`basementdb`

Key-value store with eventual consistency without trusting individual nodes

https://github.com/Spferical/basementdb

1. Abstract

`basementdb` is an eventually-consistent key-value store, composed of a simple key-value application implemented on top our implementation of the Zeno protocol (`xeno`), an eventually consistent byzantine-fault tolerant replicated system. Most of our effort was centered around `xeno`. Clients can submit requests to `xeno` that are either "weak" or "strong". Servers respond to weak requests immediately with a speculative reply, which represents the current state of the server; otherwise they reach consensus on the request. Strong requests are linearizable with respect to each other. While other Byzantine Fault Tolerant (BFT) protocols can become unavailable if even a small number of the replicas go offline, Zeno is able to handle these outages.

2. Approach

We implemented several components in Zeno. Given more time, we would like to add the rest of the Zeno protocol to `xeno`, but as it stands now, the system is functional. More specifically we have implemented the following:

1. Our system can receive requests from clients and send messages to other servers over TCP. The original Zeno implementation used UDP for most operations (other than the merge procedure, which was omitted from this implementation). We opted to use TCP to simplify our design, although we cannot rely on the invariants granted by TCP (reliability and ordering) because our outgoing message buffer may not necessarily be sent if the server goes offline.
2. Our system can verify messages sent by clients and other servers. The Zeno protocol uses public-key cryptography to vouch for the authenticity of messages. We also use public-key cryptography in a similar fashion, through Rust bindings of `libsodium`.
3. Our system can receive both strong and weak requests from clients, reach consensus, forward them to the application, and then respond to the client. Unlike the paper's implementation, we do not yet handle applications that have non-deterministic behavior (initialized with a seed), but that could be easily added.

4. Our system can continue operating as long as $\lceil \frac{N+f+1}{2} \rceil$ replicas are online. Technically we can continue operating even if there are $f+1$ replicas are online, assuming that the submitted operations are weak, although we have not implemented the procedures in Zeno that detect and merge concurrent histories. Thus, we can really need $\lceil \frac{N+f+1}{2} \rceil$ replicas to be able to make progress.
5. Our system can elect a new primary if the primary goes offline or behaves maliciously. This is a product of Zeno's view-change protocol.

In addition to the merge procedure, we also do not support Zeno's form of snapshotting, which is referred to as "garbage collection".

3. System Properties

3.1 Liveness

The Zeno paper mentions two liveness properties that are important for us.

1. Assuming that there are $f+1$ servers that eventually will get every message sent between them, then every weak request that they receive, if that the client is operating correctly, will be weakly complete.
2. Assuming that there are $2f+1$ servers that eventually will get every message sent between them, then every weak request that they receive, if that the client is operating correctly, will be strong complete.

"Weakly complete" is the claim that every operation that came before this request has already *executed*, not necessarily that they have been committed or are even consistent. "Strongly complete" encompasses our definition of linearizability.

3.2 Safety

Zeno is able to detect and merge concurrent histories. Our implementation of Zeno does not encompass this functionality; thus the safety guarantees regarding divergent histories in the original paper are not relevant.

4. Design & Implementation

5.1 Why Rust is a good language for distributed systems

All of the code for this project was written in Rust. Rust is an ideal language for distributed systems for several reasons:

- Rust is safe. For security applications, safety is far more important than performance. By design, many of the bugs that occur in memory-managed languages *cannot* occur in safe Rust. Rust disallows derferencing a dangling pointer, prevents mutable pointer aliasing, and enforces a stronger form of const-correctness. Rust even prevents some problems in languages that are not described as memory-managed. Rust cannot have null pointer exceptions – functions that may not return a value return an Option type, and Rust forces the programmer to take account of this property. Memory is not default initialized; the programmer must set the memory before it can be read – eliminating problems from default constructors and from uninitialized memory. Rust's powerful type system also enforces safe casting between types and throws exceptions on overflow (in debug mode). The ownership semantics make data races over a variable impossible, thus making race conditions more difficult to accidentally introduce. Perhaps most importantly, there is almost no undefined (or platform-specific) behavior. These protections prevent many security bugs from ever being compiled in the first place.
- Rust has a good paradigm for thread safety. The notion of data ownership carries over well to concurrent programming, forcing the programmer to consider which threads have access to certain resources. All mutable objects shared between threads must be enclosed in a mutex. The compiler must be able to reason about the lifetime of objects given to spawned threads.
- Rust is fast. The language is primarily focused on zero-cost abstractions – there is no garbage collection. Rust compiles directly into LLVM, and can benefit from all of the optimizations that LLVM performs. Benchmarks have shown that Rust code has comparable performance to analogous C code.

4.2 System components

In order to implement Zeno, we had to create/use several components that Zeno relies on. In particular, the protocol required to send signed message over the network, the means by which requests are authenticated, and the hash digest function.

The rust crate `serde` is used to serialize our messages and send them over the network. In particular, we serialize them into a binary format, convert to big-endian, and then base64-encode the result. This problem is made somewhat more complicated by the fact that some structures, such as the commit certificate (see 4.3), are difficult to compare (e.g set equality, hash map equality, etc.). To compare sets, we define an ordering on the elements contained within, insert them into a vector, and then perform a stable sort on it.

We used libsodium's implementation of the Ed25519 signature system to sign our messages. For ease of use, we created a `Signed<T>` object which allowed arbitrary serializable objects to be serialized and then signed. The digest hash function used is SHA-256; objects are first serialized, then hashed. Replicas maintain a hash-chain digest h of the requests they have received (see section 4.3 for an example).

Every replica keeps track of the highest sequence number that they have executed, n. They also keep track of every Order Request they have received from the primary, in the order specified by the primary.

4.3 Request flow

Let us consider the following sequence of events:

1. The client sends a request to all replicas (including the primary), signed with its private key.
2. The primary receives the request from the client, and then broadcasts a Order Request message to all the replicas in the cluster. Before doing so, the primary checks that the timestamp for the client is correct – i.e that the request serial number is one greater than the previous request. The primary also increments n, the largest sequence number. The Order Request message includes the hash digest of the request.
3. The replicas all receive the Order Request message from the primary. Then, the replica waits for the client to send it a request before continuing. If the message comes from the right view, it computes the next value in the hash chain – $h'_{n+1} = D(h_n, D(r))$, where r is the request currently being considered and D is the hash digest function. It then increments n, like the primary has already done, and submits the operation specified in the request to the application. If the client was issuing a weak request, the replica replies to the client immediately with the response from the application. If it is a strong request, then the replica sends a commit message to every other replica. When the replica receives at least $2f + 1$ commit messages, it stores a commit certificate consisting of all of these commits, stores it locally, and then replies to the client with the response.
4. The client returns if it gets a weak quorum (i.e at least f+1) of valid (signed) responses, assuming that the request was weak. Otherwise, the client waits until it gets a strong quorum (2f + 1) of valid responses.

There are other steps involved in sequence number assignment, such as the Fill Hole procedure, that are omitted from this paper for the purposes of brevity.

5. Evaluation

We evaluated Zeno's performance much like any other BFT system since PBFT (Practical Byzantine Fault Tolerance), by computing the average request latency under different conditions.

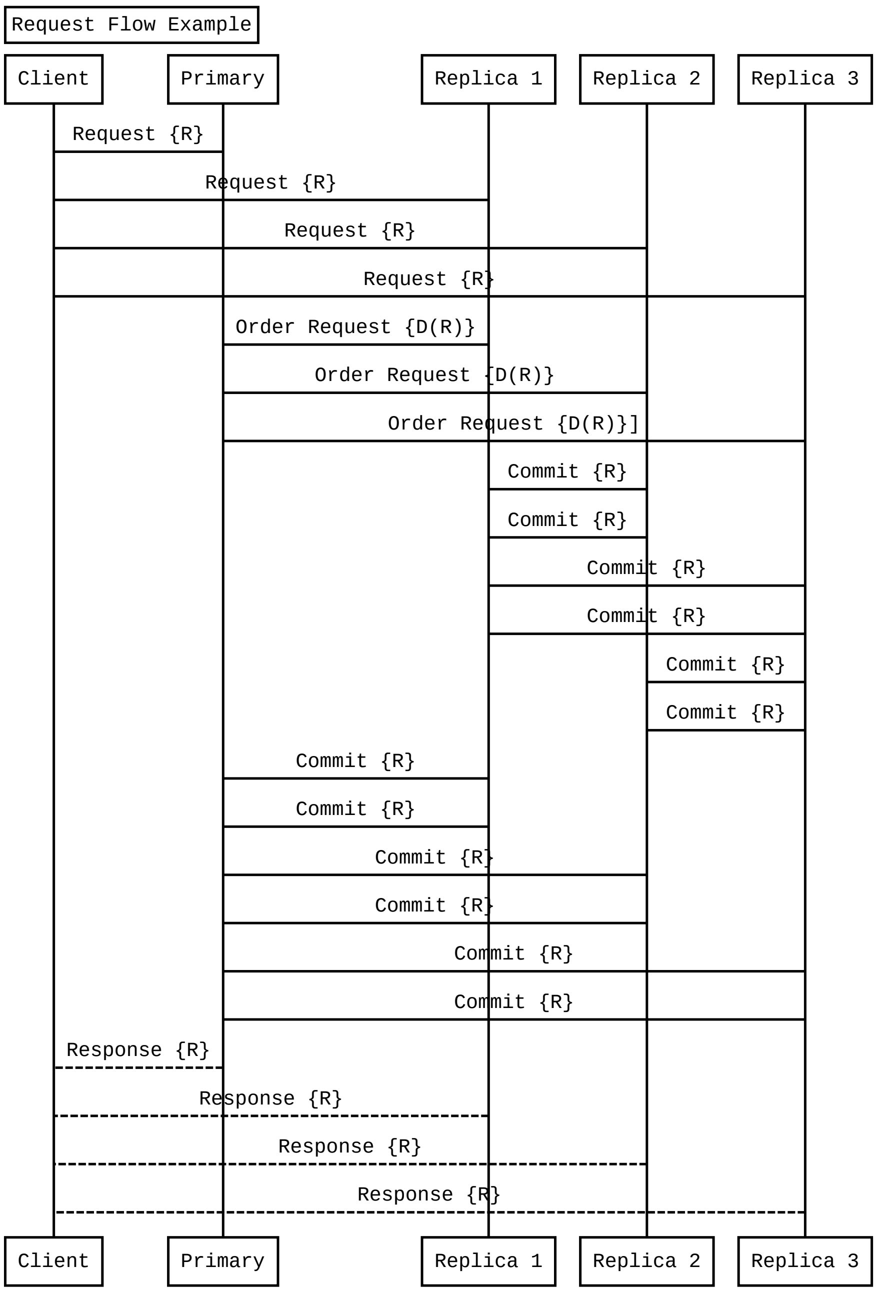

Figure 1: Request Flow for Strong Request

In particular, we tested the PUT operation into a simple Counter application. 100 trials were performed, with one primary and three replicas, on an Intel i7-4712HQ with 16 GB of memory. Each test involved a single machine going offline, either the primary or one of the replicas. Here we see the effects of the view-change protocol becoming active in the event that the primary goes offline. The replica going offline has little impact on the request latency, as the client just receives enough responses from the other nodes anyway.

Faulty	Time to reply	Standard Deviation
none	2.12 seconds	$< 10^{-2}$
primary	5.95 seconds	1.73
replica	2.14 seconds	$< 10^{-2}$

We have also created a suite of Byzantine fault tolerance tests, mimicking a variety of behaviors that a broken, compromised, or malicious node could exhibit. The BFT tests run in parallel to the main Zeno component, occasionally mutating the state in harmful ways. While there certainly are malicious behaviors that are not encompassed by these tests, we are reasonably confident about the safety of our design. For the first and the third tests, we also check that a sufficient number of faults would indeed prevent the system from functioning correctly.

The first test modifies the hash chain. It performs the following operations at random – removing the most recent hash, inserting garbage hashes, and modifying the most recent hash. This test effectively simulates the effect of servers sending each other faulty hash data. As expected, performance degraded, but not much.

Faulty	Time to reply	Standard Deviation
replica	3.60 seconds	0.95

The second test modifies the "pending commit" map. This map contains commit messages which have been received, but not been matched with a corresponding Order Request messages. Given that commit messages arrive *before* the corresponding Order Request (note that this is entirely possible even under TCP), this test simulates the effect of randomly dropping some commit messages. Interestingly enough, this test did not make much of an impact on the performance of our system, although we were able to demonstrate that a sufficiently large number of nodes with this fault were able to stall progress.

Faulty	Time to reply	Standard Deviation
replica	2.13 seconds	$< 10^{-2}$

The third test just sets the value of n to zero. n is the highest sequence number that we currently know about. This test has a variety of consequences for sequence number assignment, and should make a node temporarily unavailable. In particular, compromising the primary this way seems to greatly slow down the system.

Faulty	Time to reply	Standard Deviation
replica	4.18 seconds	0.39
primary	6.43 seconds	0.92

6. Conclusion

Much of this project was spent reinventing simple TCP networking components – methods for sending messages to clients and waiting on a response. We effectively implemented a kind of "futures" for most of our network code.

We had far fewer issues debugging `xeno` than our Raft labs during labs 2 and 3, despite the complexity of the Zeno protocol. There are two plausible explanations for this behavior – either our tests are not comprehensive enough, or our programming process for writing `xeno` was better than our process for implementing Raft in the labs. We suspect that using Rust has reduced the number of bugs that we have had to deal with.