Coordinating Distributed System Configuration Changes with Humming Consensus

Scott Lystig Fritchie
Basho Japan
<scott@basho.com>

1. INTRODUCTION

Most of the database products developed by Basho Technologies are eventually consistent systems, such as Riak KV and Riak Time Series [4]. If Basho were to apply its expertise and experience in eventual consistency systems to a new file store product, then how would such a system be managed?

Configuration management of most distributed systems today is based upon strong consistency, using services such as ZooKeeper [11], etcd [6], OpenReplica [17], and Chubby [5]. These fault tolerant systems are built on strongly consistent protocols (e.g., ZAB [15], Raft [14], and Paxos [13]). These systems aren't useful in our case: an eventual consistency system that is managed by a strong consistency system *is constrained by the availability of its manager*. We want a manager that can operate correctly in failure scenarios that would paralyze a strongly consistent system. Riak Core [12] manages Riak KV's eventual consistency but has constraints that do not fit our use case.

Humming Consensus is a new algorithm for managing configuration metadata changes in eventual consistency systems. Service is available even when only a single participant is isolated by network partition; when the network recovers, it is re-integrated safely with its peers. Humming Consensus can also manage strong consistency systems with only modest adaptation. Though further research is required, the unification across consistency modes appears novel.

Humming Consensus is implemented in Machi [3], a dual consistency mode, distributed blob/file store based on Chain Replication [18]. The algorithm safely manages Machi's administrator-directed static membership, runtime replica group membership, strict chain ordering within the replica group, and anti-entropy file re-replication activities.

2. INSIGHT IN HINDSIGHT: DON'T RECORD FINAL CONSENSUS RESULT

Consensus protocols such as Paxos and Raft use their participants' persistent storage to remember the result of a change in the system's state. For example, a Paxos system might record that the result of ballot #2 is that tax on wine has been abolished. The proposal inside each ballot specifies the protocol's final decision.

Humming Consensus uses its participants' persistent storage only to signal intention to change configuration. Each participant is free to agree with the intention, ignore it, or to make an alternative suggestion. Each participant's agree/disagree decision uses the same code and assumes no Byzantine-style misbehavior.

Humming Consensus uses write-once registers to store change intentions. When a participant reads the configuration suggestion stored at logical time epoch E, the participant makes a decision (independent of its peers) either to use E's configuration or to reject it. If rejected, then the participant might continue to use its prior configuration while either doing nothing or suggesting an alternative new configuration. However, any new configuration suggestion must be written using a later epoch number, i.e., $E + i$ where $i > 0$.

3. BASIC COMPONENTS OF HUMMING CONSENSUS

Humming Consensus is embedded into the distributed system whose configuration the algorithm manages. Unless otherwise noted, the word "system" here refers to the managed system and not to Humming Consensus.

1. The system-specific configuration, whatever that is. For example, a list of replicas of a key-value store or a flat ASCII file.

2. The epoch number, a strictly increasing integer that represents logical time. The epoch number is incremented every time that any participant believes that the system's configuration may have changed.

3. The write-once epoch register store. This store is a simple client/server service, inspired by CORFU [1], that runs collocated with each participant that needs to modify the system's configuration. The API is shown in Figure 1. It is keyed by the epoch number; its value is an opaque system-specific configuration (bullet #1). All participants must be able to read and write all other participants' epoch register stores.

4. A transition safety function, used to judge the safety of changing configuration from C_{old} to C_{new}.

PaPOC'16, April 18-21 2016, London, United Kingdom

ISBN 978-1-4503-4296-4/16/04...$15.00

DOI: http://dx.doi.org/10.1145/2911151.2911165

```
-type Epoch = non_neg_integer().
-type Config = opaque().
-type ReadError = not-written | partition.
put(Epoch,Config) -> ok | is-written | partition.
get(Epoch) -> {ok,Config} | ReadError.
get_latest() -> {ok,Epoch,Config} | ReadError.
```

Figure 1: Write-once epoch register store API.

4. ENVIRONMENTAL ASSUMPTIONS

Humming Consensus assumes a failure model of fail recovery: a process may fail and recover a finite number of times. A process may reuse its identity/name after a failure.

Message corruption by the network can be tolerated for finite periods of time, though frequent corruption can deny the system liveness. Omission failures (i.e., message drops) are permitted, though they can cause false positive failure decisions by failure detectors and affect liveness. Message delivery reordering is permitted. Late-arriving replies are discarded (e.g., after a request timeout timer has expired).

Arbitrary Byzantine behavior is not permitted. Some types of Byzantine behavior may be detected and circumvented (e.g., by verifying checksums of the epoch register store's configurations). Other kinds of Byzantine failure cannot be tolerated, such as an infinitely-fast jerk who always writes an infinite number of invalid configurations to the epoch register store.

A safety property that Humming Consensus will eventually choose a safe transition from old $\rightarrow$ new configurations depends on no Byzantine behavior, a bug-free transition safety function (see Section 3), and mitigation for the "flapping" condition described in Section 6.1. Liveness is not guaranteed by Humming Consensus: an infinitely fast jerk can make liveness impossible.

Accurate failure detection is not required. False positive failure decisions can cause additional churn by the algorithm and thus can hurt liveness, but false positives do not cause safety violations if they occur only for a limited period of time. See Section 6.2 for how Machi's failure detection strategies have evolved.

Configurations stored in the epoch register store must be durable despite arbitrary process crashes and restarts. Participants can detect durable storage amnesia by maintaining a separate persistent store in a separate failure domain that stores the latest epoch number seen by Humming Consensus.

Humming Consensus does not require any communication between participants directly. All message passing is indirect, via reads and writes to the epoch register store collocated with each participant.

5. ALGORITHM OVERVIEW

Pseudo-code for an iteration of the Humming Consensus algorithm appears in Figure 2. The figure contains a simplification of the executable code in Machi's Humming Consensus implementation, the only one available today. Due to entanglement with Machi's implementation of Chain Replication and CR's state transition safety invariants, it remains future work to create a Humming Consensus implementation separate both from Machi and from Chain Replication. This author maintains a GitHub repository at [10] for errata for this paper and source code for future work.

```
var C_cur, C_max, C_new,
    C_repair, C_x       : configuration,
    C_all_latest        : set(configuration |
                              unwritten),
    E_cur, E_max        : positive_integer,
    P_perm, P_partition, P_all,
    P_rep, P_repair     : set(participant names),
    Flapping_History,
    Flapping_History'   : failure detector plus
                          flapping detector state
iterate_once() {
  {C_all_latest, P_partition} <-
        get_latest_projections(P_all)
  C_max <- C_x where
        C_x <- member_of(C_all_latest) &&
        E_max <- epoch(C_x) &&
        E_max == max(epoch(C_all_latest))
  {C_new, Flapping_History'} <-
        make_new_config(C_cur, C_max,
                        Flapping_History,
                        P_partition)
  Flapping_History <- Flapping_History'
  if unwritten_exists(E_max, C_all_latest) {
    C_repair <- element(C_all_latest) where
        epoch(C_repair) == E_max
    P_rep <- participants(C_repair)
    P_repair <- union(P_perm, P_rep)
    _ <- store(P_repair, E_max, C_repair)
  } else if unanimous(C_all_latest) &&
            E_max > E_cur &&
            is_safe_transition(C_cur, C_max) {
    C_cur <- C_max
    E_cur <- E_max
    P_all = union(P_perm, participants(C_max))
  } else {
    _ <- store(union(P_perm, participants(C_new)),
               E_max + 1, C_new)
  }
}
```

Figure 2: Pseudo-code for a single iteration of Humming Consensus.

Additional detail on Humming Consensus can be found in a conference presentation [9], design white paper [8], and Machi's source code at GitHub [3].

We start with C_{cur}, the current configuration in use during epoch E_{cur} by this participant. Section 2 describes the epoch register store, which represents intent by a participant to change system configuration. If a participant writes a configuration C_{max} to the register store at a later epoch, E_{max} where $E_{max} = E_{cur} + i, i > 0$, then each participant makes an independent choice of what to do with C_{max}.

- **Option 1:** Some values at the latest epoch are unwritten; "read repair" is required. Choose one of the values found at the latest epoch and write it to all valid participants. All store errors are ignored. If a store's value is already written, then it cannot be altered. If a store fails due to server or network failure, then we will detect the problem during a later iteration.
- **Option 2:** Accept the new configuration: the transi-

tion from C_{cur} to C_{max} is valid and safe, so we adopt the new configuration. In the common case, all other participants will make the same judgment and also adopt C_{max}.

- **Option 3:** Reject the configuration: the transition to the new configuration is not safe, or perhaps it's a bad idea. Machi determines "bad idea" with a numeric ranking of each configuration. For example, long chains with more members rank higher than short chains with fewer members. To adopt a configuration with lower rank, there must be a good reason, such as the chain is shorter (i.e., the dynamic replica group is smaller) because a participant is believed to have crashed. In this case, we attempt to write newly-calculated configuration to the epoch register stores at epoch $E_{max}+1$. Write failures don't matter: they will be detected by a later Humming Consensus iteration.

If a new configuration is rejected, then no action is required. For the sake of liveness, however, a better alternative ought to be written to a later epoch.

Read operations always query an epoch E at all register store replicas. If there had been a race to write different configurations for epoch E, then the dissent will be obvious to any reader. If dissent is discovered by a read at E, then that epoch is ignored.

The epoch register store API's `get_latest()` function fetches the latest configuration from a single store. The client calculates an observed maximum epoch E_{max} from all stores. The client ignores any configuration written to an earlier epoch $E_{max} - i$ where $i > 0$.

In an asymmetric network partition, a participant on the minority side can write a configuration C_{part} to epoch E_{part} that may be visible to some of the majority side participants. In turn, any majority side participant can perform read repair to create replicas of C_{part} that the minority author could not write directly. Subsequently, any participant on the majority side may read a sufficient number of copies of C_{part} to be forced to consider adopting or ignoring C_{part}. In this situation, the "bad idea" evaluation in Machi's ranking function is a good idea because the majority participants should reject C_{part}.

When in eventual consistency mode, the minimum number of writes and reads for a configuration is relaxed. Even in a worst-case network partition when all participants are isolated from all others, Humming Consensus groups of one are feasible. When this happens, there is nothing stopping multiple isolated groups from using the same epoch number like E_x. There is no conflict as long as the network remains partitioned: the conflict is invisible. When the partition is healed, future reads will discover different configuration values at E_x. Humming Consensus treats such discovery as simply a case of dissent within a single epoch. The participants are now aware that the network environment has changed. Any participant may suggest a new configuration at a later epoch to trigger the change needed to adapt to the new environment.

6. EXPERIENCE WITH HUMMING CONSENSUS

Humming Consensus is used today by Machi, a distributed blob/file store. A proof of safety is not yet available. However, property-based tests have confirmed its safety so far, running in hundreds of CPU hours inside of a simulator capable of true uni-directional network partitions. When run in eventual consistency mode, even a single isolated participant can create a chain of length one. After network partitions are healed, the algorithm safely coordinates merging the isolated chains (and the files they store) back into a single chain.

6.1 Flapping

The biggest problem observed with Humming Consensus is "flapping", a liveness problem where two (or more) participants have configuration transitions that are judged safe by themselves but are judged unsafe by another. It is like bickering children: I'm right, you're wrong, and nobody ever compromises.

Flapping is easy to detect: each participant suggests the same configuration many times in a row, and no new configuration is unanimously judged safe to transition to. Flapping is more likely to happen in eventual consistency mode after a network partition has been healed. One sub-chain's safe transitions during the partition may appear unsafe to another, formerly-isolated sub-chain.

One effective solution to flapping adopts a lower-than-ideally ranked configuration that all agree is a safe transition. The current implementation uses a simple but effective strategy: fall back to the safest and shortest possible chain. In eventual consistency mode, this is a single member chain. In strong consistency mode, this is the empty chain; crash recovery logic is re-used to discover the latest quorum majority's last agreed-upon chain configuration.

6.2 Failure detection as participant fitness

Initially, Machi used a very simple failure detector: one request/response failure (including timeout) to an epoch register store was considered a failure. Failure state was not maintained between algorithm iterations. In a symmetric network partition where all participants witness the same message dropping behavior, this simple strategy works well. In an asymmetric network partition, the simple strategy triggers excessive flapping (Section 6.1).

Machi uses Chain Replication to manage its file replicas. Chain Replication defines a single path for data mutation messages to travel. If that pathway is disrupted by an asymmetric network partition, then it is worth determining where exactly the network is failing. The SWIM protocol [7] uses multiple paths to try to detect asymmetric network partitions, but SWIM does not preserve information about where it discovers network problems.

Machi includes a small fitness service that uses a naive send-all-updates-to-everyone gossip protocol to disseminate suspected network partition information. The service uses the Riak-DT [2] library's CRDT map of last-writer-wins register values, keyed by participant name. Only participant X is permitted to update the map's key X.

For example, assume a set of three participants $\{A, B, C\}$ where an asymmetric partition drops messages only in the direction of $A \rightarrow B$. An iteration of Humming Consensus running on A will report a request/response problem when attempting to read and write from B's epoch register store. A will report the problem to its local fitness service, and the result is naively spammed to everyone. Also, B sees a problem with A and creates a similar report. All par-

ticipants have full connectivity to C, so all will eventually have a CRDT that converges to information that A had a problem with B, B had a problem with A, and C has no communication problems. The same digraph is constructed independently by each participant with the problem report information, and the chain is re-ordered by later iterations of Humming Consensus to route around network problem.

6.3 Changes for Strong Consistency Mode

Humming Consensus can also be used to manage strongly consistent systems. Only two modest changes are required.

First, the minimum number of successful configuration reads of all participants' register stores is raised to a quorum majority. If the network is suffering a partition (i.e., dropped messages for any reason, including participant stall or crash), then a participant in a minority partition cannot (by definition) read from a quorum majority of register stores. Participants that are trapped in minority partitions must wait until the network partition is healed.

Second, the API of the epoch register store is modified to accommodate two register stores: *public* and *private*. The store's key then becomes a 2-tuple of the store type and epoch number. The public register store plays the role as described in Section 5. The private register store is writable only by the local participant but readable by everyone. The private register store is written only when a participant has decided make a configuration state change: it copies the configuration at epoch E from the public register store to its private register store.

The private epoch register store is read by Humming Consensus to discover the latest epoch number E_{last} where all participants had agreed to adopt the configuration in the public store at E_{last}. Flapping can occur any time that failure detectors disagree on failed participants, so there can be many configurations in the public epoch register store without unanimous agreement that any one of them is safe and usable. Therefore the private epoch register store will be more sparsely populated than the public store.

7. CONCLUSION

Humming Consensus is a technique that manages the configuration metadata of Machi, a distributed blob/file store. All participant communication is indirect, via a simple write-once register service (i.e., the epoch register store). Items in the epoch register store signal intent to make a configuration change. The decision to proceed with a configuration transition is made independently by each participant, guided by *currently available* data in the epoch register stores. Humming Consensus manages Machi in both of its operating modes: eventually consistent mode and strongly consistent mode. Future work will focus on separating the technique from its sole implementation (embedded in Machi) and developing a formal safety proof.

8. ACKNOWLEDGMENTS

My thanks to everyone who helped improve this paper with their critique: the PaPoC review committee, Mark Allen, Russell Brown, Louise Lystig Fritchie, Susan Lee, Heather McKelvey, Jon Meredith, Bill Soudan, Steve Vinoski, and Matthew Von-Maszewski. "Humming Consensus" was inspired by the IETF's practice of vocal humming as an informal substitute for voting, as described in [16].

9. REFERENCES

[1] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis. CORFU: A shared log design for flash clusters. In *USENIX NSDI '12)*, pages 1–14, 2012.

[2] Basho Technologies, Inc. Convergent replicated datatypes (CRDTs) in Erlang. Source code repository at `https://github.com/basho/riak_dt`.

[3] Basho Technologies, Inc. Machi: A robust and reliable distributed blob store. Source code repository at `https://github.com/basho/machi`.

[4] Basho Technologies, Inc. Riak: A distributed, decentralized key-value store. Source code repository at `https://github.com/basho/riak`.

[5] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 335–350. USENIX Association, 2006.

[6] CoreOS, Inc. etcd: A highly-available key value store for shared configuration and service discovery. Source code repository at `https://github.com/coreos/etcd`.

[7] A. Das, I. Gupta, and A. Motivala. SWIM: Scalable weakly-consistent infection-style process group membership protocol. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 303–312. IEEE, 2002.

[8] S. L. Fritchie. Machi chain replication: Management theory and design. `https://github.com/basho/-machi/tree/master/doc/high-level-chain-mgr.pdf`.

[9] S. L. Fritchie. Managing chain replication with Humming Consensus. Presentation at RICON San Francisco 2015, `http://ricon.io/speakers/-slides/Scott_Fritchie_Ricon_2015.pdf`.

[10] S. L. Fritchie. Papoc paper errata and future development. Repository at `https://github.com/-slfritchie/humming-consensus`.

[11] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[12] R. Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 14. ACM, 2010.

[13] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[14] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[15] B. Reed and F. P. Junqueira. A simple totally ordered broadcast protocol. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, page 2. ACM, 2008.

[16] P. Resnick. IETF RFC 7282: On consensus and humming in the IETF. 2014.

[17] E. G. Sirer and D. Altınbüken. Commodifying replicated state machines with OpenReplica. 2012.

[18] R. Van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.