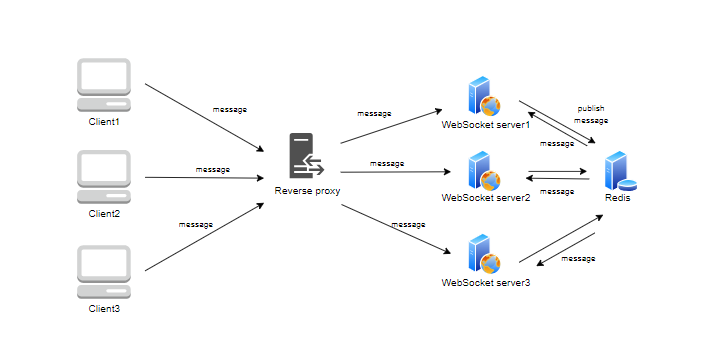
Scaling Websockets with Redis, HAProxy and Node JS

This tutorial is about how we scale WebSocket connections to run on multiple servers. To exemplify this, I used a live chat application that is scalable with docker.

**What are WebSockets?**

WebSockets are a communication protocol that provides full-duplex communication channels over a single, long-lived connection. This protocol is designed to be implemented in web browsers and web servers but can be used in any application where real-time, bidirectional communication is needed. To establish a WebSocket connection, a client sends a WebSocket handshake request to the server, and upon successful handshake, the connection is established. Once the connection is open, both the client and server can send messages to each other at any time until one of them decides to close the connection.

As we can see in Figure1, we have 3 clients, a reverse proxy (in our case is HAProxy), which listens on a specified port and manages the communication between the clients and backend servers (Server 1, Server 2, and Server 3). We want the clients to communicate with each other, so the reverse proxy facilitates load balancing and directs incoming connections from clients to the appropriate backend servers. The role of Redis in this setup is as a publish-subscribe system. When Client 1 publishes a message on a specific channel, Redis ensures that clients subscribed to that channel, such as Client 2 and Client 3, immediately receive the published message. Redis handles the distribution of messages among connected clients based on their subscriptions.



Figure

**Live chat Application**

The Docker compose configuration from the following repository <https://github.com/sofiaofimias/SOA/blob/master/live_chat_scalable_web-sockets_Redis/docker-compose.yml> includes an HAProxy load balancer exposing port 8080, distributing traffic to WebSocket applications (ws1, ws2, ws3, ws4) with different APPIDs. Each WebSocket service runs a wsapp image with a unique APPID environment variable. Additionally, a Redis service is defined for pub-sub messaging. The load balancer service utilizes the HAProxy image, and all services are encapsulated within their respective containers, facilitating scalable and modular deployment of a WebSocket-based application with load balancing and messaging capabilities.

The HAProxy configuration <https://github.com/sofiaofimias/SOA/blob/master/live_chat_scalable_web-sockets_Redis/haproxy/haproxy.cfg> has large timeouts and these are very critical because the HAProxy will terminate the connection if nobody is communicating for that amount of time. This configuration enables efficient load balancing and connection management, ensuring seamless communication between the frontend and multiple WebSocket backend servers with extended timeout settings for enhanced reliability.

In the Nodejs application itself <https://github.com/sofiaofimias/SOA/tree/master/live_chat_scalable_web-sockets_Redis/app> , The server subscribes to a 'livechat' channel in Redis and publishes a test message upon successful subscription. When a message is received on the subscribed channel, the server forwards it to connected WebSocket clients.

The WebSocket server listens on port 8080, accepting incoming requests, and establishing connections. Messages received from clients are published to the 'livechat' Redis channel, allowing real-time communication. Each WebSocket connection receives messages from Redis and broadcasts them to all connected clients. Additionally, the script creates a basic HTTP server and ties it to the WebSocket server. It logs various events like open, close, and received messages. Each connection is tracked in the 'connections' array.

The first thing we need to do is build the docker image for the Web Socket:

A screen shot of a computer

Description automatically generated

Figure

The next step is to initiate the activation of all WebSocket servers, HAProxy, and Redis by running the command docker-compose up.A screen shot of a computer program

Description automatically generated

Figure

A screen shot of a computer

Description automatically generated

Figure

Figure 5, 6 and 7 show a basic chat-like interface where users can send messages through a web browser to a WebSocket server, and receive messages from that server in real-time. When servers 3333 and 4444 are opened, the WebSocket connection established within the provided HTML and JavaScript code connects to the server running on localhost at ports 111 and 222 respectively. This connection enables bidirectional communication between the web interface and the WebSocket servers. When a connection is opened, a message is logged to the console indicating the WebSocket connection is established. Any messages received from the WebSocket servers are dynamically displayed on the webpage in real-time, with each message appearing as a new paragraph within the designated message display area. Users can input messages in the text field provided, which are then sent to the WebSocket servers when they click the "Send" button. The sendMessage() function retrieves the input message, sends it to the respective server via the WebSocket connection, and clears the input field afterward, facilitating a seamless interactive experience between the user and the WebSocket servers. If the WebSocket connection is closed for any reason, a message is logged to the console indicating the closure. Overall, this setup allows users to exchange messages with WebSocket servers 111 and 222 in real-time through the provided web interface.

A screenshot of a computer message

Description automatically generated

Figure

A screenshot of a computer screen

Description automatically generated

Figure

A screenshot of a message

Description automatically generated

Figure