# Aergia: Haskell Distributed In-Memory Key-Value Store

Vamsi Chitters, Colin Man, and Nathan James Tindall
*Department of Computer Science, Stanford University*

Submitted: 18 March 2016

## 1 Introduction

Distributed database systems have become increasingly popular in the past years and are used widely throughout the information technology industry. They run across multiple nodes and by virtue of their decentralization, provide high reliability, speed, security, and cost effectiveness. In this paper, we present *Aergia*, a Haskell implementation of a distributed in-memory key-value store running on a single coordinating (master) server and any number of worker nodes.

*Aergia* is a data-management system that stores the key-value store in main memory, while relying on features such as logging and checkpointing to remain safe against systemic failures. This is in contrast to traditional on-disk database systems. Since there is no need to constantly read from a file system, in-memory database systems can carry out functions an order of magnitude faster and necessitate lower memory and CPU requirements.

In recent years, this has been the clear trend: there are numerous in-house in-memory databases with high impact, such as *Redis*, *RocksDB (LevelDB)*, some of which are used in popular applications such as *Google Chrome*. Why? Memory has been getting cheaper over the last few years and latency SLAs have become more stringent for services.

Evidently, there is a need for such systems, but most implementations are done in languages such as Java and C++. Our goal was to take advantage of a friendly functional lanaguage such as Haskell, which offers strongly-typed semantics, lazy evaluation, IO operations, and simple yet powerful concurrency primitives. As a future extension of the project, we hope to determine how this implementation compares (varying data access patterns) with Key-Value stores implemented in other languages and carry out necessary optimizations.

## 2 Assumptions and Guarantees

There are some key assumptions that are worth stating before a detailed discussion on system internals.

- Master (coordinator) cannot incur any failures (though our system as it stands has the data integrity to recover from such a malfunction).
- Log files and checkpoint files associated with workers are never corrupted at any point. These are vital when restoring a worker to the state prior to a failure.
- Memory does not corrupt when the executable is running.
- Workers are not permanently down (come back in a reasonable time). This assumption is flexible: our implementation of *Aergia* could be modified to accomodate permanent loss of a node.

The system, under these assumptions, provides the following guarantees:

- Communication takes place via the two phase commit protocol (explained below) and as a guarantee, operations on the store are atomic (either fail or succeed completely) without external effects.
- Data storage is persistent and replication is used for fault tolerance. In other words, loss of a single node doesn't result in the loss of data.
- There is consistency between workers. The two workers that the master communicates with for a given input request will have the same view of the key-values at a given point in time.

## 3 System Overview

The following figure illustrates the high level design of the *Aergia* system.

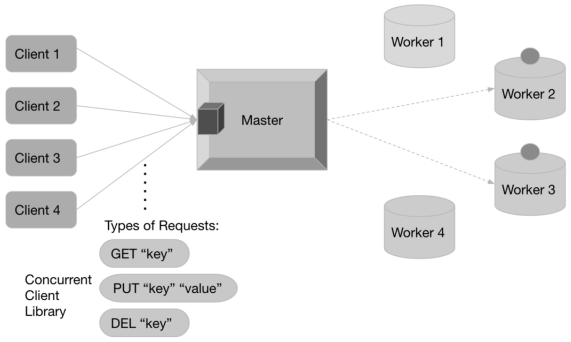


Figure 1: System Overview

The system is designed in such a way that the clients are abstracted away from the workers (and hence, the data store) and are only responsible for communication with one endpoint, the master. Similarly, the workers only need to communicate results with the master. The clients rely on a concurrent client library, `ClientLib.hs` to send either `GET`, `PUT`, or `DEL` requests to the master.

*Optimization*: In the case of GET requests, if the master cache contains the key, it can immediately return the result to the client. Otherwise, the master is responsible for forwarding these requests to the two responsible workers (determining which two workers is done through a process known as consistent hashing) and updating the cache. Thus, *Aergia* is a distributed system with replication factor, N = 2, so a particular partition of the data is stored in two different workers.

Each worker has the key-value store represented in-memory in the form of a `Data.Map.Strict`. The worker periodically checkpoints a serialized representation of the key-value store to a checkpoint file, and logs all actions during the 2PC interaction. These files are used in case of disaster recovery, when the workers come back up after the failure takes place. Failures occur often in any production level system, so it was a focal point and an important exit criteria for the system to be resilient to worker failure at any point in time.

## 4 Protocol

*Aergia* uses the two phase commit protocol (2PC). This protocol is used to ensure overall system consistency between different worker nodes.

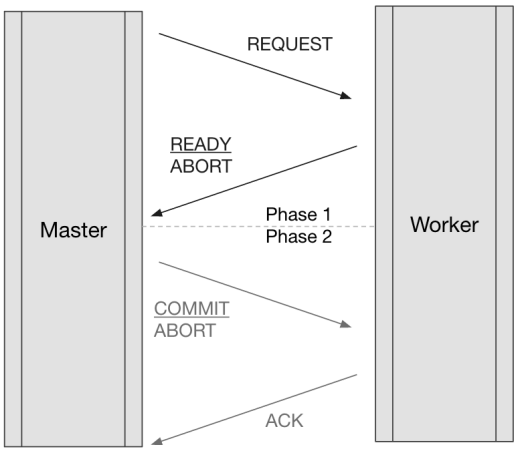


Figure 2: 2 Phase Commit Protocol

- **Phase 1:** The master forwards an incoming request to the two appropriate workers (determined by consistent hashing). It subsequently expects to receive a `READY` response from both of the workers that it communicated with. If either of the workers do not respond within a timeout threshold of 1 second, the master will treat the worker response as an `ABORT`. This is accomplished by running a timeout thread indefinitely that sleeps for a second, wakes up, and then polls the state of transactions that are currently being handled by the master and aborts any of those that have exceeded the timeout threshold.
- **Phase 2:** Based on the response the master deduces from Phase 1, the master will send both the workers either a `COMMIT` or an `ABORT` message. The workers then carry out the appropriate action and `ACK` the master, as illustrated in Figure 2. In the case that the worker dies before it is able to receive the decision or `ACK` the master, the master will repeatedly send the decision until the worker rebuilds itself and completes the pending action. The same timeout thread discussed above is involved in resending the decision repeatedly to the worker that is down. One salient facet of this process is the use of *exponential backoff* to determine the frequency of resending the master decision. The timeout wait period is multiplied by two each time there is no response from the worker. This reduces unnecessary network usage that can be used to service other requests instead.

## 5 Caching

Caching is a system performance optimization that has strong implications on GET request la-

tency. Multiple clients communicate with the master in a given messaging format (KVMessage) using a concurrent client library. The master makes use of a set-associative LRU cache to serve GET requests without involving any of the workers in the case of a cache hit. This plays well with a read-heavy client data access pattern that queries certain keys very frequently. In the case of a cache miss, the master coordinates with the respective workers to retrieve the response, and updates the cache. It is worth pointing out that PUT and DEL requests are always forwarded to workers via 2PC protocol and require a cache update every single time.

## 6 Concurrency

The concurrency architecture of both the master and worker nodes is as follows. When the node is instantiated, it creates a single thread on which it listens for new connections. Then, upon receiving a new connection, it forks a new thread that listens on the connection and then writes to a wait-free channel every message that it receives. On the master node, there are $c + w$ such channel writers, where $c$ is the number of clients and $w$ is the number of workers. The threads infinitely read until the writer to the Handle has indicated that it has no more data to send by sending an `EOF`. Additionally, there is a separate thread forked on each node that reads from the channel, examines the message type, and forks an appropriate handler thread. Each thread then manipulates the state of the node appropriately and sends whatever response is required in order for the protocol to proceed. A reference to the state is passed to each new thread through use of the the `MState` monad, as discussed in later sections.

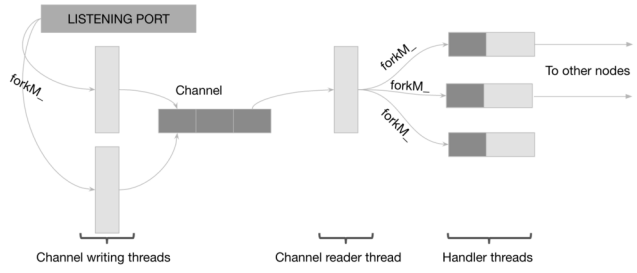


Figure 3: Concurrency Model on Master and Worker Nodes

## 7 Networking

When the cluster is started up, the master node waits until it has a connection with all of its workers, and the workers wait until they have a connection with the master. While the number of workers is assumed to be static in this iteration, clients are established dynamically through the use a simple registration protocol. When the master receives a new KVRegistration from a client, it allocates a new clientId for the client and sends this as its response. The client then uses the clientId to identify itself on future requests. The `CEREAL` library was used for serialization and deserialization of objects.

## 8 Persistence

Each worker has its own partition of the key-value store represented as an in-memory map. During worker rebuilding, the log in conjunction with the latest checkpoint is used to restore to the state prior to worker death.

1. **Periodic checkpointing:** Each worker takes a 'snapshot' of its in-memory key-value store every 10 seconds, as part of a checkpointing process. It serializes the store to a `ByteString.Lazy` representation, acquires a `FileLock` on a local checkpoint file, and writes the content to the file. When a worker dies and comes back up, the worker first rebuilds an in-memory representation of the store using the latest snapshot stored in the checkpoint file. It then relies on the log file to complete pending transactions, as described in the next subsection. While our implementation does not truncate the log file during the checkpoint, the logic required to keep the log from growing to infinite size could be implemented in future work by using a two-phase checkpointing procedure (Koo, Toueg, 1987). An additional improvement to the checkpointing logic would be to write the checkpoint to multiple file shards, rather than to one enormous file.

2. **Logging:** The table breaks down what each worker logs during the two phases of the protocol. Note: there is no need to log in the case of `GET` requests.

| Phase | Request | Log |
|---|---|---|
| 1 | PUT | `READY txn_id key newval ts` |
| 1 | DEL | `DELETE txn_id key ts` |
| 2 | PUT/DEL | `ABORT txn_id ts` |
| 2 | PUT/DEL | `COMMIT txn_id ts` |

During the 2PC interaction, if a worker dies

amidst the life of a transaction, when it comes back up, it can utilize the log to decide what action to take subsequently. There are a few different cases to consider when the worker is reading the log to rebuild. Our scheme iterates through the log in ascending timestamp order (as they were entered into the log).

(a) A particular txn_id that has `READY` or `DELETE` associated with `COMMIT`: in this case, we update the store accordingly (either add a key-value pair or delete an existing key-value pair). However, it might not be the case that the last commit written in the file for a particular `KEY` is the most recent request that was issued by a client. This would not be the case, for instance, if `PUT A B` was issued before PUT A C, but the `COMMIT` for the second message was written before the first. To resolve this, as part of the rebuild logic we keep track of the time that items are added to the map, and verify that the timestamp of the new entry is greater than that of the existing entry (read before a write). If this is not the case, then the `COMMIT` is ignored. txn_ids with `ABORT` messages are also dropped. Identical logic exists for updating the in memory map when the worker is handling normal requests.

(b) A particular txn_id that has `READY` or `DELETE` with no `ABORT` or `COMMIT` pairing: in this case, the master will repeatedly keep sending the decision again and again, so based on the master decision, the worker executes that action and sends a `ACK` back to the master. Note: the log is also updated accordingly in both the cases.

## 9 Consistent Hashing

Each key is stored using 2PC in two workers, as shown in the figure below. Given key, k:

$\texttt{workerId}_1$ `= hash(k) % numWorkers`
$\texttt{workerId}_2$ `= (workerId`$_1$ `+ 1) % numWorkers`

where hash is `HASH.hash32` (from the farmhash package).

The configuration described above is known as a 'ring' given the nature of the output range of the hash function. Each worker has a worker id that represents its position on the ring. When a request is forwarded to the master, it hashes the key to yield the position of the first worker on the ring. Subsequently, the algorithm walks the ring clockwise to find the first node with a position larger than its current position.

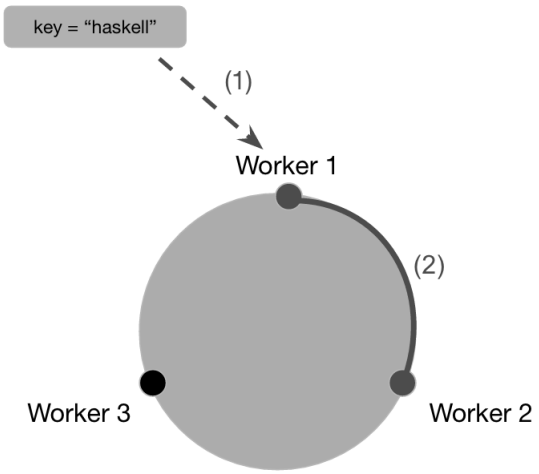


Figure 4: Consistent Hashing

Hence, a given node is responsible for the region between it and its predecessor node. As a mechanism for achieving overall system load balancing, this method allows the departure of a node to only affect its immediate neighbor (not every other node in the ring).

## 10 Implementation Details

We used the `MState` monad on both the master and worker nodes. This allowed us to give each thread reference to a modifiable state. This implementation was a result of several iterations of other Haskell concurrency primitives (`MVar`, `ReaderT`, etc.). Using `MState` made implementing functionality such as timeouts and checkpointing much more straightforward. The constructor for `MasterState` is on the subsequent page.

One benefit of programming in Haskell was that having designed the protocol messages as well-defined type interface (seen below), we saved a lot of time since the compilation guard ensured that methods weren't exposed to types that didn't meet their signature requirements. This gave us better intuition while reasoning about control flow of each node.

```haskell
data KVMessage = KVRegistration { ... }
               | KVResponse { ... }
               | KVRequest {  ... }
               | KVDecision { ... }
               | KVAck { ... }
               | KVVote { ... }
  deriving (Generic, Show)

data MasterState = MasterState {
                   -- Socket on which this node listens for incoming connections
                   receiver :: Socket
                   -- Incoming messages are received in separate threads and written
                   -- to this channel. Another thread reads from this channel and
                   -- forks handlers that then manipulate the MasterState and
                   -- send appropriate responses to other nodes.
                 , channel :: Chan KVMessage
                   -- Static config data instantiated at runtime
                 , cfg :: Lib.Config
                   -- Map representing the current state of the transcations that
                   -- the master is handling.
                 , txs :: Map.Map KVTxnId TX
                   -- Map representing the write handles from the master to each client
                 , clntHMap :: Map.Map Int (MVar Handle)
                   -- Map representing the write handles from the master to each worker
                 , wkrHMap :: Map.Map Int (MVar Handle)
                 }

data TXState = VOTE | ACK | RESPONSE
  deriving (Show, Eq, Ord)

data TX = TX {
                -- VOTE = PHASE1, ACK = PHASE 2, RESPONSE = GET
                txState :: TXState
                -- The set of worker nodes who have either responded with a VOTE
                -- or with an ACK for a decision. This set is cleared when the
                -- transaction is moved into phase 2.
              , responded :: S.Set WkrId
                -- The decision that was made for this transaction (if any).
              , kvDecision :: Maybe KVDecision
                -- The time at which the message was issued by the client
              , timeout :: KVTime
                -- The message itself
              , message :: KVMessage
              }
  deriving (Show)

-- | Channel reading thread. Reads most recent message from the channel (blocks until
-- a message exists. Then forks a handler that manipulates the MasterState appropriately.
-- The MasterState is threaded through each thread through the use of forkM_
processMessages :: MState MasterState IO ()
processMessages = get >>= \s -> do
  message <- liftIO $ readChan $ channel s
  forkM_ $ processMessage message
  processMessages
```

We faced quite a few obstacles as we went about implementing this system. Our initial network implementation was to create a new TCP connection for each message and then immediately close the connection. This allowed us to implement and test the database logic, however, when we tried to scale the number of requests, we ran into a number of machine-level issues (resource exhaustion, etc.). To fix this, we iterated on the existing logic to reuse the same connection (handle) any time a node wanted to communicate to another node. This required mutating the state to hold the handles, since Haskell automatically closes any handle that doesn't have a reference. The documentation and references about the nuances of Socket and Handle programming in Haskell is rather poor (especially about the details of what should be done if the goal is to read from the handle in a streaming fashion). This made getting to a working solution very time comsuming.

Testing that the system behaved correctly was not straightforward. In order to get an end-to-end test working via `Spec.hs`, we needed to figure out how to spawn shell processes to bring up the worker and master locally and thread the respective handles using the `StateT` monad. It was even more challenging to set up the topology (different Haskell executables for master, worker, and client) on the Stanford `corn` machines and test whether the system worked in a distributed setting with multiple workers and clients.

Since we use custom data types to store information about the state and protocol across our files, we found that we often needed to update only one field in the state. Record updating syntax in Haskell is clunky, so we experimented with using lenses to provide clean updates to the state. The introduction of lenses was successful on a standalone branch ("lens"), but we unfortunately did not have the time to merge the full lens functionality into the main code base.

Amidst getting accustomed to Haskell tricks and tools, such as monads, functors and monad transformers, we ended up spending a fair amount of time to get Shared and Exclusive locks to work properly during the logging and checkpointing phases. Identifying causes for race conditions ("Resource is busy" error messages) and handling them correctly was a challenging task. This mainly required adding `bracket` logic that caught the exception and behaved appropriately. The discovery of some useful functions such as `withFileLock` and `withMVar` made implementing race free behavior intuitive after the problem was identified.

## 11 Results

We brought the topology online the Stanford `corn` cluster (16-core Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz machines with 128 GB of RAM). While these machines are shared amongst many Stanford affiliates, we did not observe much variation in latency. The first two graphs (Figure 5 and Figure 6) demonstrate baseline results for average latency of 2000 requests issued as fast as possible from a single client. While these results are positive, further profiling and testing is required in order to determine where the biggest improvements can be gained. We did not have much time to explore alternate implementations of internal map data structures (other than `Data.Map.Strict`), which could drastically affect performance since it is used extensively on every node. Additional areas for performance improvement could include modifying the MState monad to provide atomic references to each element of the state, rather than passing a global reference to the state itself. Since some threads only read from the state, we suspect that this would substantially decrease lock contention.

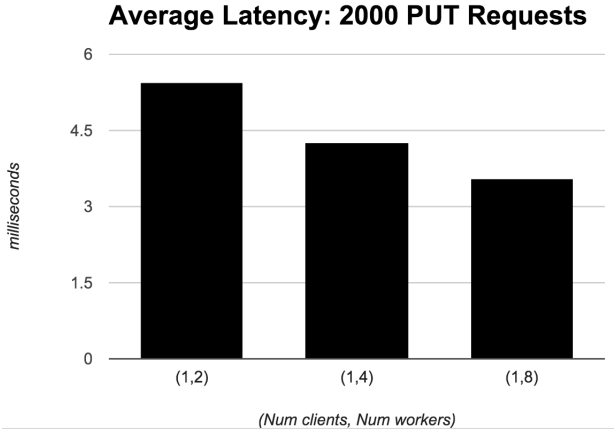


Figure 5

The figure above illustrates that as we increased the number of workers, the average latency of a PUT request decreased. This indicates that the load spread out evenly across the workers as more of them were added, which is a positive sign indicating no significant bottleneck.

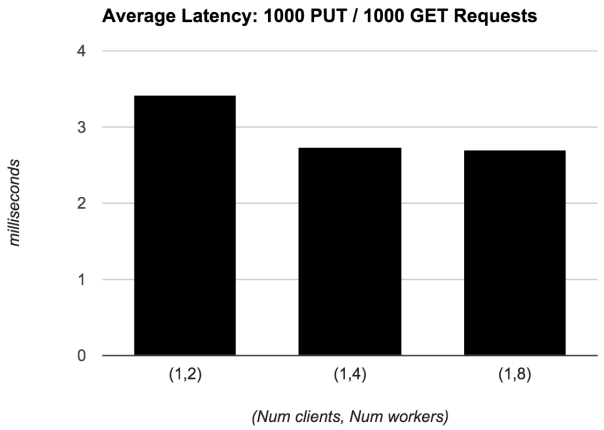


Figure 6

In order to get a sense of performance with respect to different user data access patterns, we decided to test 1000 PUT requests, followed by 1000 GET requests (as illustrated in Figure 6). We see a similar trend: increasing the number of workers (while holding number of clients constant) is inversely proportional to the average latency in milliseconds. Since GET requests are less intensive than PUT requests, it makes sense that there is a slight dip in average latency.

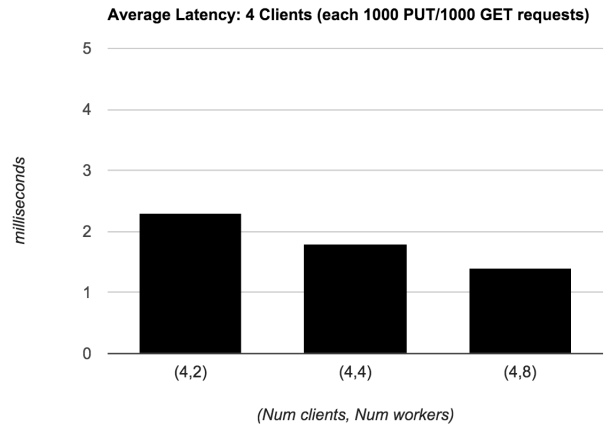


Figure 7

Finally, we decided to increase the number of clients, while doing a parameter sweep on the number of workers. Figure 7 illustrates the following scenario: 4 clients each performing 1000 PUT requests, followed by 1000 GET requests, so in total 8000 requests are sent to the master. We see a positive trend similar to the other graphs displayed above, but we do want to remark that when testing this scenario, we noticed that one or two of the clients observed some timed-out requests. We imagine there may be some sort of master bottleneck, so in order to scale the system further we would need to rectify this pain point in permit higher throughput.

## 12 Conclusion

To conclude, the development of *Aergia* has certainly been a learning experience. After overcoming the initial struggles that Haskell presents to its programmers, we began to respect and appreciate the merits that it has to offer system design.

The system, given the amount of time we had to implement, performed much better than we initially expected. On the whole, judging from our performance benchmarks, it is clear that the system is very concurrent and scales well for the most part. There are obviously numerous optimizations we can still make by using a profiler to isolate areas for improvement (for example, identify where locks are being held for too long unnecessarily and there is high contention) and techniques we can apply to reduce any systemic bottlenecks.

In comparison to the codebase of *LevelDB* (open-source), for instance, Aergia's code seems to be much easier to read because it is implemented in Haskell (albeit it has significantly fewer features), while still offering lots of the overall concurrency benefits.

### Acknowledgments

We would like to thank David Mazières and the CS240H Staff for feedback on the design/implementation and guidance on general systems theory.

### References

[1] R. Adnan, "Why are distributed databases becoming so popular?", `https://www.atlantic.net/blog/why-a-distributed-database-is-used-and-types-of-distributed-data,` (2014).

[2] R. Koo, "Checkpointing and Rollback-Recovery for Distributed Systems", `IEEE Journal` (1987).

[3] S. Gulati, "`https://dzone.com/articles/introduction-to-redis-in-memory-key-value-datastore`", *Database Zone*, (2011).

[4] I. Stoica, "`https://inst.eecs.berkeley.edu/ cs162/fa13/phase4.html`", (2013).