committed by P. This way processes can learn that a cluster member is lagging and can send it the missing tail.
2. How many `fsync()`s does each step need?

Replicated FSM

From now on we will assume that every process plays all 3 PAXOS roles (proposer, acceptor and learner). Moreover, assume there is only one process proposing new values.

The N-th step of a replicated journal proceeds this way:

1. choose a new epoch number and send prepare(),
2. wait for the other processes to reply with promise()s,
3. send propose() with the value of step N,
4. wait for the other processes to accept() it,
5. send commit() to the other processes to persist the new length of the journal,
6. wait for other processes to confirm the commit() and reply to a client that started the step N.

Questions:

1. Suppose a process lags behind the cluster (suppose it had network problems for a while). How does it catch up?
2. How many `fsync()`s does each step need?
   - the proposer needs to persist the epoch number in order to not reuse it,
   - acceptors need to persist the epoch number to NACK requests with lower epochs,
   - acceptors need to persist a value that was proposed to them,
   - acceptors need to persist the journal length upon commit.

   Overall, we need 4 `fsync()`s that go back-to-back.

Replicated FSM

From now on we will assume that every process plays all 3 PAXOS roles (proposer, acceptor and learner). Moreover, assume there is only one process proposing new values.

The N-th step of a replicated journal proceeds this way:

1. choose a new epoch number and send prepare(),
2. wait for the other processes to reply with promise()s,
3. send propose() with the value of step N,
4. wait for the other processes to accept() it,
5. send commit() to the other processes to persist the new length of the journal,
6. wait for other processes to confirm the commit() and reply to a client that started the step N.

Questions:

1. Suppose a process lags behind the cluster (suppose it had network problems for a while). How does it catch up?
2. How many `fsync()`s does each step need?
   Overall, we need 4 `fsync()`s that go back-to-back.

Problem: `fsync()`s have a pronounced negative impact on the performance:

1. PAXOS runs at least 4x slower than the storage of cluster members.
2. At steps 2-5, every member must make `fsync()`s of its own which makes PAXOS vulnerable to tail latencies.

Multi-PAXOS

From now on we will assume that every process plays all 3 PAXOS roles (proposer, acceptor and learner). Moreover, assume there is only one process proposing new values.

The N-th step of a replicated journal proceeds this way:

1. choose a new epoch number and send prepare(),
2. wait for the other processes to reply with promise()s,
3. send propose() with the value of step N,
4. wait for the other processes to accept() it,
5. send commit() to the other processes to persist the new length of the journal,
6. wait for other processes to confirm the commit() and reply to a client that started the step N.

Idea: a proposer may do steps 1 and 2 only once, and then loop through steps 3 to 6 until receiving a NACK. A NACK means that another participant decided to become the distinguished proposer and a proposer election must be repeated.

To prove the correctness, one can interpret the loop over steps 3 to 6 as a single round of PAXOS that selects a value "transitions to make in steps N_0 through N_1" where N_1 is the last round number where step 6 succeeded.

This way most appends to the journal needs only 2 `fsync()`s instead of 4.

Multi-PAXOS

From now on we will assume that every process plays all 3 PAXOS roles (proposer, acceptor and learner). Moreover, assume there is only one process proposing new values.

The N-th step of a replicated journal proceeds this way:

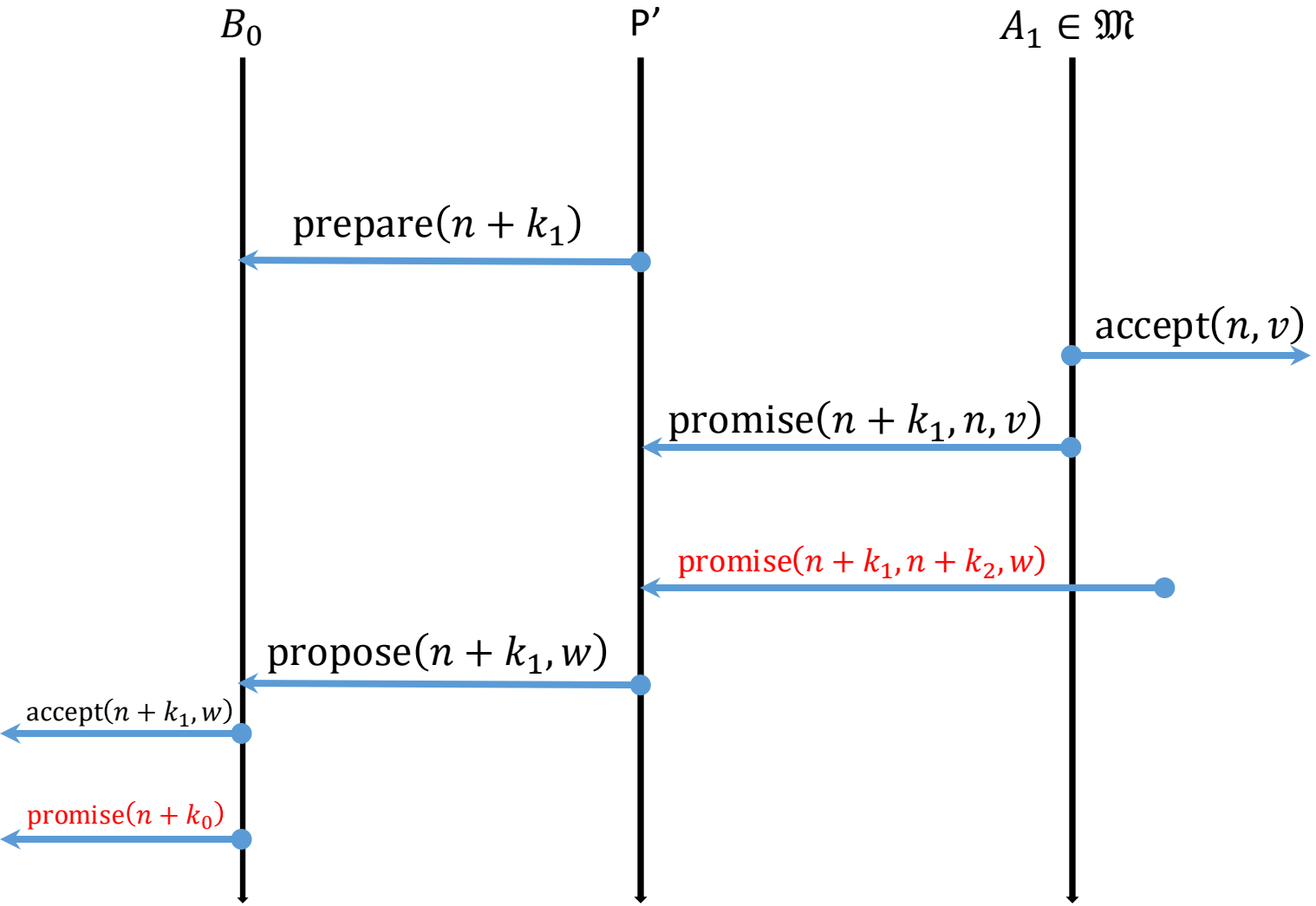
1. choose a new epoch number and send prepare(),
2. wait for the other processes to reply with promise()s,
3. send propose() with the value of step N,
4. wait for the other processes to accept() it,
5. send commit() to the other processes to persist the new length of the journal,
6. wait for other processes to confirm the commit() and reply to a client that started the step N.

Idea: the number of `fsync()`s can be reduced further if every iteration of steps 3 to 6 processes a batch of requests instead of single requests. This is possible if a service has multiple clients, and the request queue is always deep enough.

Question: how to choose the batch sizes? – Adaptive batching.

Safety (only one value can be chosen)

It is **not** possible that a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted a different value $(n + k, w)$ with $k > 0$ and $w \neq v$.

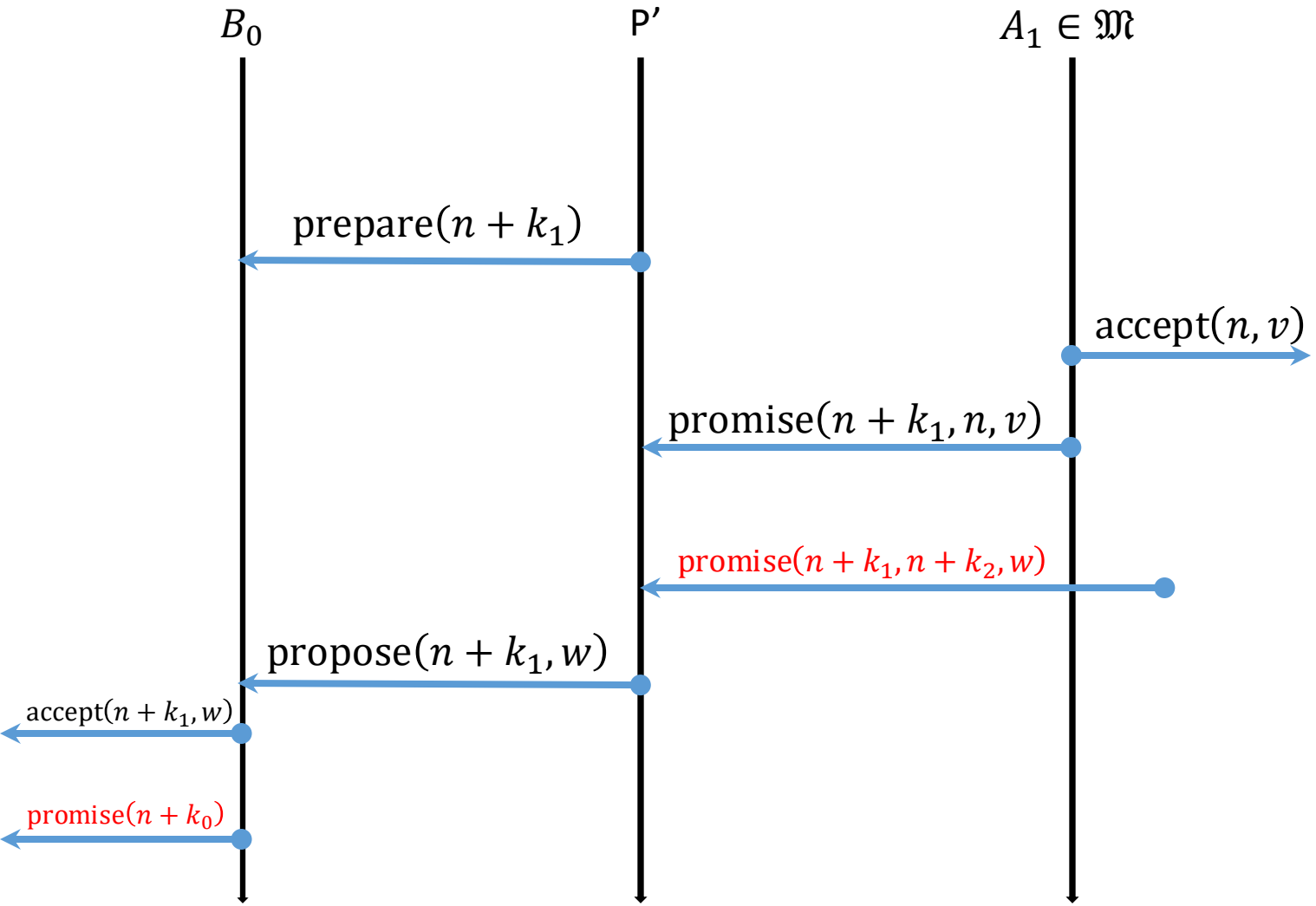


Last time we've proven the following statement:

If a value (n, v) was chosen in epoch n then all subsequent messages propose$(n + k, \cdot), k > 0$, propose the same value v.

Safety (only one value can be chosen)

It is **not** possible that a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted a different value $(n + k, w)$ with $k > 0$ and $w \neq v$.



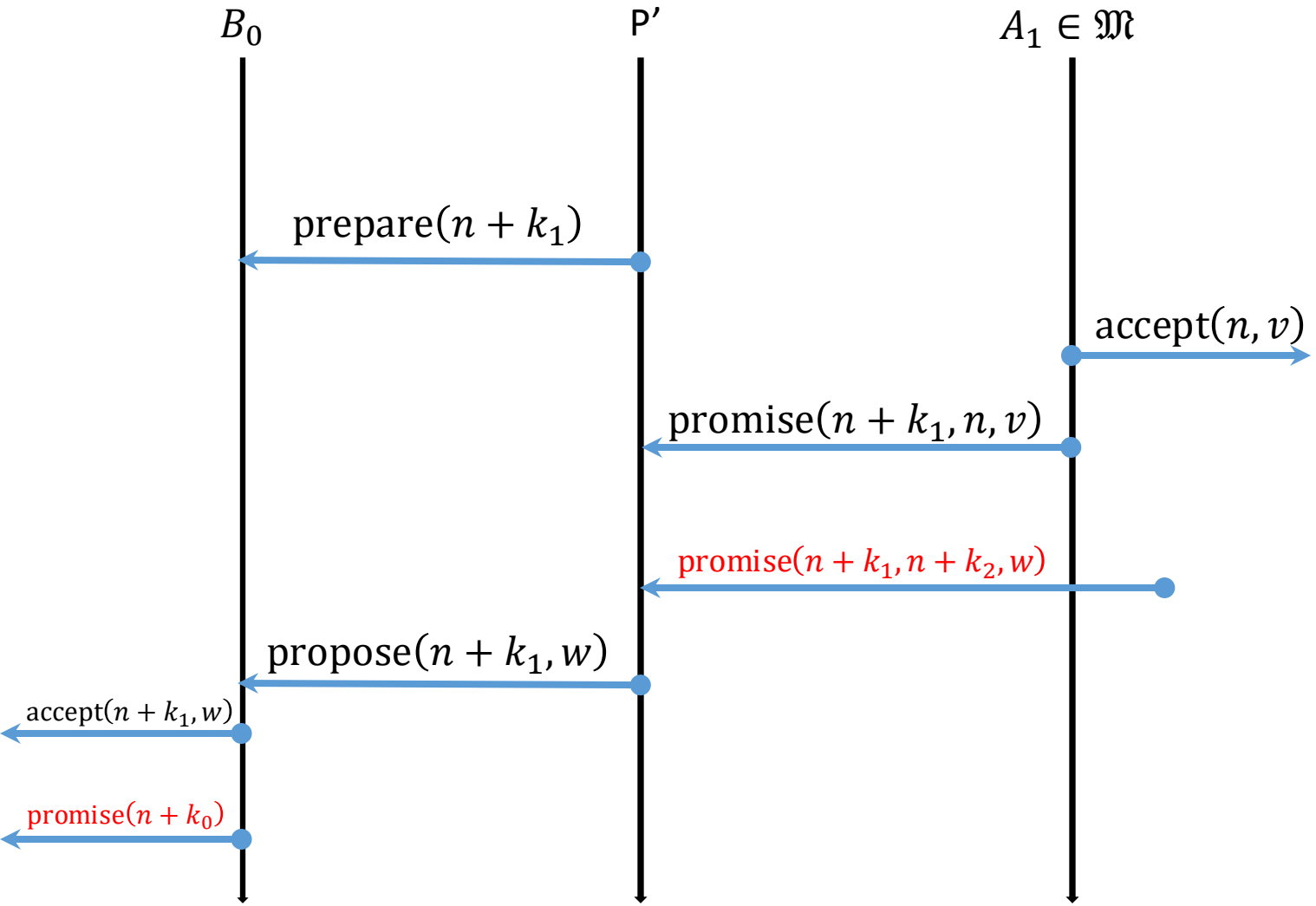
Last time we've proven the following statement:

If a value (n, v) was chosen in epoch n then all subsequent messages propose$(n + k, \cdot), k > 0$, propose the same value v.

Indeed: propose$(n + k, w), w \neq v$ must be sent to a majority of acceptors, so it must be sent to at least one acceptor from the majority that accepted (n, v). Last time we've shown that such scenario is not possible.

Safety (only one value can be chosen)

It is **not** possible that a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted a different value $(n + k, w)$ with $k > 0$ and $w \neq v$.



Last time we've proven the following statement:

If a value (n, v) was chosen in epoch n then all subsequent messages propose$(n + k, \cdot), k > 0$, propose the same value v.

Indeed: propose$(n + k, w), w \neq v$ must be sent to a majority of acceptors, so it must be sent to at least one acceptor from the majority that accepted (n, v). Last time we've shown that such scenario is not possible.

Conversely, if it is not possible to send propose$(n + k, w), w \neq v$ to an acceptor that had already accepted v, then PAXOS chooses at most one value out of proposed values.

Generalised quorums

The algorithm of a proposer:

1. Choose an epoch number n that was not used earlier.
2. Send $\mathrm{prepare}(n)$ to all acceptors. It is a request to ignore requests with the epoch number $< n$.
3. Wait for $\mathrm{promise}(n, m_i, v_i)$ from a majority of acceptors and **remember a value v with the highest epoch number**. If all replies have $v_i = nil$, the proposer may propose its value.
4. Send $\mathrm{propose}(n, v)$ to all acceptors **that have replied with a promise**.
5. Wait for $\mathrm{accept}()$ from a majority of acceptors.
6. If timed out, go to #1.

If a value (n, v) was chosen in epoch n then all subsequent messages $\mathrm{propose}(n + k, \ \cdot), k > 0$, propose the same value v.

Generalised quorums

The algorithm of a proposer:

1. Choose an epoch number n that was not used earlier.
2. Send $\mathrm{prepare}(n)$ to all acceptors. It is a request to ignore requests with the epoch number $< n$.
3. Wait for $\mathrm{promise}(n, m_i, v_i)$ from a majority of acceptors and **remember a value v with the highest epoch number**. If all replies have $v_i = nil$, the proposer may propose its value.
4. Send $\mathrm{propose}(n, v)$ to all acceptors **that have replied with a promise**.
5. Wait for $\mathrm{accept}()$ from a majority of acceptors.
6. If timed out, go to #1.

If a value (n, v) was chosen in epoch n then all subsequent messages $\mathrm{propose}(n + k, \ \cdot), k > 0$, propose the same value v.

This property is true if the set of acceptors that reply with a `promise()` at step 3 intersects the set of acceptors that had accepted (n, v). The sets of acceptors that reply in steps 3 and 5 need not be majorities. They only need to intersect.

Generalised quorums

The algorithm of a proposer:

1. Choose an epoch number n that was not used earlier.
2. Send $\mathrm{prepare}(n)$ to all acceptors. It is a request to ignore requests with the epoch number $< n$.
3. Wait for $\mathrm{promise}(n, m_i, v_i)$ from a majority of acceptors and **remember a value v with the highest epoch number**. If all replies have $v_i = nil$, the proposer may propose its value.
4. Send $\mathrm{propose}(n, v)$ to all acceptors **that have replied with a promise**.
5. Wait for $\mathrm{accept}()$ from a majority of acceptors.
6. If timed out, go to #1.

If a value (n, v) was chosen in epoch n then all subsequent messages $\mathrm{propose}(n + k, \ \cdot), k > 0$, propose the same value v.

This property is true if the set of acceptors that reply with a `promise()` at step 3 intersects the set of acceptors that had accepted (n, v). The sets of acceptors that reply in steps 3 and 5 need not be majorities. They only need to intersect.

We can introduce separate quorums for PAXOS phase 1 (prepare/promise) and phase 2 (propose/accept). The quorums are families of sets such that any phase 1 quorum intersects every phase 2 quorum.

This version of PAXOS is called “Flexible PAXOS”.

- https://arxiv.org/pdf/1608.06696.pdf,
- https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-935.pdf, chapter “Quorum intersection revised”.

Generalised quorums

The algorithm of a proposer:

1. Choose an epoch number n that was not used earlier.
2. Send $\mathrm{prepare}(n)$ to all acceptors. It is a request to ignore requests with the epoch number $< n$.
3. Wait for $\mathrm{promise}(n, m_i, v_i)$ from a majority of acceptors and **remember a value v with the highest epoch number**. If all replies have $v_i = nil$, the proposer may propose its value.
4. Send $\mathrm{propose}(n, v)$ to all acceptors **that have replied with a promise**.
5. Wait for $\mathrm{accept}()$ from a majority of acceptors.
6. If timed out, go to #1.

If a value (n, v) was chosen in epoch n then all subsequent messages $\mathrm{propose}(n + k, \ \cdot), k > 0$, propose the same value v.

This property is true if the set of acceptors that reply with a `promise()` at step 3 intersects the set of acceptors that had accepted (n, v). The sets of acceptors that reply in steps 3 and 5 need not be majorities. They only need to intersect.

We can introduce separate quorums for PAXOS phase 1 (prepare/promise) and phase 2 (propose/accept). The quorums are families of sets such that any phase 1 quorum intersects every phase 2 quorum.

Example: if a cluster has N members then we may use subsets with N-1 elements as phase 1 quorums, and subsets with 2 elements as phase 2 quorums.

This version of PAXOS is called “Flexible PAXOS”.

- https://arxiv.org/pdf/1608.06696.pdf,
- https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-935.pdf, chapter “Quorum intersection revised”.

Multi-PAXOS + Flexible PAXOS

From now on we will assume that every process plays all 3 PAXOS roles (proposer, acceptor and learner). Moreover, assume there is only one process proposing new values.

The N-th step of a replicated journal proceeds this way:

1. choose a new epoch number and send prepare(),
2. wait for the other processes to reply with promise()s,
3. send propose() with the value of step N,
4. wait for the other processes to accept() it,
5. send commit() to the other processes to persist the new length of the journal,
6. wait for other processes to confirm the commit() and reply to a client that started the step N.

Observation: the latency of phase 2 depends on the size of phase 2 quorums. We can decrease it by decreasing the size of phase 2 quorums which is possible if we make phase 1 quorums larger.

Multi-PAXOS + Flexible PAXOS

From now on we will assume that every process plays all 3 PAXOS roles (proposer, acceptor and learner). Moreover, assume there is only one process proposing new values.

The N-th step of a replicated journal proceeds this way:

1. choose a new epoch number and send prepare(),
2. wait for the other processes to reply with promise()s,
3. send propose() with the value of step N,
4. wait for the other processes to accept() it,
5. send commit() to the other processes to persist the new length of the journal,
6. wait for other processes to confirm the commit() and reply to a client that started the step N.

Observation: the latency of phase 2 depends on the size of phase 2 quorums. We can decrease it by decreasing the size of phase 2 quorums which is possible if we make phase 1 quorums larger.

Batching + small phase 2 quorums reduce the number of `fsync()`s needed per each operation and make tail latencies less pronounced.

Still, PAXOS is not a tool to increase the throughput of a system by adding more nodes. It is a way to implement fault-tolerance. Typically, PAXOS is used to persist decisions on partitioning a system and choosing masters for subsets of data.

Replicated FSM

From now on we will assume that every process plays all 3 PAXOS roles (proposer, acceptor and learner). Moreover, assume there is only one process proposing new values.

The N-th step of a replicated journal proceeds this way:

1. choose a new epoch number and send prepare(),
2. wait for the other processes to reply with promise()s,
3. send propose() with the value of step N,
4. wait for the other processes to accept() it,
5. send commit() to the other processes to persist the new length of the journal,
6. wait for other processes to confirm the commit() and reply to a client that started the step N.

Questions:

1. Suppose a process lags behind the cluster (suppose it had network problems for a while). How does it catch up?
   - Every prepare() and promise() from a process P includes the number of the last step committed by P. This way processes can learn that a cluster member is lagging and can send it the missing tail.
2. How many `fsync()`s does each step need?

Replicated FSM

Problem: the journal of an FSM grows indefinitely, but cluster members do not have unlimited disk space.

Replicated FSM

Problem: the journal of an FSM grows indefinitely, but cluster members do not have unlimited disk space.

Solution: make regular snapshots of the FSM and store only the journal tail that has changes since the last snapshot.

Replicated FSM

Problem: the journal of an FSM grows indefinitely, but cluster members do not have unlimited disk space.

Solution: make regular snapshots of the FSM and store only the journal tail that has changes since the last snapshot.

More problems:

1. Making a snapshot may not block modifications of the FSM so as to introduce no latency.
2. Cluster nodes that lag behind* can no longer catch up by copying a part of the journal. They may need to copy a whole snapshot. What happens if copying a snapshot S takes too long to transfer and S is replaced with S’ during the copy?

** A cluster need not be unhealthy for some nodes to lag behind. Consider adding a new node to a cluster.*

Reminder: the failure model of PAXOS

1. Participant processes may work at arbitrary speed.
2. Processes may crash and restart at any moment.
3. Messages may be delayed (in particular, reordered), lost, or duplicated, but they cannot be corrupted.

Network-ordered PAXOS

1. Participant processes may work at arbitrary speed.
2. Processes may crash and restart at any moment.
3. Messages may be delayed (in particular, reordered), lost, or duplicated, but they cannot be corrupted.

These assumptions are true in commodity networks, but custom hardware gives us more options.

Network-ordered PAXOS

1. Participant processes may work at arbitrary speed.
2. Processes may crash and restart at any moment.
3. Messages may be delayed (in particular, reordered), lost, or duplicated, but they cannot be corrupted.

These assumptions are true in commodity networks, but custom hardware gives us more options.

Programmable switches can modify IP packets on the fly and can order packets that arrive via different ports:

- Intel Tofino: https://hc32.hotchips.org/assets/program/conference/day2/HotChips2020_Networking_Tofino.pdf
- Microsoft Sirius (in use by Azure): https://www.usenix.org/system/files/nsdi23-bansal.pdf

Network-ordered PAXOS

1. Participant processes may work at arbitrary speed.
2. Processes may crash and restart at any moment.
3. Messages may be delayed (in particular, reordered), lost, or duplicated, but they cannot be corrupted.

These assumptions are true in commodity networks, but custom hardware gives us more options.

Programmable switches can modify IP packets on the fly and can order packets that arrive via different ports:

- Intel Tofino: https://hc32.hotchips.org/assets/program/conference/day2/HotChips2020_Networking_Tofino.pdf
- Microsoft Sirius (in use by Azure): https://www.usenix.org/system/files/nsdi23-bansal.pdf

A programmable switch can augment every IP packet with an additional header that contains a global sequence counter.

Network-ordered PAXOS

1. Participant processes may work at arbitrary speed.
2. Processes may crash and restart at any moment.
3. Messages may be delayed (in particular, reordered), lost, or duplicated, but they cannot be corrupted.

With a programmable switch, one can implement Ordered Unreliable Multicast. It is a network operation that has the following properties:

1. If two messages m and m' are sent to a set of nodes R, then every node in R receives them in the same order (if it receives them at all).
2. If a message m is sent to a set of nodes R then either
    1. Every node in R receives the message m itself or a message "m was lost".
    2. No node in R receives m, nor "m was lost".

We can use Ordered Unreliable Multicast to send `propose()`s and `accept()`s. Thanks to OUM, all acceptors receive `propose()`s in the same order and can accept them immediately. Loss of any `propose()` or `accept()` is also detected by OUM. Process(es) that detected a message loss switch to phase 1 of PAXOS and start the re-election of the distinguished proposer.

- https://www.usenix.org/system/files/conference/osdi16/osdi16-li.pdf
  Just say NO to PAXOS overhead: replacing consensus with network ordering.

Failures of cluster nodes

1. Participant processes may work at arbitrary speed.
2. Processes may crash and restart at any moment.
3. Messages may be delayed (in particular, reordered), lost, or duplicated, but they cannot be corrupted.

Failures of cluster nodes

1. Participant processes may work at arbitrary speed.
2. Processes may crash and restart at any moment.
3. Messages may be delayed (in particular, reordered), lost, or duplicated, but they cannot be corrupted.

PAXOS assumes that cluster nodes follow the rules of PAXOS even in the face of

- bit rot that may randomly corrupt the on-disk or in-memory state of nodes,
- bugs in the implementation of nodes.

It is not feasible or reasonable to prove the correctness of cluster nodes in most cases, but we can make failures rare enough:

- Consistency checks at all steps,
- Randomised tests and fault injection.

Failures of cluster nodes

1. Participant processes may work at arbitrary speed.
2. Processes may crash and restart at any moment.
3. Messages may be delayed (in particular, reordered), lost, or duplicated, but they cannot be corrupted.

PAXOS assumes that cluster nodes follow the rules of PAXOS even in the face of

- bit rot that may randomly corrupt the on-disk or in-memory state of nodes,
- bugs in the implementation of nodes.

It is not feasible or reasonable to prove the correctness of cluster nodes in most cases, but we can make failures rare enough:

- Consistency checks at all steps,
- Randomised tests and fault injection.

- Chaos engineering: https://github.com/netflix/chaosmonkey

Failures of cluster nodes

1. Participant processes may work at arbitrary speed.
2. Processes may crash and restart at any moment.
3. Messages may be delayed (in particular, reordered), lost, or duplicated, but they cannot be corrupted.

PAXOS assumes that cluster nodes follow the rules of PAXOS even in the face of

- bit rot that may randomly corrupt the on-disk or in-memory state of nodes,
- bugs in the implementation of nodes.

It is not feasible or reasonable to prove the correctness of cluster nodes in most cases, but we can make failures rare enough:

- Consistency checks at all steps,
- Randomised tests and fault injection.

- Chaos engineering: https://github.com/netflix/chaosmonkey
- Distributed systems safety research: https://jepsen.io

Failures of cluster nodes

1. Participant processes may work at arbitrary speed.
2. Processes may crash and restart at any moment.
3. Messages may be delayed (in particular, reordered), lost, or duplicated, but they cannot be corrupted.

PAXOS assumes that cluster nodes follow the rules of PAXOS even in the face of

- bit rot that may randomly corrupt the on-disk or in-memory state of nodes,
- bugs in the implementation of nodes.

It is not feasible or reasonable to prove the correctness of cluster nodes in most cases, but we can make failures rare enough:

- Consistency checks at all steps,
- Randomised tests and fault injection.

- Chaos engineering: https://github.com/netflix/chaosmonkey
- Distributed systems safety research: https://jepsen.io
- Deterministic simulation:
  - https://github.com/asatarin/testing-distributed-systems?tab=readme-ov-file#foundationdb
  - https://antithesis.com/blog/deterministic_hypervisor

Failures of cluster nodes

1. Participant processes may work at arbitrary speed.
2. Processes may crash and restart at any moment.
3. Messages may be delayed (in particular, reordered), lost, or duplicated, but they cannot be corrupted.

PAXOS assumes that cluster nodes follow the rules of PAXOS even in the face of

- bit rot that may randomly corrupt the on-disk or in-memory state of nodes,
- bugs in the implementation of nodes.

It is not feasible or reasonable to prove the correctness of cluster nodes in most cases, but we can make failures rare enough:

- Consistency checks at all steps,
- Randomised tests and fault injection.

- Chaos engineering: https://github.com/netflix/chaosmonkey
- Distributed systems safety research: https://jepsen.io
- Deterministic simulation:
  - https://github.com/asatarin/testing-distributed-systems?tab=readme-ov-file#foundationdb
  - https://antithesis.com/blog/deterministic_hypervisor
- Fuzz testing, generation of random test inputs and minimisation of testcases:
  - https://go.dev/doc/security/fuzz/
  - https://www.cse.chalmers.se/edu/year/2012/course/DIT848/files/13-GL-QuickCheck.pdf
  - https://github.com/google/AFL

Failures of cluster nodes

1. Participant processes may work at arbitrary speed.
2. Processes may crash and restart at any moment.
3. Messages may be delayed (in particular, reordered), lost, or duplicated, but they cannot be corrupted.

PAXOS assumes that cluster nodes follow the rules of PAXOS even in the face of

- bit rot that may randomly corrupt the on-disk or in-memory state of nodes,
- bugs in the implementation of nodes.

It is not feasible or reasonable to prove the correctness of cluster nodes in most cases, but we can make failures rare enough:

- Consistency checks at all steps,
- Randomised tests and fault injection.

- Chaos engineering: https://github.com/netflix/chaosmonkey
- Distributed systems safety research: https://jepsen.io
- Deterministic simulation:
  - https://github.com/asatarin/testing-distributed-systems?tab=readme-ov-file#foundationdb
  - https://antithesis.com/blog/deterministic_hypervisor
- Fuzz testing, generation of random test inputs and minimisation of testcases:
  - https://go.dev/doc/security/fuzz/
  - https://www.cse.chalmers.se/edu/year/2012/course/DIT848/files/13-GL-QuickCheck.pdf
  - https://github.com/google/AFL
- Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems: https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf

Another example of a complex state space

```
pthread_mutex_t mtx;
pthread_cond_t cv;
...
std::deque<int> queue;
constexpr sizeLimit = ...;
...


void put(int x)
{
    pthread_mutex_lock(&mtx);

    while (queue.size() >= sizeLimit)
        pthread_cond_wait(&cv, &mtx);
    queue.push_back(x);

    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&mtx);
}
```

```
int get()
{
    int x;

    pthread_mutex_lock(&mtx);

    while (queue.empty())
        pthread_cond_wait(&cv, &mtx);
    x = queue.pop_front();

    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&mtx);

    return x;
}
```

The problem: implement a bounded queue that supports multiple concurrent readers and writers.

The implementation on this slide has a problem that

1. happens only when the total number of users is greater than the maximum queue depth,
2. happens rarely even if condition #1 is satisfied.

Another example of a complex state space

```
pthread_mutex_t mtx;
pthread_cond_t cv;
...
std::deque<int> queue;
constexpr sizeLimit = ...;
...


void put(int x)
{
    pthread_mutex_lock(&mtx);

    while (queue.size() >= sizeLimit)
        pthread_cond_wait(&cv, &mtx);
    queue.push_back(x);

    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&mtx);
}
```

```
int get()
{
    int x;

    pthread_mutex_lock(&mtx);

    while (queue.empty())
        pthread_cond_wait(&cv, &mtx);
    x = queue.pop_front();

    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&mtx);

    return x;
}
```

The problem: implement a bounded queue that supports multiple concurrent readers and writers.

The implementation on this slide has a problem that

1. happens only when the total number of users is greater than the maximum queue depth,
2. happens rarely even if condition #1 is satisfied.

Another example of a complex state space

```
pthread_mutex_t mtx;
pthread_cond_t cv;
...
std::deque<int> queue;
constexpr sizeLimit = ...;
...


void put(int x)
{
    pthread_mutex_lock(&mtx);

    while (queue.size() >= sizeLimit)
        pthread_cond_wait(&cv, &mtx);
    queue.push_back(x);

    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&mtx);
}
```

```
int get()
{
    int x;

    pthread_mutex_lock(&mtx);

    while (queue.empty())
        pthread_cond_wait(&cv, &mtx);
    x = queue.pop_front();

    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&mtx);

    return x;
}
```

The problem: implement a bounded queue that supports multiple concurrent readers and writers.

The implementation on this slide has a problem that

1. happens only when the total number of users is greater than the maximum queue depth,
2. happens rarely even if condition #1 is satisfied.

Readers are supposed to wake writers, and vice versa.

However, `pthread_cond_signal()` wakes a random sleeping thread.

A model of a bounded queue

Let us model the program from the previous slide with TLA+.

See: Specifying systems, https://lamport.azurewebsites.net/tla/book-02-08-08.pdf.

We will use TLA+ Model Checker that walks the state space of a model and verifies whether or not a predicate holds in all states. It can do both breadth-first search and randomised depth-first search. Breadth-first search is exhaustive and yields shortest paths to states that invalidate a predicate. Depth-first search can explore parts of state spaces that are too big.

A model of a bounded queue

Let us model the program from the previous slide with TLA+.

See: Specifying systems, https://lamport.azurewebsites.net/tla/book-02-08-08.pdf.

We will proceed this way:

1. Declare tunable parameters of the model. Typically, these are parameters like the number of participant processes, the maximum capacity of queues, etc.
2. Declare variables that describe the state. Unlike tunables from #1, variables change upon every state transition.
3. Declare predicates that hold at every state of the model.
4. Declare a predicate that defines the initial state.
5. Declare predicates that hold iff the model makes a specific state transition. Examples: “A sends a message to B”, ”B receives a message from A”, “a message is lost”, etc.
6. Construct a formula to express “the system starts at the initial state and makes a state transition at every step”.
7. Declare predicates that we expect to follow from the rules 3-5.
8. Have TLC search the state space for counterexamples to #7.

A model of a bounded queue

TLA+ code

```
CONSTANT BufferCapacity
CONSTANTS Readers, Writers
```

LaTeX-formatted

First, let us list constants that define the properties of our system:

- the maximum queue depth,
- the sets of readers and writers.

A model of a bounded queue

TLA+ code

```
ASSUME Readers /= {}
ASSUME Writers /= {}
ASSUME (BufferCapacity \in Nat) /\ (BufferCapacity > 0)
```

LaTeX-formatted

$$\begin{aligned}&\text{ASSUME } Readers \neq \{\}\\&\text{ASSUME } Writers \neq \{\}\\&\text{ASSUME } (BufferCapacity \in Nat) \land (BufferCapacity > 0)\end{aligned}$$

- {} is the empty set,
- /= means “not equal”,
- /\ and \/ are logical “and” and “or”, respectively.

A model of a bounded queue

TLA+ code

```
VARIABLES bufferLen, threadStates
```

LaTeX-formatted

This declares variables that define the state of the system. The state space explored by TLC is the set of all possible values of these variables.

These declarations place no constraints on values of the variables. They may be arbitrary sets.

A model of a bounded queue

TLA+ code

```
Threads == Readers \cup Writers

(* "R" is for Running,
   "M" is "owns Mutex and running",
   "S" is for Sleeping *)

ThreadState == {"R", "M", "S"}

threadStates_TypeInvariant ==
  threadStates \in [Threads -> ThreadState]
```

LaTeX-formatted

$$Threads \triangleq Readers \cup Writers$$

$$ThreadState \triangleq \{\text{“R”}, \text{“M”}, \text{“S”}\}$$
$$threadStates\_TypeInvariant \triangleq threadStates \in [Threads \to ThreadState]$$

`threadStates_TypeInvariant` is a predicate which is true iff `treadStates` is a map from the set of threads to the set of thread states.

At a later stage we will require `threadStates_TypeInvariant` to be true in every state of our model.

- == means “by definition”,
- {A, B, …} defines a set by listing its elements,
- [X -> Y] is the set of maps from X to Y.

A model of a bounded queue

TLA+ code

```
Init ==
  /\ bufferLen = 0
  /\ threadStates_TypeInvariant
  /\ \A t \in Threads: threadStates[t] = "R"
```

LaTeX-formatted

$$
\begin{aligned}
&Init \triangleq \\
&\land\ bufferLen = 0 \\
&\land\ threadStates\_TypeInvariant \\
&\land\ \forall\, t \in Threads : threadStates[t] = \text{“R”}
\end{aligned}
$$

This is a predicate that is true iff the model is in its initial state: the buffer is empty and all threads are runnable.

A model of a bounded queue

TLA+ code

```
(* t acquires the mutex *)
AcquireLock(t) ==
  /\ threadStates[t] = "R"
  /\ \forall ta \in Threads: threadStates[ta] /= "M"
  /\ threadStates' = [threadStates EXCEPT ![t] = "M"]
  /\ UNCHANGED bufferLen
```

LaTeX-formatted

$$
\begin{aligned}
&AcquireLock(t) \triangleq \\
&\land\ threadStates[t] = \text{“R”} \\
&\land\ \forall\, ta \in Threads : threadStates[ta] \neq \text{“M”} \\
&\land\ threadStates' = [threadStates \text{ EXCEPT } ![t] = \text{“M”}] \\
&\land\ \text{UNCHANGED } bufferLen
\end{aligned}
$$

Predicates that describe state transitions do so by placing constraints on the current state and the next possible state.

The predicate `AcquireLock(t)` is true iff thread t acquires the mutex at the current state transition.

- F[t] is the value of map F at t,
- x and x’ denote the values of x in the current state and in the next state, respectively,
- UNCHANGED <<x, y, …>> is a shorthand for x’ = x /\ y’ = y /\ …

A model of a bounded queue

TLA+ code

```
ReadBlock(t) ==
  /\ t \in Readers
  /\ threadStates[t] = "M"
  /\ bufferLen = 0
  /\ threadStates' = [threadStates EXCEPT ![t] = "S"]
  /\ UNCHANGED bufferLen
```

LaTeX-formatted

$$
\begin{aligned}
&ReadBlock(t) \triangleq \\
&\land\ t \in Readers \\
&\land\ threadStates[t] = \text{“M”} \\
&\land\ bufferLen = 0 \\
&\land\ threadStates' = [threadStates \text{ EXCEPT } ![t] = \text{“S”}] \\
&\land\ \text{UNCHANGED } bufferLen
\end{aligned}
$$

`ReadBlock(t)` is true iff thread t attempted to read from the queue, detected that the queue was empty, and blocked. After this transition t sleeps in `pthread_condvar_wait()`.

Note that this transition is only possible if the current thread owns the mutex: `threadStates[t] = “M”`.

A model of a bounded queue

TLA+ code

```
ReadOk(t) ==
  /\ t \in Readers
  /\ threadStates[t] = "M"
  /\ bufferLen > 0
  /\ bufferLen' = bufferLen - 1
  /\ UnlockAndWakeOne(t)
```

LaTeX-formatted

$$
\begin{aligned}
&ReadOk(t) \triangleq \\
&\land\ t \in Readers \\
&\land\ threadStates[t] = \text{“M”} \\
&\land\ bufferLen > 0 \\
&\land\ bufferLen' = bufferLen - 1 \\
&\land\ UnlockAndWakeOne(t)
\end{aligned}
$$

`ReadOk(t)` is true iff thread t read a value from the queue, unlocked the mutex and (possibly) woke another sleeping thread. We will declare `UnlockAndWakeOne(t)` later.

A model of a bounded queue

TLA+ code

```
UnlockAndWakeOne(self) ==
LET
 sleepingThreads == {t \in Threads: threadStates[t] = "S"}
IN
 IF sleepingThreads = {}
 THEN
  threadStates' = [threadStates EXCEPT ![self] = "R"]
 ELSE
  \E t \in sleepingThreads:
  threadStates' = [threadStates EXCEPT ![self] = "R", ![t] = "R"]
```

LaTeX-formatted

$$
\begin{array}{l}
UnlockAndWakeOne(self) \triangleq \\
\text{LET} \\
sleepingThreads \triangleq \{t \in Threads : threadStates[t] = \text{“S”}\} \\
\text{IN} \\
\text{IF } sleepingThreads = \{\} \\
\ \text{THEN} \\
threadStates' = [threadStates \text{ EXCEPT } ![self] = \text{“R”}] \\
\ \text{ELSE} \\
\exists\, t \in sleepingThreads : \\
threadStates' = [threadStates \text{ EXCEPT } ![self] = \text{“R”}, ![t] = \text{“R”}]
\end{array}
$$

A model of a bounded queue

TLA+ code

```
Next == \E t \in Threads:
  \/ AcquireLock(t)
  \/ WriteOk(t)
  \/ WriteBlock(t)
  \/ ReadOk(t)
  \/ ReadBlock(t)
```

LaTeX-formatted

$$
\begin{aligned}
&Next \triangleq \exists\, t \in Threads : \\
&\lor\ AcquireLock(t) \\
&\lor\ WriteOk(t) \\
&\lor\ WriteBlock(t) \\
&\lor\ ReadOk(t) \\
&\lor\ ReadBlock(t)
\end{aligned}
$$

This predicate is true iff the system made a state transition. A state transition may be one of the following actions:

- a thread acquires a mutex,
- a thread attempted to write a value and succeeded, or blocked,
- a thread attempted to read a value and succeeded, or blocked.

A model of a bounded queue

TLA+ code

```
Spec == Init /\ [][Next]_<<bufferLen, threadStates>>
```

LaTeX-formatted

$$Spec \triangleq Init \land \Box[Next]_{\langle bufferLen,\, threadStates \rangle}$$

This is the definition of our model: `Init` is true in the initial state and `Next` is true at every state transition.

- []E is true iff E is true at every state transition.

A model of a bounded queue

TLA+ code

```
AtMostOneMutexOwner ==
 LET owners == {t \in Threads: threadStates[t] = "M"}
 IN Cardinality(owners) <= 1

Liveness ==
 LET
 runnable == {t \in Threads: threadStates[t] \in {"R", "M"}}
 IN runnable /= {}
```

LaTeX-formatted

$$AtMostOneMutexOwner \triangleq$$
$$\text{LET } owners \triangleq \{t \in Threads : threadStates[t] = \text{“M”}\}$$
$$\text{IN } \quad Cardinality(owners) \leq 1$$

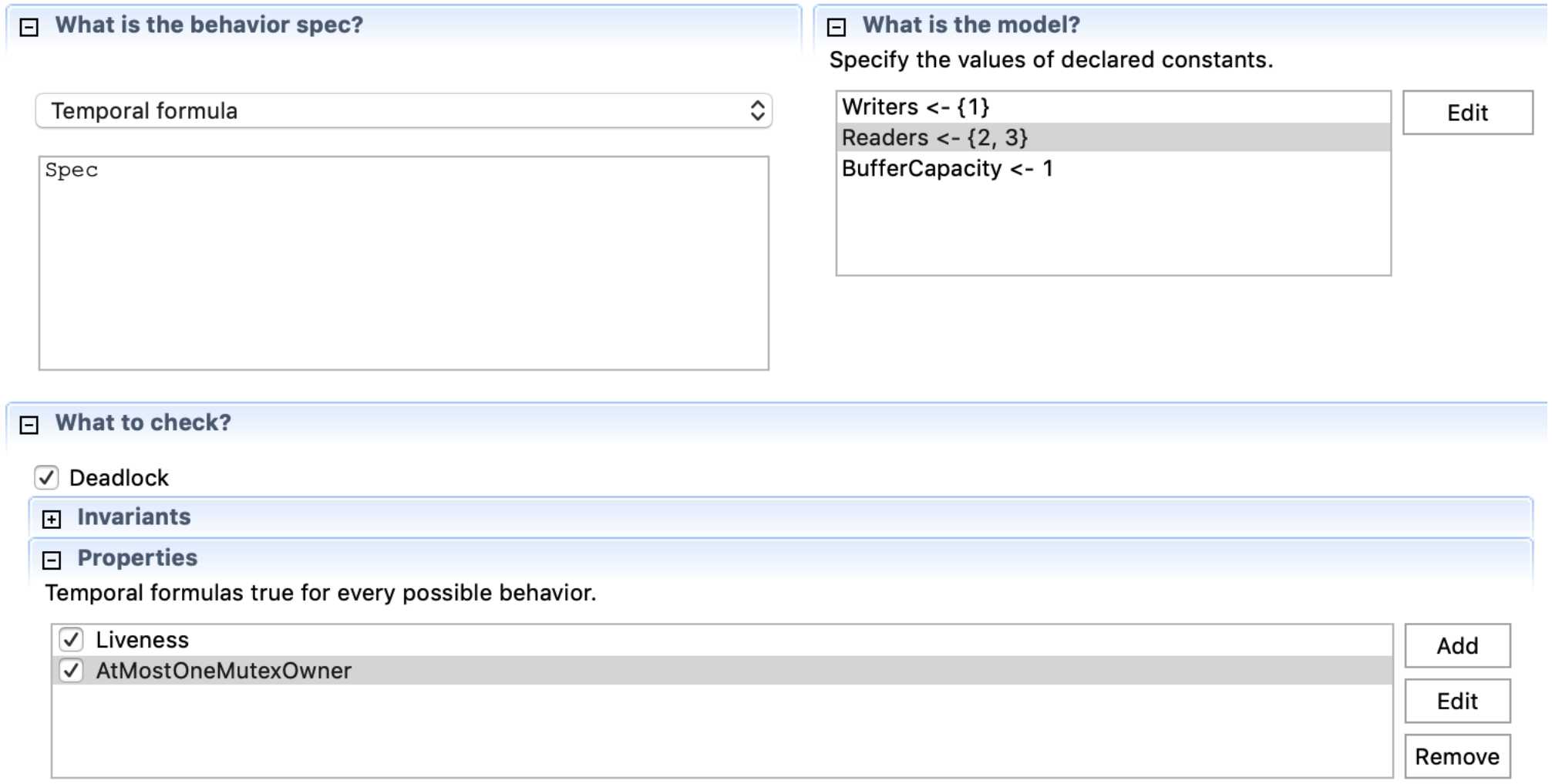
$$Liveness \triangleq$$
$$\text{LET } runnable \triangleq \{t \in Threads : threadStates[t] \in \{\text{“R”}, \text{“M”}\}\}$$
$$\text{IN } \quad runnable \neq \{\}$$

Now let us define predicates that describe desired properties of our model:

- at most one thread owns the mutex,
- there is at least one runnable thread.

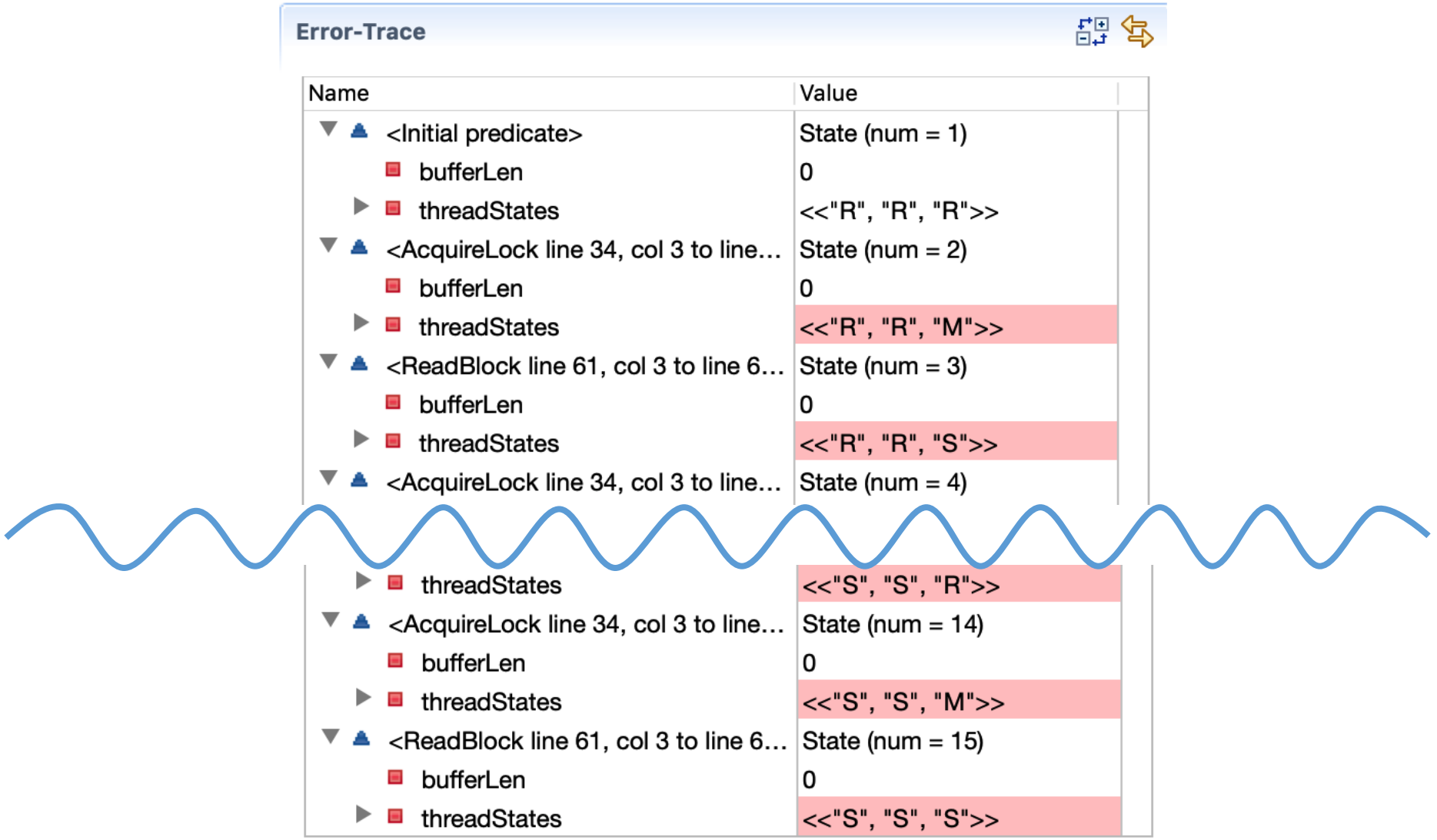
A model of a bounded queue

Now let us have TLA+ Model Checker walk the states of our model:



A model of a bounded queue

Now let us have TLA+ Model Checker walk the states of our model:



To do at home

Use TLC to find a way to transport a goat, a wolf, and a cabbage from one shore of a river to the other one.

Idea:

1. Declare predicates that describe the rules of transporting,
2. Declare a predicate “the goat, the wolf and the cabbage are not on the other shore”.
3. Use TLC to find a counterexample to #2.

To read

1. Specifying systems, https://lamport.azurewebsites.net/tla/book-02-08-08.pdf.
2. TLA+ Toolbox https://lamport.azurewebsites.net/tla/toolbox.html.