# The Palladio access protocol

Richard Golding

HPL–SSP–99–2
1 November 1999

**Abstract**

This document presents the Palladio access protocol, which provides for reading and writing data on multiple devices.

## Contents

# 1 Assumptions

- We are assuming a synchronous system, where processors execute at a rate bounded both above and below and where message delivery—if it happens—occurs in a bounded time.
- Network nodes have loosely-synchronized clocks.
- Timestamps derived from clocks are of sufficient resolution that no two timestamps derived on the same host will have the same value. The actual timestamp includes the host id appended to the clock value, so that each draw of a timestamp from a clock is unique over all time and all nodes in the system.

# 2 Constants

- $\delta$: maximum clock drift
- $\delta_{to}$: a timeout value. When an operation has been on a chunk's operation queue longer than $\delta_{to}$, the chunk presumes that a host has crashed and not committed a write and therefore gets a manager to reconcile that block.
- $\delta_{ml}$: a timeout value to detect probable message loss.

# 3 Players

The host and chunk are the primary players in the access protocol. Managers play an incidental role: they control aspects of host and chunk behavior through the layout control protocol. Chunks can also request managers to perform reconciliation, again through the LCP.

# 4 Interfaces

There are interfaces between the access protocol and two other protocols.

With the layout control protocol. This interface is in the chunk. A chunk doesn't serve data when it has no lease. The access protocol makes use of

the chunk's size, and of the epoch number, which are set by the LCP. Finally, chunks involved in the access protocol can use the LCP to ask their manager to reconcile data that may have gotten out of sync.

With the layout retrieval protocol. The host can request a refresh of its layout cache, and the access protocol uses the layout and epoch number it gets from the LRP in generating low-level messages.

# 5 Hosts

## 5.1 External operations

There are three events that can be injected from the outside:

- read(block): reads one block from the store.
- write(block,value): writes one block to the store.
- fail: the host fails.

There are two events that can be provided to the outside:

- readresp(block,value,status): a read has completed; this is the result. Status is either complete or failed.
- writeresp(block,status): a write has completed; this is the result. Status is either complete or failed.

The sequence of interleavings of these five events is restricted in a correct execution of the system. Informally, an execution is correct iff the following properties hold:

- (FIFO per host) For a single host, the readresp and writeresp events for a particular block are in the same order as the corresponding read and write events, as long as failures don't occur.
- (Bounded response, fairness) Any read or write request completes within a bounded time, either by there being a corresponding readresp or writeresp, or by there being a fail event that matches all unresponded reads and writes on that host.
- (Single-copy serializability) Across hosts, there is a global serialization of operations such that each read returns the value of the preceding write for that block in the serialization order.

Note that the FIFO condition is per-block, not over all blocks: requests for different blocks may be reordered arbitrarily. The finite-response-time condition means that it is possible to construct a barrier in the execution sequence by allowing the request queue to drain before issuing new requests. This could be

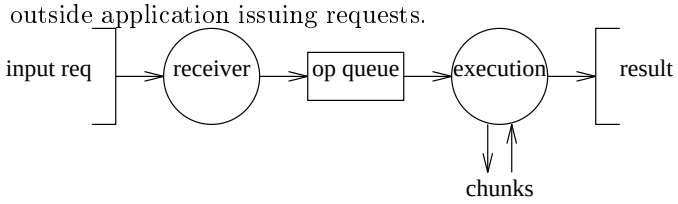
improved by adding additional ordering indications to the request events, and accounting for this in the ordering conditions; this is an extension for later.

Note also that this model does not allow for multi-block atomicity; this is an extension for later.

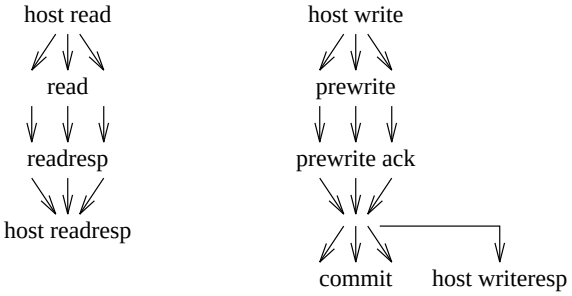
The status values that can be returned on a read or write response are OK and FAIL. An OK value means that the operation definitely succeeded. On a write, in particular, this means that the data in the write have been persistently recorded to storage (barring excessive failure.) A FAIL value means, on a read, that the data returned is invalid. A FAIL on a write means that the data may or may not have been written. A well-designed, properly-functioning system will generate few false failures but there will always be some. These semantics are the same as on current disks and disk arrays. This design decision was made because providing stronger meanings for a FAIL result would require the use of heavyweight consensus mechanisms on every I/O.

## 5.2 General structure

The host side generally consists of two parts: the first receives and queues up operations as they are requested; the second takes operations from that queue one at a time and issues low-level read and write requests to chunks. In this way the processing of requests is decoupled from, and occurs asynchronously with, the outside application issuing requests.




The execution machine in the host uses two BSTs [Amiri99], or patterns of message passing, to perform high-level reads and writes. The patterns are encoded in the execution engine's state transitions.




## 5.3 State

All state on the host is transient: it is lost on failure.

The following state is shared among all parts of a host:

| variable | type | use |
|---|---|---|
| clock | time | a clock that is synchronized to within some global $\delta$ of all other clocks in the system. |

The following define the interface between the receiver and execution machines:

| variable | type | use |
|---|---|---|
| opqueue | queue of op = (type, block, value, sender) | queue of operations that have been requested from the outside but not yet processed; initially empty |

The following define the state of the execution machine:

| variable | type | use |
|---|---|---|
| currop | op = (type, block, value, sender); type is none, read, or write | the current operation, if any; initially none |
| currts | timestamp | the timestamp of the current operation; initially 0 |
| resultokay | boolean | whether the result is okay or not |
| resultrestart | boolean | if resultokay is false, whether the operation should be restarted or not |
| resultvalue | value | the value to be returned |
| outstanding | (chunk, block) | a set of physical blocks for which some kind of reply is expected; initially empty |

The following define the interface between the access protocol and the layout retrieval protocol:

| variable | type | use |
|---|---|---|
| layout | layout: (block, operation) $\rightarrow$ (chunk,block) | a cached copy of the mapping of block numbers to the physical blocks in chunks that make it up; initially empty |
| version | epoch number | the epoch number of the cached layout information; initially -1 |
| layout-current | boolean | whether the cached layout mapping is believed to be current or not; initially false |

## 5.4 Interfaces to other protocols

In the host, the access protocol interfaces only to the layout cache coherence mechanism. The chunks provide indications of out-of-date layout version numbers, which cause the access protocol to use the layout retrieval protocol to obtain a better copy of the layout. The interface is through three variables,

layout, version, and layout-current. Layout and version are read-only to the access protocol; layout-current can be read or set to false. To obtain a new layout, the access protocol sets layout-current to false; this triggers the layout retrieval protocol to obtain a new copy of the layout, changing the layout and version variables. When the layout retrieval protocol is done, it will set the layout-current variable to true, which is the signal that the layout data can be used again. While the layout-current variable is false, the access protocol must assume that the layout and version variables may change at any time and need not be current. While the layout-current variable is true, these variables are stable.

## 5.5 Receiver transitions

**read(block)**

*An application wants to read a particular block.*

```
op ← (READ, block, null, sender)
opqueue.append(op)
```

**write(block, value)**

*An application wants to write a value to a particular block.*

```
op ← (WRITE, block, value, sender)
opqueue.append(op)
```

## 5.6 Execution transitions

**(currop == NONE) ∧ (!opqueue.empty) ∧ layout-current**

*The execution engine isn't busy, believes it has a current copy of the layout, and there is an operation ready to go. Initiate the first stage of the appropriate transaction.*

```
currop ← opqueue.pop
resultvalue ← null
resultokay ← true
resultrestart ← true
currts ← clock
outstanding ← layout(currop.block, currop.op)
if currop.op == WRITE
     foreach (chunk, block) ∈ outstanding
          send prewrite(block, version, currts, currop.value) to chunk
else
     foreach (chunk, block) ∈ outstanding
          send read(block, version, currts) to chunk
set timeout to δml
```

**versionmismatch(block, timestamp)**

*Some chunk has indicated that the host is operating with out-of-date layout information. Set the flag that will cause the layout retrieval protocol to do its thing, and set the operation up to restart.*

```
if (timestamp == currts)
     resultokay ← false
     outstanding ← outstanding - (sender, block)
     layout-current ← false
```

**error(block,timestamp)**

*A chunk has reported that there is a grevious error in a request. When this happens, quit the operation.*

```
if (timestamp == currts)
     resultokay ← false
     resultrestart ← false
     outstanding ← outstanding - (sender, block)
```

**outoforder(block, timestamp)**

*Some chunk has indicated that an operation request has been received out of order. Set this operation up for being restarted.*

```
if (timestamp == currts)
     resultokay ← false
     outstanding ← outstanding - (sender, block)
```

**timeout**

*Some message has not been responded-to within the $\delta_{ml}$ timeout period. Restart the current operation.*

```
resultokay ← false
outstanding ← empty
```

**readresp(block, timestamp, value, status)**

*A chunk is responding with read data. Combine this with any other data received so far.*

```
if (timestamp == currts)
     outstanding ← outstanding - (sender, block)
     resultvalue ← F(resultvalue, value)
```

**(outstanding == empty) ∧ (currop.op == READ)**

*A read transaction has collected all the responses it is going to get. Determine the outcome of the transaction.*

```
cancel timeout
if resultokay
     send readresp(currop.block, resultvalue, OK) to currop.sender
else
     if resultrestart
          opqueue.prepend(currop)
     else
          send readresp(currop.block, null, FAIL) to currop.sender
currop ← NONE
currts ← 0
```

**prewriteack(block, timestamp)**

*A chunk is responding that it has received a prewrite. Once all prewrites have been acknowledged, commit the write.*

```
if (timestamp == currts)
     outstanding ← outstanding - (sender, block)
```

**(outstanding == empty) ∧ (currop.op == WRITE)**

*A write transaction has collected all the responses it is going to get. Determine the outcome of the transaction: if all acks have been received, commit the data, otherwise abort the prewrites and try to restart if possible; otherwise just return failure.*

```
cancel timeout
if resultokay
     foreach (chunk, block) ∈ layout(currop.block)
          send commit(block, version, currts) to chunk
     send writeresp(currop.block, OK) to currop.sender
else
     foreach (chunk, block) ∈ layout(currop.block)
          send abort(block, version, currts) to chunk
     if resultrestart
          opqueue.prepend(currop)
     else
     send writeresp(currop.block, FAIL) to currop.sender
currop ← NONE
currts ← 0
```

### 5.7 Extensions

**Dealing with message loss.** The current version of protocol will retry an operation forever. A more appropriate approach would be to retry up to a fixed number of times. This can be done by including a retry-count value in the opqueue and currop structures. This would be initialized to zero in the read() and write() transitions in the receiver, and incremented in the read and write finalization code (when the currop is put back on the operation queue.)

**Host failure and recovery.** The model as presented does not provide a way for hosts to fail and recover. This can be added by having a "functioning" state variable, predicating all existing transitions and that variable, and adding fail and recover transitions to change that variable.

**Multiple outstanding operations.** The engine currently only allows one (high-level) operation to be in process for any host. In practice this is needlessly restrictive, and standard techniques for having multiple outstanding operations in disk device drivers can be applied here. For example, any two operations that do not overlap could proceed concurrently. Alternately, read and write requests could be amended to include parallelization constraints and any ready operation could be executed.

This would necessitate changes to the receiver machine, to put more information in the operation queue. It would require extensive changes to the execution engine, to track more than one operation.

## 6 Chunks

### 6.1 General structure

The chunk portion of the access protocol processes read, prewrite, and commit requests coming from hosts. It depends on the layout control protocol to dictate when it can act, and when it can't. It is purely reactive.

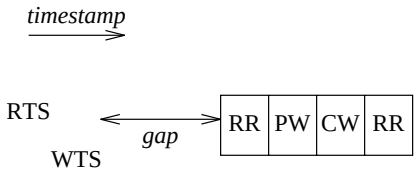
It consists of two machines: one that receives requests from hosts (read, prewrite, commit, and abort messages), checks their basic validity, and places them in the appropriate queue. The second machine takes operations that are ready to be executed (e.g. committed writes, reads with no preceding prewrites) out of the queue(s) and executes them.

Because of the drift in clocks and message latency variability, chunks cannot simply apply operations in the order they are received and still get consistent serialization in timestamp order. To ensure the proper order:

- each chunk keeps track of the timestamp of the last read and write operations applied to each block.
- when trying to apply a read with a timestamp before the last write, or a write with a timestamp before the last read or write, the chunk rejects

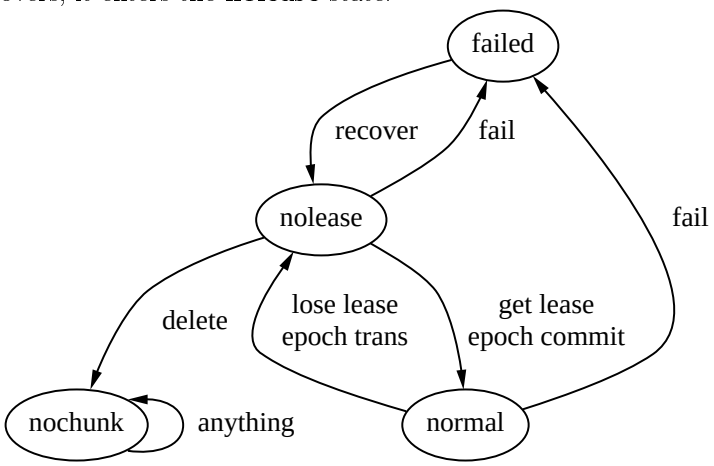
the operation as out of order. This means that writes are strictly ordered with respect to other operations, but reads of data from the same write are arbitrarily interchangeable.

- when operations must be queued behind uncommitted writes, the operations are kept in timestamp order.
- When processing an operation at the head of the queue, the operation's timestamp must fall in the gap between the max(RTS,WTS) and the oldest operation in the queue.




The current model puts all operations on the queue, so it is only necessary to check against the RTS and WTS. This differs from the explanation in [Amiri99], which tries to optimize the case when an operation does not need to be queued.

The chunk, as far as the access protocol is concerned, operates in one of four states. The **normal** state occurs when the chunk has a lease and can serve data. The chunk moves to a **nolease** state when it loses a lease, or during an epoch transition. It moves back to **normal** state when an epoch transition commits and the chunk is issued a new lease. Failures can occur in both the **normal** and **nolease** states, moving the chunk to the **failed** state. When the chunk recovers, it enters the **nolease** state.




When a chunk is deleted, as part of an epoch transition or by a manager while the chunk is attempting recovery, the chunk goes away. We model the absence of a chunk by the **nochunk** state, which specifies what the device will do when requests are made on a chunk that doesn't exist.

## 6.2 State

Some state variables are persistent; other variables are not.

| variable | type | use |
|---|---|---|
| clock | time | a clock that is synchronized to within some global $\delta$ of all other clocks in the system. |
| state | one of **failed**, **normal**, **nolease**, or **nochunk** | the overall state of the chunk; initially **nolease** |

The interface to the layout control protocol is:

| variable | type | use |
|---|---|---|
| haslease | bool | whether the chunk has a regular lease, and so can serve requests. Set asynchronously by the layout control protocol. Transient. |
| version | epoch number | the current epoch number; set by the layout control protocol; stable while haslease is true. |
| length | count | number of blocks in the chunk; set by the layout control protocol; stable while haslease is true. Persistent. |
| needreconciling | set of block | a set of blocks that are believed to be in need of reconciliation. This set is added-to by the access protocol and decreased by the LCP, when the chunk uses the LCP to contact a manager to initiate reconciliation. Initially empty; transient. |

Persistent data storage is:

| variable | type | use |
|---|---|---|
| blocks[b] | array [0..length-1] of value | the data storage |
| rts[b] | array [0..length-1] of timestamp | the time of last read for each block |
| wts[b] | array [0..length-1] of timestamp | the time of last write for each block |

Transient data is:

| variable | type | use |
|---|---|---|
| opqueue[b] | array [0..length-1] of priority queue of (op, ready, block, version, timestamp, value, sender) ordered by timestamp | for each block, a priority queue of pending read and prewrite operations. Supports a mints operation that returns the minimum timestamp in the queue, or +infinity if it is empty. Initially empty. |

## 6.3 Interfaces to other protocols

The access protocol in the chunk interfaces only with the layout control protocol, via the haslease and related variables. The access protocol is solely reactive to the LCP: when the LCP sets the haslease variable to true, the access protocol proceeds; when the LCP sets the variable to false, the access protocol ignores all events.

The chunk can request that the manager initiate reconciliation for a particular block by adding that block to a "needreconciling" set. Chunks do not directly initiate reconciliation because they aren't supposed to treat the layout metadata as opaque, which implies they don't know what other chunks would need to be involved.

## 6.4 General transitions

**fail $\wedge$ state $\neq$ nochunk**

*The chunk is failing.*

```
state ← failed
```

**recover $\wedge$ state == failed**

*The chunk is recovering after a failure. Note that the layout control protocol will be making a similar transition in parallel with this one, which will re-establish the variables that define the interface between the two protocols.*

```
needreconciling ← empty
for each b,
     opqueue[b] ← empty
state ← nolease
```

**haslease == false $\wedge$ state == normal**

*The chunk has lost its lease.*

```
state ← nolease
```

**haslease == true $\wedge$ state == nolease**

*The chunk has gotten a new lease.*

```
state ← normal
```

**delete**

*The chunk is being deleted. This is actually triggered by the layout control protocol. The main finalization needed is to send a version mismatch notice on any reads that are outstanding, which will trigger the reading host to retry the read on another copy of the data. Note that it is not incorrect to not send this message, since the host will eventually time out and retry the message anyway; sending the version mismatch notice will simply speed things up.*

```
for each b
     for each operation (READ, true, block, version,
               opts, null, sender) in opqueue[b]
          send versionmismatch(block, opts) to sender
state ← nochunk
```

**state == nochunk ∧ read(block, hversion, opts)**

*The host is requesting a data read on a nonexistent chunk.*

```
send versionmismatch(block, opts) to sender
```

**state == nochunk ∧ prewrite(block, hversion, opts, value)**

*The host is requesting a data write on a nonexistent chunk.*

```
send versionmismatch(block, opts) to sender
```

Note that other messages received while in the nochunk state are ignored.

## 6.5 Receiver machine transitions

Note that all these transitions implicitly include "∧ state == normal".

**read(block, hversion, opts)**

*The host is requesting that data be read.*

```
if (hversion != version)
     send versionmismatch(block, opts) to sender
elseif ((0 > block) or (block >= length))
     send error(block, opts) to sender
elseif (opts < wts[block])
     send outoforder(block, opts) to sender
else
     opqueue[block].append (READ, true, block, hversion, opts, null, sender)
```

**prewrite(block, hversion, opts, value)**

*The host is indicating is trying to begin a write; either a commit or abort with a matching opts will follow.*

```
if (hversion != version)
     send versionmismatch(block, opts) to sender
elseif ((0 > block) or (block >= length))
     send error(block, opts) to sender
elseif (opts < wts[block]) or (opts < rts[block])
     send outoforder(block, opts) to sender
else
     opqueue[block].append (WRITE, false, block, hversion, opts, value, sender)
     send prewriteack(block, opts) to sender
```

**abort(block, hversion, opts)**

*The host is aborting a previously-issued prewrite.*

```
op ← opqueue[block].lookup(opts)
if (op != null) ∧ (op.operation == WRITE)
     op.operation ← ABORT
     op.ready ← true
```

**commit(block, hversion, opts)**

*The host is aborting a previously-issued prewrite.*

```
op ← opqueue[block].lookup(opts)
if (op != null) ∧ (op.operation == WRITE)
     op.operation ← COMMIT
     op.ready ← true
```

## 6.6 Execution machine transitions

Note that all these transitions implicitly include "∧ state == normal".

**opqueue[b].notempty ∧ opqueue[b].head.ready ∧ (opqueue[b].head.op == READ)**

*Execute a ready read operation, for some block b.*

```
op ← opqueue[b].pop
if (op.timestamp < wts[block])
     send outoforder(b, op.timestamp) to sender
else
     value ← blocks[block]
     rts[b] ← max(rts[b], op.timestamp)
     send readresp(op.block, op.timestamp, value, OK) to op.sender
```

**opqueue[b].notempty ∧ opqueue[b].head.ready ∧ (opqueue[b].head.op == COMMIT)**

*Execute a committed write operation, for some block b.*

```
op ← opqueue[b].pop
blocks[b] ← op.value
wts[b] ← max(wts[b], op.timestamp)
```

**opqueue[b].notempty ∧ opqueue[b].head.ready ∧ (opqueue[b].head.op == ABORT)**

*Remove an aborted write operation from the queue.*

```
op ← opqueue[b].pop
```

**opqueue[b].notempty ∧ not opqueue[b].head.ready ∧ ((clock - opqueue[b].head.timestamp) > $\delta_{to}$)**

*Some operation has timed out. Note that in the IOA model this action could fire infinitely often until the problem is reconciled. A real implementation should be more selective.*

```
needreconciling ← needreconciling ∪ {b}
```

## 6.7 Extensions

**Separating data transmission from control.** As written, the data to be written is transmitted as part of the prewrite request. This is different from the protocol in [Amiri99], which transmits the data on the commit. This change is done to ensure that writes are properly recoverable on host failure. However, this has two drawbacks: first, if the prewrite is rejected, the data transmission is waste; second, this makes it hard for the chunk to control the flow of bulk data transmission.

A solution to this is to make writes three-phase rather than two: the first phase is a dataless prewrite and a response from the device when it is ready to accept the data. The second phase is purely data transmission, followed by an acknowledgment. The third phase is a dataless commit, followed by an ack. (This is inspired by the decoupling of command and data transfer in SCSI.)

**Inter-block operation scheduling.** As currently written, all operations only involve a single block, and the order of service is arbitrary when multiple operations, at multiple blocks, are concurrently ready. Traditional disk and array operation scheduling would apply here.

**Multiblock writes.** The current protocol model does not allow for serialization between operations on multiple blocks. There are several semantics for multiblock writes, each requiring its own extension to the protocol. These include:

- multiblock writes consisting of $n$ sequential blocks in the virtual address space;
- an arbitrary mix of block in the virtual address space; or
- explicit dependency relations among multiple separate write (and possibly read) operations.

If the host is supporting explicit inter-operation dependencies, it is probably useful to have the chunk support them as well. This might get the maximal possible parallelism out of the chunks, rather than having hosts reduce parallelism in order to avoid potential races among multiple hosts.

**Opqueue persistence.** As written, the opqueue is lost when a device fails. When the device recovers, the layout control protocol ensure that the reconciliation protocol is executed before the chunk is allowed to serve data again, which will update any committed writes that the chunk has lost—if a mirror copy is available. In general, handling $n$ individual concurrent transient failures requires $n + 1$ transient copies of the data. Alternately, a single persistent copy of the opqueue, to capture committed writes that have not yet been applied to the main persistent block store, will do the job. (Note that mass power failure amounts to all chunks getting a transient failure, so having at least one persistent copy of the opqueue is probably worth having.) This could be done in two different ways: by making the chunk's opqueue persistent, or by creating a lightweight "log chunk" that doesn't have a persistent block store, but only a persistent copy of the opqueue. If MRAM becomes widely available, it may make an effective medium for storing a persistent opqueue.

**Multiple opqueues.** Reserving space for one opqueue for every block is too expensive for actual implementation. There is engineering work to be done in condensing these multiple queues into a more efficient data structure.

**Timestamp writebacks.** [Amiri99] notes there are ways to optimize out the overhead of making the RTS and WTS data persistent.

**Messages while without a lease.** At present any read, prewrite, abort, and commit messages that are received while the chunk is without a lease are ignored. This can be improved by sending negative acknowledgments back to the sender, indicating that the chunk is temporarily blocked. A host could use this to adjust the period at which it retries an operation (for example, no write will succeed while even one chunk is blocked), or allow it to give up on the operation without having to wait a full timeout period.

# 7 Correctness

In this section we argue several correctness properties of the access protocol.

## 7.1 One-copy serialization

To show serialization, we must show that:

1. no two chunks process a pair of write operations in different orders; and
2. for any pair of read and write operations, all chunks either process the read first or the write first.

That is, reads are serialized with respect to writes, and writes are mutually serialized, but reads of the same data can be processed in any order. This is well known to produce a result equivalent to a one-copy serialization [ElAbaddi89].

Note that this argument is relative to a single block in a store, since the protocol as presently written only allows operations on single blocks. This simplifies the conflict and ordering constraints considerably.

### 7.1.1 Notation

- The set of chunks storing the block in question is $C$.
- Operations are denoted $o_x$. An operation is either a read or a write.
- An operation causes the host to make a series of *attempts* to perform the operation. Each attempt gets a new timestamp. Generally, all the attempts but the last fail; however, in practice on rare occasions more than one attempt to perform the operation will succeed. (For example, consider a prewrite that reaches all chunks just before a partition happens. The host will not receive acks, so will time out and try to abort the write; however, being partitioned, the aborts will not take effect. The chunks will eventually initiate recovery, which will run reconciliation, which will commit the write since all chunks received the appropriate prewrite. However, the host will still be trying to perform the write, and once the partition is repaired the host may try again and succeed.)
- The $j$th attempt for operation $o_i$ involves a set of chunks $C_{ij} \subseteq C$.

- The $j$th attempt for operation $o_i$ will, in the absence of any failures, cause a set of actions $a_{cij}, d \in C_{ij}$. Each action is either a read, a committed prewrite, or an aborted prewrite. When it is clear from context we may omit one or more of the subscripts on an action.
- The sequence of actions $a_1, a_2, \ldots, a_n$ performed at a chunk is the *history* at that chunk. This sequence is totally ordered, since the chunk's implementation only processes as single action at a time. For any action $a_i$, there are two well-defined sets $pred(a_i)$ and $succ(a_i)$, defining those actions that precede (respectively, follow) $a_i$ in the chunk's history.
- The timestamp associated with an action $a$ is written $ts(a)$.

### 7.1.2 Mutual write serialization with no failures

At this point, we consider only systems that do not experience failures.

**Lemma 1** *Chunks perform committed write actions for a single block in strict timestamp order. That is, for any two committed write actions $a_i$ and $a_j$, $i \neq j$, $a_i \in pred(a_j) \Rightarrow ts(a_i) < ts(a_j)$.*

**Proof:** Proof by induction over the sequence of actions $a_0, a_1, \ldots, a_j$. Note that we are only considering committed write actions at a chunk $c$, ignoring read and aborted write actions.

Base case: immediately before executing the first action, $a_0$, `wts` $= 0$ and the opqueue contains some set of actions. Since the opqueue is ordered by $ts(a)$, $ts(a_0)$ must be less than $ts(a_k)$, for the remaining $a_k$ in the opqueue. After executing $a_0$, `wts` $= ts(a_0)$ and $a_0$ has been removed from the opqueue. Thus $\forall a \in$ `opqueue`$, ts(a) >$ `wts` $= ts(a_0)$.

Induction step: The next action related to writes will either be the receipt of another action $a_k$, or the execution of some operation $a_k$ in the opqueue. On receipt of $a_k$, the action is rejected if $ts(a_k) <$ `wts`; otherwise, $a_k$ is added to the opqueue. This preserves the condition $\forall a \in$ `opqueue`$, ts(a) >$ `wts`. When $a_k$ is selected for execution, it must be the action with the least timestamp in the opqueue, so afterwards the condition still holds.

Thus $a_0, a_1, \ldots, a_n$ are executed in increasing timestamp order, so $a_i \in pred(a_j) \Rightarrow ts(a_i) < ts(a_j)$. ■

Note that this lemma does not address the fairness of execution; only that those actions that are executed are done so in timestamp order.

**Lemma 2** *$\exists ts.t. \forall c \in C_{ij}, ts(a_{cij}) = t$. That is, all actions $a_{cij}$ associated with the $j$th attempt of operation $o_i$ have the same timestamp.*

**Proof:** In the first transition of Section 5.6, the host draws a fresh timestamp for each attempt and sends the same timestamp to all chunks $c \in C_{ij}$. The network does not modify this value in transit. Chunks receive this timestamp and do not modify it, so the timestamp associated with every action is the one drawn by the host when the attempt begins. ■

**Theorem 1** *For any two operations $o_i$ and $o_j$, $i \neq j$, the associated write action at every chunk affected by both operations is performed in the same order. That is, either $\forall c \in (C_i \cap C_j), a_{ci} \in pred(a_{cj})$ or $\forall c \in (C_i \cap C_j), a_{ci} \in succ(a_{cj})$.*

**Proof:** By contradiction. Assume that there is some chunk $c \in (C_i \cap C_j)$ s.t. $a_{ci} \in succ(a_{cj})$ and some chunk $d \in (C_i \cap C_j)$ s.t. $a_{di} \in pred(a_{dj})$. By Lemma 1, $ts(a_{ci}) > ts(a_{cj})$. By Lemma 2, this implies $ts(a_{di}) = ts(a_{ci}) > ts(a_{cj}) = ts(a_{dj})$. However, by Lemma 1, $ts(a_{di}) < ts(a_{dj})$, which is a contradiction. Thus there can be no such pair of chunks $c$ and $d$. ■

### 7.1.3 Read-write serialization with no failures

To show that reads and writes are mutually serialized, we extend the results in the previous section concerning writes.

Chunks alternate between performing a committed write action and a set of read actions, for a single block at a single chunk, with the timestamp of the write less than the timestamps of all reads in the following set of reads and greater than the timestamps of all reads in the preceding set of reads. More formally:

**Lemma 3** *For any pair of read and write actions $a_r$ and $a_w$, $a_r \in pred(a_w) \Rightarrow ts(a_r) < ts(a_w)$, and $a_r \in succ(a_w) \Rightarrow ts(a_r) > ts(a_w)$.*

**Proof:** Case 1: $a_r \in pred(a_w)$. Consider the values of `rts` for the block. After the execution of $a_r$, `rts` $\geq ts(a_r)$, since `rts` is only updated in the processing of a read, and then to the max of the previous `rts` and $ts(a_r)$. At the time of executing $a_r$, there are two cases for action $a_w$. In the first case, it would have already have been received by the chunk and been placed in the opqueue, and since the opqueue is ordered by $ts(a)$, $ts(a_w) > ts(a_r)$. In the second case, the chunk receives $a_w$ sometime after executing $a_r$. On receipt, the write $a_w$ is rejected if $ts(a_w) <$ `rts`; thus for any $a_w \in succ(a_r)$, $ts(a_w) >$ `rts` $\geq ts(a_r)$. Thus $a_r \in pred(a_w) \Rightarrow ts(a_r) < ts(a_w)$.

Case 2: $a_r \in succ(a_w)$. This case is similar to the previous case, exchanging read for write. Consider the value of `wts` for the block. After the execution of $a_w$, `wts` $\geq ts(a_w)$. Immediately after executing $a_w$, either $a_r$ is in the opqueue, implying that $ts(a_r) > ts(a_w)$, or $a_r$ has yet to be received at the chunk. When $a_r$ is received, it is rejected if $ts(a_r) <$ `wts`, and so for any $a_r$ that is executed, $ts(a_r) >$ `wts` $\geq ts(a_w)$. Thus $a_r \in succ(a_w) \Rightarrow ts(a_r) > ts(a_w)$.

■

**Theorem 2** *For any two operations $o_r$ and $o_w$, the associated actions at every chunk affected by both operations is performed in the same order. That is, either $\forall c \in (C_r \cap C_w), a_{cr} \in pred(a_{cw})$ or $\forall c \in (C_r \cap C_w), a_{cr} \in succ(a_{cw})$.*

The proof is identical to that of Theorem 1.

### 7.1.4 Extending serialization for failures

The previous section showed that the protocol serializes writes, and reads with respect to writes, in timestamp order in the absence of failure. This section extends that result to show that operations are serializable in the presence of failures, though they are not serialized in strictly timestamp order.

**Theorem 3** *The protocol always serializes two write operations in the same order at all functioning chunks.*

This is easy to show, since chunks are strict about processing writes in strict timestamp order at all times.

Serializability of reads with respect to writes is a little more involved. We want to show that there is no disagreement in the reads-from relation: if one chunk says that read 1 $reads - from$ write 2, then no chunk should say that a later read 2 $reads - from$ some earlier write.

Note that this is weaker than saying that all chunks serialize actions in the same way, since some chunks may not process an action associated with one of the operations, either because of failure or because the chunk simply wasn't involved (for example, in a read).

**Theorem 4** *For any two reads $a_{c1}$ and $a_{d2}$, $ts(a_{c1}) < ts(a_{d2})$, if write $a_{cw} \in pred(a_{c1})$, then $\nexists d$ s.t $a_{dw} \in succ(a_{d2})$.*

**Proof:** By contradiction.

By Lemma 4, below, if $a_{cw}$ commits on chunk $c$, the matching action $a_{dw}$ on chunk $d$ will also commit, and $ts(a_{cw}) = ts(a_{dw})$. On chunk $c$, $ts(a_{c1}) > ts(a_{cw})$ since any individual chunk is strict about ordering reads wrt writes. Likewise $ts(a_{d2}) < ts(a_{dw})$. This implies that $ts(a_{d2}) < ts(a_{dw}) = ts(a_{cw}) < ts(a_{c1})$, which is a contradiction. ■

Note that this result does admit the possibility of a read with timestamp greater than that of a write, but where the read is serialized before the write. Consider chunk $c$, which receives a read request with timestamp $t_1$. Chunk $c$ responds to the read, then crashes. Just after $c$ crashes, some host sends a prewrite message with $t_2 < t_1$ due to clock drift, and all chunks except $c$ receive the message. Some time later an epoch transition occurs, removing chunk $c$ from the layout and then running the reconciliation protocol. The prewrite will commit, since all chunks in the new layout have received it, but its timestamp will be less than the read at $c$, which reflected data before the write but had a later timestamp.

**Lemma 4** *In the absence of permanent failure, for a write operation $o_w$, if any associated write action $a_{cw}$ at any chunk commits, all associated write actions will commit. That is, $\exists c \in C_w$ s.t. $a_{cw}$ commits $\Rightarrow \forall d \in C_w$, $a_{dw}$ commits.*

**Proof:** In the absence of failure, the only way for a prewrite action in chunk $c$'s opqueue to commit is for a host to send a commit message to the appropriate

chunk. The host only sends a commit once it has received an acknowledgment from every chunk involved in the write, which implies that there is a prewrite action in every chunk's opqueue, and that the host has sent commit messages to every chunk. Thus in the absence of failure, every chunk will have received a commit message for the prewrite and will thus have committed the action.

If there is some transient failure, then the host may time out and give up on the write, leaving the chunks in an inconsistent state. However, if some some functioning chunk does not have a copy of the prewrite, then no chunk will have committed since there must be a chain of causality from all chunks receiving the prewrite to the host issuing the commit. Likewise, if any chunk has received a commit, then there is a causal chain to all chunks receiving the prewrite. This leaves the chunks in one of three states, which the reconciliation protocol will use to drive the system to consistency:

- At least one has not received prewrite, and none have committed. The reconciliation protocol will abort those prewrites that have been received.
- All have received prewrite, but none have committed. The reconciliation protocol will commit the prewrites.
- All have received prewrite, and at least one has committed. The reconciliation protocol will commit those prewrites that have not yet been committed.

Thus in all cases where at least one chunk commits, all functioning chunks commit. ■

The key idea in serialization in this protocol is that the universality of commits forms a kind of interlock among the execution sequences at each chunk, and serialization is built on top of this interlock. When it doesn't exist, which can happen during epoch transitions, serialization in timestamp order is lost—but serializability remains intact. This implies that applications must not be able to see the timestamps used internally to order operations, lest they be tempted to try to deduce whether a write has committed or not.

## 7.2 Liveness

The access protocol is free of deadlock, as follows. The timestamps attached to each action sent to chunks partition the actions into a set of equivalence classes, and induce a total ordering on the equivalence classes. Within a chunk, an action can only be blocked on actions ahead of it in the opqueue, which have lesser timestamps and are therefore in lesser equivalence classes. Within an equivalence class, the only way blocking is possible is if at least one of the actions in the equivalence class is an uncommitted prewrite. Prewrites do not dependencies on each other; only on the host that initiated the write, or if that fails, on the completion of a reconciliation procedure to either commit or abort any prewrites in the class. Thus circular dependencies cannot be formed, and deadlock will not occur.

Note, however, that this does not guarantee unbounded waiting on any particular action. If hosts can execute arbitrarily fast, and timestamps are taken from a continuous domain, then it is easy to construct an infinitely-delayed write:

1. Host 1 sends a prewrite with timestamp $t$.
2. Host 2 repeatedly sends a read $n$ with timestamp $t - (1/n)$.

Host 2 will be able to generate an infinite amount of traffic that will be sequenced before Host 1's prewrite. However, the host must be able to execute arbitrarily fast for this to occur. If there is a bound on processor rates, or equivalently, timestamps are taken from a discrete domain, then no processor can insert more than a bounded number of actions in a chunk's opqueue ahead of any other processor's action, and with a bounded population of hosts one can establish an upper bound on the time before any arbitrary action will be executed.

## 7.3 Meeting interface requirements

The access protocol has interfaces with three other protocols: layout control, layout retrieval, and reconciliation. We ignore reconciliation here, deferring that to the document on that protocol.

The interface with the layout control protocol requires that the chunk not process operations when it does not have a lease. This is accomplished by conditioning all the transitions in Sections 6.5 and 6.6 on the presence of a valid lease. Any read, prewrite, commit, or abort messages received while there is no valid lease are ignored, and no action is executed. This may cause a host to time out and make another attempt to complete the operation – but the chunk that has lost its lease will remain correctly stopped. Note that this behavior can be improved, as noted in Section 6.7.

Chunks provide support for consistency control of hosts' cached metadata copies by checking the version number sent in read and prewrite messages against the chunk's current version (epoch) number, and only accepting messages with the correct version. However, there may be some time between accepting the message and actually executing the associated action – a read can sit in the opqueue for a while, and a prewrite needs a matching commit message. The check is *not* repeated when execution time comes along. This is acceptable, as follows:

- For a read action, when the version number changes, then either the chunk is remaining in the layout or it is being removed. If it remains in the layout, by the end of the epoch transition in which the version number is changed, the chunk will have a consistent copy of the block and can properly execute the read action. If the chunk is being removed from the layout, it rejects all reads in the opqueue by sending version mismatch messages.
- For a committed write, when the version number changes, the data should still be written to permanent store.

- For an uncommitted prewrite, the reconciliation process that accompanies the epoch transition that changes the version number will determine whether the prewrite should be committed or aborted, and so no uncommitted prewrites will survive into the new epoch.

# References

[Amiri99] Khalil Amiri, Garth A. Gibson, and Richard Golding. Scalable concurrency control and recovery for shared storage arrays. Technical report CMU–CS–99–111. Department of Computer Science, Carnegie-Mellon University, 1999.

[ElAbaddi89] Amr El Abaddi and Sam Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems*, **14**(2):264–90, June 1989.