HARoost

Arkadiy Frasinich, Istvan Chung, Mark Theng
6.824 Final Project

Introduction

Zephyr is a messaging system used throughout MIT which relies on the Kerberos system for secure authentication. Roost is a web-based Zephyr client made by David Benjamin as part of his master's thesis on Kerberos authentication in the browser. In Benjamin's implementation, the Roost backend runs on a single machine. If that server crashes, clients will be unable to access Roost, and incoming messages will be lost until the server restarts.

HARoost is a highly-available reimplementation of Roost based on the replicated database system RethinkDB. RethinkDB achieves consistency and high availability by replicating data on multiple machines using the Raft consensus protocol. We use the database both as persistent storage and as the sole means of communication between different HARoost instances.

The aim of HARoost is to remove Roost's single point of failure, so that the service remains available while individual server instances may crash or become disconnected. Multiple instances of HARoost on different machines interact with the Zephyr backend to manage user subscriptions and receive messages. In normal operation, incoming Zephyr messages are received by multiple HARoost instances and are replicated in the database. HARoost instances tail the database for new messages and display them to clients. If one of the instances crashes, the other instances continue to receive messages from Zephyr and serve clients.

Zephyr

Zephyr is a messaging system originally designed as an automatic notification system which is today used as a chat system by various members and groups within the MIT community. In Zephyr, clients broadcast and receive messages on *Zephyr triples*, which play the role of channels in similar messaging systems.

Zephyr is itself a distributed, highly available system. Clients connect to a cluster of Zephyr servers through one of a number of Zephyr Host Managers (zhms). We refer to the entire system as the Zephyr backend.

While the Zephyr backend provides the infrastructure to distribute messages from one user to all subscribed recipients, it does not persistently store messages. Clients must store the messages locally in order to refer to them later. Roost solves this problem for individual users by persisting messages in its database so that users may read them at any time.

Design

HARoost's design constraints pair well with the use of a distributed database, which allows for replicating messages and user credentials across all instances. Another crucial feature provided by RethinkDB is the ability for HARoost instances to subscribe to changefeeds of updates to particular tables. This allows the database to serve as a reliable, scalable communication channel between different HARoost instances.

The HARoost backend is logically separated into four components -- the client managers, the websocket managers, the Zephyr listeners and the database. Clients make Roost API requests to the client managers, which query and update the database to fulfill them. The API includes actions like authenticating to the HARoost server, updating Zephyr credentials, subscribing to Zephyr triples, and sending messages. Client managers communicate directly with the Zephyr backend to send messages, logging the outgoing message and replying to the frontend only after the Zephyr backend acknowledges its receipt.

Clients also connect via websockets to the HARoost backend to receive live updates about incoming Zephyr messages. The websocket managers use changefeeds to be informed of any new messages logged into the database so that they can be forwarded to the user.

The Zephyr listeners are in charge of subscribing to the Zephyr backend and receiving messages. They tail the database to be informed of any new subscriptions or updates to user credentials. When a message is received, HARoost calculates the list of users that were subscribed to the message's destination at the time and persists this information to the database along with the message itself. The former is stored in a table of pairs of message-ids and users, which is indexed by user so that users can quickly retrieve past messages that they have access to. Although this scheme necessitates two separate database inserts for each incoming message, transactional behavior is not needed since the operations are idempotent. The message is inserted first to ensure that in case of failure, the database contains references only to persisted messages.

Challenges

While writing HARoost, we made the decision to write the backend in Go. While the benefits of static type checking and concurrency support made development easier, it also restricted the choice of libraries for some of the functions needed for Roost's function. In particular, our choice of distributed database was heavily influenced by the availability of drivers in Go.

Many of the difficulties we encountered stemmed from interfacing with Zephyr. These were exacerbated by a relative lack of documentation and hard-to-decipher behavior from the Zephyr backend. For instance, one issue we ran into involved the subscription model that Zephyr uses. If we controlled the design of the Zephyr backend, we would ideally opt for a model in which subscriptions are made by a cohort of Zephyr clients, where if one client fails the Zephyr backend will reestablish the connection with another from the cohort. However, the existing Zephyr backends do not support this model, so all subscriptions must be made by each HARoost instance in order to ensure that no messages are missed if some of the HARoost instance fails.

Additionally, the lack of widespread usage for systems such as Zephyr and Kerberos means that development of libraries is similarly uncommon. HARoost uses the zephyr-go library, a reimplementation of the Zephyr protocol in Go that hadn't been developed on for three years. We had to update zephyr-go to be compatible with the latest version of Go, and also translate between the JSON-encoded credentials provided to us by Webathena and the ASN1-encoded Kerberos tickets expected by zephyr-go.

Discussion

HARoost is limited by the capabilities provided by the Zephyr protocol. When a Zephyr client subscribes to a Zephyr triple, it has to provide the Zephyr backend with valid Kerberos credentials. The client only receives messages related to that subscription as long as the client maintains an active connection with the Zephyr backend. If the client loses its connection to the Zephyr backend, it has to re-subscribe all its subscriptions in order to continue receiving messages.

This poses a fundamental problem for the implementation of a highly-available Zephyr client like HARoost. While users provide HARoost with valid credentials at the time of subscription, these credentials expire in about a day. After that, if an HARoost instance loses and subsequently re-establishes its connection to the Zephyr server, it can no longer use those credentials to re-subscribe to existing subscriptions.

Following Roost, HARoost alleviates this for public subscriptions by making those subscriptions with a dedicated HARoost principal instead of the user's provided credentials. However, for private subscriptions, Zephyr's security model requires that the user's own credentials be provided. This means that if all HARoost instances lose and re-establish connection with the Zephyr backend at some point of time after a user's credentials expires and before the user provides HARoost with fresh credentials, there is no way for HARoost to receive messages for that user's private subscriptions. This both impacts HARoost's availability and also utilizes more network bandwidth, as each HARoost instance must connect to Zephyr separately.

The limits imposed on HARoost by the choice of methods exposed by Zephyr highlight the difficulty in composing highly available distributed systems. While both Zephyr and HARoost are highly available systems, the Zephyr API is designed for individual clients and has difficulty composing with another distributed system. This suggests that the analysis of properties of distributed systems such as consistency and availability should take into account how they can be composed and extended to create larger systems.

Bibliography

David Benjamin, 2013. *Adapting Kerberos for a Browser-based Environment*. Master's thesis, MIT. Retrieved from https://davidben.net/thesis.pdf. Code at https://github.com/roost-im and https://github.com/zephyr-im.

Robert S. French, John T. Kohl, 1989. *The Zephyr Programmer's Manual*. Retrieved from http://web.mit.edu/keithw/Public/progman.pdf.

RethinkDB. https://www.rethinkdb.com/

Repository Links

HARoost: https://github.mit.edu/ikdc/haroost

HARoost Client: https://github.mit.edu/arkadiyf/haroost-client