A Secure Replicated Name-Profile Store
6.824 Final Project

Andres Erbsen Daniel Ziegler

May 9, 2014

1 Problem

Many applications rely on some form of directory service for connecting human-meaningful user identifiers (names) with application data associated with that user. When trying to provide security, the lack of a sufficiently trusted directory can easily become bottleneck. We previously developed a solution for having a group of servers efficiently vouch for the value assigned to some particular name in a shared directory. The goal of this project is to finish the protocol using which the servers can update this shared state and maintain consistency in presence of arbitrary server and network failures, as long as at least one server is not malicious. While we built a prototype that functions well in good network conditions during IAP, it would get stuck after certain patterns of network and server failures. Furthermore, the security audit protocol requires that any correct server must not engage in inconsistent behavior even if other servers are malicious; this limitation makes migrating the same state to a new set of servers very complicated. When we had to do this manually, it took as the better part of a day, and manually controlling the timings will only get harder as the number of servers increases. Therefore, we build a reconfiguration protocol that adheres to the auditability constraints.

2 State machine replication

Unfortunately, a replicated log managed using a series Paxos or instances of Raft does not satisfy our requirements. As we saw on quiz 2, two completely correct Paxos servers can decide on a different value if a third server is not following the protocol. PBFT and other byzantine fault tolerance protocols would not admit that case, but in every protocol we are aware of, a majority of incorrect servers could cause a correct server to be inconsistent with other correct servers. For the security-critical application we have in mind, this is not acceptable. There is a related impossibility result that says to have a replicated state machine that is correct with probability 1.0 in presence of f byzantine faults one needs to have $3f + 1$ servers. However, we are satisfied with being correct with probability at least $1 - 2^{-128}$ and therefore this result is not applicable. Still, the problem in its full generality seems to be very difficult and while we think that a solution is probably possible, we do attempt to achieve one right now. Instead, we allow for loss of liveness during failures; specifically, we permit

write requests to be denied if all servers cannot communicate to each other. This may seem like a huge simplification, but it is far from sufficient to solve all issues we are facing. Note that we require both service consistency and auditability in the presence of failures.

3 The protocol

Changes to the user directory happen in discrete rounds: at regular time intervals (currently every $\Delta t = 3$ seconds) the servers propose changes and apply them in lockstep. We use a verified broadcast primitive (described below) to ensure that all servers receive the same set of requested changes and the algorithm for handling them is deterministic. The physical analogy of verified broadcast is a public announcement: everybody learns what the announcer has to say and can be sure that others heard the same thing. In computer networks allowing only point-to-point communication we can emulate this using a two-phase protocol: first the announcer broadcasts the message, then every server broadcasts an acknowledgment of what they received from the announcer. In our system, all n servers announce exactly one set of changes $\Delta_1 \dots \Delta_n$ during each round, so we can group each server's acknowledgments of all messages it received into one message. Furthermore, as just the equality of the sets of announcements received by different servers is important (not the actual contents), we can sign a cryptographic hash $h(\Delta_1 \parallel \dots \parallel \Delta_n)$ of all received announcements in an acknowledgment instead of the announcements themselves. The consensus protocol can be seen in figure 1. Encrypting proposals is needed to prove fairness of the treatment of clients and is irrelevant to consistency.

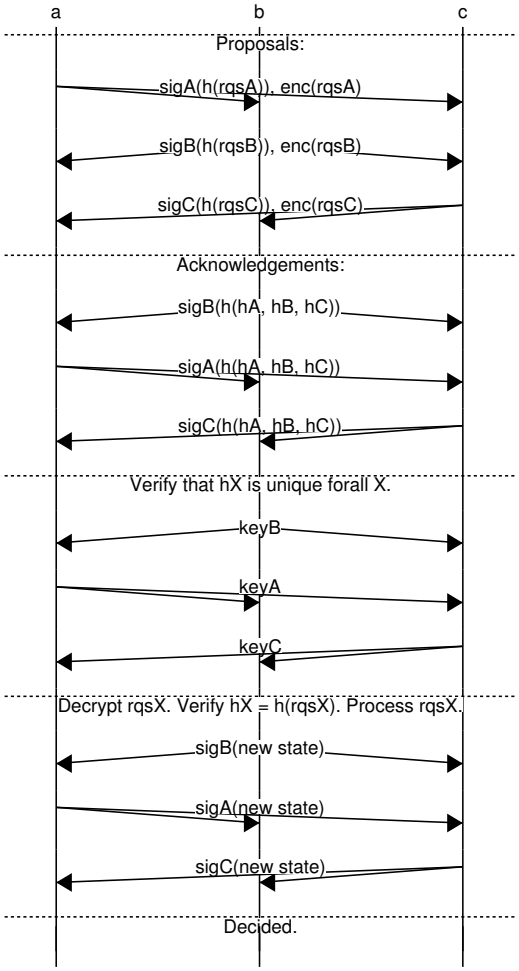

Figure 1: Consensus protocol

3.1 Message resending and crash recovery

We do not use a RPC abstraction. Instead, a server tries to keep at least one TCP connection open to any other server, which it uses to send messages in our protocol. Message handling is idempotent: duplicate messages are harmless. Every time a new connection is established, all messages that the other server has not reacted to are resent. To demonstrate the correctness of this approach, consider a strawman system that would resend all messages it ever sent every time it establishes a new connection. It would be inefficient, but trivially correct. However, we know that if server A has received from B a publish for round i, B must have received all messages it needs for round $i-1$, because otherwise it would not have stared round i. Therefore, it is safe not to resend its messages. While this dependency between consecutive rounds can limit the speed at which our system can commit operations, it does not limit the actual performance because to execute an operation from round i, we would have to wait for $i-1$ to complete anyway.

3.2 Reconfiguration and auditability

Unlike in Paxos, a server only acknowledges one set of proposals during a round. If there were only two correct servers and an unknown number of incorrect servers, and a correct server accepted two different sets of proposals (maybe before and after a crash), the adversaries could present either one to the other correct server. Therefore, we must require that acknowledgments are final. This creates an obstacle for reconfiguration: the obvious mechanism of stopping all the servers, changing their configuration and restarting them is no longer guaranteed to work, because once a server has acknowledged some servers' requests during one round, it cannot acknowledge any new server's requests during that round. Just doing it anyway would result in a situation where there are two signed acknowledge messages from one server for the same round, and we have defined this to be bad. To make reconfiguration work without protocol support, one would have to stop all servers when

- all of them have started the same rounds
- all of them have decided the same rounds
- no acknowledgement messages for any other rounds have been sent.

While we managed to do this with 2 servers controlled by us, it involved quite a bit of luck and would not be feasible when the servers were run by loosely coordinated parties (a situation that would be advantageous for security). Fortunately, an approach similar to the online reconfiguration mechanism often used with Paxos can work here: Since the servers are mutually untrusted, each server operator has to input the new set of servers separately, and then their server will start proposing it every round. Once all the servers propose the same updated configuration during a round r, they agree to switch to the new configuration in round $r+3$. Each server finishes processing all rounds before $r+3$, connects to the new set of servers and resumes processing round $r+3$. It would not be possible to reconfigure in $r+1$ or $r+2$, since processing for those rounds could already have been started concurrently, but $r+3$ starts only after r is finished.

3.3 Persistence

To ensure that each server issues and acknowledges only once, these actions need to be committed to persistent storage. The data in client requests and the final signed "decided" messages are required the lookup protocol, so they are also saved persistently. We chose a postgresql database to store the relevant messages. This was probably not the best choice – while it enabled us to develop quickly and spend less time thinking about atomicity of modifications to the local state, it also quickly became a performance bottleneck (see the Implementation section). In addition, we saved the Merkle prefix tree which summarizes the set of name registrations as a persistent compact tree on disk which supports all-or-nothing atomic writes by being copy-on-write: Changing the tree creates new nodes rather than modifying existing nodes, and new versions of the tree are referenced by the ID of the new root node. Then, this ID is stored atomically in the database.

3.4 Lookups

While we accept that we have to give away availability for writes to achieve consistency in presence of incorrect servers, we would like to serve reads even when some subset of the servers is partitioned or crashed. Therefore, modeling reads as no-op write transactions is not possible. Instead, we adopt an approach similar to Spanner: We fix an amount of time ϵ which is greater than the maximum time difference between the client's clock and all server clocks. Correct servers respond to reads with the results of the latest round; every round has a signed timestamp from every server. However, a client has to accept timestamps up to ϵ ago, since its clock could be ahead. Since its clock could actually be *behind* by ϵ, this means the write could actually have happened an additional ϵ farther in the past. In all, when a client receives a timestamped read, it is guaranteed to be at most $\Delta t + 2\epsilon$ old: it will take at most Δt for the write to be timestamped by the servers, the client will accept timestamps up to ϵ old according to its clock, and its clock may be ϵ behind the server time.

As in Spanner, we can also use this bound on the age of the read to provide a notion of external consistency: When a client performs a write (i.e. an updated name registration), it should only complete once all reads are guaranteed to include the newest value, i.e. $\Delta t + 2\epsilon$ in the future. While our system does not utilize GPS or atomic clocks, high time accuracy is not required for correctness. We argue that write latency is relatively unimportant in a user directory, because the writes or batches of writes are issued by humans and the clock inaccuracy of a NTP-managed computer is negligible compared to the time it takes for somebody to type an username. Read performance can matter, but as the reads do not have to go through the consensus protocol, they can be served quickly.

3.5 Implementation and performance

We did our best to keep the consensus-related code separate from application logic, but because all application-level choices a server makes have to be uniquely determined by the inputs, there is a considerable amount of dependencies between the two modules. We have not tested the consensus code separately. The early prototype we started with could do 20 writes per second. To improve on this, we created more in-process state to reduce the number

of database queries. Our current implementation can handle 300 writes operations per second on a machine that can do less than 2000 bytea-indexed postgresql operations per second. For each request, the following database queries are made:

- Determine whether the name being modified is already claimed by some other client
- Determine whether the current client is allowed to claim a new name
- A write to save the operation after it has been decided on
- Modify the lookup consistency proof data structure
- Modify the directory entry the name resolves to
- Set the modification time of the name (used for expiration)

While it is probably possible to combine the last two and apply some optimizations for the others, the performance issues are clearly caused by the poor choice of the storage layer and not by a distributed systems issue. We also expect the current performance not to be an issue in practice: at this rate, all Gmail users could have been registered in less than a month; Facebook's average registration load is 30 times lower.

3.6 Testing

Our server implementation is *validating*; that is, it checks that what other servers do (and occasionally its own past behavior) is consistent with its own view of the shared state. The only correctness bug we have observed so far that was not caught by these checks was in the client. The majority of the issues were found using an automated stress test similar to the "failures, unreliable" cases in the labs. Other property tests are ran while this stress test is in progress. The following processes are ran in parallel:

- 3 servers
- a 100-thread client sending write requests
- A processes that kills (`-9`) a random server at random intervals. The first server is never killed to detect possible bugs that appear when other servers die.
- A process that at random intervals interacts with the kernel traffic shaping system and toggles the network from reliable and fast to high-latency and unreliable.

We also tested that the writes one client makes are observed by other clients after the specified amount of time and that the client code successfully emulates the read-my-writes property. We did not test that a server does not get tricked by a malicious server, because doing so would be very time-consuming and unlike with bugs that cause problems due to bad coincidences, fixing 99.9% of the buggy cases does not mean that 99.9% of the adversaries cannot exploit any remaining bugs, if there are any. All code passes go race detector on go 1.3 (and crashes due to a known go race detector on earlier versions).