Chain Replication In Theory and in Practice

Working Title, rough draft

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 running code can be difficult. Staff at Google have documented the difficulties in implementing the Paxos algorithm: making the implementation correct andreaching performance goals.

In the spirit of justaposing the purity of theory and proof with the practice of implementation and real-world deployment, this paper explores the implementation of the chain replication protocol in a distributed key-value database called Hibari. In theory, the chain replication algorithm is simpler than the Paxos algorithm. In practice, there are plenty of practical implementation details that have hampered both a correct and a sufficiently-fast implementation. I hope that the lessons learned here will help others in the Erlang community and distributed systems implementors in general to create robust distributed systems while avoiding traps and pitfalls that lurk at the boundaries of the real-world computing.

This draft is likely somewhere between a draft extended abstract and rough draft full paper quality. Reviewers should assume the latter and thus be merciless in their critique.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

Lynch's *Distributed Algorithms* [5] is one of several collections of formal specifications and proofs for distributed algorithms. As Lynch is the first to admit, translating any algorithm into running code can be difficult. Lampson's Paxos algorithm [4, 7] is now a well-known distributed algorithm for maintaining shared consensus, but as staff at Google have documented in [2], there are at least two problems that any implementation must solve: maintaining the algorithm's correctness and reaching acceptable performance goals.

Continuing in the spirit of justaposing the purity of theory and proof with the practice of implementation and real-world deployment, this paper explores the implementation of the chain replication protocol for high throughput and strong consistency in a distributed key-value database called Hibari. In theory, the chain replication algorithm is simpler than the Paxos algorithm, but in practice, there are plenty of practical implementation details that have hampered both a correct and a sufficiently-fast implementation. We hope that the lessons learned here will help others in the Erlang community and distributed systems implementors in general to create robust distributed systems that [blah blah needs good final].

2. Chain Replication Overview

Chain replication was first described by Van Renesse and Schneider [11]. It describes a variation on master/slave replication where all servers that are responsible for storing the replica of a key are arranged in a strictly-ordered chain. The head of the chain makes all decisions about updates to any particular key. The head's decision is propagated down the chain in strict order. The principle of strong consistency (CITE??) is preserved if all read-only queries are handled by the tail of the chain: by definition, all updates have already been processed by replicas earlier in the chain (or "upstream") of the tail brick.

The number of replicas for a given key is directly determined by the length of the replica chain that is responsible for that key. To tolerate F replica server failures, a chain must be at least F+1 servers long.

Hibari, a distributed key-value store written in Erlang, implements the chain replication algorithm as described by Van Renesse and Schneider with little change. Erlang's message passing makes it trivial to support the original algorithm's asymmetric message passing for updates: client updates are sent to the head of the chain, but the head's reply is sent to the client via the tail of the chain. The paper's description of local update logging and repair in case of failure of a member brick are implemented as described.

3. Hibari Overview

Hibari is a distributed key-value database that also provides high throughput: several thousand updates per second of 1KB values per second per server on typical commodity 1-2U rack-mountable server hardware. Client operation throughput increases linearly as servers are added to the cluster. High availability is achieved through using replication chains of more than one server, combined with quick reaction times (e.g. under 10 seconds) when a member server fails.

Hibari maintains the principle of strong consistency by using the chain replication protocol as originally described: all updates are sent to the head of a chain, all read-only queries are sent to the tail of a chain, and all replies for all client operations are sent by the tail.

Hibari uses a consistent hashing technique to map a TableName, Key tuple to the name of the chain that is responsible for storing that key Key in table TableName. Two such mappings are used to implement data migration (or key repartitioning). In normal operation, the maps are the same. During data migration, the maps are used to calculate the current and new/desired location of a key. Data migration is a dynamic, online process and can be used to expand or contract a cluster as well as change relative balancing weights of chains.

4. Disk Writing Latency Is a Big Deal

The price of RAM-based and flash RAM-based storage is too expensive for many big data applications. At a price/gigabyte ratio today (US dollars) of TODO for flash RAM storage, only traditional hard disks are cost-effective for an application such as free/advertising-supported email services. The market now routinely provides storage quotas in the range of 1-10 gigabytes per user, with "unlimited" storage available at extremely low monthly fees.

Traditional 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 4.5 milliseconds for top-of-the-line SCSI disks and at least double that amount for slower SATA disks. As a result, it's in an application's best interest to minimize the number of random I/O operations that it generates.

The write-ahead log has been used by database systems for decades to aggregate disk write operations and flush them to stable storage, minimizing random disk I/O operations by appending log entries to a log file. A "group commit" technique is frequently used to use a single fsync(2) operation (or OS equivalent) to flush many transactions' worth of log entries to the write-ahead log simultaneously.

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

An Erlang process that is based on an Erlang/OTP gen_server behavior can usually utilize one full CPU core when writing data to a file in the write-ahead log. The fsync(2) operation, however, can block a gen_server process for up to tens of milliseconds per call. Such blocking delay is unacceptable in almost any latency-constrained application. To solve this problem, each gen_server-based Hibari data server sends all writes to a central write-ahead log (WAL) process for write(2) and fsync(2) call management. The WAL process sends messages back to the data server when an fsync(2) operation has been completed.

The chain repliation protocol already requires each write operation to have a serial number associated with it, and that each update is propagated down the chain in serial number order. The WAL process uses these serial numbers to signal each data server the largest serial number that has been safely flushed to disk via fsync(2). Each server is then free to send those updates to downstream bricks at its leisure. However, this update communication between data server and WAL server processes has been fraught with subtle, difficult-to-find race conditions where writes are written out of order, fsync(2) operations are acknowledged with wrong log serial numbers, and data servers sending updates downstream out of order. The QuickCheck software testing tool [1] has been invaluable for helping create the conditions necessary to exploit the very small windows of vulnerability of many of these bugs.

5. Disk Reading Latency Is a Big Deal

For the purposes of this paper, "big data" means storing more data than comfortably fits in RAM. In the context of a key-value database, it means that the total size of all keys, values, and related metadata is larger than can be stored in RAM of a single machine. For a distributed key-value store, the total amount of data and metadata is larger than can be stored in the RAM of all machines in the clustered store.

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 value blob read operations may also generate disk I/O. Caching strategies by the OS or application can only be as good as the size of cache available (e.g. RAM) and the temporal locality patterns of the client application(s).

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 for a value blob read by always storing the write-ahead log file number, starting byte offset, and value blob size in RAM. The log file path can be calculated from the file number, so the application need only use a single open(2), lseek(2), read(2), and close(2) call each to read any value blob.

Client-driven workload is not the only source of disk read I/O demand. The two most significant ones are:

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 from disk before sending downstream to the repairee.

2. Data migration. Sometimes call data repartitioning, resharding, or rebalancing, the Hibari data migration process moves keys (and associated values and metadata) from one chain to another. All keys that must be moved during a data migration must have their values read from disk by a brick in the source chain before transmission to the destination chain.

As described above, a gen_server process can be blocked by disk I/O. Any file open(2) or read(2) system call can block a Hibari gen_server process for many milliseconds each. For extremely overloaded systems, the cost can be easily over 100 milliseconds each. The effect on latency-sensitive applications is enormous.

To avoid blocking gen_server processes with read-only disk I/O, Hibari borrows techniques used by the Squid HTTP proxy [8] and Flash HTTP servers [6]. Before a gen_server attempts to open or read a file, it first spawns a "primer" process which asynchously 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 file API, e.g. open(2) and read(2). When finished, the primer process sends a message to the main gen_server process that the priming action is complete. Then the gen_server process can open and read the file (using the same system calls) with very little probability of blocking.

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

For both chain repair and data 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 by the lexicographic sort order of the keys stored by the brick, whereas the order the value blobs are stored on disk is related to the time at which their respective updates were received.

The resulting mismatch can create a significant amount of random I/O workload, as far as the underlying disks are concerned. For data migration workload, the latency is largely unavoidable in the current implementation. For brick repair workload, the latency can be quite low or extremely high. If the brick under repair was down for only a short while, 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 sorted by location within the write-ahead log. This can help reduce the amount of random read disk I/O required to read the value blobs. The savings can be very significant when the total size of value blobs is in the hundreds or thousands of megabytes.

For repair tasks that fall in the middle, the number of keys to repair is large but the cost of starting repair completely from scratch is too high. In this middle case, there is no choice other than wait for the standard repair technique to finish and to accept the amount of read disk I/O required to do so. And in any case, for a chain that contains terabytes worth of data, 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 (i.e. how long should each chain be).

6. Rate Control Is Effectively Mandatory

It's almost certain that the storage server will be the primary performance bottleneck in a distributed client application system. Modern hard disks are simply orders of magnitude slower than other components in the system: CPU, RAM, system buses, and even commercial gigabit Ethernet interfaces and switches are less likely to be the slowest system component. To avoid overloading disk subsystems even further, it is effectively mandatory that rate control mechanisms be added to control just about 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) for the following: number of "primer" processes for prefetching value blobs from disk, chain repair operations, data migration operations, and log "scavenging" operations (which reclaim space from the theoretically infinite sized write-ahead long).

Hibari also has an implicit limit on the number of application client operations that a single brick can support. The simple technique is borrowed from SEDA [12]: if the client request is too old, then drop it silently. Each Hibari client request contains a wall clock timestamp. If that timestamp is too far in the past, the Hibari server will ignore the request, under the assumption that the request waited in the server's Erlang mailbox for so long that the server must be overloaded. To send a reply to the client will create even more work for the server to do, so the cheapest thing to do is to do nothing.

Also, during data 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, 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.

7. Cluster Management and Monitoring

The original chain replication paper describes a single master entity that is responsible for managing the state of each server brick within each chain. Such single entities are also single points of failure within the system and, for high availability applications, need to be avoided altogether or at least minimized.

Hibari's implementation implements the single logical service 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, called the "Admin Server", in an active/standby manner. This indeed creates a single point of failure: if the machine running the Admin Server crashes, the Admin Server's services are lost.

In balance, however, failure of the Admin Server 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, history, or reconfigure the cluster. Hibari client nodes and applications may continue operation without error, as long as other bricks do not fail simultaneously.

The single application instance has another convenient consequence: it makes behavior during network partition events easier to reason about. For clients and server bricks on the "far" side of a network partition (relative to the running Admin Server), those clients and bricks will be severely affected by the partition and will be unable to perform most operations(*). For clients and bricks on the "near" side of the partition, the Admin Server will be able to reconfigure chains to maintain service to all clients on the near side of the partition, so long as each chain has at least one live brick that is also on the near side.

(*) Footnote: If all bricks in a chain are on the "far" side of a partition, then the Admin Server can communicate with none of them, so the chain can operate in its current state ... as long as none of its member bricks fail while the network partition exists.

The CRAQ paper [10] proposes 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. Other distributed techniques were considered at the time, but all of them were discarded in favor of implementation simplicity and reasoning about them in case of network partition. If the entity/enties that are responsible for managing chain state make faulty decisions, it is **surprisingly easy** to lose data in a very short period of time.

The original chain replication paper 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 will cause data loss.

The biggest problem with detecting halted nodes is the problem of network partition. With message passing alone, it's impossible to tell the difference between a network partition, a failed node, or merely a really slow node. The built-in Erlang/OTP message passing and network distribution mechanisms cannot adequately handle network partition events by themselves. Svensson and Fredlund [9] expand on the Erlang messaging limitations described in the Erlang/OTP documentation [3].

To combat the worst problems caused by network partition, Hibari includes an OTP application called "partition_detector". 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 the application assumes that a partition of network 'A' is in progress.

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

Failure of the "fail stop" assumption have also caused problems for Hibari. Gemini Mobile Technologies is not responsible for day-to-day operations and monitoring of its customer's systems, so all the data we have received has been during "post mortem" analysis of past failures of a customer's lab or production system. In these post mortem analyses, we have 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 failover.
2. Interference in process scheduling and high inter-node message passing latencies created by 'busy_dist_port' events. All Erlang message passing traffic between two Erlang nodes is transmitted over a single TCP connection. If the sending node detects congestion (e.g. a slow receiver, intermittent network failure), then any Erlang process on that 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 TCP port is no longer congested.

The trio of 'net_kernel' deadlock, wild variations of message passing latency, and process de-scheduling create a "fail stutter" (CITE, Lynch??) environment instead of "fail stop". The result is that Hibari's largest deployment has suffered from availability failures but not data loss.

One solution to this problem has been a small patch to the Erlang virtual machine to make the buffer size for network distribution ports configurable. The default size of the 'erts_dist_busy' constant is 128 kilobytes. Even with a value of 4 megabytes appears to be too small for the amount of data that Hibari servers move in bursty traffic patterns.

Another solution is to use information from Hibari's partition detector application to supplement the monitoring info that the Admin Server uses once it has initialized itself and is running normally. If a 'nodedown' message is received (as a system monitoring event, requested via the erlang:system_monitor() BIF), the partition dectector's state is queried to check if a network partition was a possible trigger for the message. The same is done if a query of a remote brick's general health status fails due to 'timeout' or 'nodedown' reasons.

As a whole, the single Admin Server process has had more problems in production than we had anticipated. The solutions outlined above have not been in production long enough to judge their effectiveness. However, given the problems that we know have happened in production networks, it is likely that a manager distributed across many nodes would likely have been fooled by the same conditions into making similarly bad decisions.

8. Erlang Messaging Is Not Always Reliable

The original chain replication paper says, "Assume reliable FIFO links." There is no such thing in the real world. So how do we avoid this problem?

Erlang's "send and pray" messaging semantics are well known and documented. Why do people who should know better (including this author) write code that assumes that Erlang's messaging is more reliable than it is?

The original chain replication paper suggests that the tail of the chain acknowledge each update after its processing has been successfully completed by the tail. This acknowledgement is forwarded all the way back to the head, which must keep track of such acknowledgements for chain repair purposes. The Hibari implementation avoids the overhead per update (which can be as high as several thousand updates per second per chain) by using a once/second acknowledgement of the largest update serial number processed by the tail.

This section specifically deals with the nature of gen_server:cast() and its use within Hibari. Under the surface, the gen_server:cast() call is a very thin layer of wrapper around Erlang's message send() BIF (built-in function). The nature of an error in Hibari is not an assumption about message ordering but of when a message between two Erlang nodes may be dropped.

Svensson and Fredlund describe clearly in [9] under what conditions the usually-reliable messaging between two Erlang nodes can turn unreliable.

Fortunately, the Erlang process monitoring BIF, monitor(), will deliver a {'DOWN',...} message to the receiver when the connection between nodes may have dropped messages.

9. Micro-Transactions Are Valuable

While not mentioned in the original chain replication paper, the CRAQ paper mentions the possibility of implementing a "micro-transaction" fairly easily. Because all updates are sent to the head of a chain, and because the head can make any decision it wishes (including non-deterministic decisions), that single decision-making entity can also decide on the fate of multiple key operations that are sent in a single client query: "commit" by applying all of their changes, or "abort" to apply none of them.

Hibari has implemented a similar micro-transaction feature. A client can send multiple primitive key query and update operations in a single protocol operation to a Hibari data server. As long as all keys for all primitive operations are for keys that the server is responsible for, then any additional pre-conditions (e.g. fail if the key already exists/does not exist, "test and set" condition) can also be checked and, if satisfactory, then all primitive operations are applied by the head and propagated down the rest of the chain.

Each Hibari server must be aware of what range of keys it is responsible for in order to implement request forwarding, i.e. when a client sends a request to the wrong server. Micro-transactions introduce a second reason why Hibari server's must be fully aware of the consistent hashing algorithm used to map keys to chains: if a server detects that a micro-transaction is attempting to operate on keys managed by two or more chains, the micro-transaction must be aborted as quickly as possible.

Hibari's micro-transaction feature has been heavily used by a custom Webmail application, developed by Gemini Mobile Technologies. The application was developed under a very tight time schedule. It is unlikely that it could have been delivered on time if client-side logic were always required to handle inter-key data consistency.

10. Automatic Data Partitioning and Re-partitioning Is Mandatory

Some key-value stores in the open source world (CITE? Tokyo Tyrant, memcached, redis, ...) do not include automatic support for data partitioning (also called "data sharding"): they assume the client will implement that task. Unfortunately, coordinating the actions of multiple distributed clients in a 100 percent bug-free manner is a very difficult task.

Other distributed storage systems place significant restrictions on data re-partitioning. The MySQL Cluster RDBMS did not support re-partitioning until 2009, and then only to expand the size of the cluster. But reduction in cluster size is also a valuable features. Similarly, support for homogenous hardware is very desirable: it's nearly impossible to buy the same hardware more than three months after an system has been deployed in the field, much less three years or more in the future.

Hibari provides support for data migration as well as heterogenous hardware. Both are accomplished by its consistent hashing implementation. Hibari's consistent hash algorithm uses the following inputs to calculate a chain name output: table name, key name, and chain weighting structure.

The key name (or a configurable key prefix) is hashed using the MD5 algorithm and mapped onto the unit interval of 0.0 – 1.0. The unit interval is divided into an arbitrary number of ranges, where each range represents a chain name. (SEE unfinished diagram) Each chain can appear 0 or more times in the range map, and the relative size of each range is determined by the chain weighting factor. If the chain weighting factor for chain C1 as the weighting factor for chain C5, then the total sum of sizes of range intervals found in the range map will be twice as large for chain C1 as for C5.

To implement data migration, each server and client node maintains two complete consistent hashing maps for each Hibari data table: one "current" view and one "new" view. During normal operations, the two views are identical. However, during a data migration period, the two views will be different: the "current" view describes where keys are mapped in the current scheme, and the "new" view describes where keys are mapped in the newly desired scheme.

Hibari's data migration is performed dynamically, while all servers and clients are in full operation. Due to the realities of message passing asynchrony, it's possible that clients will send queries to the wrong chains. Each brick will determine if a query has been mis-forwarded and, if so, re-forward to the appropriate server. Most forwarding loops are three-legged only or exist for very small periods of time (typically much less than one second). The forwarding delay and maximum hop mechanism described earlier takes care of the rare, long-lived forwarding loop.

11. Data Placement Is Flexible

(Draft section, sorry) Multiple logical bricks for the same chain on same box -¿ greater CPU/multi-core utilization is possible, but it also makes for greater management overhead.

Brick "placement": sketches are discussed in CRAQ paper and somewhat in orig paper. Part of "niftiness" of chain replication is that it's very flexible and therefore allows placement however you wish. E.g. Japan customer using group-of-three-boxes strategy. "Each logical brick in a physical machine in different rack" would be easy to implement, as would many other physical placement policies (host placement, rack placement, data center placementj)

Ring placement strategy is efficient, but then adding/removing machines can cause asymmetric migration workload. That's why Japan customer chose group-of-three-boxes strategy.

12. Ops Section

(Sketch section, sorry) How do you monitor the thing? We needed/need more ops tools.

13. Smaller But Important Observations

(Draft/sketch section, sorry) Chain reordering doesn't appear in either (?) paper, but it's valuable from an ops perspective.

Timestamping of each key for client use was also useful for server repair (instead of using e.g. Merkle trees, which were found to be expensive in pure Erlang in Dynomite's implementation)

Per-op async option was not a good idea: self-inflicted inconsistency within a chain

- It is too easy (and especially when subject to schedule pressures) to shot yourself in Erlang. Don't mix complex code with side-effects. It is difficult to understand, to test, and to support.

(comment: In some ways, Haskell is the right way to go ... but I prefer Erlang's approach for modeling concurrency, fault tolerance, etc.)

- Read-ahead optimizations of the operating system's disk subsystem can (often) be unhelpful (?).

- The application shouldn't trust the network, shouldn't trust the hardware, and shouldn't trust the software (OS, virtual machine, ...).

Local defects do impact remote processes and vice versa. A process's understanding of the remote outside world might not be true in reality.

- Anything can and does happen in a production environment. The "impossible" is possible.

Single logical brick -¿ single chain design decision ... was mostly useful? One weakness might be making migration easier?

Reliance on Erlang distribution -¿ max. size of single cluster -¿ short term easy of implementation but long term unhappiness

References

[1] Q. AB. Quickcheck property-based software testing tool. URL `http://www.quviq.com/`.

[2] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *In Proc. of PODC*, pages 398–407. ACM Press, 2007.

[3] Ericsson AB. Erlang/otp documentation. URL `http://www.erlang.org/doc/`.

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

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

[6] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server, 1999.

[7] R. D. Prisco and B. Lampson. Revisiting the paxos algorithm. In *In Marios Mavronicolas and Philippas Tsigas, editors, 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.

[8] Squid. Squid http proxy. URL `http://www.squid-cache.org/`.

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

[10] 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.

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

[12] M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well-conditioned, scalable internet services, 2001.