The Palladio layout control protocol

Richard Golding and Liz Borowsky

HPL–SSP–99–1 rev. 2
30 August 1999

Abstract

This document presents the Palladio layout control protocol, which provides for failure recovery and coordinates layout changes.

Contents

1 Assumptions **2**

2 Constants **3**

3 Players **3**

4 Chunks **4**

4.1 State 4
4.2 Transitions 6
4.2.1 Regular lease state 6
4.2.2 Reconciling state 8
4.2.3 Regular transition state 8
4.2.4 Garbage collect state 9
4.2.5 Newly awake state 9
4.2.6 No lease state 9
4.2.7 Recovery lease state 10
4.2.8 Recovery transition state 12
4.2.9 Recovery reconciling state 13

5 Managers **13**

5.1 State 13
5.2 State overview 15
5.3 Transitions 16
5.3.1 None state 16
5.3.2 Newly awake state 16
5.3.3 Into reconciling state 17
5.3.4 Into transition state 17
5.3.5 Out of reconciling state 17

5.3.6 Out of transition state 17
5.3.7 Manager state 18
5.3.8 Relayout reconciling state 20
5.3.9 Relayout transition state 21
5.3.10 Gathering chunks state 21
5.3.11 Gathered state 24
5.3.12 Gathering leases state 25
5.3.13 Recovery reconciling state 26
5.3.14 Recovery transition state 26

1 Assumptions

- Partitions partition nodes into mutually communicating subsets. This means that we don't have to worry about non-transitivity in the communication graph.

- Devices can tell with complete certainty that they are restarting after they have failed. (That is, a device knows when it reboots.)

- The commit protocol is assumed to always eventually terminate, even if some participants fail. If a manager fails during a run of the commit protocol, chunks will observe a lease expiry some time after the commit protocol terminates.

- The network is fairly reliable (perhaps by a bounded, small bit error rate?). Message delivery occurs within bounded time when not partitioned. Messages are not corrupted in transit; nor are they replayed or spontaneously generated.

- All parties (chunks and manager nodes) respond to events within bounded time.

- There is a way for any chunk in a partition to find, within a bounded time, all manager nodes that exist in that partition. It can do so an infinite number of times.

- Leases (both normal and recovery) last much longer than the time it takes to renew them. As a result, there will eventually be a period when all leases are not in the middle of being renewed. Moreover, the timer used to detect the failure of a chunk to acknowledge a lease renewal is longer than the time it takes to do a successful renewal, so that there are never acknowledgement messages coming back to a manager after the timer goes off.

2 Constants

δ_c	maximum difference between any two clocks in the system.
Δt_{msg}	bound on message delivery latency
ct_{end}	a bound on the latency between the first and last commit terminations
cta_{end}	analogous to ct_{end}, but just for aborts
ctc_{end}	analogous to ct_{end}, but just for commits
ct_{start}	a bound on the latency between the first and last commit start messages from different participants
abort lease window	Δt for how long leases are extended after transaction abort; want $\Delta t > cta_{end}$.
commit lease window	Δt for how long leases are extended after transaction commits; want $\Delta t > ctc_{end}$
d_l	the length of a lease. (Practically speaking it can vary; for this presentation we assume a constant.)

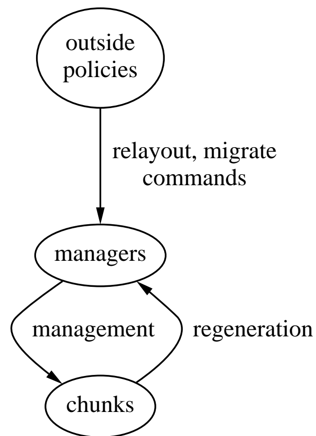
Note that the lease windows imply $\exists \delta t > 0$ such that commit has ended and lease renewals are not yet needed.

Note that the commit protocol latency is assumed bounded, but potentially large. The commit protocol always eventually terminates.

3 Players

There are three kinds of participants in the failure recovery protocol:

1. Chunks. These are the units of data storage. They have persistent state.
2. Managers. These coordinate changes to the layout of the chunks, and serve as a single point where outside users can find out about layout. They have no persistent state. There are many managers, but at most one manager node can manage a particular virtual store.
3. Outside policies. These policies determine when the layout for a virtual store should change, and when management function should migrate from management node to management node. These policies are arbitrary and outside the scope of this protocol, except for specifying the interface they use.



4 Chunks

4.1 State

Device state. Devices as a whole do not figure much into this protocol; rather, most of the protocol is concerned with the behaviour of the chunks within a device.

set of chunks	the chunks on this device
clock	a (loosely-)synchronised clock

Chunk state. Most of the protocol deals with the behaviour of individual chunks. Much of the state in a chunk is persistent – that is, it can be counted on to remain intact as long as the chunk exists. Persistent state variables are in **bold face** below. Other state variables are not persistent: they are expected to be erased whenever the chunk (or the device on it) fails, and some are only used while in one or two states.

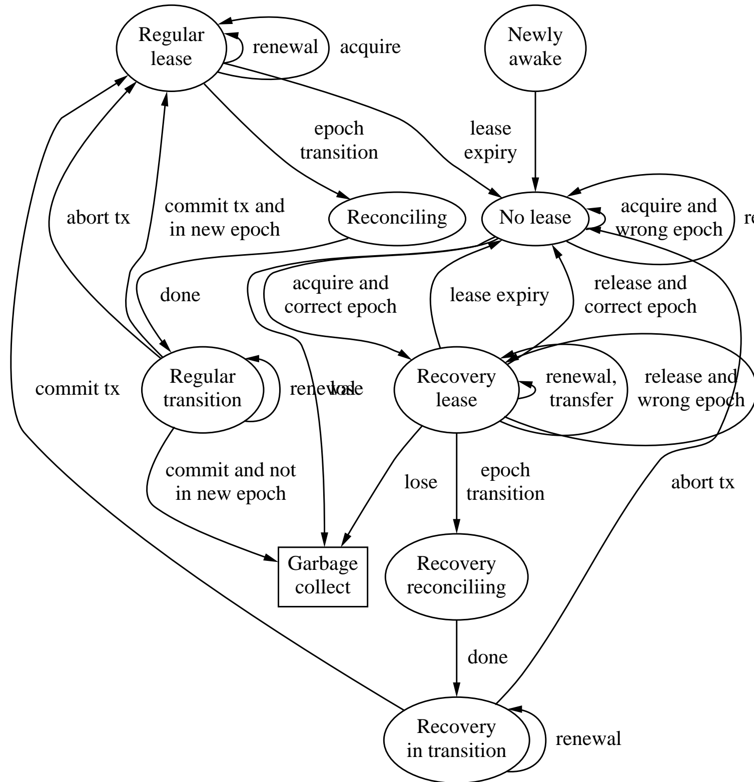
Each chunk has a unique identity, and that identity is never reused. In practice, this can mean that each chunk is given a serial number within the device, and the device must be careful to record any advance in the serial number on stable storage before making a new serial number available to the outside world.

epoch	integer	the current epoch number
metadata	–	the metadata giving the layout of the store in the named epoch
lease expiry	timer	when the chunk's current lease expires
lease renewal	timer	when the chunk's current lease should be renewed
lease manager	manager id	the manager that granted the current lease (or none)
state	one of none, regular, recovery, newly awake, reconciling, regular transition, recovery reconciling, recovery transition	the state of the chunk. This corresponds, roughly, to the kind of lease the chunk has.
epoch transition log	record { metadata, epoch, manager }	temporary used during epoch transitions
mgr_attempt_queue	priority queue of manager id, ordered by decreasing manager precedence	a priority queue of ids of managers that might be useful in regenerating manager state; used only during manager regeneration

The state of the chunk is a special case: it is reset on each state transition. Note that we have assumed, above, that devices can detect that they have restarted with complete certainty.

Note that the access protocol blocks unless the state is `regular`.

4.2 Transitions



Note that any state can transition to the `newly awake` state at any time when a `fail` event occurs; for clarity, these transitions are not shown in the transition diagram and have been omitted from some of the state transition actions that follow.

4.2.1 Regular lease state

In this state, the chunk is in regular operation and has a valid lease. This is the only state in which the access protocol will allow data to be read or written.

renewal(mgr epoch, new expiry time)

The chunk's manager is renewing the chunk's lease.

```
if (mgr epoch == current epoch)
     lease expiry time ← new expiry time
     lease renewal ← now + lease renewal period
     state ← regular lease
```

epoch transition(old epoch, new epoch, new metadata, new mgr id)

> *A manager is beginning an epoch transition. The first step is to perform reconciliations.*
>
> ```
> if (old epoch >= curr epoch)
>      log arguments in transient storage
>      state ← reconciling
> ```

lease renewal

> *The chunk's lease will expire soon, and so the chunk should request renewal.*
>
> `send` **renewal request(curr epoch)** `to manager`
> `state ← regular lease`

lease expiry

> *This transition occurs when the lease has expired, and the chunk now suspects its manager of having failed.*
>
> `state ← no lease`

fail

> `state ← newly awake` *`(after some delay)`*

acquire(new lease expiry, new mgr, new epoch number)

> *A manager node is trying to regenerate the metadata for the virtual store of which this chunk is part, but this chunk is already properly being managed by somebody. Note that in this case the new epoch number should always be less than or equal to the current epoch number.*
>
> `send` **nack(new epoch number, current manager, regular lease)** `to sender`
> `state ← regular lease`

lose

> *This message should not be received while the chunk has an active lease.*
>
> *`ignore this message`*
> `state ← regular lease`

release(epoch number, (opt) new mgr hint set)

> *This message should not be received while the chunk has an active lease.*
>
> *ignore this message*
> state ← regular lease

4.2.2 Reconciling state

In this state, the manager has begun an epoch transition, and any reconciliations are being performed. The run of the reconciliation protocol is omitted; it triggers a **done** event when when all reconciliation for this chunk is complete.

done

```
initiate commit protocol, voting commit
state ← regular transition
```

4.2.3 Regular transition state

commit tx

```
if self ∈ logged new metadata
     using logged state
          epoch ← new epoch
          metadata ← new metadata
          lease expiry time ← now + lease commit window
          lease renewal ← now + lease renewal period
          lease grantor ← new manager
     clear log
     state ← regular lease
else
     state ← garbage collect
```

abort tx

```
lease expiry time ← now + lease abort window
lease renewal ← now + lease renewal period
clear log
state ← regular lease
```

fail

```
state ← newly awake (after some delay)
```

release(epoch number, (opt) new mgr hint set)

> *This message should not be received while the chunk has an active manager. It is thus ignored in this state.*
>
> `state ← regular lease`

4.2.4 Garbage collect state

In this state, the chunk should not be there. The chunk should delete itself.

The device may need to respond to one message on behalf of the now-deleted chunk:

acquire(lease expiry, mgr, epoch)

> `send` **nochunk(chunk)** `to sender`

All other messages to the chunk are ignored.

4.2.5 Newly awake state

The device places all chunks in this state immediately upon rebooting after a failure, and before any messages are processed. On entering this state the chunk does the following:

```
clear log and any other transient data
mgr_attempt_queue ← lease mgr
state ← no lease
```

4.2.6 No lease state

In this state the chunk believes that its manager has failed, and so it must try to regenerate a manager. It will continue to attempt regeneration until it is able to do so successfully, or until it is informed that it is no longer needed.

We assume that the random selection of a director node will eventually try all director nodes in the system. An actual design will want to be more clever: attempting to only contact reachable director nodes, avoiding generating too much traffic, and so on. Note that we believe that this sequence will eventually terminate with a fail, acquire, or lose transition as long as partitions are eventually repaired and the chunk retries regeneration forever.

Upon entry to this state the chunk does the following:

```
repeatedly:
     if mgr_attempt_queue is empty
          pick a director node d at random
     else
          d ← mgr_attempt_queue.pop
     send help(epoch number, metadata) to d
     wait long enough for d to respond if it is available
```

This loop terminates when the chunk transitions out of the no lease state.

The restriction on the wait ensures that at most one help message is outstanding from any chunk at time.

fail

```
state ← newly awake (after some delay)
```

lose

A manager has determined that this chunk should be garbage collected.

```
state ← garbage collect
```

acquire(new lease expiry, new mgr, new epoch number)

```
if new epoch number < current epoch number
     send nack(current epoch number, nil, no lease) to sender
     state ← no lease
else
     if new epoch number == current epoch number
          send ack(current epoch number) to new mgr
     else
          send ack-conditional(new epoch number, current epoch number, metadata) to
     lease expiry time ← new lease expiry
     lease renewal ← now + lease renewal period
     mgr ← new mgr
     mgr_attempt_queue ← nil
     state ← recovery lease
```

redirect(new mgr)

```
append new mgr to mgr_attempt_queue
state ← no lease
```

4.2.7 Recovery lease state

epoch transition(old epoch, new epoch, new metadata, new mgr id)

This is identical to the transition of the same name in the regular lease state.

```
if (old epoch >= curr epoch)
     log arguments in transient storage
     state ← recovery reconciling
```

lease renewal

```
send recovery renewal request(curr epoch) to mgr
state ← recovery lease
```

lease expiry

```
state ← no lease
```

fail

```
state ← newly awake
```

renewal(mgr epoch, new expiry time)

```
if (mgr epoch == curr epoch)
     lease expiry time ← new expiry time
     lease renewal ← lease renewal period
     state ← recovery lease
```

release(epoch number, (opt) new mgr hint set)

The manager with which this chunk has been interacting has determined that another manager node, which has precedence over it, is also attempting to regenerate a manager. This message informs the chunk that its current manager is abandoning its attempt to regenerate, and that the chunk should contact the new manager node to complete regeneration.

```
if (epoch number != current epoch number) or
          (sender != current mgr)
     state ← recovery lease
else
     append new mgr hints to mgr_attempt_queue
     state ← no lease
```

lose

```
state ← garbage collect
```

acquire(new lease expiry, new mgr, epoch number)

Another manager is trying to acquire this chunk during regeneration.
Inform it that the chunk has already been acquired.

```
if (new mgr != current mgr) or
        (epoch number != current epoch number)
    send nack(current epoch number, current mgr, recovery lease) to sender
state ← recovery lease
```

transfer lease(epoch number, new mgr, expiry)

Another manager has won the right to be the manager for the store.
Change the recovery lease over to that new manager.

```
if (epoch number != current epoch number)
    send nack(current epoch number, current mgr) to sender
else
    send transfernotice(epoch number, new mgr) to lease mgr
    send transferack(epoch number, new mgr) to sender
    lease mgr ← new mgr
    lease expiry time ← new expiry
    lease renewal ← renewal period
state ← recovery lease
```

4.2.8 Recovery transition state

commit tx

```
if self ∈ logged new metadata
    using logged state
        epoch ← new epoch
        metadata ← new metadata
        lease expiry time ← now + lease commit window
        lease renewal ← now + lease renewal period
        lease grantor ← new manager
    clear log
    state ← regular lease
else
    state ← garbage collect
```

abort tx

```
clear log
state ← no lease
```

fail

```
state ← newly awake (after some delay)
```

4.2.9 Recovery reconciling state

In this state, the manager has begun an epoch transition at the end of a recovery sequence, and any reconciliations are being performed. The run of the reconciliation protocol is omitted; it triggers a **done** event when when all reconciliation for this chunk is complete.

done

```
initiate commit protocol, voting commit
state ← recovery transition
```

5 Managers

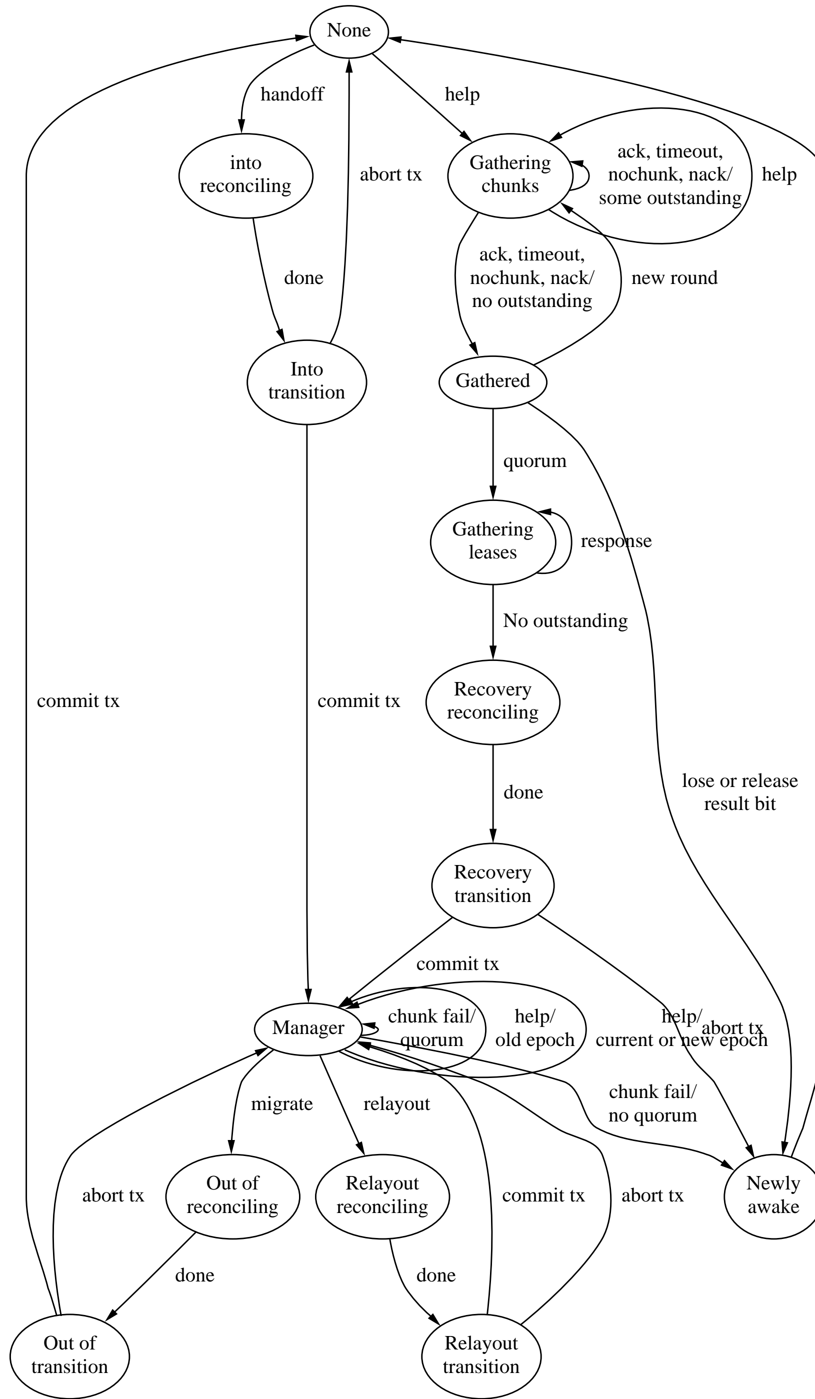
The manager nodes provide fast access to the metadata for a store, and are the entities that can coordinate changes to the layout of a store. The manager function can migrate from node to node, and this ability is reflected in the protocol here. There is also a distributed data structure for finding the manager, even as it moves around; this is not included in this protocol.

5.1 State

The state of a manager is one of new, newly awake, mgr, gathering chunks, gathering leases, into reconciling, into transition, out of reconciling, out of transition, recovery reconciling, recovery transition, relayout reconciling, or relayout transition.

clock	–	a (loosely-)synchronised clock
epoch number	integer	the current epoch number
metadata	metadata	the currently active metadata for the virtual store.
failed chunks	list of chunks	the chunks in the current epoch that the manager believes have failed.
recovered chunks	list of chunks	the chunks in the current epoch that the manager believes have failed and since recovered.
forwarding address	manager id	where a manager can be found (if none or newly awake)
new mgr id	manager id	temporary manager id variable used during manager migration.
better mgr id	set of manager id	set of managers with higher precedence; used during manager regeneration.
lesser mgr id	set of manager id	set of managers with lower precedence; used during manager regeneration.
new metadata log	metadata	a temporary copy of new metadata
release result	boolean	during regeneration, indicates that the chunks involved in regeneration are from an out-of-date epoch and causes the manager to give up its regeneration attempt.
outstanding chunks	set of chunks	during regeneration, the chunks that have not yet been heard from
won chunks	set of chunks	during regeneration, the chunks that have agreed to this manager as their manager
renewal iffy chunks	set of chunks	during lease gathering, the chunks to which lease transfers have been sent, but not yet acknowledged.
chunkfail(c)	one-shot timer	a set of timers for detecting when individual chunks' leases have expired.
timeout(i)	one-shot timer	a set of timers used to bound response to acquire messages.
old mgr	manager id	the node that was previously manager for this store; used during manager migration.

5.2 State overview



5.3 Transitions

Note: all states can transition to the newly awake state on a fail event.

fail

```
state ← newly awake (after some delay)
```

5.3.1 None state

This node is not a manager for the store in question.

handoff(old epoch number, new metadata, failed chunks)

The manager function is being migrated into this node.

```
epoch number ← old epoch number + 1
metadata ← new metadata
old mgr ← sender
failed chunks ← new failed chunks
state ← into reconciling
```

help(epoch number, metadata)

A device is requesting that management be regenerated.

```
outstanding chunks ← (chunks ∈ metadata)
won chunks ← nil
for all d ∈ outstanding chunks
     send acquire(expry, me, epoch number) to d
     set timeout(d) to a reasonable message timeout
better mgr id ← nil
lesser mgr id ← nil
epoch number ← new epoch number
metadata ← new metadata
state ← gathering chunks
```

5.3.2 Newly awake state

This node has restarted; initialise all state.

```
erase all state
state ← none
```

5.3.3 Into reconciling state

Manager function is being migrated into this node; as the first step in this, chunks are being reconciled.

done

Reconciliation has completed.

```
initiate commit, voting commit
state ← into transition
```

5.3.4 Into transition state

Manager function is being migrated into this node, and reconciliation is done; execute a commit protocol.

commit tx

```
failed chunks ← nil
state ← manager
```

abort tx

```
erase epoch number, metadata
state ← none
```

5.3.5 Out of reconciling state

Manager function is being migrated out of this node; as the first step, chunks are being reconciled.

done

Reconciliation has completed.

```
initiate commit, voting commit
state ← out of transition
```

5.3.6 Out of transition state

commit tx

```
erase epoch number, metadata
forwarding address ← new mgr id
state ← none
```

abort tx

```
erase new mgr id
state ← mgr
```

5.3.7 Manager state

migrate(node)

An outside policy has decided that the manager running here should migrate to another node.

send **handoff(epoch number, metadata, failed chunks)** to node
send **epoch transition(epoch number, epoch number + 1, metadata, node)**
        to all chunks ∈ metadata - failed
new mgr id ← node
state ← out of transition

chunkfail(chunk)

A chunk has failed to renew its lease.

```
if chunk ∉ failed
     a new failure
     failed chunks += chunk
     send chunkfail(chunk) to outside policy
     if (coverage and quorum)
          state ← manager
     else
          state ← newly awake (suicide)
else
     failure of a recovered chunk
     recovered -= chunk
     state ← manager
```

renewal request(chunk epoch)

A chunk is requesting that its lease be renewed.

if sender ∉ failed and chunk epoch == curr epoch
     t ← now $+d_l$
     send **renewal(epoch,t)** to sender
     set timer chunkfail(sender) to t
     state ← manager
else
     ignore message

relayout(new metadata)

An outside policy has decided that the metadata has changed.

```
send epoch transition(epoch number, epoch number + 1, new metadata, me)
          to (all chunks ∈ metadata - failed) ∪
               (all chunks ∈ new metadata)
new metadata log ← new metadata
state ← relayout reconciling
```

help(chunk epoch number, metadata)

A chunk has lost its lease and wants to regenerate. This can occur in two cases: when a chunk from a previous epoch becomes able to communicate and chooses the current manager to try regeneration; or when a lease renewal message to a chunk active in this epoch has been lost, the chunk's lease has expired, and that chunk happens to choose the current manager for manager regeneration.

If the chunk is part of the current layout, then the manager issues the chunk a recovery lease to keep the chunk from trying other recovery steps until it can be reconciled and brought back into active use. If the chunk is not part of the current layout, it has been deleted and is sent a lose message so that it garbage collects.

```
if sender ∈ chunks(metadata)
     if sender ∉ failed
          note this is the same as the chunkfail(chunk) event.
          failed chunks += sender
          send chunkfail(sender) to outside policy
          if not (coverage and quorum)
               state ← newly awake (suicide)
     exprry ← now + d_l
     send acquire(exprry, self, curr epoch) to sender
else
     send lose to sender
     state ← manager
```

ack(chunk epoch)

A chunk has been issued a recovery lease and is acknowledging it.

```
if sender ∈ failed
     recovered += sender
     exprry ← now + d_l
     set chunkfail(sender) to exprry
```

ack-conditional(mgr epoch, chunk epoch, chunk metadata)

A chunk from a previous epoch has been issued a recovery lease and is acknowledging it. Note that the behaviour is the same as for an ordinary ack.

```
if sender ∈ failed
     recovered += sender
     exprry ← now + d_l
     set chunkfail(sender) to exprry
```

nack(chunk epoch, chunk mgr, lease type)

A chunk has been issued a recovery lease but is not accepting it.

do nothing

recovery renewal request(chunk epoch)

A chunk is requesting that its recovery lease be renewed.

```
if sender ∈ recovered
     t ← now +d_l
     send renewal(epoch,t) to sender
     set timer chunkfail(sender) to t
     state ← manager
else
     ignore message
```

5.3.8 Relayout reconciling state

A store layout change is being committed; as the first step, chunks are being reconciled.

done

Reconciliation has completed.

```
initiate commit, voting commit
state ← relayout transition
```

5.3.9 Relayout transition state

commit tx

```
metadata ← new metadata
erase new metadata log
increment epoch
send relayout commit to outside policy
state ← manager
```

abort tx

```
erase new metadata log
send relayout abort to outside policy
state ← mgr
```

5.3.10 Gathering chunks state

This state is the first in the recovery sequence. In it, the manager node tries to become a recovering manager for the store, contending with other managers to acquire as many chunks as possible.

ack(epoch number)

A chunk has agreed to be acquired by this manager node. Note that the epoch number in the message should always equal the current epoch.

```
oustanding -= sender
won chunks += sender
cancel timeout(sender)
set chunkfail(sender) to now + d_l
if outstanding = nil
     state ← gathered
else
     state ← gathering chunks
```

ack-conditional(old epoch, chunk epoch, chunk metadata)

A chunk has agreed to be acquired by this manager node, but it has a different epoch number than the manager. If the chunk's epoch is newer than the one the manager is currently recovering, the manager brings forward its recovery attempt to that epoch.

```
oustanding -= sender
won chunks += sender
cancel timeout(sender)
```

```
set chunkfail(sender) to now + d_l
if chunk epoch > epoch number
     cnew ← chunks(chunk metadata) - chunks(metadata)
     foreach c ∈ cnew
          send acquire(expry, self, chunk epoch) to c
          outstanding chunks += c
          set timeout(c) to a reasonable message timeout
     metadata ← chunk metadata
     epoch ← chunk epoch
if outstanding = nil
     state ← gathered
else
     state ← gathering chunks
```

timeout(chunk)

A chunk has failed to respond to an acquire message.

```
oustanding -= chunk
if outstanding = nil
     state ← gathered
else
     state ← gathering chunks
```

nochunk(chunk)

A device is responding that a chunk doesn't exist.

```
oustanding -= chunk
cancel timeout(sender)
if outstanding = nil
     state ← gathered
else
     state ← gathering chunks
```

recovery renewal request(chunk epoch)

It is time to renew the lease on some chunk that has been acquired by this manager.

```
if sender ∈ won
     t ← now +d_l
     send renewal(epoch,t) to sender
     set timer chunkfail(sender) to t
state ← gathering chunks
```

chunkfail(c)

A chunk has failed to acknowledge a lease renewal.

```
won -= c
state ← gathering chunks
```

help(epoch number, metadata)

A chunk is asking for management to be regenerated. If this is from a chunk from an old epoch that has since been removed from the layout, signal the chunk to garbage-collect itself. Otherwise, as long as the chunk does not already have an acquire message on the way to it, try to acquire the chunk. Note that the chunk may have previously been acquired by this manager, but since failed during the recovery process. Note also that if the chunk is from another epoch, trying to acquire the chunk will provoke the chunk into replying with an ack-conditional, which may advance the recovering manager to a newer epoch.

```
if (chunk epoch < epoch) and (sender ∉ chunks(metadata))
     send lose to sender
else if sender ∉ outstanding
     if sender ∈ won
          won -= sender
     expry ← now + d_l
     send acquire(expry, self, curr epoch) to sender
     set timeout(sender) to a reasonable message timeout
     outstanding += sender
state ← gathering chunks
```

nack(chunk epoch number, chunk mgr, chunk lease type)

A chunk is responding that it has already been acquired by a different manager. This transition implements the arbitration between managers based on a precedence relation, which is assumed to be well-known. Note: Chunk lease type can be regular lease, no lease, recovery lease.

```
oustanding -= sender
cancel timeout(sender)
if lease type = regular lease
     if chunk mgr = this mgr
          (do nothing)
     else (someone else has a regular lease on this chunk)
```

```
            better mgr id += chunk mgr
    else (recovery lease or no lease )
        if chunk mgr = this mgr
            (shouldn't happen)
        else if chunk mgr has precendence over this mgr
            better mgr id += chunk mgr
        else
            lesser mgr id += chunk mgr
    if outstanding = nil
        state ← gathered
    else
        state ← gathering chunks
```

transfernotice(chunk epoch number, chunk mgr)

A chunk is informing the manager that a different manager has won the contention for completing recovery, and that the chunk has transferred its lease to that manager.

```
    if (chunk epoch number == current epoch number)
        state ← newly awake
    else
        state ← gathering chunks
```

5.3.11 Gathered state

Immediately on entry to this state, the manager does the following:

```
    if (coverage and quorum)
        state ← gathering leases
    else if |better manager id| = 0 and
            (won ∪ failed) ⊂ chunks(metadata)
        for each c ∈ chunks(metadata) - won
            send acquire(expiry, me, epoch number) to c
            outstanding += c
            set timeout(c) to a reasonable message timeout
        state ← gathering chunks
    else
        send release(better mgr id, epoch number) to all chunks ∈ won
        state ← newly awake
```

This transition depends on two predicates – coverage and quorum – on the set of won chunks. Quorum must ensure that the manager in at most one partition is allowed to proceed with regeneration. The coverage predicate must ensure that data is present for all of the virtual store.

If this manager has not clearly lost the competition to become manager, but has not yet acquired coverage and quorum, it starts another round of acquisitions rather than proceeding with recovery.

5.3.12 Gathering leases state

In this state, the manager has won the right to be the one true manager for the store. Now it needs to go through all the chunks that are still active and get their leases, in preparation for taking them all through an epoch transition.

Immediately on entry to this state, the manager does the following:

```
renewal iffy ← nil
t ← metadata - (failed chunks ∪ won)
if |t| > 0
     for each chunk c ∈ t
          t ← now + d_l
          send transfer lease(epoch, me, t) to c
          renewal iffy += c
          set timer chunkfail(c) to t
     state ← gathering leases
else
     send epoch transition(epoch, epoch+1, metadata, me)
          to all chunks in won
     state ← recovery transition
```

recovery renewal request(chunk epoch)

It is time to renew the lease on some chunk that has been acquired by this manager. This transition is the same as the similar transition in the gathering chunks state.

```
if sender ∉ failed
     t ← now +d_l
     send renewal(epoch,t) to sender
     set timer chunkfail(sender) to t
state ← gathering chunks
```

chunkfail(chunk)

```
renewal iffy -= chunk
won -= chunk
if !(coverage and quorum)
     state ← newly awake
else
     state ← gathering leases
```

transfer ack(chunk epoch number, chunk mgr)

```
assert that epoch number and mgr match this mgr
renewal iffy -= sender
won += sender
if renewal iffy == nil
     send epoch transition(epoch, epoch+1, metadata, me)
          to all chunks in won
     state ← recovery transition
else
     state ← gathering leases
```

5.3.13 Recovery reconciling state

This manager is performing an epoch transition to complete recovery; as the first step in this, chunks are being reconciled.

done

Reconciliation has completed.

```
initiate commit, voting commit
state ← recovery transition
```

5.3.14 Recovery transition state

commit tx

```
increment epoch number
failed chunks ← nil
state ← manager
```

abort tx

```
state ← newly awake
```