CPE 400 Final Project

Microservice Simulation Software (MSS)

Curtis Chung and Carl Montemayor

Instructors: Shamik Sengupta

Due: December 7

Project Overview:

In recent years, the micro-service architecture has been favored by many software companies as a way to structure large projects and applications so that independent pieces of software can be developed asynchronously and with little to no dependencies. In this type of architecture, pieces of software are broken apart into smaller "services" that are self-contained and functional (usually deployed via cloud services). The result is a graph-like network of several services that communicate with one another via HTTP or RPC. Although this provides many benefits to developers, opting into this architecture presents many difficult tasks such as how these services communicate with other services and how to share such data and information in an efficient manner.

This project aims to develop a network monitoring system specific to the micro-service architecture. It presents itself as an online web application that allows for users to simulate the passing of data, messages, and information to these services and the creation of new services in a user-friendly manner. Users will be able to simulate the creation of their own "services" whether it be an API, database, or website, and track the metrics associated with these interactions such as delays, errors, and latency in their system.

Technical Report: Implementation

For the project, we built a system where users can simulate their own micro service network system and build all the different connections to each node. Each node can act as different aspects of a network system, such as databases, frontend, etc. This project is to give a visual simulation of the test cases and scenarios rather than lines on a command line. When on the web application, the user will have the option to run the simulation and choose between three different scenarios. Each scenario will break down a different node, while the nodal re-routing algorithm will choose an optimal node to make a new connection to without interrupting any nodal processes. This also shows the data flow (which in the context of microservices and software architecture can simulate HTTP requests) between the different nodes.

The purpose of this project is to give a novel solution to network testing scenarios and problems within business networks. The visual simulation allows software developers an easier way to see how each node (network structures) interacts with one another with their connections and if one crashes, it's easier to see which nodes it will affect. Each node in our simulation has a different weight to it. Higher weights mean it is more important to the network, and therefore is much less likely to crash. If a node were to crash, the node itself and any connections will turn red as displayed in Figure 1.

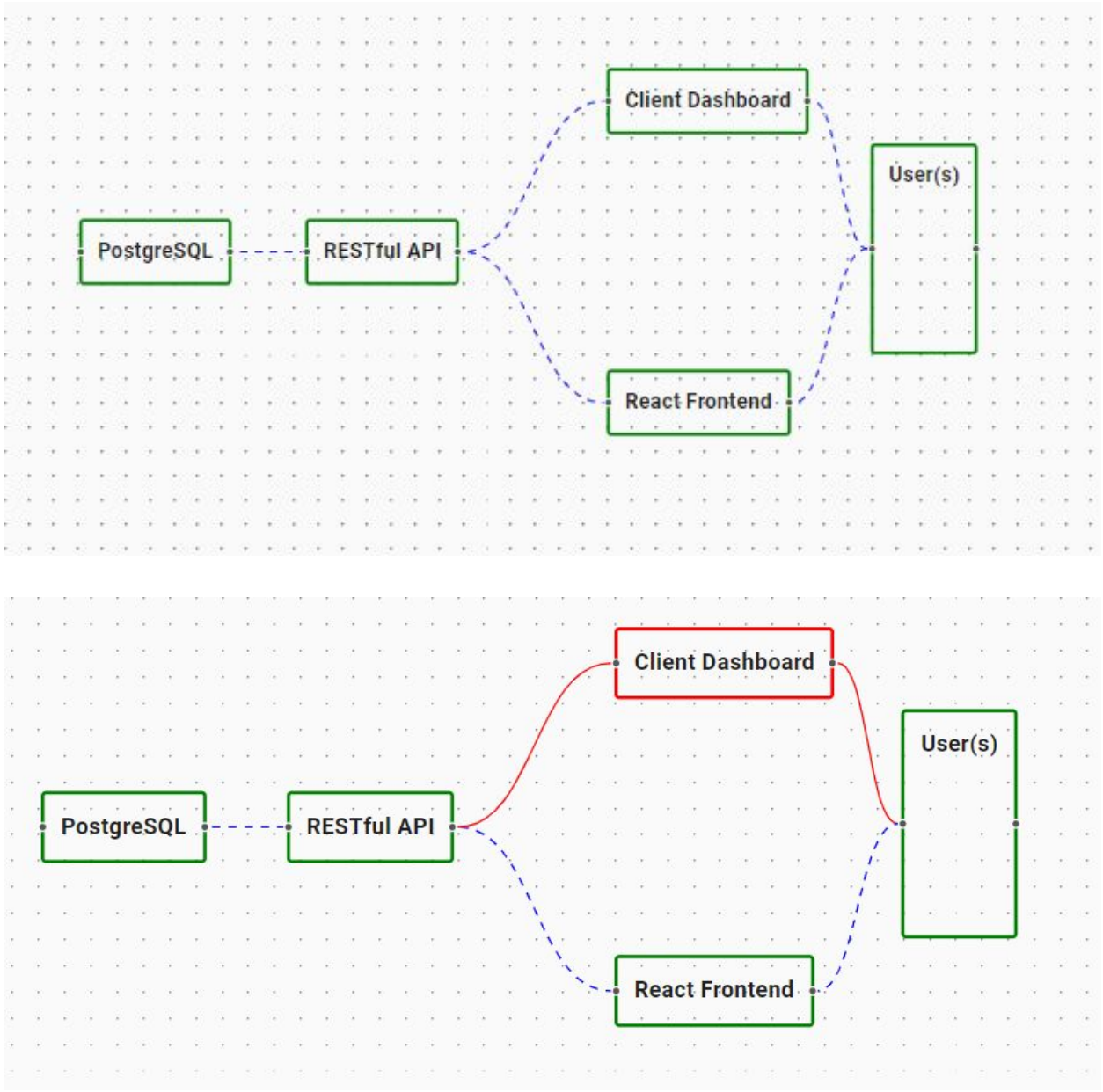

Figure 1: “Simple” Simulation before and after user clicks run and crashes one node

Technical Report: Simulation and Protocol Walkthrough

Our simulation is in the context of microservices. As stated in our problem statement/overview, our goal was to find out if it was possible at all to monitor and check if pieces of a software or web application (which can have multiple components) can be monitored. A microservice architecture application can actually present itself as a large network that is made up of servers and components such as databases, APIs, and connections to a client-side website, etc. In our simulation, we have developed three situations/simulations that are possible configurations of software. In our first ("Basic") simulation, we have a typical client-server architecture that is not broken up into smaller services. In our second ("Intricate") simulation, we have a more complex software architecture that is made up of more nodes which simulates the process of converting to fully microservice architecture. Lastly, we have a third ("Complicated") simulation that is meant to simulate a complete microservice-based application.

When running the simulation, there are several notable areas of interest. First, our simulation simply picks, at random, which node is to be faulty. Once a node is determined to be faulty, the corresponding incoming and outgoing edges, along with the node itself is painted red. This showcases the dataflow coming from all of the nodes and their connections. Visualizing these connections visually is a great option for system administrators to see or locate which service is faulty. Having a visual application of this large network is very important as it is hard to discern a network if its mode of delivery is through a command line application. This demonstrates out-of-the-box thinking as users of the application are able to visualize appropriately their designated software system and/or network.

In addition, there is additional complexity in terms of which nodes become faulty. There are several nodes that are showcased in the three simulations and the result of their faultiness is explained in the following table.

Node Type	Faulty Effect
Datastore (Postgres, Redis, etc.)	No connections to other nodes are affected since it is a “terminal” node.
API/Service/Middle Layer (RESTful, GraphQL, Gateway)	All outgoing and incoming edges are affected. If faulty, this is detrimental to the entire system as it is the middelmost layer typically.
Frontend/Consumption Layer (React, other frontends/client sites)	This is a similar case to the terminal nodes for the Datastore.

In general, in larger applications, the need for a microservice architecture becomes greater as software based in the web should possess high availability. This also simulates a relatively real-life application of large networks and services as the chance for these components to fail can be quite random which is how we also pick which node is to be faulty.

Analysis:

When certain aspects of the network crash, appropriate actions will occur in our simulation. For example, if the database or main node were to crash in our simulation, there are no other databases to reroute any connections to, so there is no choice but to wait for it to come back up in order for the service to continue to function properly. If smaller aspects, such as the API, were to crash, then the graph will choose an appropriate node to reroute the connections to until the service is back on. In the case of Figure 2, if the node labeled as the database (PostgreSQL) were to become red and crash, there would be no way to reroute the nodes

connected to it to another node. Even if there were multiple databases, each one would fill a specific role. One database node cannot take over the functions of another database node, therefore there is no choice but to wait for it to boot back up.

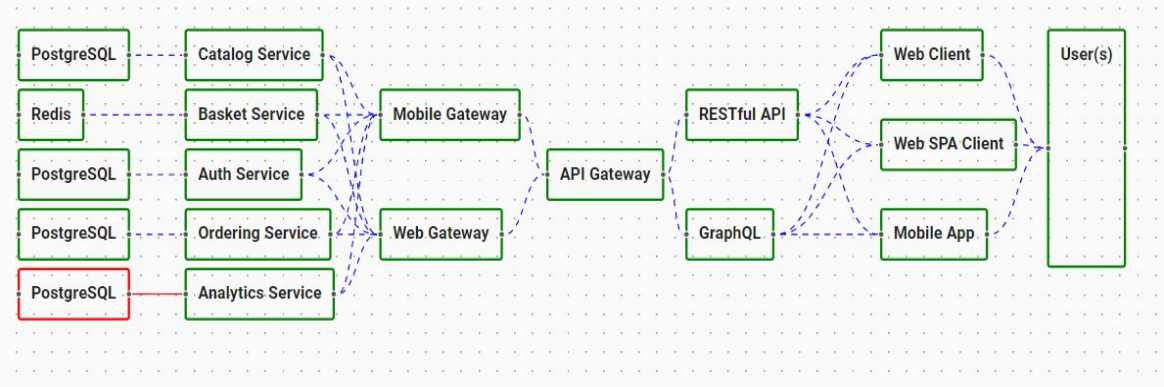

Figure 2: A more intricate network layed out in our simulation. Other PostgreSQL nodes cannot be rerouted to connect to other nodes if one were to crash. Each PostgreSQL node has a specific purpose and cannot cover for one another.

When running our simulation, a few things came to note of real-life network failures and some that we may not be able to completely resolve in our simulation. Table 1 states a few cases that we can implement our project.

Real Life Cases	**Consequences**
Nodes can crash and lose all of it's connections to other nodes.	When a node crashes, the ability to communicate the information necessary for other nodes. Can be rerouted
Main/Important nodes crash	Nothing can be done except to wait until that main node/service comes back up. Cannot reroute
Multiple nodes crash	Difficult to reroute. Depending on the multitude of nodes crash and the specifics of which one, can be hard or impossible to reroute in real network situations

Table 1: Scenarios of nodal failures and the consequences of how they may affect the final result.

As stated before and in our problem statement, our simulation is in the context of microservices and largely software architecture. Given this context, it is clear from runs of our simulation/visualization that the nodes in our simulation run very independently of one another. The practicality of our software/application is that users, developers or system administrators are able to simulate a large number of servers or pieces of architecture to accurately find out which other servers or pieces are affected by a faulty node. As a result, our simulation and analysis showed the advantage and high-availability that microservice applications have. This shows that if software is broken down into smaller services, the advantage is that there is high availability of the software itself. For example, if there is a singular service of an application that is down (similar to the "Complicated" simulation scenario), the other services within the application will stay alive and will continue providing data. In contrast, with the "Simple" simulation that is run, there are no smaller microservices in place, therefore, if a critical portion of the application, say an API is faulty or down, the whole application, will therefore also have downtime. As a result, our analysis and reports of our simulations demonstrate that when developing applications with a large network it is best to implement the software architecture in such a way that it is a microservice-based architecture so that all the nodes and components of the software are high-availability.