Chain Replication in Theory and in Practice

Scott Lystig Fritchie
Gemini Mobile Technologies, Inc.
slfritchie@snookles.com

Abstract

When implementing a distributed storage system, using an algorithm with a formal definition and proof is a wise idea. However, translating any algorithm into effective code can be difficult because the implementation must be both correct and fast.

This paper is a case study of the implementation of the chain replication protocol in a distributed key-value store called Hibari. In theory, the chain replication algorithm is quite simple and should be straightforward to implement correctly. In practice, however, there were many implementation details that had effects both profound and subtle. The Erlang community, as well as distributed systems implementors in general, can use the lessons learned with Hibari (specifically in areas of performance enhancements and failure detection) to avoid many dangers that lurk at the interface between theory and real-world computing.

Categories and Subject Descriptors H.2.4 [*Database Management*]: Systems—Distributed Databases; C.4 [*Performance of Systems*]: Reliability, availability, and serviceability

General Terms Algorithms, Design, Reliability, Theory

Keywords Chain replication, Erlang, Hibari, key-value store

1. Introduction

A data store, whether a key-value store or file system or other kind of database, may be distributed across two or more machines for any combination of the following reasons:

- Availability — It is unacceptable for data to be inaccessible or lost when a single machine fails.
- Performance — A single machine cannot service its intended workload within acceptable limits (e.g., minimum throughput or maximum latency limits).
- Capacity — A single machine cannot physically store the required amount of data (e.g., RAM capacity or disk capacity).
- Cost — A single machine may be meet all of the above goals but is too expensive to purchase and/or maintain. Both hardware and software (e.g., software license fees) are considered.

The challenge of building any distributed, mutable-state service is managing changes to replicated copies of state data in a predictable manner. A distributed systems architect might turn to a collection of formal specifications and proofs such as Lynch's *Distributed Algorithms* [13]. Unfortunately, translating a distributed algorithm into reliable running code is a subtle, poorly understood art.

Lamport's Paxos algorithm [12, 16] is now a well-known distributed algorithm for maintaining shared consensus, but as staff at Google [5] and Microsoft [11] have written, it is very difficult to preserve the algorithm's correctness and simultaneously reach performance goals. In the Erlang community, Arts et al. [2, 3] presented leader election algorithms, their implementations, the testing methods they used, and the flaws that they discovered long after the code was considered finished.

This paper is a case study of the implementation of the chain replication protocol in a distributed key-value store called Hibari[1]. In theory, the chain replication algorithm is simpler than the Paxos algorithm. In practice, however, there are plenty of implementation details that have hampered creating a product that is both correct and sufficiently fast. The experience presented here can help others in the Erlang community and, more broadly, all distributed systems developers to create robust distributed systems that actually work correctly.

An outline of this paper's topics is as follows:

- Summaries of the chain replication technique and of an Erlang application, Hibari, that uses chain replication for replica management.
- Practical problems caused by disk latency and the need for rate control.
- Status monitoring, including how Erlang messaging infrastructure can hide network failures.
- Hibari's implementation of consistent hashing and replica placement strategies.
- Observations about Hibari that don't merit their own sections.
- Related work and concluding remarks.

2. Chain Replication

The chain replication algorithm is described by van Renesse and Schneider [24]. The paper specifies a variation of master/slave replication where all servers that are responsible for storing a replica of an object are arranged in a strictly-ordered chain. The head of the chain makes all decisions about updates to an object. The head's decision is propagated down the chain in strict order. See Figure 1 for a diagram of message flows.

The number of replicas for an object is determined by the length of the replica chain that is responsible for that object. To tolerate f replica server failures, a chain must be at least $f + 1$ servers long.

[1] In Japanese, "hibari" means "meadowlark." The two Kanji characters used for "hibari", 雲雀, literally mean "cloud sparrow."

Erlang'10, September 30, 2010, Baltimore, Maryland, USA.

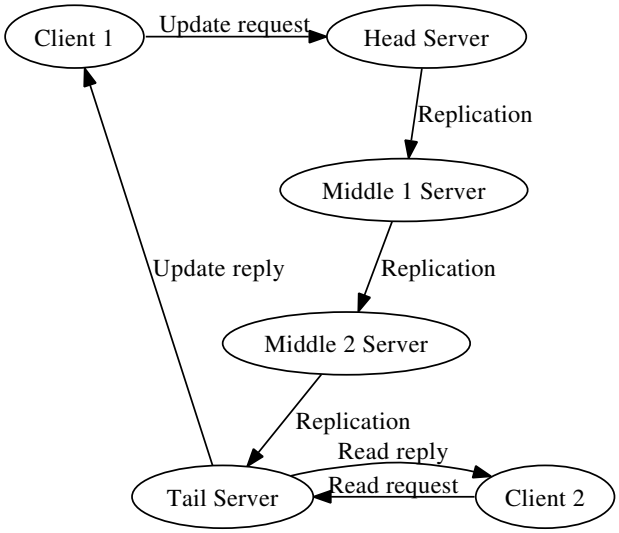

Figure 1. Write and read operations upon a chain of length four.

Operations on objects within the chain are linearizable (chapter 13 of [13]) when all updates are processed by the head of the chain and all read-only queries are processed by the tail of the chain. Strong consistency[2] is maintained because read requests are processed only by the tail of the chain.

When a chain member fails, the following steps are taken to repair the chain.

1. The chain is shortened to remove the failed brick (call it B) from service, and un-acked updates are re-sent down the chain. All clients are notified about the new chain configuration.
2. When brick B restarts, it is added to the end of the chain, and all out-of-date keys are copied to B. Meanwhile, brick B is ignored by all clients. If B receives a client request by mistake, the request is ignored. (See also section 12.3.)
3. When the key copying phase is complete, the chain is reconfigured to make B a full member of the chain in the tail role, and all clients are notified about the new chain configuration.

See [24] for a full description of procedures necessary to recover from chain member failure.

Client workloads with extremely large read/write ratios can potentially imbalance individual server workloads: 100% of read operations are sent to the same server, the tail of the chain. The chain replication implementation in CRAQ [22] allows read operations to be handled by other servers in the chain without violating strong consistency. Hibari does not use CRAQ's optimizations but instead uses data placement policies to balance server workloads; see section 11 for more detail.

3. Hibari Overview

Hibari is a distributed, fault tolerant, high availability key-value store written in Erlang. Through use of chain replication (section 2), all operations by Hibari clients read strongly consistent updates. Hibari is one of the few distributed key-value stores that can atomically update multiple keys in a single client operation (section 9). By default, all updates are persistent: each server flushes all updates to local stable storage before replying to a client.

[2] Read operations can only return an object's last update.

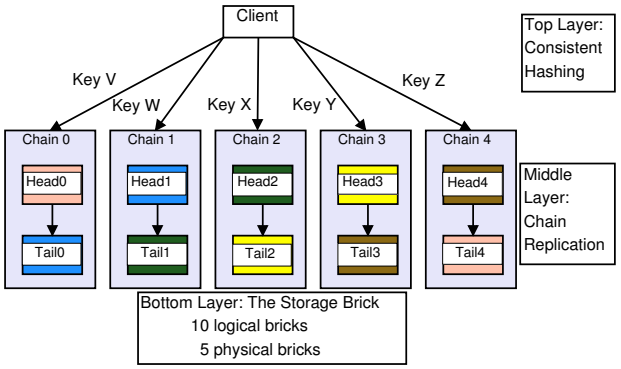

Figure 2. Hibari logical architecture: consistent hashing, chain replication, and basic storage.

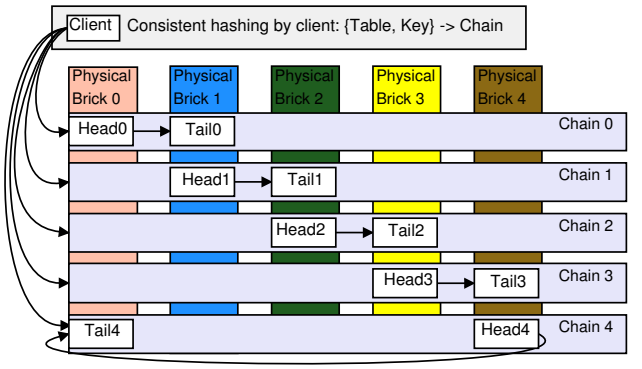

Figure 3. Hibari architecture: an alternate view of Figure 2 with each physical brick represented in a vertical column.

Hibari faithfully implements the chain replication algorithm as described in [24]. Erlang's messaging model makes it trivial to support the original algorithm's asymmetric message passing for updates. Local data logging, inter-server messaging, and chain replica repair (after the failure of a server) are implemented as described.

Hibari's performance is typically quite good, individually and as a cluster. On typical commodity 1-2U rack-mountable server hardware, Hibari can sustain a throughput of several thousand updates of 1 kilobyte values per second per server. Clients see throughput increase linearly as servers are added to the cluster.[3]

High availability is achieved by using replication chains longer than one server and by reacting quickly (e.g., within a few seconds) when a server fails. Availability is further assisted by distributing keys across the cluster using consistent hashing: when a chain has failed, all keys managed by other chains are unaffected by the failure. Also, machines may be added to or removed from Hibari clusters without interrupting service.

3.1 Physical and Logical Bricks

A brick is a server that stores persistent data. Figures 2 and 3 portray a cluster of five physical bricks (mapping one-to-one onto physical machines, typically Linux-based) with five replication chains. Each chain is of length two; the chain's data is available for reads and writes as long as the total number of failures within the chain is less than two.

[3] Assuming that chains are evenly balanced across physical machines; see also section 11.

```
CREATE TABLE foo (
  BLOB key;
  BLOB value;
  INTEGER timestamp;
  INTEGER expiration_time;        -- 0 = no expiry
  ERLANG_PROPERTY_LIST proplist;  -- Usually empty
) UNIQUE PRIMARY KEY key;
```

Figure 4. SQL-like representation of a Hibari table.

As far as Hibari Admin Server (see section 7) is concerned, data is stored by a "logical brick." The five physical machines, called "physical bricks," in Figure 2 are each configured with two logical bricks. The logical bricks are striped across the physical bricks so that each physical brick contains one logical brick in a "head" role and one logical brick in a "tail" role. Using consistent hashing (section 3.3) and techniques described in section 11, each physical brick's CPU, RAM, and disk workloads will be in balance under most conditions.

The Admin Server monitors the status of each logical brick. If a physical brick crashes, then clearly all logical bricks on that machine will fail. The reason for a physical brick's crash is not usually important: any hardware failure, such as power supply or disk volume, that disrupts a logical brick is sufficient to trigger reaction by the Admin Server.

If deployed on virtualized hardware, "physical brick" could mean either the physical hardware or the virtualized hardware. Virtualized hardware complicates management of the actual physical locations of logical bricks. Hibari's current code base does not make any attempt to enforce replication on distinct physical bricks. During development, it is useful to test clusters of hundreds or thousands of logical bricks on a single physical machine (with or without hardware virtualization). But right now, it is ultimately a human administrator's responsibility to ensure the physical diversity of each logical brick within a chain.

3.2 A Client's View of a Hibari Cluster

Each key-value pair is stored in a Hibari table. Tables were first implemented to provide separate key namespaces, that is, to permit storing the key `"foo"` multiple times, each in a different table. Later, tables became a convenient administration tool for configuring behaviors such as consistent hashing (section 3.3) and key migration (section 10).

Data in a Hibari table is stored in one or more chains. Each chain stores data for only one Hibari table. Each logical brick stores data for only one Hibari chain. Figure 3 depicts a typical layout for chains for a single Hibari table; see section 11 for discussion of replica placement strategies.

Each key in a Hibari cluster has the attributes depicted in pseudo-SQL in Figure 4; technically, each key stored by Hibari is actually part of a 5-tuple. Going forward, the more familiar term key-value pair will be used instead, and the other attributes will be mentioned only when the context requires it.

Each Hibari client receives status updates from the Hibari Admin Server that contain server status updates. Using the mapping data within each status update, each client knows the head and tail bricks for all chains in all tables within the cluster. Clients usually send their requests directly to the correct brick and do not incur intra-cluster query forwarding penalty, except during cases of key migration (also called key repartitioning, see section 10).

All attributes in Figure 4 except `value` are always stored in RAM by the logical brick. The `value` attribute may be stored on disk or in RAM as a per-table configuration option. As a result, Hibari logical bricks can consume a lot of RAM, proportional to the number of keys stored in the brick and (for RAM-based value blob storage) the sum of all value blob sizes.

3.3 Consistent Hashing

Hibari uses a consistent hashing technique [10] to map a $\{T, K\}$ tuple to the name of the chain that is responsible for storing that key K in table T. The key K, or configurable prefix of K, for example, the first 4 bytes, or all bytes between the first two ASCII '/' (slash) characters, is hashed using the MD5 algorithm[4] and mapped onto the unit interval $[0.0, 1.0)$. The unit interval is divided into an arbitrary number of ranges, where each range represents a chain name. Each chain can appear one or more times in the range map. The top third of Figure 5 depicts a range mapping of two chains onto the unit interval; each of the two chains has an identical chain weighting factor. (The bottom two-thirds of Figure 5 is discussed in section 10.)

The relative size of each range is determined by the chain weighting factor. Assume a hypothetical chain mapping where the chain weighting factor for chain C_1 is twice as large as the weighting factor for chain C_9. The sum of the size of range interval(s) found in the range map will be twice as large for chain C_1 as the sum for chain C_9. Smaller weighting factors can be used to bias distribution of keys away from some chains (and therefore away from some physical bricks/machines) that have lower capacity (e.g., slower CPU, less RAM, or smaller disk capacity).

Two hash mappings are used to implement key migration (or key repartitioning). In normal operation, the maps are the same. During key migration, the maps are used to calculate the current and new/desired location of a key. Key migration is a dynamic, online process that can expand or shrink a cluster as well as to change the relative chain weighting factors.

3.4 Single Data Center

Hibari was designed to provide strong consistency within a single data center. All deployments to date are in a single data center. A Hibari chain can have members in multiple data centers, but there are several practical complications in such a scheme:

- By definition, each update operation must traverse the entire replication chain. If chain members are in different data centers, each update operation will pay a penalty of the sum of the wide-area network latencies of all network links between the data centers.
- A client application in data center D_1 that attempts to read a key stored by a tail brick in data center D_2 must pay the penalty of the wide-area network latency between D_1 and D_2.
- The Hibari Admin Server is not currently designed to run simultaneously in multiple data centers.

4. Problems with Disk Write I/O Latency

All distributed systems architects have to face the tough facts of economic reality: if a system costs too much, then it won't be built. Deployment on cheap-enough hardware can also mean deployment on not-fast-enough hardware. The pressure of meeting performance goals can make cutting algorithmic corners very tempting. Architects must never forget that any change to a distributed algorithm, no matter how small or innocent the change seems, may in fact invalidate the algorithm.

For the purposes of this paper, "big data" means that the total amount of data and metadata stored (including all replicas) is larger

[4] MD5 was chosen for convenience and relatively low computation cost. Neither the larger output range or collision resistance of more recent cryptographic checksum algorithms is necessary.

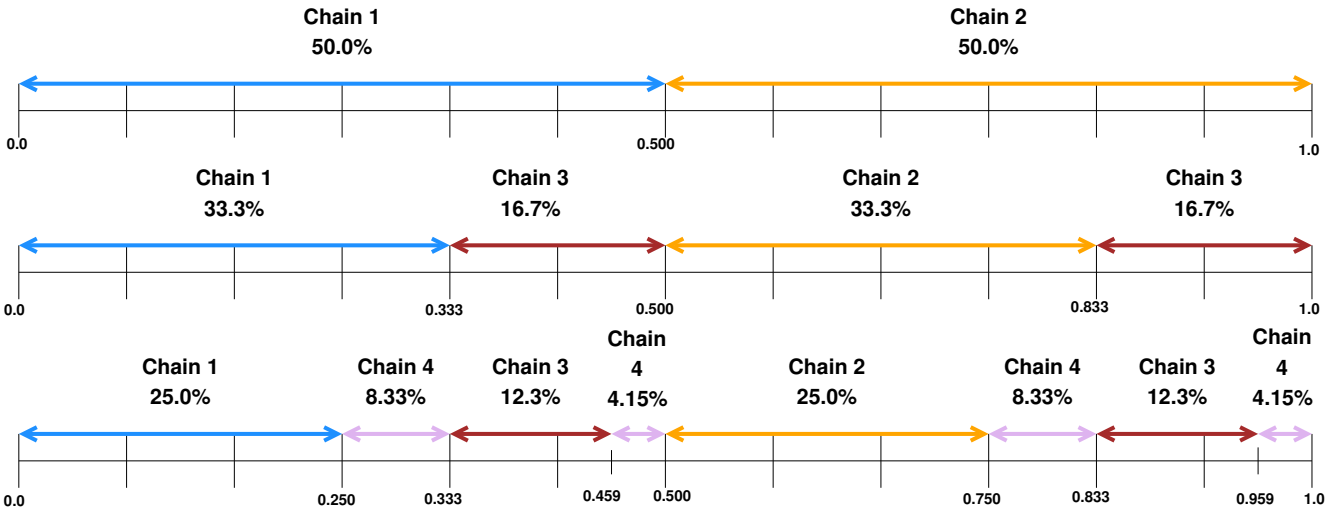

Figure 5. Consistent hashing map after two migrations: 2 chains, then 3 chains, then 4 chains. All chains have equal weighting factors.

than the sum of RAM in the cluster. The price of RAM-based and flash RAM-based storage is too high for many big data applications. Only traditional hard disks are cost-effective enough to support free or advertising-supported applications such as email services. Free email services now routinely provide email users with storage quotas in the range of 1-25 gigabytes per user, with "unlimited" storage available at extremely low monthly fees.

Rotating disk media can provide total storage capacity at a low-enough cost for most big data applications, but their average random I/O operation latencies are quite high: at least 4 milliseconds for top-of-the-line, 15K RPM SCSI disks and at least double that amount for slower, cheaper disks. As a result, it is in any application's best interest to minimize the number of random I/O operations that it generates.

The write-ahead log technique has been used by database systems for decades to aggregate disk write operations, minimizing random disk I/O operations by appending log entries to a log file. Another common technique is group commit, which flushes many transactions' worth of log entries to stable storage with a single `fsync(2)` system call (or OS equivalent).

Hibari uses both techniques, a write-ahead log and group commit, to minimize random disk I/O required to store reliably all updates received by a brick. The latency penalty of the `fsync(2)` system call is required to avoid data loss in the event of a cluster-wide catastrophe, such as a data center power failure.

Systems calls, such as `write(2)` and `fsync(2)`, that operate on local file systems can block for tens of milliseconds (or more) per call. Such blocking delay is unacceptable in almost any latency-sensitive application. To solve this problem, each `gen_server`-based Hibari logical brick sends all write and sync requests to a central write-ahead log (WAL) process (shared by all logical bricks on that Erlang node) to request `write(2)` and `fsync(2)` system call execution. The `gen_server` process is then free from blocking to work on other tasks while the I/O operations are pending. The WAL process sends messages back to the brick when the I/O calls have finished.

The chain replication protocol already requires that each write operation have a serial number associated with it and that each update propagates down the chain in serial number order. The WAL process uses these serial numbers to signal to each brick the largest serial number that has been safely flushed to disk. Each brick can then send those updates to downstream bricks at its leisure.

Unfortunately, Hibari's early implementations of the communication between brick and WAL server processes were fraught with subtle, difficult-to-find race conditions: writes were written to the local WAL out of order, `fsync(2)` operations were acknowledged with wrong log serial numbers, and bricks sent log replay messages downstream in incorrect order. The QuickCheck software testing tool [17] has been invaluable for helping create the conditions necessary to exploit the very small windows of vulnerability of many of these bugs.

Many of the bugs above have caused data loss. Most would never have been created if the code could ignore the reality of dealing with slow disk devices. Few data buffering hacks go unpunished.[5]

5. Problems with Disk Read I/O Latency

Each Hibari data server maintains key and key metadata in RAM but stores value blobs either in RAM or on disk. For big data purposes, Hibari must store value blobs on disk. As a consequence, any client `'get'` or `'get_many'` operation may trigger disk I/O. If the available cache is too small, and/or if the client application's access pattern doesn't provide sufficient temporal locality of reference, then disk read I/O operations are inevitable.

Hibari tries to minimize the number of disk operations required to read a key's value blob by always storing in RAM the value blob's exact storage location: write-ahead log file number, starting byte offset, and blob size. The brick can read any value blob with a single pair of `lseek(2)` and `read(2)` system calls.

Hibari's `'get'` and `'get_many'` operations are not the only source of disk read I/O workload. Two more significant sources are the following.

1. Chain repair — Chain repair can generate a huge amount of disk read I/O. For each value blob that a brick under repair does not have, the upstream brick must read the blob from disk before sending it downstream to the repairee brick.
2. Key migration — Sometimes called key repartitioning, resharding, or rebalancing, the Hibari key migration process moves keys (and associated values and metadata) from one chain to another (see section 10). Any key that must be moved during a

[5] Borrowed very loosely from Patrik Nyblom.

key migration must have its value read from disk by a brick in the source chain before it is copied to the destination chain.

5.1 Read Priming

As described in section 4, many system calls involving the local file system can block the caller. Any file `open(2)` or `read(2)` system call has the potential to block a Hibari logical brick process for many milliseconds each. On extremely overloaded systems, each call can easily take over 100 milliseconds, which in turn can have enormous negative effects on latency-sensitive applications.

To avoid blocking brick processes with read-only disk I/O, Hibari borrows techniques used by the Squid HTTP proxy [19] and Flash HTTP servers [15]. Before a brick attempts to open or read a file, it first spawns a "primer" process that asynchronously opens the file and reads the desired data. This process acts like adding water to a pump to "prime" the pump: all necessary file metadata and data is read into the OS page cache. The primer process uses the standard Erlang `file` API to do its work. When finished, the primer process sends a message to the logical brick process that the priming action is complete. Then the brick process can read the blob (using the same API) with very little probability of blocking.

This priming technique has the disadvantage of reading the same data twice: once by the short-lived primer process and once by the long-lived brick process. However, even with value blobs up to 16 megabytes in size, the overhead isn't big enough to worry about. The major advantages are that the Erlang `file` module already supports all operations that the primer process requires, and the probability of blocking the brick process is reduced to practically zero. The reduction of average read latency significantly outweighs the disadvantages.

5.2 Access by Lexicographic vs. Temporal Orders

For both chain repair and key migration workloads, the primer technique only hides a portion of the latency required to read large numbers of value blobs from disk. Both workloads generate I/O based on the lexicographic sort order of the keys. However, the value blobs are stored on disk in temporal order, that is, relative to the time when they were received.

The mismatch between lexicographic and temporal orderings can create a significant amount of random I/O workload, as far as the underlying disks are concerned. For key migration workloads, the I/O cost is largely unavoidable. Brick repair times can take a few seconds or several days, depending on several factors.

- If the brick under repair was down for only a short time, the total number of keys that require repair is likely to be small, and their value blobs are likely to be in the OS page cache.
- For a brick that is completely empty (e.g., a new machine with a new, empty file system), a manual function is provided that transmits keys and value blobs in an order that is sorted by each value blob's location within the write-ahead log. The sorting can help reduce the amount of random pattern read disk I/O required to read a large number of value blobs. The savings can be very significant when the total size of value blobs is in the range of hundreds or thousands of gigabytes.
- For repair tasks that fall in the middle, the number of keys to repair is high, but the cost of starting repair completely from scratch is even higher. In this middle case, there is no choice other than wait for the standard repair technique to finish and to accept the amount of random read disk I/O required to do it. For a chain that contains a terabyte of data or more, the time required to finish chain repair can be minutes (best case), hours, or even days (worst case). System planners and operations staff must keep this in mind as they plan their data redundancy strategy, that is, how big should each brick be and how long should each chain should be.

6. Rate Control

Modern hard disks are orders of magnitude slower than other components in the system: CPU, RAM, system buses, and even commercial gigabit Ethernet interfaces and switches are all less likely to be the slowest system component. To avoid overloading disk subsystems even further, rate control mechanisms are necessary to control anything that can generate disk I/O.

Hibari has explicit controls for both batch sizes (e.g., number of keys per iteration of an algorithm loop) and bandwidth (e.g., total number of bytes/second) for the following.

1. Chain repair operations, key migration operations (see section 5), and number of primer processes for prefetching value blobs from disk.
2. Log "scavenging" operations, which reclaim space from Hibari's otherwise infinite-sized write-ahead log. The scavenger's activity can create a large amount of extra disk I/O, more than enough affect clients by increasing latency. See the Hibari Systems Administrator's Guide at [9] for a detailed description.

During key migration periods, it is possible for a client's request to be forwarded back and forth between a key's old chain location and its new chain location. This forwarding loop is usually quickly broken once the key has been stably written to the new chain. If a forwarding loop is detected (using a simple hop counter), an exponential delay is added at each forwarding hop to try to avoid overloading bricks in either chain. Also, the loop will be broken if the total number of hops exceeds a configurable number.

Hibari also has an implicit limit on the number of simultaneous client operations that a single brick can support. The simple technique is borrowed from SEDA [26]: if the client request is too old, then drop the request silently. Sending a reply to the client will create even more work for an overloaded server to do, so the cheapest thing to do is to do nothing. Each Hibari client request contains a wall clock timestamp. If that timestamp is too far in the past, the Hibari brick will ignore the request, assuming that the request waited in the brick's Erlang mailbox for so long that the brick must be overloaded.

To help synchronize system clocks, it is strongly recommended that all Hibari machines, servers and clients, run the NTP (Network Time Protocol) service. Synchronization down to the femtosecond is not necessary; all clocks within even 100 milliseconds is good enough. However, deployment on virtual machines, such as Xen or VMware, should be avoided unless the guest OS's clock can reliably match the host OS's clock (which is assumed to be stable).

Unsynchronized guest OS clocks can cause bricks to drop client requests silently, via the mechanism described two paragraphs earlier. The silent drops cause client-side timeouts that can be very confusing, unless you happen to be looking at "operation too old" counter statistics. That counter should only ever increment during periods of server overload; any other time is a near-certain symptom of unsynchronized OS clocks.

7. Cluster Management and Monitoring

The original chain replication paper [24] describes a single master server that is responsible for managing chain state and monitoring the status of each server within each chain. To be accepted commercially, however, single points of failure must be avoided or minimized to the greatest extent possible.

Hibari implements the single master entity as a single Erlang/OTP application that is managed by the Erlang kernel's "application controller." The application controller coordinates multiple

Erlang nodes to run the management/monitoring application, the Admin Server, in an active/standby manner. This indeed creates a single point of failure: if the machine running the active Admin Server instance crashes, the Admin Server's services are lost.

Failure of the Admin Server is not usually a significant problem. The Admin Server is required only when bricks crash or restart within the cluster, or if an administrator wishes to query cluster status or to reconfigure the cluster. Without the Admin Server, Hibari client nodes may continue operation without error, as long as other bricks also fail while the Admin Server is down.

The Admin Server requires approximately 10 seconds to restart. The Admin Server's private state (including histories of the down/-up status of each logical brick and chain) is also distributed across bricks within the cluster. If the Admin Server's private state were managed with chain replication, then there would be a "chicken and the egg" problem when the Admin Server bootstraps itself. To avoid a circular dependency, the private state is replicated using quorum voting-style replication.

7.1 Detecting Brick Failure and Network Partition

The original chain replication paper [24] makes two assertions that are extremely problematic in the real world. The first is "Servers are assumed to be fail-stop." The second is "A server's halted state can be detected by the environment." If either assumption is violated, the system can quickly make bad decisions that can cause data loss.

The biggest problem with detecting halted nodes is the problem of network partition. Partitions are usually caused by failure of network equipment, such as an Ethernet switch or failure of network links such as a telecom data circuit. However, any failure of hardware and/or software that creates arbitrary message loss can be considered a network partition.

With message passing alone, it's impossible to tell the difference between a network partition, a failed node, or merely a very slow node. The built-in Erlang/OTP message passing and network distribution mechanisms cannot adequately handle network partition events by themselves.

To fix the worst problems caused by network partition, Hibari includes an OTP application called "partition_detector." Running on all Hibari servers, the sole task of this application is to monitor two physical networks, the 'A' and 'B' networks, for possible partition. All Erlang network distribution traffic is assumed to use network 'A' only. UDP broadcast packets are sent periodically on both networks. When broadcasts by an Erlang node are detected on network 'B' but have stopped on network 'A', then a partition of network 'A' may be in progress.

The Erlang/OTP application controller can still make faulty decisions when a network partition happens; the application controller does not interact with partition_detector application. However, after the application controller restarts an Admin Server instance, the partition_detector application can abort the initialization of that instance when it believes there is a partition in effect, raise an alarm, and leave the Admin Server processes in an idle state. This situation must then be resolved by a human administrator.

7.2 Fail Stop Means . . . Stop?

Violation of the "fail stop" assumption have also caused problems for Hibari. Hibari's sponsor, Gemini Mobile Technologies, is not responsible for day-to-day operations and monitoring of its customer's systems, so all we know and theorize comes from after-the-fact analysis of failures in customer lab or production systems. In these post mortem analyses, we identified two significant problems:

1. A bug within the Erlang/OTP `'net_kernel'` process that can cause deadlock and thus cause communication failures between Erlang nodes. One instance of this bug hit a customer's system on at least 10 different machines within a 30 minute interval, including both nodes that managed the Admin Server's active/standby fail-over.
2. System `'busy_dist_port'` events can trigger interference in process scheduling and extremely high inter-node messaging latencies. All Erlang messaging traffic to a remote Erlang node is sent through a single Erlang port which represents a TCP connection. If the sending node detects congestion (e.g., a slow receiver, intermittent network failure), then any Erlang process on the local node that attempts to send a message to the remote node will be blocked: the Erlang process is removed from the scheduler and will remain unschedulable until the distribution port is no longer congested.

The combination of `'net_kernel'` deadlock, wild variations in message passing latency, and process de-scheduling can create a situation that is difficult to diagnose. If a brick is merely slow to respond to status queries by the Admin Server, Arpaci-Dusseau et al. suggest calling it "fail stutter" [1]. But if the brick responds too slowly, the Admin Server may interpret a performance problem as a failure instead.

One such problem, affecting both the Admin Server node and many others within a cluster, caused Hibari's largest deployment to suffer from multi-hour transient availability failures. If a brick does not respond to a status query by the Admin Server, it is considered failed and removed from the chain. A few seconds later, the brick would catch up and answer new queries. The Admin Server would force the brick to crash, triggering automatic repair and eventual rejoining the chain. If the situation is bad enough, the chain can be (and has indeed been) whittled down to zero bricks.

One solution to this problem has been a small patch to the Erlang virtual machine to make the buffer size for inter-node network distribution ports configurable. The default size of the `erts_dist_busy` constant is 128 kilobytes. However, even a value of 4 megabytes appears to be too small for the amount of messaging data that Hibari bricks send during bursty traffic patterns.

Another solution uses information from Hibari's partition_-detector application to supplement the monitoring info that the Admin Server uses. If a system monitor `'nodedown'` message is received, the partition detector's state is queried to check if a network partition was a possible cause of the message. The same is done if a query of a remote brick's general health status fails due to timeout or `'nodedown'` reasons.

In hindsight, the single Admin Server process has had more problems in production than we had anticipated. The solutions outlined above have not been in use long enough to judge their effectiveness. However, given the problems that we know have happened in production networks, it is likely that a distributed manager application would likely have been fooled by the same conditions and made similarly bad decisions.

7.3 The Admin Server as a Single Entity

The single running Admin Server instance has a convenient consequence: behavior during network partition events is easy to reason about. An administrator knows where the Admin Server might run: all eligible nodes are configured statically, so there are (typically) only two or three machines where the Admin Server may be running. Furthermore, using Figure 6 as an example:

- If an entire chain is on the same side of a partition as the Admin Server, then Chain 1 is healthy and usable by Client 1. Client 2 is the far side of the partition and therefore cannot access Chain 1.

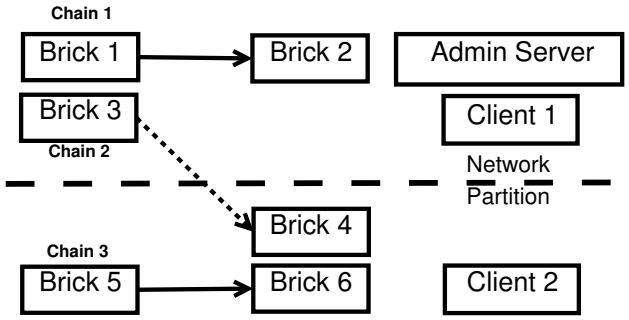

Figure 6. A network partition scenario.

- If the entire chain is on the far side of the partition relative to Admin Server, then Chain 3 is healthy and usable by Client 2. Client 1 is the far side of the partition and therefore cannot access Chain 3.
- For chains that are split by the partition, the bricks on the same side of the partition as the Admin Server will be reconfigured into a new chain. In Figure 6, Chain 2 will be reconfigured to a chain of length one that contains only Brick 3. The new Chain 2 is accessible by Client 1 but not by Client 2.

The CRAQ paper [22] proposed a distributed chain monitoring and management scheme. Hibari's Admin Server pre-dates the CRAQ paper and therefore couldn't take advantage of its suggestions.

8. Erlang Messaging Is Not Reliable

The original chain replication paper [24] says, "Assume reliable FIFO links." There is no such thing in the real world. Erlang's messaging model is frequently described anecdotally as "send and pray."

The original chain replication paper says that the tail of the chain, after processing an update successfully, acknowledges the update back "upstream" all the way to the head. These acknowledgments are vital for chain repair purposes. The Hibari implementation avoids the cost of an acknowledgement per update (which can be as high as several thousand updates per second per chain) by using a once per second acknowledgment that contains the largest update serial number that has been processed by the tail.

The once per second optimization is fine if all communication links are indeed reliable and FIFO. However, Erlang's communications links are not reliable.

Svensson and Fredlund describe in [21] under what conditions the usually-reliable messaging between two Erlang nodes can turn unreliable. It is possible for a brick to send three updates downstream, with serial numbers S_1, S_2, and S_3, respectively. It is possible for the Erlang network distribution mechanism to deliver the messages containing S_1 and S_3 and drop the message containing S_2. In this situation, a Hibari tail brick might acknowledge serial S_3 as the last processed serial number, and the head brick will mistakenly assume that the S_2 update has been processed by all members of the chain.

Fortunately, the Erlang process monitoring BIF `monitor()` will deliver a {`'DOWN',...`} message to the receiver when the connection between nodes has been broken and may have dropped messages. Hibari now uses this mechanism to detect dropped messages. If a {`'DOWN',...`} message is received, then subsequent replication log replay messages must pass a series of strict sanity checks. If the checks fail, then the receiving brick will crash itself.

Sample key	Data stored in value blob
`/42/1`	Text of post #1
`/42/1/1`	Text of comment #1 on post #1
`/42/1/2`	Text of comment #2 on post #1
`/42/2`	Text of post #2
`/42/summary`	Next post number, number of active posts, number of deleted posts, ...

Figure 7. A sample Web blogging application's `posts` table.

It is imperative that all bricks replay all replication log messages in exactly the same serial order. By crashing immediately, the brick that first notices a dropped message will avoid propagating out-of-order messages downstream. Hibari's automatic chain repair will compensate for the lost message.

9. Micro-Transactions

Though not mentioned in the original chain replication paper [24], [22] mentions the possibility of implementing a "micro-transaction": an atomic update of multiple keys by a single transaction in certain limited situations. All update operations are sent to the head of a chain, and the head can make any decision it wishes, including a non-deterministic decision.[6] Therefore, the head can decide on the fate of multiple operations that are sent in a single client request: "commit" by applying changes for all operations in the request, or "abort" by applying none of them.

Hibari has implemented a similar transaction feature. A client can send multiple primitive query and update operations in a single protocol request to a Hibari data server. The limiting factor is that all keys for the primitive operations in the request must be keys that the brick is responsible for. This limit is the reason for using the word "micro-transaction" instead of "transaction."

To implement request forwarding, for example, when a client sends a request to the wrong brick, each Hibari brick is already aware of what range of keys it is responsible for. Micro-transactions introduce a second reason why Hibari bricks must maintain this awareness: if a brick detects that a micro-transaction attempts to operate on keys stored in multiple chains, the micro-transaction must be aborted.

To use micro-transactions effectively, the client application must be aware of the key prefix scheme used by each table. It is the client's responsibility to create micro-transactions where all keys are managed by the same chain. This implicit knowledge could be made explicit by changing the client API: add a parameter to specify the consistent hash string, similar to a "bucket" in the Riak client API [18]. By using either implicit key prefixing or an explicit bucket-like grouping, the client controls whether any two keys must be stored in the same chain.

For example, assume a need to build a simple Web blogging application. On a per-user basis, the application requires storage for user authentication data, biographical data and preferences settings, blog postings, and comments on blog postings. The blog postings and comments would be stored in a single table called `posts`. The hashing key prefix, configured when the `posts` table was created, would be a variable prefix delimited by two slash characters.

See Figure 7 for example keys that would be stored in the `posts` table for user #42. The value of the `/42/summary` key would contain metadata for the user's collection of postings: the number to assign to the user's next post, the number of active/undeleted posts, the number of deleted posts, etc. All comments for post #1 would be retrieved by a `'get_many'` operation with the {`binary_prefix,`

[6] Non-deterministic choices are mentioned in [24].

```
add_new_post(UserID, PostText) ->
  Prefix = "/" ++ integer_to_list(UserID) ++ "/",
  MetaKey = Prefix ++ "summary",
  {ok, OldTS, OldVal} =
      brick_simple:get(posts, MetaKey),
  #post{next_id = NextID, active = Active} =
      OldMeta = binary_to_term(OldVal),
  NewMeta = OldMeta#post{next_id = NextID + 1,
                         active = Active + 1},
  PostKey = Prefix ++ integer_to_list(NextID),
  %% replace op: Abort if the key does not exist
  %%             or if current timestamp /= OldTS.
  %% add op: Abort if the key already exists.
  Txn = [brick_server:make_txn(),
         brick_server:make_replace(MetaKey,
                          term_to_binary(NewMeta),
                          0, [{testset, OldTS}]),
         brick_server:make_add(PostKey, PostText)],
  [ok, ok] = brick_simple:do(posts, Txn).
```

Figure 8. Example code to add a new Web blog posting using a micro-transaction.

`"/42/1/"}` option to limit results to only those keys that have a prefix that matches post #1's comments.

To create a new posting, the micro-transaction feature would be used to keep the metadata in the `summary` key consistent despite races with other metadata updates. A simple function, without error handling code for sake of simplicity, is shown in Figure 8.

10. Automatic Key Partitioning and Migration

Some key-value stores in the open source world [14, 23] do not include automatic support for key partitioning (also called "key sharding"): they assume the client will implement it. Unfortunately, coordinating the actions of many distributed clients in a 100% bug-free manner is a very difficult task.

Other distributed storage systems place significant restrictions on key migration/repartitioning. For example, the MySQL Cluster RDBMS did not support repartitioning until April 2009, and then only to expand the size of the cluster [20] — reducing cluster size was not supported.

Reducing cluster size is a valuable feature. Also, support for heterogeneous hardware is very desirable. It is nearly impossible to buy the same hardware more than three months after a system has been deployed in the field, much less three years or more in the future.

The original chain replication paper [24] is silent on the subject of key migration. Hibari provides support for key migration as well as support for heterogeneous hardware. Both are accomplished by its consistent hashing implementation.

Each Hibari server and client node maintains two complete consistent hashing maps for each Hibari data table: one old/current map and one new map. During normal operations, the two maps are identical. However, during a key migration period, the two maps will be different: the current map describes where keys are stored in the current scheme, and the new map describes where keys are stored in the desired scheme.

The bottom two-thirds of Figure 5 shows an example of the chain mappings used to migrate a table from two chains to three chains and later four chains. A key K_1 with an MD5 hash that maps to 0.1 on the unit interval would be stored in Chain 1 and would not move in either key migration. A key K_2 with an MD5 hash that maps to 0.49 on the unit interval would initially be stored on Chain 1. The first key migration would move K_2 from Chain 1 to Chain 3. The second key migration would move K_2 from Chain 3 to Chain 4.

The method demonstrated in Figure 5 attempts to minimize key movement and to evenly distribute migration workload. However, the Admin Server API permits the flexibility to choose arbitrary map definitions for a key migration. As a planning tool, an API function is provided to calculate how many keys would be moved between all pairs of chains, given a specific hash map.

Hibari's key migration is performed dynamically, while all bricks and clients are in full operation. The chain head brick selects a "sweep window," a range of keys (in lexicographic sort order), and copies the keys to their respective destination chains. When all destination chains have acknowledged successful writes, the sweep window is advanced, and the process repeats. All chain heads perform key migration sweeps in parallel. Operations by clients on keys inside the sweep window are deferred until the sweep window advances.

Due to the realities of message passing asynchrony, it is possible for clients to send queries to the wrong brick in the wrong chain. Each brick will determine if a query has been sent incorrectly and, if so, forward the query to the appropriate brick. Most forwarding involves only one extra hop or are loops that exist for very small periods of time (typically much less than one second). The forwarding delay and maximum hop mechanism described in section 6 take care of rare, long-lived forwarding loops.

Hibari's key migration implementation is currently missing a feature requested by at least one customer: the ability to halt a migration. If the I/O workload caused by migration causes severe latency problems for normal Hibari client applications, the customer wishes to suspend migration until peak client workload subsides.

Aborting a migration entirely would be much more difficult. The sweep key mechanism would have to "run backward": a sweep of the key space in reverse order would send each key-value pair from its destination chain backward to its source chain. Rather than implement this complex feature, it is much easier to permit the current migration to map M_n to finish, and then trigger a new migration to map M_{n+1} where $M_{n-1} = M_{n+1}$ to move all keys back to their original location.

11. Replica Placement

Terrace and Freedman [22] have discussed replica placement strategy: where should various chain members be located physically and logically? For example, all replicas within a chain should not be within the same physical data center rack: rack-wide power failures and network outages are too common, even in well-managed data centers.

One of chain replication's nice features is that it doesn't make many demands on replica placement policy, giving an administrator great flexibility. For example, a Gemini customer decided that chain lengths of three would be sufficient to meet its availability goals. The Hibari default replica placement arranges bricks within chains as if the underlying physical machines were in a ring: chain 1 uses machines $A \rightarrow B \rightarrow C$, chain 2 uses machines $B \rightarrow C \rightarrow D$, and so on. In the case of 26 physical machines, the final chain would use machines $Z \rightarrow A \rightarrow B$. (See also Figure 2 for an example of a ring of five machines.)

Using the above ring strategy, the resources of each machine are likely to be used equally: each physical machine would host an equal number of head, middle, and tail bricks. This balance was pleasing to the customer. However, the operations impact of key migration did not appear so pleasing when considering expanding the size of the cluster. If the new machines are inserted into the ring between A and Z, then the machines nearby, that is, machines A, B, Y, and Z, will endure greater load caused by key migration than the other 22 original nodes.

Original Machines									Original Machines									New Machines								
Machine A			Machine B			Machine C			Machine D			Machine E			Machine F			Machine G			Machine H			Machine I		
Head0	→		Middle0	→		Tail0.			**Head3**	→		Middle3	→		Tail3.			**Head6**	→		Middle6	→		Tail6.		
Tail1.			**Head1**	→		Middle1	→		Tail4.			**Head4**	→		Middle4	→		Tail7.			**Head7**	→		Middle7	→	
Middle2	→		Tail2.			**Head2**	→		Middle5	→		Tail5.			**Head5**	→		Middle8	→		Tail8.			**Head8**	→	

Figure 9. Replica placement using groups-of-3-machines strategy: start with six machines, then add three more. Machines that maintain head bricks are bold-faced to highlight the striping pattern.

This customer decided to use a different placement strategy, called groups-of-three-machines strategy. For each Hibari table, a group of three chains is striped across a small group of three machines. This process would repeat until all machines were accounted for. See Figure 9 for an example. The result provides equal workload sharing: each machine still has an equal number of head, middle, and tail bricks. Also, adding new machines (in groups-of-three) will create a balanced workload during key migration: assuming that all chain weightings are equal, then roughly 50% of all keys in chains on machines A through F will migrate to chains on machines G through I.

On top of the groups-of-three placement strategy, the customer is free to use rack-aware placement also. For example, each physical machine in a group-of-three can be placed in a different rack.

12. Other Observations

This section contains a number of observations about Hibari's implementation and production deployments that don't merit entire sections to themselves.

12.1 Using `gen_server`

Hibari's implementation makes heavy use of the Erlang/OTP `gen_server` behavior, largely because its serial method of handling messages maps very well onto the serialization that a well-behaved chain replication server must do. However, a single Erlang process cannot consume more than a single CPU core's worth of computation resources. Due to Hibari's one-to-one mapping of logical bricks to Erlang processes, an administrator who wishes to take full advantage of multi-core and multi-CPU systems must provision more chains than strictly necessary so that many logical bricks will be assigned to a single physical brick.

The extra logical bricks come at a cost of management complexity. The Admin Server now must keep track of more bricks and chains than is otherwise strictly necessary. The overhead of monitoring each brick is small, but when monitoring a few thousand bricks, the total cumulative workload can cause problems. The biggest single bottleneck is updating the Admin Server's private state storage bricks. For the sake of simplicity, updates to the private state bricks are serialized. When a cluster with over 3,000 logical bricks are booted simultaneously, the number of state transitions that are generated each second can exceed the state storage bricks' maximum update rate. A future release of Hibari will fix this problem.

12.2 Chain Reordering

Chain reordering doesn't appear in either the original chain replication paper [24] or the CRAQ paper [22], but it's valuable from an operations perspective. As originally described, a chain can become reordered by the failure and repair of member servers. In the long term, such reordering can destroy an administrator's intended balance of workload across hardware resources.

For example, if a chain is configured as $B_1 \rightarrow B_2 \rightarrow B_3$ and brick B_2 fails, then after repair is finished, the chain's order will be $B_1 \rightarrow B_3 \rightarrow B_2$. If brick B_1 fails later, the chain's order will be $B_3 \rightarrow B_2 \rightarrow B_1$ (again, after repair). Without reordering, the chain will remain in this order until yet another brick fails. Hibari, however, will reorder the chain back to $B_1 \rightarrow B_2 \rightarrow B_3$ once the repair of B_1 is complete.

12.3 Key Timestamps

Each key in a Hibari server has a timestamp associated with it. Each server enforces a rule that each update must strictly increase the key's timestamp. This feature prevents multi-client races that attempt to update the same key. The timestamp can also be used for "test and set" behavior, which will abort a micro-transaction if the key's current timestamp does not exactly match a timestamp observed in an earlier operation.

Key timestamps have subsequently become extremely important for optimizing brick repair. Other projects such as Dynomite [7] and Riak [18] use Merkle trees to quickly calculate which keys two servers share.

Hibari uses a simple "I have"/"Please send" iterative protocol to identify keys that need repair. Each key and its timestamp are sent in the "I have" phase. Because all keys and timestamps are stored in RAM, no disk I/O is required (by either the upstream/online brick or the downstream/repairing brick) to complete the "I have" phase. Disk I/O (to retrieve value blobs) is required only for keys that are missing or out-of-date on the downstream brick.

12.4 Read-Ahead

Read-ahead optimizations by the operating system's disk subsystem can often degrade performance. Most of Hibari's read-only disk operations are random in nature across mostly small pieces of data, usually only a few kilobytes each. Read-ahead mechanisms that try to read hundreds or thousands of kilobytes merely create higher latency for all disk operations.

There is one case where Hibari could use very aggressive read-ahead buffering by the OS: during brick initialization's sequential scan of the brick's write-ahead log. The Erlang virtual machine does not support system calls like `fadvise(2)` and `mincore(2)`, so it has no custom control over read-ahead behavior. We have not yet been desperate enough to try using the newer R13B Erlang NIF feature or an older-style driver to implement these system calls, but we probably will, someday.

12.5 File Checksums

Hibari stores all log data on disk with MD5 checksums. Any data corruption detected by an MD5 checksum will cause a brick to take itself out of service. Automatic chain repair will identify the keys lost due to corruption and re-replicate those keys. The bad file is moved to a separate directory to prevent future access. By not deleting the bad file, Hibari hopes to avoid reusing the bad disk block(s) that caused the original problem.

Unlike GFS [8], Hibari's network messages are not yet protected by checksums. It is possible for a bit error to escape the network protocols stack's checksum regime and affect Hibari clients and/or downstream bricks.

12.6 Murphy's Law

Anything can and does happen in a production environment. The "impossible" is possible, and if something can go wrong, it will.

In one memorable case, the Erlang/OTP "kernel" application's error_logger process was overwhelmed by over 85,000 error messages that were triggered by Erlang's system_monitor() BIF in under one minute. We know that such system messages can be generated extremely quickly by the Erlang virtual machine. Hibari's process that receives the system events will only forward 40 events per second to the error_logger. [7]

So, how can a process that throttles itself to generate only 40 error messages/second send over 85,000 messages to error_logger in under one minute? After all, (40 messages/sec)(1 minute = 60 seconds) = 2,400 messages. Intensive code review of the rate limiting mechanism has found no fault. And we know for certain that all 85,000 messages were generated in under 60 seconds. This mystery will probably never be solved.

12.7 Other observations

- It is too easy, especially when subject to schedule pressures, to shoot yourself in the foot with Erlang. Code with side-effects is difficult to understand, to test, and to support ... yet management of side-effects (i.e., mutable state) is Hibari's reason for being. If we were to rewrite Hibari from scratch, we would be extra careful to segregate code with and without side-effects to simplify testing by QuickCheck and other tools.
- An early implementation decision for Hibari, left ambiguous by the original chain replication paper and discarded by the CRAQ paper, was that any single logical brick can be a member of only one chain. In hindsight, it was a good decision. The complexity of implementing the key migration logic for a logical brick that stores keys for multiple chains would have been painful.
- Hibari relies on Erlang's network distribution service for all significant cluster communication. The short term impact is positive: Erlang message passing "just works."[8] The long term impact is negative: nobody knows the largest practical size of a single Erlang cluster. To build a Hibari cluster with thousands of nodes, we may have to move away from Erlang's built-in messaging or, perhaps, re-write Ericsson's network distribution code.

13. Related Work

Citations of related work have appeared throughout the paper; however, a few other prior works should be mentioned.

As described in the introduction, others have written about the experience of implementing distributed algorithms; citations of [3, 5, 11] only scratch the surface of recent publications. Any Erlang developer who attempts to implement a distributed algorithm from scratch or modify one should read [2] and [21] before starting.

Consistent hashing was introduced by Karger et al. in [10]. The technique has become popular for replacing central directory services for many key → location mapping needs. Central directory servers, in most cases, run on a single host and are therefore a single point of failure for availability and also a likely performance bottleneck.

Amazon's Dynamo distributed hash table [6] uses a layer of indirection in its consistent hashing implementation. Hibari's implementation is quite similar. The main differences are in naming and that Hibari's number and size of hash interval partitions can be changed by using key migration. Also, Hibari's hash partition sizes may be heterogeneous, as demonstrated in Figure 5.

Replica placement is the main topic of [25] and is also discussed in [8] and [22]. The Cassandra distributed database has plug-in API [4] that helps encourage experiments with different placement policies.

[7] The message flow is: Erlang VM → system event receiver process → error_logger process.

[8] An important exception is described in section 8.

14. Conclusion

Using the Hibari distributed key-value store as a case study, we have shown that the path from a pure, proven algorithm to real-world implementation is not smooth. Most of the problems we've encountered with Hibari, both with implementation correctness and with performance, apply not only to Erlang but all distributed computing environments.

We all share the same limitations, such as hard disk drives that grow ever slower relative to the computers they are paired with, failure-prone networks, the maximum speed of light, the fundamental properties of asynchronous messaging, and the problem of making theoretical ideas such as "fail stop" into an equivalent reality. We must never forget that any change to a distributed algorithm or its environment or implementation, no matter how small or innocent the change may appear, may in fact invalidate the algorithm ... and it may be weeks, months, or even years before we notice the error.

15. Availability

In July 2010, Gemini released the Hibari source code under the Apache Public License version 2.0. Its source code and documentation, including Systems Administrator's Guide and Developer's Guide, is available at `http://hibari.sourceforge.net/`.

Acknowledgments

Many thanks to Gemini Mobile Technologies, Inc.: to the Shibuya development team for code and documentation reviews and to the Shibuya field engineer team for their invaluable customer support. I also owe thanks (and probably quite a few drinks) to the ACM's reviewers and to Olaf Hall-Holt, Satoshi Kinoshita, James Larson, Romain Lenglet, Jay Nelson, Joseph Wayne Norton, Gary Ogasawara, Mark Raugas, Justin Sheehy, Ville Tuulos, and Jordan Wilberding. Finally, thank you to Louise Lystig Fritchie for her unstinting patience and editing wizardry. All remaining errors are mine.

References

[1] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *The Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII)*, pages 33–38, 2001.

[2] T. Arts, K. Claessen, J. Hughes, and H. Svensson. Testing implementations of formally verified algorithms. In *Software Engineering Research and Practice*, 2005.

[3] T. Arts, K. Claessen, and H. Svensson. Semi-formal development of a fault-tolerant leader election protocol in erlang. In *Lecture Notes in Computer Science*, pages 140–154. Springer, 2005.

[4] Cassandra Wiki. URL `http://wiki.apache.org/cassandra/-Operations`. Accessed on 31 July 2010.

[5] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, 2007.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of SOSP*, pages 205–220, 2007.

[7] Dynomite key-value store. URL `http://github.com/-cliffmoon/dynomite`. Accessed on 31 July 2010.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on operating systems principles*, pages 29–43, New York, NY, 2003.

[9] Hibari. URL `http://hibari.sourceforge.net/`. Accessed on 31 July 2010.

[10] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. P. Abstract. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *In Proc. 29th ACM Symposium on Theory of Computing (STOC)*, pages 654–663, 1997.

[11] I. Keidar and L. Zhou. Building reliable large-scale distributed systems: When theory meets practice. *ACM SIGACT News*, 40(3), September 2009.

[12] L. Lamport and K. Marzullo. The part-time parliament. *ACM Transactions on Computer Systems*, 16:133–169, 1998.

[13] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[14] Memcached. URL `http://memcached.org/`. Accessed on 31 July 2010.

[15] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Annual Technical Conference*. USENIX, 1999.

[16] R. D. Prisco and B. Lampson. Revisiting the paxos algorithm. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG 97), volume 1320 of Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, 1997.

[17] Quviq AB. QuickCheck property-based software testing tool. URL `http://www.quviq.com/`. Accessed on 01 August 2010.

[18] Riak key-value store. URL `http://wiki.basho.com/display/-RIAK/Riak`. Accessed on 31 July 2010.

[19] Squid. Squid http proxy. URL `http://www.squid-cache.org/`. Accessed on 31 July 2010.

[20] Sun Microsystems. Sun announces mysql cluster 7.0 for real-time, mission-critical database applications. URL `http://www.mysql.com/news-and-events/generate-article.php?id=2009_06`. Accessed on 01 August 2010.

[21] H. Svensson and L.-A. Fredlund. Programming distributed erlang applications: Pitfalls and recipes. In *ACM Erlang Workshop*. ACM Press, 2007.

[22] J. Terrace and M. J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *USENIX Annual Technical Conference*, San Diego, CA, 2009.

[23] Tokyo Tyrant. URL `http://1978th.net/tokyotyrant/`. Accessed on 31 July 2010.

[24] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *USENIX OSDI*, 2004.

[25] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*. ACM, 2006.

[26] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on operating systems principles*, pages 230–243, New York, NY, 2001.