A Dynamic Low-Latency Chat Server Framework for Distributed Networks

Porom Kamal[1], Seron Athavan[3] and Jean-Sebastien Dandurand[1]

[1, 2, 3] Department of Computer Science, University of Toronto

Abstract

This paper presents the design and implementation of a Dynamic, Distributed Low-Latency Chat Server framework, developed to address the challenges of latency optimization and adaptability in distributed network environments. Using Mininet, we simulated a network comprising multiple clients and chat servers, emphasizing real-time communication efficiency and fault tolerance. Our system dynamically selects the optimal server for client communication, minimizing overall latency by identifying the closest accessible server. Furthermore, the platform adapts to dynamic network conditions, such as traffic congestion or link failures, by seamlessly relocating chat sessions to alternative servers.

Demo

A demo of our system can be found here on youtube.

Introduction

One of the biggest challenges in real-time communications, whether it is chat servers, online competitive games, or video conferencing, is ensuring low-latency and reliable communication while adapting to dynamic network conditions such as congestion, link failures, or topology changes.

Our team wanted to provide a proof-of-concept for a framework which allows real-time communication systems to dynamically optimize server selection and adapt to changing network conditions, ensuring low-latency and reliable interactions in distributed environments.

Our solution provides a plug-and-play framework for integrating servers into a network, enabling the system to maintain resilience against various network and server failures with minimal impact on the user experience.

The Components

First let's look at the different components of our system at a high level.

Central

Central is the brains of the entire distributed system, and is the most important core component. System architects should make it a top priority to ensure that an instance of Central is actively maintained and operational at all times.

At a high level, some of the responsibilities of central include:

- Load balancing across servers
- Client & Server Registration
- Service Discovery
- Network Congestion Monitoring
- Server selection and matchmaking for clients
- Failure detection and recovery

Overall, Central is an immortal, all knowing central server, which knows the following about the system:

- The available Servers
- The registered clients
- The ongoing chat instances
- The latency vectors for a given client to all of the available servers

All new servers and clients that are added to the network must know where Central exists, by starting with Central's IP address as a configured value.

Servers

These are the main servers which facilitate a given activity that clients can engage in (for our proof-of-concept we chose chat). When a server is first started, it will register itself with Central and then periodically send a heartbeat every 3 seconds to Central. From Central's POV a Server is considered offline if it hasn't reported a heartbeat within the last 15 seconds.

With our system, we can operate on the assumptions that servers are feeble beings. They can crash, restart, overheat at any time. Therefore, the only responsibilities that a server has is:

- Report a heartbeat to central every 5 seconds
- Accept connections from clients and then serve them the activity (in our example, chat)

Clients

A Client is a component that represents the end-user interacting with the system. When a client is first started, it will register itself with Central, along with a unique identifier (in our example, we use a username).

The client will have the following background jobs which will run periodically:

- Retrieve a list of available server IPs from Central
- Measure the delay for a ping from the Client to each server, and save this information
- Report the ping delay data back to Central
- Actively listen for messages from Central. More on what these messages are later.

For interactive functionality a client can:

- Send a request to another client to engage in some activity
- View and accept requests that it has received from other clients.

Our System in Action: The Setup

To show how our system behaves, and the different client/server/central interactions which take place, let's take a look at an example (which is provided with the starter code)

The Network Environment

We will simulate our network using MiniNet. This network has been provided in the starter code (network/chat_network.py).

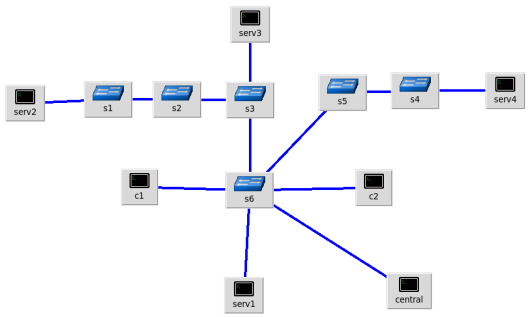

Figure 1: *The Network Environment (Sorry! You might need to zoom in to read the labels)*

Our network simulation contains 4 servers, Serv1, Serv2, Serv3, and Serv4. It contains a single central server and two clients (c1 and c2).

Each of the links are configured with 10ms of delay. The closest server (in terms of ping latency) to both clients is Serv1, with Serv3 being the second closest.

Central Startup

The first service to start will be Central. Central will startup the following sub services:

- A REST API for Client & Server registration and other associated CRUD
- A REST API for Updating/Storing client delay lists
- An open socket connection which listens for client connections. Clients will request a server to connect to through this socket.

All of these different services can be found under the "Central/internal" folder.

Server Startup

Next all of the servers will startup. **For this example, we will assume that Serv1 (the most optimal server) hasn't started up.**

When a Server starts up, and registers itself with Central, it will start the following:

- A job which sends a heartbeat to Central every 3 seconds
- A socket which will listen for incoming client connections
  - In our example, our socket will listen for client connections which will contain a RoomId and their Username.

Client Startup

When a client starts up, it will prompt the user to enter a username. It will then register itself with Central and begin the following background jobs:

- Fetch the available servers from Central
- Ping all of the other Servers, and measure how long it takes. Send the results to Central
- Open a socket connection which listens for chat requests from Central
- Open a socket connection which will listen for messages from Central
  - In our POC, Central will use this socket to relocate clients to a new server. We will go into more detail about how this is done with this example.

Clients will then present options to the user to "Begin a Chat" or view/accept a chat request. For beginning a chat, they will enter the username, which will start the server selection and matchmaking flow on Central.

Our System in Action: The Chat Request/Match Making Flow

Client A Initiates Chat

Suppose Client A initiates a chat request by entering Client B's username. This will then send the request to Central via a TCP socket. Central will identify the Client which has the provided username, and then forward the request to the client.

Central will then keep the TCP connection with Client A alive and periodically send updates of the form:

- **ACK_CONN**, When Central accepts the request from Client A
- **REQ_SENT**, When the request has been sent to Client B
- **USER_NOT_FOUND**, when the user Client A is request can't be found
- **AWAITING_REQ**, When Central is still waiting for Client B to accept the request
- **SERVER_ERR**, If there is a server error at any point
- **REQ_ACCEPTED**, When Client B accepts the chat request

These signals are streamed back to Client A to improve the users observability into the status of their request.

Client B Accepts Chat

Once Client B accepts the request, Central will send a REQ_ACCEPTED to Client A, and both Clients will then wait for Central to send back a RoomID, and a Server IP.

The ServerIP will be determined using Central's Server Selection Algorithm.

Central's Server Selection Algorithm

Central will retrieve the delay list for Client A, and Client B. A delay list is a map with key ServerIP and value float (representing the ping delay from Client to a ServerIP).

A naive algorithm we can use is to simply minimize the combined latency from both clients.

This can be modeled as the following:

Let S be a list of chat servers $S = \{s_1, s_2, ..., s_n\}$.
Let C_1 and C_2 represent two clients.
Let $delay(C_j, s_i)$ be the ping delay from some Client j to a server i.

let $combined_latency(s_i) = delay(C_1, s_i) + delay(C_2, s_i)$

We will find the optimal server s_i which minimizes combined_latency(s_i).
The issue with this algorithm is that it does not account for the chat experience of both clients equally. One client might enjoy a very low-latency chat experience, while the other experiences significantly higher latency. However, since the algorithm only minimizes the combined latency values, it can lead to imbalanced user experiences, where one client suffers poor performance.

Therefore, the algorithm we chose for server selection will consider the experience for both clients.
We will select the server which minimizes the maximum latency experienced by either client. This can be modeled as the following:

minimize $\max(delay(C_1, s_i), delay(C2, s_i))$. To break a tie between two servers, we will then consider the combined latency.
If multiple servers are eligible, one will be selected at random to ensure a probabilistically even distribution of traffic among them.

The algorithm for this can be found in the compute_optimal_server method, under the "Central/Internal/matchmaking/matchmaking.go" file.

After a Server has been selected, Central will generate a random RoomId and send both the RoomId and Server to both Client A and B over a TCP socket.

Chat Connected

When the clients receive the RoomId, and ServerIp to connect on, they will initialize the connection with the server by sending "RoomID#Username" to the ServerIp. The server will acknowledge and will store a the client's TCP connection under a RoomID.

From this period onwards, any messages sent/received on this socket connection will be broadcasted to all sockets under a RoomID.

Dynamic Chat Relocation

In this example, Client A, and Client B will most likely connect to Server 3 (Serv3), since it provides the lowest latency for both clients.

As you may recall from earlier, we held off on starting up Server 1 (Serv1) which provides even better latency for both Client A and Client B.

Let's take a look at how our system performs a "Chat Relocation". First, we startup Server 1. This will register with Central, and eventually clients will add the server to it's "Ping/Delay list".

A job is scheduled every 5 seconds on Central which searches through all active chats and recomputes the optimal server. Since Client A, and Client B are speaking on Server 3, and Central now knows (through Client A and B's delay list) that Server 1 will provide the best overall latency, it will do the following:

- Send the new server IP to Client A and Client B. This will be through the open socket connection on both Clients which was mentioned earlier, as a connection "which waits for server messages".
- Client A and Client B will now create a new socket connection to the new optimal server, and perform steps similar to the "Chat Connected" section.
- Client A and Client B will swap out their current socket connection with the old server, with the new one. They will now continue chatting on the new server.

Similarly, if a server crashes, it will stop sending heartbeats to Central. Eventually Central will find out that it crashed (because of the 15 second rule for server heartbeats which was mentioned earlier). Client's will then remove the server from their delay list, and the background job which was described above will relocate all the chats on the dead server.

Analysis of the Results

With our system, we found that a client in a chat system will spend no longer than 4 seconds in a bad state. A bad state is defined as:

- The chat is hosted on a server that has crashed.
- The chat is hosted on a suboptimal server.

This is achieved because our central server continuously scans for bad states across ongoing chat instances every 4 seconds.

Reducing this interval can decrease the time a client remains in a bad state, but it comes with tradeoffs. More frequent scans increase server resource usage, as the system must compute optimal servers for all chat instances more frequently. Additionally, clients may experience disruptions or resource inefficiencies if they are repeatedly moved to new servers in quick succession.

TLS Implementation

As part of our project, we aimed to implement Transport Layer Security (TLS 1.2) to enable confidentiality between client-server communication. The TLS module was designed to be a wrapper around existing connection objects, enabling clients and servers to secure their data transmissions.
Unfortunately, we were not able to fully integrate the TLS module into the main application. However, we have included a standalone TLS implementation code in a separate folder within the project repository, along with a mini-demo to showcase its functionality.

Documentation

Some more technical, low-level description of the provided code.

The Tech

Here is a list of technologies we used for our project:

- Golang
- Mininet
- Python
- TCP

We chose Golang as our language of choice since it provided us with the best balance of high-level language support, as well as, low-level concurrency control.

Central

/cmd/main.go: This is the main entry point for Central. It starts and initializes the different services for central.

/internal/client/store.go: This is the storage layer for our in-memory client store. This handles the CRUD operations for our Clients.
/internal/client/api.go: This is the API layer for our Client REST API. It registers several CRUD endpoints on Central.

/internal/service/store.go: This is the storage layer for our in-memory Server store. This handles the CRUD operations for our Servers.
/internal/service/api.go: This is the API layer for our Server REST API. It registers several CRUD endpoints on Central.

/internal/matchmaking/matchmaking.go: Starts the matchmaking server. This will listen for TCP connections on port 8081. This class handles

everything which was described in the example we looked at previously, such as chat request handling (handleConnection method), server selection (compute_optimal_server method), and server relocation (backgroundAnalysis method).

Server

/cmd/main.go: This is the main entry point for Central. It starts and initializes the different services for a Server.

/internal/chat: Listens for client connections on port 3002. This handles the actual chat activity for clients.

/jobs/heartbeat.go: Contains a background job which sends a heartbeat to the central server.

Client

/main.go: The main entry point to the Client application. Starts up the client_runner.

/runner/client_runner.go: The main UI for the Client application. This acts as the presentation/view layer.

/client/client.go: The actual client object which interfaces directly with Central and Chat Servers. This is the ViewModel and API layer.

/service/server_registry_api.go: This contains an API which sends requests to Central for a list of active servers.

Network

/chat_network.py: The mininet network simulation. This starts up the network, starts the chat servers, starts Central, and starts an sshd daemon on all of the hosts.

Startup Instructions

Quick Start

1. Start a VM instance on Virtual Box with MiniNet installed and configured.
2. Install Golang version 1.22.2
3. Clone the Repo on the machine
4. cd $PROJECT_ROOT/network && sudo python chat_network.py
5. Open two more terminals
6. In the first terminal run "ssh 10.0.0.1". The password will be the same as your VM password
7. Similarly in the second terminal, run "ssh 10.0.0.2"
8. On both terminals: $PROJECT_ROOT/Client && sudo ./build.sh –run
9. You will be promoted to enter a username, enter a different username on each terminal
10. From one terminal enter the "Begin Chat" interface and input the username of the second client/terminal
11. One the second terminal, use the arrow keys to select the second option (viewing chat requests), and accept the chat request
12. You can now begin chatting.

Manual Start

1. Get a physical or virtualized/simulated network.
2. Clone the repo on all the respective hosts.
3. Select a host to be the central server, and run cd $PROJECT_ROOT/Central && sudo ./build.sh –run
4. Take the IP for the host which Central is running on, and replace the IP in both /Client/client/config.txt and /Server/config.txt.
5. Designate some hosts to be the servers. Run the following commands on each of them "cd $PROJECT_ROOT/Server && sudo ./build.sh –run
6. Select two hosts to be the clients. Run the clients similarly to the example given in the quick start.

Contributions

Overall our team contributed equally through the project. It's difficult to provide exact partitions of the project which each person was responsible for, since everyone helped out with ever aspect. Here is a general gist of what everyone worked on:

Porom Kamal

Porom was responsible for:

- System Architecture/Framework
- Central Matchmaking algorithm
- Client UI
- Chat Relocation feature
- MiniNet Simulation

Jean-Sebastien Dandurand

JS was responsible for:

- Client Registration CRUD APIs
- Service Registration CRUD APIs
- Chat Request Flow

- Storage Layer for Client and Service
- Debugging critical issues

Seron Athavan

Seron was responsible for:

- Helping out with the System Architecture/Framework
- Build infrastructure for the system
- Client Concurrency Issues Debugging
- Performance Improvements for Client
- Helped out with Client Relocation Feature
- Built the TLS Client

Concluding Remarks

During this project our team got the chance to build a real world system which applies core concepts in distributed computing. We solved practical challenges related to resiliency in a degraded network environment, and leveraged tools we learned in CSCD58, such as MiniNet, to bring our system to life.
This project was a chance for us to experience all the different hardships that building network resilient systems comes with, including dynamic routing, load balancing, failure detection and recovery, as well as latency optimization. As aspiring software engineers, this is an invaluable experience that allowed us to bridge theoretical knowledge with practical application. Tackling these challenges gave us a deeper appreciation for the complexities of building resilient, efficient, and scalable distributed systems.