Chain Replication metadata management in Machi, an immutable file store

Introducing the "humming consensus" algorithm

Basho Japan KK

1. Origins

This document was first written during the autumn of 2014 for a Basho-only internal audience. Since its original drafts, Machi has been designated by Basho as a full open source software project. This document has been rewritten in 2015 to address an external audience. For an overview of the design of the larger Machi system, please see [5].

2. Abstract

TODO Fix, after all of the recent changes to this document.

Machi is an immutable file store, now in active development by Basho Japan KK. Machi uses Chain Replication to maintain strong consistency of file updates to all replica servers in a Machi cluster. Chain Replication is a variation of primary/backup replication where the order of updates between the primary server and each of the backup servers is strictly ordered into a single "chain". Management of Chain Replication's metadata, e.g., "What is the current order of servers in the chain?", remains an open research problem. The current state of the art for Chain Replication metadata management relies on an external oracle (e.g., ZooKeeper) or the Elastic Replication algorithm.

This document describes the Machi chain manager, the component responsible for managing Chain Replication metadata state. The chain manager uses a new technique, based on a variation of CORFU, called "humming consensus". Humming consensus does not require active participation by all or even a majority of participants to make decisions. Machi's chain manager bases its logic on humming consensus to make decisions about how to react to changes in its environment, e.g. server crashes, network partitions, and changes by Machi cluster admnistrators. Once a decision is made during a virtual time epoch, humming consensus will eventually discover if other participants have made a different decision during that epoch. When a differing decision is discovered, new time epochs are proposed in which a new consensus is reached and disseminated to all available participants.

[]

3. Introduction

3.1 What does "self-management" mean?

For the purposes of this document, chain replication self-management is the ability for the N nodes in an N-length chain replication chain to manage the chain's metadata without requiring an external party to perform these management tasks. Chain metadata state and state management tasks include:

- Preserving data integrity of all metadata and data stored within the chain. Data loss is not an option.
- Preserving stable knowledge of chain membership (i.e. all nodes in the chain, regardless of operational status). A systems administrator is expected to make "permanent" decisions about chain membership.
- Using passive and/or active techniques to track operational state/status, e.g., up, down, restarting, full data sync, partial data sync, etc.
- Choosing the run-time replica ordering/state of the chain, based on current member status and past operational history. All chain state transitions must be done safely and without data loss or corruption.
- As a new node is added to the chain administratively or old node is restarted, adding the node to the chain safely and perform any data synchronization/repair required to bring the node's data into full synchronization with the other nodes.

3.2 Ultimate goal: Preserve data integrity of Chain Replicated data

Preservation of data integrity is paramount to any chain state management technique for Machi. Even when operating in an eventually consistent mode, Machi must not lose data without cause outside of all design, e.g., all particpants crash permanently.

3.3 Goal: Contribute to Chain Replication metadata management research

We believe that this new self-management algorithm, humming consensus, contributes a novel approach to Chain

Replication metadata management. The "monitor and mangage your neighbor" technique proposed in Elastic Replication (Section 4.1) appears to be the current state of the art in the distributed systems research community. Typical practice in the IT industry appears to favor using an external oracle, e.g., using ZooKeeper as a trusted coordinator.

See Section 4 for a brief review.

3.4 Goal: Support both eventually consistent & strongly consistent modes of operation

Machi's first use cases are all for use as a file store in an eventually consistent environment. In eventually consistent mode, humming consensus allows a Machi cluster to fragment into arbitrary islands of network partition, all the way down to 100% of members running in complete network isolation from each other. Furthermore, it provides enough agreement to allow formerly-partitioned members to coordinate the reintegration and reconciliation of their data when partitions are healed.

Later, we wish the option of supporting strong consistency applications such as CORFU-style logging while reusing all (or most) of Machi's infrastructure. Such strongly consistent operation is the main focus of this document.

3.5 Anti-goal: Minimize churn

Humming consensus's goal is to manage Chain Replication metadata safely. If participants have differing notions of time, e.g., running on extremely fast or extremely slow hardware, then humming consensus may "churn" rapidly in different metadata states in such a way that the chain's data is effectively unavailable.

In practice, however, any series of network partition changes that case humming consensus to churn will cause other management techniques (such as an external "oracle") similar problems. **[Proof by handwaving assertion.]** (See also: Section 5.3)

4. Review of current Chain Replication metadata management methods

We briefly survey the state of the art of research and industry practice of chain replication metadata management options.

4.1 "Leveraging Sharding in the Design of Scalable Replication Protocols" by Abu-Libdeh, van Renesse, and Vigfusson

Multiple chains are arranged in a ring (called a "band" in the paper). The responsibility for managing the chain at position N is delegated to chain N-1. As long as at least one chain is running, that is sufficient to start/bootstrap the next chain, and so on until all chains are running. This technique is called "Elastic Replication".

The paper then estimates mean-time-to-failure (MTTF) and suggests a "band of bands" topology to handle very large clusters while maintaining an MTTF that is as good or better than other management techniques.

NOTE: If the chain self-management method proposed for Machi does not succeed, this paper's technique is our best fallback recommendation.

4.2 An external management oracle, implemented by ZooKeeper

This is not a recommendation for Machi: we wish to avoid using ZooKeeper and any other "large" external service dependency. See the "Assumptions" section of [5] for Machi's overall design assumptions and limitations.

However, many other open source software products use ZooKeeper for exactly this kind of critical metadata replica management problem.

4.3 An external management oracle, implemented by Riak Ensemble

This is a much more palatable choice than option 4.2 above. We also wish to avoid an external dependency on something as big as Riak Ensemble. However, if it comes between choosing Riak Ensemble or choosing ZooKeeper, the choice feels quite clear: Riak Ensemble will win, unless there is some critical feature missing from Riak Ensemble. If such an unforseen missing feature is discovered, it would probably be preferable to add the feature to Riak Ensemble rather than to use ZooKeeper (and for Basho to document ZK, package ZK, provide commercial ZK support, etc.).

5. Assumptions

Given a long history of consensus algorithms (viewstamped replication, Paxos, Raft, et al.), why bother with a slightly different set of assumptions and a slightly different protocol?

The answer lies in one of our explicit goals: to have an option of running in an "eventually consistent" manner. We wish to be able to make progress, i.e., remain available in the CAP sense, even if we are partitioned down to a single isolated node. VR, Paxos, and Raft alone are not sufficient to coordinate service availability at such small scale. The humming consensus algorithm can manage both strongly consistency systems (i.e., the typical use for Chain Replication) as well as eventually consistent data systems.

5.1 The CORFU protocol is correct

This work relies tremendously on the correctness of the CORFU protocol [2], a cousin of the Paxos protocol. If the implementation of this self-management protocol breaks an assumption or prerequisite of CORFU, then we expect that Machi's implementation will be flawed.

5.2 Communication model

The communication model is asynchronous point-to-point messaging. The network is unreliable: messages may be arbitrarily dropped and/or reordered. Network partitions may occur at any time. Network partitions may be asymmetric, e.g., a message can be sent successfully from $A \rightarrow B$, but

messages from $B \rightarrow A$ can be lost, dropped, and/or arbitrarily delayed.

System particpants may be buggy but not actively malicious/Byzantine.

5.3 Time model

Our time model is per-node wall-clock time clocks, loosely synchronized by NTP.

The protocol and algorithm presented here do not specify or require any timestamps, physical or logical. Any mention of time inside of data structures are for human/historic/diagnostic purposes only.

Having said that, some notion of physical time is suggested for purposes of efficiency. It's recommended that there be some "sleep time" between iterations of the algorithm: there is no need to "busy wait" by executing the algorithm as quickly as possible. See also Section 10.2.2.

5.4 Failure detector model

We assume that the failure detector that the algorithm uses is weak, it's fallible, and it informs the algorithm in boolean status updates/toggles as a node becomes available or not.

5.5 Data consistency: strong unless otherwise noted

Most discussion in this document assumes a desire to preserve strong consistency in all data managed by Machi's chain replication. We use the short-hand notation "CP mode" to describe this default mode of operation, where "C" and "P" refer to the CAP Theorem [9].

However, there are interesting use cases where Machi is useful in a more relaxed, eventual consistency environment. We may use the short-hand "AP mode" when describing features that preserve only eventual consistency. Discussion of strongly consistent CP mode is always the default; exploration of AP mode features in this document will always be explictly noted.

5.6 Use of the "wedge state"

A participant in Chain Replication will enter "wedge state", as described by the Machi high level design [5] and by CORFU, when it receives information that a newer projection (i.e., run-time chain state reconfiguration) is available. The new projection may be created by a system administrator or calculated by the self-management algorithm. Notification may arrive via the projection store API or via the file I/O API.

When in wedge state, the server will refuse all file write I/O API requests until the self-management algorithm has determined that humming consensus has been decided (see next bullet item). The server may also refuse file read I/O API requests, depending on its CP/AP operation mode.

5.7 Use of "humming consensus"

CS literature uses the word "consensus" in the context of the problem description at [20] . This traditional definition differs from what is described here as "humming consensus".

"Humming consensus" describes consensus that is derived only from data that is visible/known at the current time. The algorithm will calculate a rough consensus despite not having input from all/majority of chain members. Humming consensus may proceed to make a decision based on data from only a single participant, i.e., only the local node.

See Section 10 for detailed discussion.

5.8 Concurrent chain managers execute humming consensus independently

Each Machi file server has its own concurrent chain manager process embedded within it. Each chain manager process will execute the humming consensus algorithm using only local state (e.g., the $P_{current}$ projection currently used by the local server) and values observed in everyone's projection stores (Section 6).

The chain manager communicates with the local Machi file server using the wedge and un-wedge request API. When humming consensus has chosen a projection P_{new} to replace $P_{current}$, the value of P_{new} is included in the un-wedge request.

6. The projection store

The Machi chain manager relies heavily on a key-value store of write-once registers called the "projection store". Each Machi node maintains its own projection store. The store's keyspace is divided into two halves (described below), each with different rules for who can write keys to that half of the store.

The store's key is a 2-tuple of a positive integer and the half of the partition, the "public" half or the "private" half. The integer represents the epoch number of the projection stored with this key. The store's value is either the special 'unwritten' value[1] or else a binary blob that is immutable thereafter; the projection data structure is serialized and stored in this binary blob.

The projection store is vital for the correct implementation of humming consensus (Section 10). The write-once register primitive allows us to reason about the store's behavior using the same logical tools and techniques as the CORFU ordered log.

6.1 The publicly-writable half of the projection store

The publicly-writable projection store is used to share information during the first half of humming consensus algorithm. Projections in the public half of the store form a log

[1] We use $\perp$ to denote the unwritten value.

of suggestions[2] by humming consensus participants for how they wish to change the chain's metadata state.

Any chain member may write to the public half of the store. Any chain member may read from the public half of the store.

6.2 The privately-writable half of the projection store

The privately-writable projection store is used to store the Chain Replication metadata state (as chosen by humming consensus) that is in use now by the local Machi server as well as previous operation states.

Only the local server may write values into the private half of store. Any chain member may read from the private half of the store.

The private projection store serves multiple purposes, including:

- Remove/clear the local server from "wedge state".
- Act as a world-readable indicator of what projection that the local server is currently using. The current projection will be called $P_{current}$ throughout this document.
- Act as the local server's log/history of its sequence of $P_{current}$ projection changes.

The private half of the projection store is not replicated.

7. Projections: calculation, storage, and use

Machi uses a "projection" to determine how its Chain Replication replicas should operate; see [5] and [2]. At runtime, a cluster must be able to respond both to administrative changes (e.g., substituting a failed server with replacement hardware) as well as local network conditions (e.g., is there a network partition?).

The projection defines the operational state of Chain Replication's chain order as well the (re-)synchronization of data managed by by newly-added/failed-and-now-recovering members of the chain. This chain metadata, together with computational processes that manage the chain, must be managed in a safe manner in order to avoid unintended data loss of data managed by the chain.

The concept of a projection is borrowed from CORFU but has a longer history, e.g., the Hibari key-value store [7] and goes back in research for decades, e.g., Porcupine [18].

7.1 The projection data structure

NOTE: This section is a duplicate of the "The Projection and the Projection Epoch Number" section of [5].

The projection data structure defines the current administration & operational/runtime configuration of a Machi cluster's single Chain Replication chain. Each projection is identified by a strictly increasing counter called the epoch projection number (or more simply "the epoch").

```
-type m_server_info() :: {Hostname, Port,...}.
-record(projection, {
            epoch_number    :: m_epoch_n(),
            epoch_csum      :: m_csum(),
            creation_time   :: now(),
            author_server   :: m_server(),
            all_members     :: [m_server()],
            active_upi      :: [m_server()],
            repairing       :: [m_server()],
            down_members    :: [m_server()],
            dbg_annotations :: proplist()
        }).
```

Figure 1. Sketch of the projection data structure

Projections are calculated by each server using input from local measurement data, calculations by the server's chain manager (see below), and input from the administration API. Each time that the configuration changes (automatically or by administrator's request), a new epoch number is assigned to the entire configuration data structure and is distributed to all servers via the server's administration API.

Pseudo-code for the projection's definition is shown in Figure 1. To summarize the major components:

- `epoch_number` and `epoch_csum` The epoch number and projection checksum are unique identifiers for this projection.
- `creation_time` Wall-clock time, useful for humans and general debugging effort.
- `author_server` Name of the server that calculated the projection.
- `all_members` All servers in the chain, regardless of current operation status. If all operating conditions are perfect, the chain should operate in the order specified here.
- `active_upi` All active chain members that we know are fully repaired/in-sync with each other and therefore the Update Propagation Invariant (Section A.1 is always true.
- `repairing` All running chain members that are in active data repair procedures.
- `down_members` All members that the `author_server` believes are currently down or partitioned.
- `dbg_annotations` A "kitchen sink" proplist, for code to add any hints for why the projection change was made, delay/retry information, etc.

7.2 Why the checksum field?

According to the CORFU research papers, if a server node S or client node C believes that epoch E is the latest epoch, then any information that S or C receives from any source that an epoch $E + \delta$ (where $\delta > 0$) exists will push S into

[2] I hesitate to use the word "propose" or "proposal" anywhere in this document ... until I've done a more formal analysis of the protocol. Those words have too many connotations in the context of consensus protocols such as Paxos and Raft.

the "wedge" state and C into a mode of searching for the projection definition for the newest epoch.

In the humming consensus description in Section 10, it should become clear that it's possible to have a situation where two nodes make proposals for a single epoch number. In the simplest case, assume a chain of nodes a and b. Assume that a symmetric network partition between a and b happens. Also, let's assume that operating in AP/eventually consistent mode.

On a's network-partitioned island, a can choose an active chain definition of `[A]`. Similarly b can choose a definition of `[B]`. Both a and B might choose the epoch for their proposal to be #42. Because each are separated by network partition, neither can realize the conflict.

When the network partition heals, it can become obvious to both servers that there are conflicting values for epoch #42. If we use CORFU's protocol design, which identifies the epoch identifier as an integer only, then the integer 42 alone is not sufficient to discern the differences between the two projections.

Humming consensus requires that any projection be identified by both the epoch number and the projection checksum, as described in Section 7.1.

8. Managing multiple projection store replicas

An independent replica management technique very similar to the style used by both Riak Core [12] and Dynamo is used to manage replicas of Machi's projection data structures. The major difference is that humming consensus *does not necessarily require* successful return status from a minimum number of participants (e.g., a quorum).

8.1 Read repair: repair only unwritten values

The idea of "read repair" is also shared with Riak Core and Dynamo systems. However, Machi has situations where read repair cannot truly "fix" a key because two different values have been written by two different replicas. Machi's projection store is write-once, and there is no "undo" or "delete" or "overwrite" in the projection store API.[3] Machi's projection store read repair can only repair values that are unwritten, i.e., storing $\bot$.

The value used to repair $\bot$ values is the "best" projection that is currently available for the current epoch E. If there is a single, unanimous value V_u for the projection at epoch E, then V_u is use to repair all projections stores at E that contain $\bot$ values. If the value of K is not unanimous, then the "highest ranked value" V_{best} is used for the repair; see Section 10.6 for a description of projection ranking.

[3] It doesn't matter what caused the two different values. In case of multiple values, all participants in humming consensus merely agree that there were multiple suggestions at that epoch which must be resolved by the creation and writing of newer projections with later epoch numbers.

8.2 Writing to public projection stores

Writing replicas of a projection P_{new} to the cluster's public projection stores is similar, in principle, to writing a Chain Replication-managed system or Dynamo-like system. But unlike Chain Replication, the order doesn't really matter. In fact, the two steps below may be performed in parallel. The significant difference with Chain Replication is how we interpret the return status of each write operation.

1. Write P_{new} to the local server's public projection store using P_{new}'s epoch number E as the key. As a side effect, a successful write will trigger "wedge" status in the local server, which will then cascade to other projection-related activity by the local chain manager.
2. Write P_{new} to key E of each remote public projection store of all participants in the chain.

In cases of `error_written` status, the process may be aborted and read repair triggered. The most common reason for `error_written` status is that another actor in the system has already calculated another (perhaps different) projection using the same projection epoch number and that read repair is necessary. The `error_written` may also indicate that another server has performed read repair on the exact projection P_{new} that the local server is trying to write!

8.3 Writing to private projection stores

Only the local server/owner may write to the private half of a projection store. Also, the private projection store is not replicated.

8.4 Reading from public projection stores

A read is simple: for an epoch E, send a public projection read API request to all participants. As when writing to the public projection stores, we can ignore any timeout/unavailable return status.[4] If we discover any unwritten values $\bot$, the read repair protocol is followed.

The minimum number of non-error responses is only one.[5] If all available servers return a single, unanimous value $V_u, V_u \neq \bot$, then V_u is the final result for epoch E. Any non-unanimous values are considered complete disagreement for the epoch. This disagreement is resolved by humming consensus by later writes to the public projection stores during subsequent iterations of humming consensus.

We are not concerned with unavailable servers. Humming consensus only uses as many public projections as are available at the present moment of time. If some server S is unavailable at time t and becomes available at some later $t + \delta$, and if at $t + \delta$ we discover that S's public projection store

[4] The success/failure status of projection reads and writes is *not* ignored with respect to the chain manager's internal liveness tracker. However, the liveness tracker's state is typically only used when calculating new projections.

[5] The local projection store should always be available, even if no other remote replica projection stores are available.

for key E contains some disagreeing value V_{weird}, then the disagreement will be resolved in the exact same manner that would be used as if we had found the disagreeing values at the earlier time t.

9. Phases of projection change, a prelude to Humming Consensus

Machi's projection changes use four discrete phases: network monitoring, projection calculation, projection storage, and adoption of new projections. The phases are described in the subsections below. The reader should then be able to recognize each of these phases when reading the humming consensus algorithm description in Section 10.

9.1 Network monitoring

Monitoring of local network conditions can be implemented in many ways. None are mandatory, as far as this RFC is concerned. Easy-to-maintain code should be the primary driver for any implementation. Early versions of Machi may use some/all of the following techniques:

- Other FLU file and projection store API requests.
- Internal "no op" FLU-level protocol request & response.
- Network tests via ICMP `ECHO_REQUEST`, a.k.a. `ping(8)`

Output of the monitor should declare the up/down (or alive/unknown) status of each server in the projection. Such Boolean status does not eliminate fuzzy logic, probabilistic methods, or other techniques for determining availability status. A hard choice of boolean up/down status is required only by the projection calculation phase (Section 9.2).

9.2 Calculating a new projection data structure

A new projection may be required whenever an administrative change is requested or in response to network conditions (e.g., network partitions, crashed server).

Projection calculation is a pure computation, based on input of:

1. The current projection epoch's data structure
2. Administrative request (if any)
3. Status of each server, as determined by network monitoring (Section 9.1).

Decisions about *when* to calculate a projection are made using additional runtime information. Administrative change requests probably should happen immediately. Change based on network status changes may require retry logic and delay/sleep time intervals.

9.3 Writing a new projection

In Machi's case, the writing a new projection phase is very straightforward; see Section 8.2 for the technique for writing projections to all participating servers' projection stores. Humming Consensus does not care if the writes succeed or not: its final phase, adopting a new projection, will determine which write operations are usable.

9.4 Adoption a new projection

It may be helpful to consider the projections written to the cluster's public projection stores as "suggestions" for what the cluster's new projection ought to be. (We avoid using the word "proposal" here, to avoid direct parallels with protocols such as Raft and Paxos.)

In general, a projection P_{new} at epoch E_{new} is adopted by a server only if the change in state from the local server's current projection to new projection, $P_{current} \rightarrow P_{new}$ will not cause data loss, e.g., the Update Propagation Invariant and all other safety checks required by chain repair in Section 12 are correct. For example, any new epoch must be strictly larger than the current epoch, i.e., $E_{new} > E_{current}$.

Machi first reads the latest projection from all available public projection stores. If the result is not a single unanmous projection, then we return to the step in Section 9.2. If the result is a *unanimous* projection P_{new} in epoch E_{new}, and if P_{new} does not violate chain safety checks, then the local node may replace its local $P_{current}$ projection with P_{new}.

Not all safe projection transitions are useful, however. For example, it's trivally safe to suggest projection P_{zero}, where the chain length is zero. In an eventual consistency environment, projection P_{one} where the chain length is exactly one is also trivially safe.[6]

10. Humming Consensus

Humming consensus describes consensus that is derived only from data that is visible/available at the current time. It's OK if a network partition is in effect and not all chain members are available; the algorithm will calculate a rough consensus despite not having input from all chain members. Humming consensus may proceed to make a decision based on data from only one participant, i.e., only the local node.

- When operating in AP mode, i.e., in eventual consistency mode, humming consensus may reconfigure a chain of length N into N independent chains of length 1. When a network partition heals, the humming consensus is sufficient to manage the chain so that each replica's data can be repaired/merged/reconciled safely. Other features of the Machi system are designed to assist such repair safely.
- When operating in CP mode, i.e., in strong consistency mode, humming consensus would require additional restrictions. For example, any chain that didn't have a minimum length of the quorum majority size of all members would be invalid and therefore would not move itself out of wedged state. In very general terms, this requirement

[6] Although, if the total number of participants is more than one, eventual consistency would demand that P_{self} cannot be used forever.

for a quorum majority of surviving participants is also a requirement for Paxos, Raft, and ZAB. See Section 11 for a proposal to handle "split brain" scenarios while in CP mode.

If a projection suggestion is made during epoch E, humming consensus will eventually discover if other participants have made a different suggestion during epoch E. When a conflicting suggestion is discovered, newer & later time epochs are defined to try to resolve the conflict.

The next portion of this section follows the same pattern as Section 9: network monitoring, calculating new projections, writing projections, then perhaps adopting the newest projection (which may or may not be the projection that we just wrote). Beginning with Section 10.5, we provide additional detail to the rough outline of humming consensus.

This section will refer heavily to Figure 2, a flowchart of the humming consensus algorithm. The following notation is used by the flowchart and throughout this section.

Author The name of the server that created the projection.

Rank Assigns a numeric score to a projection, see Section 10.6.

E The epoch number of a projection.

UPI "Update Propagation Invariant". The UPI part of the projection is the ordered list of chain members where the Update Propagation Invariant of the original Chain Replication paper [19] is preserved. All UPI members of the chain have their data fully synchronized and consistent, except for updates in-process at the current instant in time. The UPI list is what Chain Replication usually considers "the chain". For strongly consistent read operations, all clients send their read operations to the tail/last member of the UPI server list. In Hibari's implementation of Chain Replication [7], the chain members between the "head" and "official tail" (inclusive) are what Machi calls the UPI server list. (See also Section A.1.)

Repairing The ordered list of nodes that are in repair mode, i.e., synchronizing their data with the UPI members of the chain. In Hibari's implementation of Chain Replication, any chain members that follow the "official tail" are what Machi calls the repairing server list.

Down The list of chain members believed to be down, from the perspective of the author.

$\mathbf{P_{current}}$ The projection actively used by the local node right now. It is also the projection with largest epoch number in the local node's private projection store.

$\mathbf{P_{newprop}}$ A new projection suggestion, as calculated by the local server (Section 10.2).

$\mathbf{P_{latest}}$ The highest-ranked projection with the largest single epoch number that has been read from all available public projection stores, including the local node's public projection store.

Unanimous The P_{latest} projection is unanimous if all replicas in all accessible public projection stores are effectively identical. All major elements such as the epoch number, checksum, and UPI list must the same.

$\mathbf{P_{current} \rightarrow P_{latest}}$ **transition safe?** A predicate function to check the sanity & safety of the transition from the local server's $P_{current}$ to the P_{latest} projection.

Stop state One iteration of the self-management algorithm has finished on the local server.

The flowchart has three columns, from left to right:

Column A Is there any reason to change?

Column B Do I act?

Column C How do I act?

C1xx Save latest suggested projection to local private store, unwedge, then stop.

C2xx Ask the author of P_{latest} to try again, then we wait, then iterate.

C3xx Our new projection $P_{newprop}$ appears best, so write it to all public projection stores, then iterate.

The Erlang source code that implements the Machi chain manager is structured as a state machine where the function executing for the flowchart's state is named by the approximate location of the state within the flowchart. Most flowchart states in a column are numbered in increasing order, top-to-bottom. These numbers appear in blue in Figure 2. Some state numbers, such as $A40$, describe multiple flowchart states; the Erlang code for that function, e.g. `react_to_env_A40()`, implements the logic for all such flowchart states.

10.1 Network monitoring

The actions described in this section are executed in the top part of Column A of Figure 2. See also, Section 9.1.

In today's implementation, there is only a single criterion for determining the alive/perhaps-not-alive status of a remote server S: is S's projection store available now? This question is answered by attemping to read the projection store on server S. If successful, then we assume that all S is available. If S's projection store is not available for any reason (including timeout), we assume S is entirely unavailable. This simple single criterion appears to be sufficient for humming consensus, according to simulations of arbitrary network partitions.

10.2 Calculating a new projection data structure

The actions described in this section are executed in the top part of Column A of Figure 2. See also, Section 9.2.

Execution starts at "Start" state of Column A of Figure 2. Rule $A20$'s uses recent success & failures in accessing other public projection stores to select a hard boolean up/down status for each participating server.

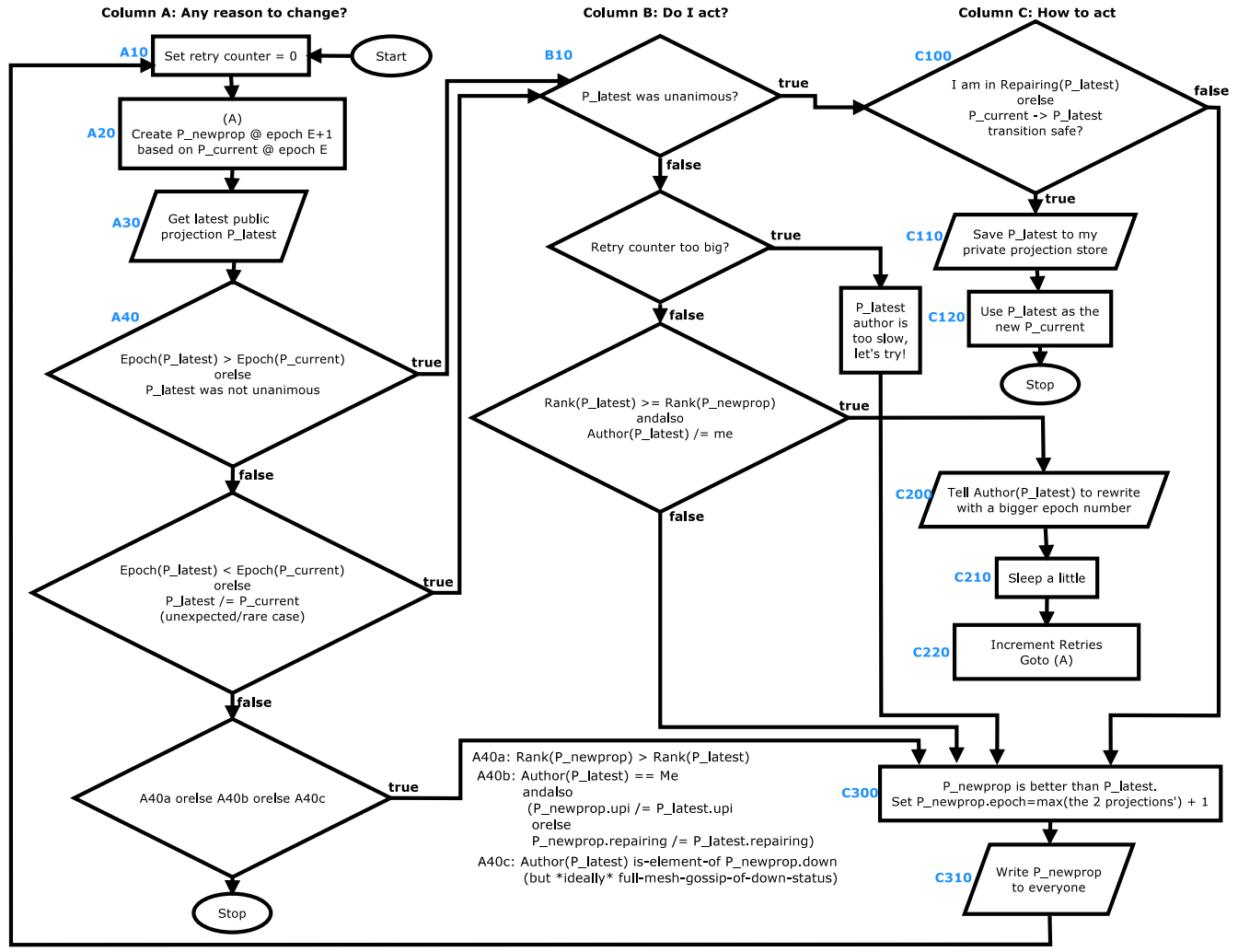

Figure 2. Humming consensus flow chart

10.2.1 Calculating flapping state

Also at this stage, the chain manager calculates its local "flapping" state. The name "flapping" is borrowed from IP network engineer jargon "route flapping":

> "Route flapping is caused by pathological conditions (hardware errors, software errors, configuration errors, intermittent errors in communications links, unreliable connections, etc.) within the network which cause certain reachability information to be repeatedly advertised and withdrawn." [21]

Flapping due to constantly changing network partitions and/or server crashes and restarts Currently, Machi does not attempt to dampen, smooth, or ignore recent history of constantly flapping peer servers. If necessary, a failure detector such as the ϕ accrual failure detector [10] can be used to help mange such situations.

Flapping due to asymmetric network partitions The simulator's behavior during stable periods where at least one node is the victim of an asymmetric network partition is . . . weird, wonderful, and something I don't completely understand yet. This is another place where we need more eyes reviewing and trying to poke holes in the algorithm.

In cases where any node is a victim of an asymmetric network partition, the algorithm oscillates in a very predictable way: each server S makes the same P_{new} projection at epoch E that S made during a previous recent epoch $E - \delta$ (where δ is small, usually much less than 10). However, at least one node makes a suggestion that makes rough consensus impossible. When any epoch E is not acceptable (because some node disagrees about something, e.g., which nodes are down), the result is more new rounds of suggestions that create a repeating loop that lasts as long as the asymmetric partition lasts.

From the perspective of S's chain manager, the pattern of this infinite loop is easy to detect: S inspects the pattern

of the last L projections that it has suggested, e.g., the last 10. Tiny details such as the epoch number and creation timestamp will differ, but the major details such as UPI list and repairing list are the same.

If the major details of the last L projections authored and suggested by S are the same, then S unilaterally decides that it is "flapping" and enters flapping state. See Section 10.5 for additional disucssion of the flapping state.

10.2.2 When to calculate a new projection

The Chain Manager schedules a periodic timer to act as a reminder to calculate a new projection. The timer interval is typically 0.5–2.0 seconds, if the cluster has been stable. A client may call an external API call to trigger a new projection, e.g., if that client knows that an environment change has happened and wishes to trigger a response prior to the next timer firing.

It's recommended that the timer interval be staggered according to the participant ranking rules in Section 10.6; higher-ranked servers use shorter timer intervals. Staggering sleep timers is not required, but the total amount of churn (as measured by suggested projections that are ignored or immediately replaced by a new and nearly-identical projection) is lower when using staggered timers.

10.3 Writing a new projection

See also: Section 9.3.

To focus very specifically about writing a projection, Figure 2 shows that writing a private projection is done by state $C110$ and that writing a public projection is done by states $C300$ and $C310$.

Broadly speaking, there are a number of decisions made in all three columns of Figure 2 to decide if and when any type of projection should be written at all. Sometimes, the best action is to do nothing.

10.3.1 Column A: Is there any reason to change?

The main tasks of the flowchart states in Column A is to calculate a new projection P_{new} and perhaps also the inner projection P_{new2} if we're in flapping mode. Then we try to figure out which projection has the greatest merit: our current projection $P_{current}$, the new projection P_{new}, or the latest epoch P_{latest}. If our local $P_{current}$ projection is best, then there's nothing more to do.

10.3.2 Column B: Do I act?

The main decisions that states in Column B need to make are:

- Is the P_{latest} projection written unanimously (as far as we call tell right now)? If yes, then we consider using it for our new internal state; go to state $C100$.
- We compare P_{latest} projection to our local P_{new}. If P_{latest} is better, then we wait for a while. The waiting loop is broken by a local retry counter. If the counter is small enough, we wait (via state $C200$). While we wait, the author of the P_{latest} projection will have an opportunity to re-write it in a newer epoch unanimously. If the retry counter is too big, then we break out of our loop and go to state $C300$.
- Otherwise we go to state $C300$, where we try to write our P_{new} to all public projection stores because, as far as we can discern, our projection is best and everyone else ought to know it.

It's notable that if P_{new} is truly the best projection available at the moment, it must always first be written to everyone's public projection stores and only then processed through another monitor & calculate loop through the flowchart.

10.3.3 Column C: How do I act?

This column contains three variations of how to act:

C1xx Try to adopt the P_{latest} suggestion. If the transition between $P_{current}$ to P_{latest} isn't safe, then jump to $C300$. If it is completely safe, we'll use it by storing P_{latest} in our local private projection store and then adopt it by setting $P_{current} = P_{latest}$.

C2xx Do nothing but sleep a while. Then we loop back to state $A20$ and step through the flowchart loop again. Optionally, we might want to poke the author of P_{latest} to ask it to write its proposal unanimously in a later epoch.

C3xx We try to replicate our P_{new} suggestion to all local projection stores, because it seems best.

10.4 Adopting a new projection

See also: Section 9.4.

The latest projection P_{latest} is adopted by a Machi server at epoch E if the following two requirements are met:

#1: All available copies of P_{latest} are unanimous/identical If we read two projections at epoch E, P_E^1 and P_E^2, with different checksum values, then we must consider $P_E^2 \neq P_E^1$ and therefore the suggested projections at epoch E are not unanimous.

#2: The transition from current $\rightarrow$ new projection is safe Given the projection that the server is currently using, $P_{current}$, the projection P_{latest} is evaluated by numerous rules and invariants, relative to $P_{current}$. If such rule or invariant is violated/false, then the local server will discard P_{latest}.

The transition from $P_{current} \rightarrow P_{latest}$ is checked for safety and sanity. The conditions used for the check include:

1. The Erlang data types of all record members are correct.
2. The members of the UPI, repairing, and down lists contain no duplicates and are in fact mutually disjoint.
3. The author node is not down (as far as we can observe).

4. There is no re-ordering of the UPI list members: the relative order of the UPI list members in both projections must be strictly maintained. The same re-reordering restriction applies to all servers in P_{latest}'s repairing list relative to $P_{current}$'s repairing list.
5. Any server S that was added to P_{latest}'s UPI list must appear in the tail the UPI list. Furthermore, S must have been in $P_{current}$'s repairing list and had successfully completed file repair prior to the transition.

10.5 Additional discussion of flapping state

All P_{new} projections calculated while in flapping state have additional diagnostic information added, including:

- Flag: server S is in flapping state.
- Epoch number & wall clock timestamp when S entered flapping state.
- The collection of all other known participants who are also flapping (with respective starting epoch numbers).
- A list of nodes that are suspected of being partitioned, called the "hosed list". The hosed list is a union of all other hosed list members that are ever witnessed, directly or indirectly, by a server while in flapping state.

10.5.1 Flapping diagnostic data accumulates

While in flapping state, this diagnostic data is gathered from all available participants and merged together in a CRDT-like manner. Once added to the diagnostic data list, a datum remains until S drops out of flapping state. When flapping state stops, all accumulated diagnostic data is discarded.

This accumulation of diagnostic data in the projection data structure acts in part as a substitute for a separate gossip protocol. However, since all participants are already communicating with each other via read & writes to each others' projection stores, the diagnostic data can propagate in a gossip-like manner via the projection stores.

10.5.2 Flapping example (part 1)

Any server listed in the "hosed list" is suspected of having some kind of network communication problem with some other server. For example, let's examine a scenario involving a Machi cluster of servers a, b, c, d, and e. Assume there exists an asymmetric network partition such that messages from $a \rightarrow b$ are dropped, but messages from $b \rightarrow a$ are delivered.[7]

Once a participant S enters flapping state, it starts gathering the flapping starting epochs and hosed lists from all of the other projection stores that are available. The sum of this info is added to all projections calculated by S. For example, projections authored by a will say that a believes that b is down. Likewise, projections authored by b will say that b believes that a is down.

10.5.3 The inner projection (flapping example, part 2)

. . . We continue the example started in the previous subsection. . .

Eventually, in a gossip-like manner, all other participants will eventually find that their hosed list is equal to $[a, b]$. Any other server, for example server c, will then calculate another projection, P_{new2}, using the assumption that both a and b are down in addition to all other known unavailable servers.

- If operating in the default CP mode, both a and b are down and therefore not eligible to participate in Chain Replication. This may cause an availability problem for the chain: we may not have a quorum of participants (real or witness-only) to form a correct UPI chain.
- If operating in AP mode, a and b can still form two separate chains of length one, using UPI lists of $[a]$ and $[b]$, respectively.

This re-calculation, P_{new2}, of the new projection is called an "inner projection". The inner projection definition is nested inside of its parent projection, using the same flapping disagnostic data used for other flapping status tracking.

When humming consensus has determined that a projection state change is necessary and is also safe (relative to both the outer and inner projections), then the outer projection[8] is written to the local private projection store. With respect to future iterations of humming consensus, the innter projection is ignored. However, with respect to Chain Replication, the server's subsequent behavior *will consider the inner projection only*. The inner projection is used to order the UPI and repairing parts of the chain and trigger wedge/unwedge behavior. The inner projection is also advertised to Machi clients.

The epoch of the inner projection, E^{inner} is always less than or equal to the epoch of the outer projection, E. The E^{inner} epoch typically only changes when new servers are added to the hosed list.

To attempt a rough analogy, the outer projection is the carrier wave that is used to transmit the inner projection and its accompanying gossip of diagnostic data.

10.5.4 Outer projection churn, inner projection stability

One of the intriguing features of humming consensus's reaction to asymmetric partition: flapping behavior continues for as long as an any asymmetric partition exists.

10.5.5 Stability in symmetric partition cases

Although humming consensus hasn't been formally proven to handle all asymmetric and symmetric partition cases, the

[7] If this partition were happening at or below the level of a reliable delivery network protocol like TCP, then communication in *both* directions would be affected by an asymmetric partition. However, in this model, we are assuming that a "message" lost during a network partition is a uni-directional projection API call or its response.

[8] With the inner projection P_{new2} nested inside of it.

current implementation appears to converge rapidly to a single chain state in all symmetric partition cases. This is in contrast to asymmetric partition cases, where "flapping" will continue on every humming consensus iteration until all asymmetric partition disappears. A formal proof is an area of future work.

10.5.6 Leaving flapping state and discarding inner projection

There are two events that can trigger leaving flapping state.

- A server S in flapping state notices that its long history of repeatedly suggesting the same projection will be broken: S instead calculates some differing projection instead. This change in projection history happens whenever a perceived network partition changes in any way.
- Server S reads a public projection suggestion, P_{noflap}, that is authored by another server S', and that P_{noflap} no longer contains the flapping start epoch for S' that is present in the history that S has maintained while S has been in flapping state.

When either trigger event happens, server S will exit flapping state. All new projections authored by S will have all flapping diagnostic data removed. This includes stopping use of the inner projection: the UPI list of the inner projection is copied to the outer projection's UPI list, to avoid a drastic change in UPI membership.

10.6 Ranking projections

A projection's rank is based on the epoch number (higher always wins), chain length (larger wins), number & state of any repairing members of the chain (larger wins), and node name of the author server (as a tie-breaking criteria).

11. "Split brain" management in CP Mode

Split brain management is a thorny problem. The method presented here is one based on pragmatics. If it doesn't work, there isn't a serious worry, because Machi's first serious use case all require only AP Mode. If we end up falling back to "use Riak Ensemble" or "use ZooKeeper", then perhaps that's fine enough. Meanwhile, let's explore how a completely self-contained, no-external-dependencies CP Mode Machi might work.

Wikipedia's description of the quorum consensus solution[9] is nice and short:

> A typical approach, as described by Coulouris et al.,[4] is to use a quorum-consensus approach. This allows the sub-partition with a majority of the votes to remain available, while the remaining sub-partitions should fall down to an auto-fencing mode.[10]

[9] See `http://en.wikipedia.org/wiki/Split-brain_(computing)`.

[10] Any server on the minority side refuses to operate because it is, so to speak, "on the wrong side of the fence."

Partition "side" Quorum UPI	Partition "side" Minority UPI
$[S_1, S_0, S_2]$	[]
$[W_0, S_1, S_0]$	$[W_1, S_2]$
$[W_1, W_0, S_1]$	$[S_0, S_2]$

Figure 3. Illustration of witness servers: on the left side, witnesses provide enough servers to form a UPI chain of quorum length. Servers on the right side cannot suggest a quorum UPI chain and therefore wedge themselves. Under real conditions, there may be multiple minority "sides".

This is the same basic technique that both Riak Ensemble and ZooKeeper use. Machi's extensive use of write-once registers are a big advantage when implementing this technique. Also very useful is the Machi "wedge" mechanism, which can automatically implement the "auto-fencing" that the technique requires. All Machi servers that can communicate with only a minority of other servers will automatically "wedge" themselves, refuse to author new projections, and refuse all file API requests until communication with the majority can be re-established.

11.1 The quorum: witness servers vs. real servers

In any quorum-consensus system, at least $2f + 1$ participants are required to survive f participant failures. Machi can borrow an old technique of "witness servers" to permit operation despite having only a minority of "real" servers.

A "witness server" is one that participates in the network protocol but does not store or manage all of the state that a "real server" does. A "real server" is a Machi server as described by this RFC document. A "witness server" is a server that only participates in the projection store and projection epoch transition protocol and a small subset of the file access API. A witness server doesn't actually store any Machi files. A witness server's state is very tiny when compared to a real Machi server.

A mixed cluster of witness and real servers must still contain at least a quorum $f + 1$ participants. However, as few as one of them may be a real server, and the remaining f are witness servers. In such a cluster, any majority quorum must have at least one real server participant.

Witness servers are always placed at the front of the chain.

When in CP mode, any server that is on the minority side of a network partition and thus cannot calculate a new projection that includes a quorum of servers will enter wedge state and remain wedged until the network partition heals enough to communicate with a quorum of FLUs. This is a nice property: we automatically get "fencing" behavior.

11.2 Witness server data and protocol changes

Some small changes to the projection's data structure are required (relative to the initial spec described in [5]). The projection itself needs new annotation to indicate the oper-

ating mode, AP mode or CP mode. The state type notifies the chain manager how to react in network partitions and how to calculate new, safe projection transitions and which file repair mode to use (Section 12). Also, we need to label member server servers as real- or witness-type servers.

Write API requests are processed by witness servers in *almost but not quite* no-op fashion. The only requirement of a witness server is to return correct interpretations of local projection epoch numbers, via the `error_bad_epoch` and `error_wedged` error codes. In fact, a new API call is sufficient for querying witness servers: {`check_epoch, m_epoch()`}. Any client write operation sends the `check_epoch` API command to witness servers and sends the usual `write_req` command to real servers.

11.3 Restarting after entire chain crashes

There's a corner case that requires additional safety checks to preserve strong consistency: restarting after the entire chain crashes.

The default restart behavior for the chain manager is to start the local server S with $P_{current} = P_{zero}$, i.e., S believes that the current chain length is zero. Then S's chain manager will attempt to join the chain by waiting for another active chain member S' to notice that S is now available. Then S''s chain manager will automatically suggest a projection where S is added to the repairing list. If there is no other active server, then S will suggest projection P_{one}, a chain of length one where S is the sole UPI member of the chain.

The default restart strategy cannot work correctly if: a). all members of the chain crash simultaneously (e.g., power failure), or b). the UPI chain was not at maximum length (i.e., no chain members are under repair or down). For example, assume that the cluster consists of servers S_a, S_b, and witness W_0. Assume that the UPI chain is $P_{one} = [W_0, S_a]$ when a power failure halts the entire data center. When power is restored, let's assume server S_b restarts first. S_b's chain manager must suggest neither $[S_b]$ nor $[W_0, S_b]$. Clients must not access S_b at this time because we do not know how much stale data S_b may have.

The chain's operational history is preserved and distributed amongst the participants' private projection stores. The maximum of the private projection store's epoch number from a quorum of servers (including witnesses) gives sufficient information to know how to safely restart a chain. In the example above, we must endure the worst-case and wait until S_a also returns to service.

12. File Repair/Synchronization

There are some situations where read-repair of individual byte ranges of files is insufficient and repair of entire files is necessary.

- To synchronize data on servers added to the end of a chain in a projection change. This case covers both adding a new, data-less server and re-adding a previous, data-full server back to the chain.
- To avoid data loss when changing the order of the chain's existing servers.

1. Projection P_E says that chain membership is $[S_a]$.
2. A write of bytes B to file F at offset O is successful.
3. An administration API request triggers projection P_{E+1} that expands chain membership is $[S_a, S_b]$. Al file repair/resyncronization process is scheduled to start sometime later.
4. FLU S_a crashes.
5. The chain manager on S_b notices S_a's crash, decides to create a new projection P_{E+2} where chain membership is $[S_b]$; S_b executes a couple rounds of Humming Consensus, adopts P_{E+2}, unwedges itself, and continues operation.
6. The bytes in B are definitely unavailable at the moment. If server S_a is never re-added to the chain, then B are lost forever.

Figure 4. An illustration of data loss due to careless handling of file repair/synchronization.

Both situations can cause data loss if handled incorrectly. If a violation of the Update Propagation Invariant (see end of Section A) is permitted, then the strong consistency guarantee of Chain Replication can be violated. Machi uses write-once registers, so the number of possible strong consistency violations is smaller than Chain Replication of mutable registers. However, even when using write-once registers, any client that witnesses a written → unwritten transition is a violation of strong consistency. Avoiding even this single bad scenario can be a bit tricky; see Figure 4 for a simple example.

12.1 Just "rsync" it!

A simple repair method might loosely be described as "just `rsync` all files to all servers in an infinite loop."[11] Unfortunately, such an informal method cannot tell you exactly when you are in danger of data loss and when data loss has actually happened. However, if we always maintain the Update Propagation Invariant, then we know exactly when data loss is imminent or has happened.

We intend to use Machi for multiple use cases, in both require strong consistency and eventual consistency environments. For a use case that implements a CORFU-like service, strong consistency is a non-negotiable requirement.

[11] The file format suggested in [5] does not actually permit `rsync` as-is to be sufficient. A variation of `rsync` would need to be aware of the data/metadata split within each file and only replicate the data section ... and the metadata would still need to be managed outside of `rsync`.

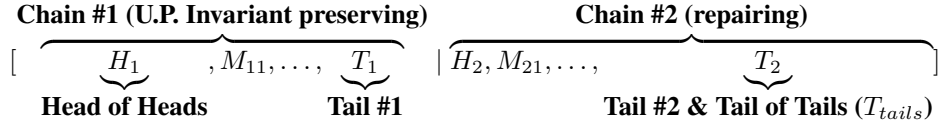

Figure 5. A general representation of a "chain of chains": a chain prefix of Update Propagation Invariant preserving FLUs ("Chain #1") with FLUs under repair ("Chain #2").

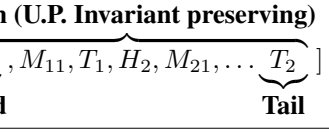

Figure 6. Representation of Figure 5 after all repairs have finished successfully and a new projection has been calculated.

Therefore, we will use the Update Propagation Invariant as the foundation for Machi's data loss prevention techniques.

12.2 Whole file repair as servers are (re-)added to a chain

Machi's repair process must preserve the Update Propagation Invariant. To avoid data races with data copying from "U.P. Invariant-preserving" servers (i.e. fully repaired with respect to the Update Propagation Invariant) to servers of unreliable/unknown state, a projection like the one shown in Figure 5 is used. In addition, the operations rules for data writes and reads must be observed in a projection of this type.

- The system maintains the distinction between "U.P. Invariant-preserving" and "repairing" FLUs at all times. This allows the system to track exactly which servers are known to preserve the Update Propagation Invariant and which servers do not.
- All "repairing" FLUs must be added only at the end of the chain-of-chains.
- All write operations must flow successfully through the chain-of-chains in order, i.e., from "head of heads" to the "tail of tails". This rule also includes any repair operations.
- All read operations that require strong consistency are directed to Tail #1, as usual.

While normal operations are performed by the cluster, a file synchronization process is initiated to repair any data missing in the tail servers. The sequence of steps differs depending on the AP or CP mode of the system.

12.2.1 Repair in CP mode

In cases where the cluster is operating in CP Mode, CORFU's repair method of "just copy it all" (from source FLU to repairing FLU) is correct, *except* for the small problem pointed out in Appendix B. The problem for Machi is one of time & space. Machi wishes to avoid transferring data that is already correct on the repairing nodes. If a Machi node is storing 170 TBytes of data, we really do not wish to use 170 TBytes of bandwidth to repair only 2 *MBytes* of truly-out-of-sync data.

However, it is *vitally important* that all repairing FLU data be clobbered/overwritten with exactly the same data as the Update Propagation Invariant preserving chain. If this rule is not strictly enforced, then fill operations can corrupt Machi file data. The algorithm proposed is:

1. Change the projection to a "chain of chains" configuration such as depicted in Figure 5.
2. For all files on all FLUs in all chains, extract the lists of written/unwritten byte ranges and their corresponding file data checksums. Send these lists to the tail of tails T_{tails}, which will collate all of the lists into a list of tuples such as $\{$`FName`, O_{start}, O_{end}, `CSum`, `FLU_List`$\}$ where `FLU_List` is the list of all FLUs in the entire chain of chains where the bytes at the location $\{$`FName`, $O_{start}, O_{end}\}$ are known to be written (as of the beginning of the current repair period).
3. For chain #1 members, i.e., the leftmost chain relative to Figure 5, repair all file byte ranges for any chain #1 members that are not members of the `FLU_List` set. This will repair any partial writes to chain #1 that were interrupted, e.g., by a client crash.
4. For all file byte ranges B in all files on all FLUs in all repairing chains where Tail #1's value is written, send repair data B & metadata to any repairing FLU if the value repairing FLU's value is unwritten or the checksum is not exactly equal to Tail #1's checksum.
5. For all file byte ranges B in all files on all FLUs in all repairing chains where Tail #1's value is unwritten $\perp$, force B on all repairing FLUs to also be $\perp$.[12]

When the repair is known to have copied all missing data successfully, then the chain can change state via a new projection that includes the repaired FLU(s) at the end of the U.P. Invariant preserving chain #1 in the same order in which they appeared in the chain-of-chains during repair. See Figure 6. This transition may progress one server at a time, moving the server formerly in role H_2 to the new role

[12] This may appear to be a violation of write-once register semantics, but in truth, we are fixing the results of partial write failures and therefore must be able to undo any partial write in this circumstance.

T_1 and adjusting all downstream chain members to "shift left" by one position.

The repair can be coordinated by the T_{tails} FLU or any other FLU or cluster member that has spare capacity to manage the process.

There is no race condition here between the enumeration steps and the repair steps. Why? Because the change in projection at step #1 will force any new data writes to adapt to a new projection. Consider the mutations that either happen before or after a projection change:

- For all mutations M_1 prior to the projection change, the enumeration steps #3 & #4 and #5 will always encounter mutation M_1. Any repair must write through the entire chain-of-chains and thus will preserve the Update Propagation Invariant when repair is finished.
- For all mutations M_2 starting during or after the projection change has finished, a new mutation M_2 may or may not be included in the enumeration steps #3 & #4 and #5. However, in the new projection, M_2 must be written to all chain of chains members, and such in-order writes will also preserve the Update Propagation Invariant and therefore is also be safe.

12.2.2 Repair in AP Mode

In cases the cluster is operating in AP Mode:

1. In general, follow the steps of the "CP Mode" sequence (above).
2. At step #3, instead of repairing only FLUs in Chain #1, AP mode will repair the byte range of any FLU that is not a member of the `FLU_List` set.
3. Do not use step #5; stop at step #4; under no circumstances go to step #6.

The end result is a big "merge" where any $\{$`FName`, $O_{start}, O_{end}\}$ range of bytes that is written on FLU S_w but unwritten from FLU S_u is written down the full chain of chains, skipping any FLUs where the data is known to be written and repairing all such S_u servers. Such writes will also preserve Update Propagation Invariant when repair is finished, even though AP Mode does not require strong consistency that the Update Propagation Invariant provides.

12.3 Whole-file repair when changing server ordering within a chain

This section has been cut — please see Git commit history of this document for discussion.

13. Additional sources for information about humming consensus

- "On Consensus and Humming in the IETF" [11], for background on the use of humming by IETF meeting participants during IETF meetings.
- "On 'Humming Consensus', an allegory" [8], for an allegory in homage to the style of Leslie Lamport's original Paxos paper.

14. Acknowledgements

We wish to thank everyone who has read and/or reviewed this document in its really terrible early drafts and have helped improve it immensely: Mark Allen, John Daily, Zeeshan Lakhani, Chris Meiklejohn, Jon Meredith, Mark Raugas, Justin Sheehy, Shunichi Shinohara, Andrew Stone, and Kota Uenishi.

References

[1] Abu-Libdeh, Hussam et al. Leveraging Sharding in the Design of Scalable Replication Protocols. Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC'13), 2013. `http://www.ymsir.com/papers/sharding-socc.pdf`

[2] Balakrishnan, Mahesh et al. CORFU: A Shared Log Design for Flash Clusters. Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12), 2012. `http://research.microsoft.com/pubs/157204/corfumain-final.pdf`

[3] Balakrishnan, Mahesh et al. CORFU: A Distributed Shared Log ACM Transactions on Computer Systems, Vol. 31, No. 4, Article 10, December 2013. `http://www.snookles.com/scottmp/corfu/corfu.a10-balakrishnan.pdf`

[4] Basho Japan KK. Machi Chain Self-Management Sketch `https://github.com/basho/machi/tree/master/doc/chain-self-management-sketch.org`

[5] Basho Japan KK. Machi: an immutable file store `https://github.com/basho/machi/tree/master/doc/high-level-machi.pdf`

[6] Calder, Brad et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11), 2011. `http://sigops.org/sosp/sosp11/current/2011-Cascais/printable/11-calder.pdf`

[7] Fritchie, Scott Lystig. Chain Replication in Theory and in Practice. Proceedings of the 9th ACM SIGPLAN Workshop on Erlang (Erlang'10), 2010. `http://www.snookles.com/scott/publications/erlang2010-slf.pdf`

[8] Fritchie, Scott Lystig. On Humming Consensus, an allegory. `http://www.snookles.com/slf-blog/2015/03/01/on-humming-consensus-an-allegory/`

[9] Seth Gilbert and Nancy Lynch. Brewers conjecture and the feasibility of consistent, available, partition-tolerant web services. SigAct News, June 2002. `http://webpages.cs.luc.edu/ pld/353/gilbert_lynch_brewer_proof.pdf`

[10] Naohiro Hayashibara et al. The accrual failure detector. Proceedings of the 23rd IEEE International Symposium on. IEEE, 2004. `https://dspace.jaist.ac.jp/dspace/bitstream/10119/4784/1/IS-RR-2004-010.pdf`

[11] Internet Engineering Task Force. RFC 7282: On Consensus and Humming in the IETF. https://tools.ietf.org/html/rfc7282

[12] Klophaus, Rusty. "Riak Core." ACM SIGPLAN Commercial Users of Functional Programming (CUFP'10), 2010. http://dl.acm.org/citation.cfm?id=1900176 and https://github.com/basho/riak_core

[13] Kreps, Jay. The Log: What every software engineer should know about real-time data's unifying abstraction http://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

[14] Kreps, Jay et al. Kafka: a distributed messaging system for log processing. NetDB11. http://research.microsoft.com/en-us/UM/people/srikanth/netdb11/netdb11papers/netdb11-final12.pdf

[15] Lamport, Leslie. The Part-Time Parliament. DEC technical report SRC-049, 1989. ftp://apotheca.hpl.hp.com/gatekeeper/pub/DEC/SRC/research-reports/SRC-049.pdf

[16] Lamport, Leslie. Paxos Made Simple. In SIGACT News #4, Dec, 2001. http://research.microsoft.com/users/lamport/pubs/paxos-simple.pdf

[17] Miranda, Alberto et al. Random Slicing: Efficient and Scalable Data Placement for Large-Scale Storage Systems. ACM Transactions on Storage, Vol. 10, No. 3, Article 9, July 2014. http://www.snookles.com/scottmp/corfu/random-slicing.a9-miranda.pdf

[18] Saito, Yasushi et al. Manageability, availability and performance in Porcupine: a highly scalable, cluster-based mail service. 7th ACM Symposium on Operating System Principles (SOSP99). http://homes.cs.washington.edu/%7Elevy/porcupine.pdf

[19] van Renesse, Robbert et al. Chain Replication for Supporting High Throughput and Availability. Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI'04) - Volume 6, 2004. http://www.cs.cornell.edu/home/rvr/papers/osdi04.pdf

[20] Wikipedia. Consensus ("computer science"). http://en.wikipedia.org/wiki/Consensus_(computer_science)#Problem_description

[21] Wikipedia. Route flapping. http://en.wikipedia.org/wiki/Route_flapping

A. Chain Replication: why is it correct?

See Section 3 of [19] for a proof of the correctness of Chain Replication. A short summary is provide here. Readers interested in good karma should read the entire paper.

A.1 The Update Propagation Invariant

"Update Propagation Invariant" is the original chain replication paper's name for the $H_i \succeq H_j$ property mentioned in Figure 7. This paper will use the same name. This property may also be referred to by its acronym, "UPI".

A.2 Chain Replication and strong consistency

The basic rules of Chain Replication and its strong consistency guarantee:

1. All replica servers are arranged in an ordered list C.
2. All mutations of a datum are performed upon each replica of C strictly in the order which they appear in C. A mutation is considered completely successful if the writes by all replicas are successful.
3. The head of the chain makes the determination of the order of all mutations to all members of the chain. If the head determines that some mutation M_i happened before another mutation M_j, then mutation M_i happens before M_j on all other members of the chain.[13]
4. All read-only operations are performed by the "tail" replica, i.e., the last replica in C.

The basis of the proof lies in a simple logical trick, which is to consider the history of all operations made to any server in the chain as a literal list of unique symbols, one for each mutation.

Each replica of a datum will have a mutation history list. We will call this history list H. For the i^{th} replica in the chain list C, we call H_i the mutation history list for the i^{th} replica.

Before the i^{th} replica in the chain list begins service, its mutation history H_i is empty, []. After this replica runs in a Chain Replication system for a while, its mutation history list grows to look something like $[M_0, M_1, M_2, ..., M_{m-1}]$ where m is the total number of mutations of the datum that this server has processed successfully.

Let's assume for a moment that all mutation operations have stopped. If the order of the chain was constant, and if all mutations are applied to each replica in the chain's order, then all replicas of a datum will have the exact same mutation history: $H_i = H_J$ for any two replicas i and j in the chain (i.e., $\forall i, j \in C, H_i = H_J$). That's a lovely property, but it is much more interesting to assume that the service is not stopped. Let's look next at a running system.

If the entire chain C is processing any number of concurrent mutations, then we can still understand C's behavior. Figure 7 shows us two replicas in chain C: replica R_i that's on the left/earlier side of the replica chain C than some other replica R_j. We know that i's position index in the chain is smaller than j's position index, so therefore $i < j$. The restrictions of Chain Replication make it true that length(H_i) $\geq$ length(H_j) because it's also that $H_i \supset H_j$, i.e, H_i on the left is always is a superset of H_j on the right.

When considering H_i and H_j as strictly ordered lists, we have $H_i \succeq H_j$, where the right side is always an exact prefix of the left side's list. This prefixing propery is exactly what

[13] While necesary for general Chain Replication, Machi does not need this property. Instead, the property is provided by Machi's sequencer and the write-once register of each byte in each file.

On left side of C		On right side of C
Looking at replica order in chain C:		
i	$<$	j
For example:		
0	$<$	2
It *must* be true: history lengths per replica:		
length(H_i)	$\geq$	length(H_j)
For example, a quiescent chain:		
length(H_i) = 48	$\geq$	length(H_j) = 48
For example, a chain being mutated:		
length(H_i) = 55	$\geq$	length(H_j) = 48
Example ordered mutation sets:		
$[M_0, M_1, \ldots, M_{46}, M_{47}, \ldots, M_{53}, M_{54}]$	$\supset$	$[M_0, M_1, \ldots, M_{46}, M_{47}]$
Therefore the right side is always an ordered subset of the left side. Furthermore, the ordered sets on both sides have the exact same order of those elements they have in common.		
The notation used by the Chain Replication paper is shown below:		
$[M_0, M_1, \ldots, M_{46}, M_{47}, \ldots, M_{53}, M_{54}]$	$\succeq$	$[M_0, M_1, \ldots, M_{46}, M_{47}]$

Figure 7. The "Update Propagation Invariant" as illustrated by Chain Replication protocol history.

1. Destroy all data on the repair destination FLU.
2. Add the repair destination FLU to the tail of the chain in a new projection P_{p+1}.
3. Change the active projection from P_p to P_{p+1}.
4. Let single item read repair fix all of the problems.

Figure 8. Simplest CORFU-style repair algorithm.

strong consistency requires. If a value is read from the tail of the chain, then no other chain member can have a prior/older value because their respective mutations histories cannot be shorter than the tail member's history.

B. Divergence from CORFU's repair

The original repair design for CORFU is simple and effective, mostly. See Figure 8 for a complete description of the algorithm Figure 9 for an example of a strong consistency violation that can follow.

A variation of the repair algorithm is presented in section 2.5 of a later CORFU paper [3]. However, the re-use a failed server is not discussed there, either: the example of a failed server S_6 uses a new server, S_8 to replace S_6. Furthermore, the repair process is described as:

> "Once S_6 is completely rebuilt on S_8 (by copying entries from S_7), the system moves to projection (C), where S_8 is now used to service all reads in the range $[40K, 80K)$."

The phrase "by copying entries" does not give enough detail to avoid the same data race as described in Figure 9. We believe that if "copying entries" means copying only written pages, then CORFU remains vulnerable. If "copying entries" also means "fill any unwritten pages prior to copying them", then perhaps the vulnerability is eliminated.[14].

1. Write value V to offset O in the log with chain $[S_a]$. This write is considered successful.
2. Change projection to configure chain as $[S_a, S_b]$. All values on FLU S_b are unwritten, $\perp$. We assume that S_b's unwritten values will be written by read-repair operations.
3. FLU server S_a crashes. The new projection defines the chain as $[S_b]$.
4. A client attempts to read offset O and finds $\perp$. This is a strong consistency violation: the value V should have been found.

Figure 9. An example scenario where the CORFU simplest repair algorithm can lead to a violation of strong consistency.

[14] SLF's note: Probably? This is my gut feeling right now. However, given that I've just convinced myself 100% that fill during any possibility of split brain is *not safe* in Machi, I'm not 100% certain anymore than this "easy" fix for CORFU is correct.