

## EECS 4222: Project 2

**Name:** Jayant Varma

**Student id:** 218350819

**Note to the reader:** The project does NOT work on Kubernetes, I believe that it's because in the Kubernetes environment, the Redis image couldn't be pulled from Harbor private registry. I've also run out of memory for my virtual machines, so that's another separate problem. I have attached a video in the submitted folder, where I explain the deployment and some interesting parts of my implementation. However, I've submitted the chatapp.yaml to the best of my capabilities.

### How the project can be tested:

- (1) Create the images from the Dockerfiles:

Run :

```
docker build -t harbor.pacslab.ca/218350819/chatapp-backend:latest .
docker build -t harbor.pacslab.ca/218350819/chatapp-frontend-server:latest .
```

In the terminal of the directory where their DockerFiles lie.

- (2) Run the docker-compose.yaml using 'docker compose up' command in its directory.
- (3) Go to <http://localhost:30222> and you have one client online, on a new tab, go to the same link and enjoy the conversation!
- (4) Kindly refer to Jamie's post on eclass or the attached video where I'm seen explaining the docker deployment and testing of the project for more reference.

### Before we begin:

The overall flow of the application is this:

- (1) A user's browser requests the chat application web page from the frontend service.
- (2) The frontend service responds with the static files needed to load the web application in the user's browser.
- (3) When the user sends a message, the browser uses the WebSocket connection established by dripper.js to send the message to the backend service.
- (4) The backend service receives the message and uses the Redis client to publish it to a specific channel that it's subscribed to for chat messages
- (5) Redis processes the published message and broadcasts it to all subscribers, which, in this case, include the backend service instances.
- (6) The backend service instances receive the message from the Redis channel and then broadcast it to all connected clients via their respective WebSocket connections.

### System Architecture:




A blend of:

- **Publish-subscribe:**

- Event based: When a message is 'sent', it is passed through redis for real-time communication by allowing messages to be published to a channel and immediately received by all subscribers of that channel. This is essential for a chat system where users expect instantaneous message delivery.
  - Achieves horizontal scaling: This way of using pub/sub from redis supports more scalability as many nodes/clients can chat simultaneously where each client is implemented the same way as any other.
  - Achieves distribution transparency: Since the pub/sub decouples the backend from who the subscribers are, not only does it create modularity, but also gives the illusion of a centralized backend handling all broadcasting. → Increasing scalability as well.
- **Service Oriented Architecture: Object based style:**
  - Components:
    - Web server service(chatapp-frontend-server):
      - Statically holds the frontend files for the web UI.
    - Backend (chatapp-backend): contains the logic, the structure of the messages, interfaces with redis for pub/sub, upgrades HTTP to websocket, holds a map of all clients for broadcasting.
      - Web-socket: Helps achieve real-time bidirectional communication between clients and the backend (server)
    - Redis as a message broker: Use of redis for pub/sub essentially is a way to use it as a message broker. (check resources section below)
- **Layered Architecture:**
  - Splitting the project into 3 layers, each having a separate role to play:
    - Frontend: Statically holds files for web UI
    - Redis: For real-time messaging as a message broker
    - Backend: For web-sockets, and business logic, etc

## Design of each component:

Focusing on the roles within our architecture:

-  Chatapp-Frontend-server:
  - Serves static files to each client's web browser
  - Interfaces with the backend service to establish proxy for WebSocket connections allowing real-time communication
-  Chatapp-backend:
  - Upgrades from HTTP to WebSocket protocol.
  - Handles connections, and handles business logic.
-  Redis:
  - Enables message broadcasting to multiple subscribers, which is essential for the chat functionality

## How components interact/connect with each other:

The components of the system are connected via the network and communicate with each other through websockets.

### Backend:

The backend listen and serves on port 14222 of the whole network.

```
var addr = flag.String("addr", "0.0.0.0:14222", "HTTP service address")
```

It holds a websocket there:

```
log.Printf("WebSocket server started on %s\n", *addr)
if err := http.ListenAndServe(*addr, nil); err != nil {
    log.Fatal("ListenAndServe: ", err)
}
```

### Redis:

Redis is on port 6379 of the redis service

```
var redisAddr = flag.String("redis-addr", "redis:6379", "Redis server address")
```

### Frontend:

Initiates the websocket connection to backend in 'dripper.js' where a websocket connection is opened on port 14222 (where the backend is listening).

```
// Use window.location.hostname which will be dynamic based on where the
// frontend is being accessed from
new_uri += "://" + window.location.hostname + ":14222/websocket";

websocket = new WebSocket(new_uri);
```

## How I implemented the additional feature (feature #7):

- Made a web server in Go, using the http package which provides HTTP client and server implementations. The server serves static files such as HTML, CSS, and JavaScript to clients' browsers, replacing this job of Nginx:

```
• var staticDir = flag.String("static-dir", "./html", "The directory
  of static files to host")
• //... code in between...
• fs := http.FileServer(http.Dir(*staticDir))
•
• /* Register the file server handler to respond to HTTP requests on all
  routes ("/").This *effectively* makes this Go program work/pretend to
```

```

be a web server, directly addressing Part 2's requirement to build my
own web server with Go, therefore avoiding external solutions like
Nginx for serving static content.
•      */
•
•      http.Handle("/", fs)

```

*This above code is taken from staticserver.go file.*

- The proxying of WebSocket connections was inherently handled by the Go server, which accepts HTTP connections and upgrades them to WebSocket connections if the request is for WebSocket communication.

```

• var upgrader = websocket.Upgrader{
•     CheckOrigin: func(r *http.Request) bool {
•         return true // Allow connections from any origin.
•     },
• }
•
• http.HandleFunc("/websocket", func(w http.ResponseWriter, r
• *http.Request) {
•     ws, err := upgrader.Upgrade(w, r, nil)
•     if err != nil {
•         log.Fatal(err)
•     }
•     defer ws.Close()
•     // WebSocket connection handling logic...
• })

```

*The above code snippet has been taken from chatapp.go file*

## **Efforts you have made to improve the performance, if any:**

I believe the current implementation is quite performant, I just tested for correctness in a systematic manner, I never tested for performance. So, unfortunately, I don't have any benchmarks to compare improvement/worsening of performance.

## **Other requirements in report for feature 7:**

I have explained the architecture, components, and their interactions. The code has been beautifully documented.

## **Resources:**

- Articles:
  - [Building a websocket chat application in Go](#)
  - [Creating your first API: A step by step guide using Go](#)
  - [How to use Websockets in Golang](#) (very helpful)
  - [Redis as a message broker](#)
  - Many, many more...

- Github repository:
  - [Pacslab chatapp repository](#)
- Go documentation: <https://go.dev/doc/>
- Videos:
  - [Go tutorial](#)
  - [Build a Golang chat application using WebSockets](#)
- Jamie's post on eclass was a massive help with deployment over docker:
 

2023\_LE\_EECS\_W\_4222\_\_3\_M\_EN\_A\_LECT\_01: Local testing with Docker Compose Inbox x



**Jamie Fletcher (via eClass)** <myforum@yorku.ca>

to me ▾

[2023\\_LE\\_EECS\\_W\\_4222\\_\\_3\\_M\\_EN\\_A\\_LECT\\_01](#) » [Forums](#) » [Class Forum](#) » [Local testing with Docker Compose](#)

[JF](#) Local testing with Docker Compose

by [Jamie Fletcher](#) - Saturday, 30 March 2024, 6:25 PM

While working on [Project 2](#), it's nice to be able to test locally. This procedure worked for me and I thought others might benefit but I'm certainly no Docker expert.

- On your local machine, build each of your Docker images. Follow the appropriate instructions in [Appendix 2](#).
- Create a Docker compose file ("compose.yaml") that references the Docker images you have built. A few tips:
  - The yaml format is very picky. Keep the right number of spaces at each indent, don't use tabs, etc.
  - You will probably want to forward at least the port from the frontend to localhost so you can see your app
- Within the directory that has the compose.yaml file, run the command: **docker compose up**.
  - This will launch all the services specified in the yaml file. You should see errors, logs, etc. get printed to the command prompt.
  - If you forwarded the port, the frontend should be accessible at <http://localhost:4222> (or whatever other port number you picked)
- When you're done, you can use **Ctrl+C** to stop Docker and then run the command **docker compose down** to clean up.

The compose.yaml file can be very simple. Mine basically looks like this:

```
services:
  redis:
    image: harbor.pacslab.ca/123456789/redis:7.0.8-alpine

  chatapp-backend:
    image: harbor.pacslab.ca/123456789/backend:latest

  chatapp-frontend:
    image: harbor.pacslab.ca/123456789/frontend:latest
    ports:
      - "4222:4222"
```

<https://docs.docker.com/compose/gettingstarted/>