CS 5412 Final Report

Debarghya Das
dd367

Dominick Twitty
dkt36

Introduction

Our proposal was to create a scalable, fault-tolerant system for real-time location updates, with an interface inspired by the "Marauder's Map" from Harry Potter. We wanted clients to be able to push location updates to the system, which would then inform relevant clients of said update via markers on a map. The motivation behind this project was exploration of the real-time cloud. That is, we do not have strong consistency requirements, but we do require the ability to handle a fair number of simultaneous connections, each frequently updating a very small amount of data. This is in contrast with several other cloud computing staples such as "big data" and "social networks", which are designed to deal with large amounts of data relatively slowly.

We have created an application that addresses these requirements. We present an API for pushing and receiving location updates as well as several client implementations including a web app, an iPhone app, and Python testing scripts.

Our service either implements our requirements, or is architecturally structured to easily evolve to meet said requirements. Our application consists of two tiers. The first is an outer, soft-state tier that generates web pages and handles pushing and pulling updates with clients. These outer-tier servers connect to an internal pub/sub network implemented using a Redis cluster. The idea is to allow outer-tier servers to fail with little trouble.

Application

Our application is a Marauder's map. Clients (players) log in with a name and are assigned an internal id. Clients can create new maps or join existing ones. Whenever a client pushes a location update, every map they are a part of is informed, and other players will have their local maps updated. We also support keeping track of the previous locations of players. Clients see the name and location of all other players registered in the map.

System

Our system has three parts: clients, handlers, and brokers. Clients present an interface for users to publish location updates, a map for viewing other's locations, and facilities for real-time reception of new updates. We expose a relatively simple API to clients. They can create new maps and players, query existing ones, look up all the locations on a map for initialization, etc. This is all done through a REST API. Updates are pushed to clients over websockets.

Handlers are the middlemen of the system. They are the only servers that have application-specific logic. Upon a REST request, handlers query a database to either acquire or update the

needed information. Handlers also deal with websocket connections. Upon a successful handshake, a pub/sub connection is established for the duration of the websocket connection. Handlers are therefore soft-state - they store map data off-site, and their in-memory data only relates to efficiently managing connections.

Brokers form the core layer of our application. Handlers register pub/sub connections to a central broker entity, which may be made of one or a cluster of servers. We use channel-based pub/sub - publishers push data to a channel, and the broker repeats that information to all subscribed connections. The broker is a stand-alone service - it knows nothing of our application and handlers don't need to know the specifics of its implementation.

The database and brokers must be able to handle many small updates from a small number of sources very quickly. However, consistency and data loss is not a major concern, especially when data is expected to go stale quickly. Therefore we use an in-memory database and only flush to disk on major events like the creation of a new player or map. The application is also structured such that different broker implementations can be dropped in with little fuss. Therefore, the consistency and fault tolerance of our application is tied to that of our database and brokers.

Our complete schema is a snowflake. Every client connects to a single handler server, and handlers maintain a connection to the broker. Clients need not connect to each other directly, even to themselves. That is, when a client pushes a location update, it makes no assumptions and will wait for the update to come down the websocket before updating its map. Likewise, handlers know nothing about each other, and rely on the brokers for communication.

Implementation

We chose to implement this system by using existing tools rather than build each part ourselves. This approach allows us to make certain assumptions and also lessens the amount of work needed to fit in a semester. We decided to model our system on a web chat application, as there are many similarities between the two applications. We based our research on this for piecing together our application.

Previous Implementations

Initially, we wanted to take a very high level of abstraction, and treat a cluster of computers as a single machine. we attempted to use the Akka framework in Scala, which implements a configure-and-go distributed version of the Actor Model of concurrency. We found that we were making our lives more complicated than necessary, as we were trying to obtain fault tolerance and scaling in a single step. We ditched this for a more modular approach.

Our second attempt involved a "one server per map" mentality, inspired by games such as World of Warcraft. The thinking was that we would not have to deal with cross-server communication if all clients in one map were guaranteed to be on the same server. We ran into the problem of replication - if a server failed, we would need to have another server up and running with the same information quickly. We thought to use chain replication, but the problem was that we would have backup servers keeping up-to-date but not actually handling clients. Furthermore, it would be awkward to allow clients to be a part of multiple maps at once.

Current Implementation

Our clients can be in any language that supports HTTP calls and websockets. As said before, updating the state of the application is done through REST calls, and updates are received in real time via websockets. This design was chosen to allow easy creation of new client implementations. We have a web application that runs on JavaScript, which is our main product. We also have a mostly-functional iPhone application that demonstrates the ability to push periodic updates. Finally, our testing scripts use the same API in a purely command-line application.

Our handlers run on Facebook's Tornado framework for Python. Tornado is a single-threaded, IOloop-based framework with heavy focus on statelessness and asynchrony. The Tornado websockets implementation allows us to asynchronously place a callback function on pub/sub connection. We use a Redis database for storage of game information, which all the handlers connect to. Redis was chosen because it is very fast (it is essentially a huge hash table in memory) and we do not need strong data persistence. Also, Redis allows a simple abstraction layer to a database cluster, which would provide a layer of fault tolerance.

We use Redis again for our pub/sub requirements, as Redis is a well-known and respected pub/sub broker. Upon creation of a map, a new pub/sub channel is created. Upon a location update, the data is published to a channel, and every websocket connection object is registered as a subscriber to the appropriate channel. In order to keep the total number of connections low and actions asynchronous, we handle publishing in one thread and subscribing in another thread.

Extending the Implementation

There is also the issue of clustering the broker network. Currently the broker is implemented on a single Redis instance. However, there exist clustered pub/sub systems that we could drop in to our application given more time. First is Redis cluster, which is in beta stage. This would involve configuration and setup that we did not have time to do, but would still provide us with a standalone pub/sub application. Another distributed broker we looked into was ZeroMQ, which can be used with or without a central arbiter.

Discussion

We believe that the strength of our implementation is modularity and how it was designed in a logical progresion. Remember earlier that we likened our project to a chat engine. The core problem of a chat engine is receiving an update from one use and getting that update to several other users in a timely fashion.

This lends itself nicely to one of two design patterns - push/pull and pub/sub. In push/pull, clients update a central broker, and periodically ask it for new information. We thought this to be conceptually ugly, hence we come to pub/sub, which has roots in asynchrony and interrupts. Rather than having many clients bombard us with requests for new information that may not exist, we simply promise to inform clients when there is new data.

Next we derive the snowflake schema. Our belief is that systems in a vacuum are the easiest to reason about. Thus, our clients should know as little as possible about the internals of the application. This implies the use of a single endpoint (a load balancer). Next, we decided that as soon as handlers know about each other, we have to start dealing with internal consistency. The same is true if handlers keep sensitive information in memory. This leads us to the use of a

data-handling agent, also known as a database. If handlers essentially know nothing about each other, then we must pass off communication to some other service layer. Herein lies the decision to make handlers essentially middlemen to the pub/sub broker system.

Modularity is a strength of this implementation. As discussed previously, we use standard REST and websockets to keep our front-end distinct from our back-end, and allow multiple front-end implementations. Internally, we can change our database and pub/sub broker to suit new needs without compromising other parts of the application.

Experiments

We experimented to attempt to find bottlenecks in our system. The numbers in our benchmarks should be taken with a grain of salt - there are a lot of assumptions being made. For instance, servers may be VMs or bare metal, we have no control over connection times, and the Python concurrency model is dodgy to say the least. We justify our findings where possible and offer our own insights gained from researching this project.

Setup

We found that testing locally did not reflect the speed of the application, as there was simply too much contention for processor time between the app, Redis, and the test threads. As such, we set ourselves up on a Digital Ocean VM with four cores. We commit up to three cores solely to application instances (standalone programs with no cross-talk) and one for Redis to emulate multiple servers. We load-balance between servers using NGinx.

To best emulate many connections (with no processor contention between them) and isolate our findings from client speed versus application speed, we run our benchmark on both a 2-core Macbook Air and an 8-core i7 Dell laptop.

The Benchmark

Our testing procedure is as follows:

- Create n clients. For each client open a thread for sending updates and another for receiving. Make sure everything is synchronized.
- Record the time. Set the pushers to send m random location updates, sleeping s seconds in between each one. Have the receivers log the number of updates they have seen.
- Stop each receiver when it has gotten nm updates.
- A client is done when it has sent all updates and received all updates. We use various synchronization techniques learned in OS to make sure everything happens in the correct order. Record the time when all clients are done.
- Optionally, check the clients for consistency (all players should have the same last position ideally).

Our parameters are the number of clients n, the number of updates per client m, and the wait time s. Let t be the time elapsed between the two recordings. The plan is to get an estimate of the

number of the total number of updates we can push per number of listening clients. The ideal value of t is just sm. Ideally we would run a test with $n = 1000$, $m = 100$, and $s = 1$, to emulate a large game, but we are limited by the processing power of the computer running the test. Therefore, we include a test where both testing machines are run at once.