Mylar Distributed DB

Jordan Moldow, Louis Lamia
https://github.com/jmoldow/MylarDistributedDB

Problem

Email is a very common form of communication in today's digital society. Unfortunately, it is also very insecure. Most email services store all the information unencrypted on the server, and an attacker (say, the NSA) who gets access to the server can read all of your personal email.

The Mylar framework, an extension of Meteor, exists to provide application developers with the capability to easily implemented a secure webmail service. However, Meteor is only able to connect with a MongoDB backend, and MongoDB only has out-of-the-box replication support for primary-backup replication. In order for Mylar to truly be able to support an email service, it must have built-in support for a better mode of replication.

Mylar Distributed DB is a Meteor package, compatible with Mylar, which adds support for MongoDB collections that are replicated across multiple servers, as well as application-agnostic support for seamlessly connecting with those servers.

Design Goals

Availability: To a user, being unable to access email for an extended period of time could be catastrophic. However, short outages are usually not an issue and can often go completely undetected. We should optimize MylarMail for the best availability possible without compromising on important correctness or security issues. We should also prioritize quick recovery from outages.

Consistency: To a user, it is often unnoticeable or at worst slightly inconvenient to need to wait a few minutes to read a message originally sent to another server. Thus we can rely on eventual consistency for application data, while making sure such weak guarantees does not introduce any security flaws. We simplify the mechanisms for achieving eventual consistency by noting that in the case of emails, most data about a message is either immutable (i.e. message contents are never rewritten after a message is sent) or results in low user impact if a mistake is made (i.e. accidentally resetting a message to "unread" after it was already read is a very minor issue for the user), allowing us to rely on simple conflict resolution policies for different versions.

Durability: Disappearing emails can be a huge problem for a user expecting to be able to communicate reliably. Thus, we need strong durability guarantees. The sender of a message should eventually know whether their message was successfully sent, and if so,

they should be guaranteed that the message will not be lost later, especially before the recipient reads it. If this cannot be practically guaranteed, there should be a mechanism in place for notifying the sender of a message whose message was lost.

System Overview

The system consists of the following components:

- A set of machines, running the Mylar Identity Provider (IDP).
- A cluster of machines, all running identical copies of the server-side application, which itself is running in the Mylar/Meteor Node.js environment with mylar-distributed-db enabled.
- Client machines that use their browsers to connect to the application servers, then download/run the client-side Javascript application.
- A Go library that provides support for the distributed data storage. There is both a server-side and a client-side library, which communicate with each other and their respective Mylar/Meteor Environments and client application to implement the distributed logic

Server-Side Software Stack

The server-side code is organized into vertical components as follows:
Mylar/Meteor application code: Initialization code, which needs to run when a server is started. Defines Meteor collections, calls the Mylar API to set up encrypted fields, and defines published data sets that clients can subscribe to.

Our package exposes a method called wrap_insert(), which can be called on an application's collections. It redefines some of the standard collection APIs on both the client and the server, so that database modification operations go through our database stack instead of the normal Meteor stack. It also registers an object resolution handler with the collection.

Mylar/Meteor client: Runs the application in the browser, and occasionally communicates with one of the server nodes to get and push data.

Distributed Database Local API: The distributed database server, running in Go, exports GetList(username) (get an ordered list of replicas for a user's data) and CoordinatorPut (communicates with the other replicas to store the data). The Meteor server sends commands to the Go server via a Unix socket.

Distributed Database Network API: The nodes of the distributed database communicate with each other over sockets and a Go RPC API.
Collections API: Each Meteor server listens on a Unix socket for messages from its associated Go server. Upon receipt of an insert() or remove() operation, it will perform the operation, with the conflict resolution handler, and push the change to the underlying database via Meteor's usual API to MongoDB.

Distributed Message Database - Simplified Dynamo

Dynamo struck us as providing a good tradeoff amongst our goals, achieving key/value storage with high availability, durability, and scalability at the cost of relaxed consistency. This is perfect for storing messages in a distributed database. We will adopt the techniques from Dynamo in the following way

Consistent Hashing - We will use the consistent hashing scheme as described in Dynamo to achieve incremental scalability

Simplified Application-level Conflict Resolution - The final write to the MongoDB database happens in the Meteor environment. Thus, application logic is available at the time of the write, and in the face of conflicting copies of data, the application can decide the best way to resolve the problem. All collections using the distributed database register conflict resolution handlers with Meteor. When an insert occurs and a previous version of that object already exists, the handler is called to construct the object that will be written to the database. By default, if no custom handler is given, the application will choose the object with the later timestamp. If a data object is more complicated and requires a more powerful approach, the application developer can implement a vector-timestamp resolution handler, or any other suitable handler that meets the needs of the application.

Sloppy Quorum and Hinted Handoff - We will implement this as described in Dynamo in order to achieve Availability and Durability in the face of transient failures.

Database API:

GetCoordinatorList(username) - Returns a priority list of servers to try to create replicas on
CoordinatorPut(username, message) - Asks a server to store the message and to ensure that nReplicas - 1 other servers store a replica before returning to the client. After a successful CoordinatorPut, there will be nReplicas replicas in the system (where nReplicas is a configurable parameter by the system)