CPE 400 - Microservice Simulation + Dataflow in a Faulty Network

This is a the project documentation made by Curtis Chung and Carl Montemayor. This documentation file contains how to run the simulation, the structure of the code (and how it was built), and also the general thought process behind our programming.

Installation

There are two ways of running this simulation: website or locally. These produce the same outcome regardless, however, accessing the simulation through the website is most preferred.

NOTE: If accessing the website, it is preferred to use `Safari`, `Google Chrome`, or `Firefox`.

Website

If you would like to view the the simulation software via the website, simply visit the following link: MSS.

Once at the website, you have several options listed in the homepage. You can visit the documentation such as our reports and Github. Most importantly, you can access the simulation itself by clicking the "Simulation" card.

Homepage

Local

If you want to launch the `development` version of the simulation software you are free to do so. Please follow the instructions below:

```
// Clone the repository
git clone https://github.com/carlmontemayor/mss

// Move into the /mss directory (there are two, go to the directory with the package.json)
cd mss

// Install the aproppriate packages
npm install

// Run the development server
npm run dev

// Finally go to your http://localhost:3000
// to access the website locally.
```

Regardless of which way the simulation is run, you have the option of finding looking at the simulation page.

Simulation

In this section, a set of nodes is already pre-built for the user. The nodes themselves can be dragged and moved around as needed for a clearer view. The nodes are indicated by `green` boxes and their data flow is shown in `blue`.

To the right of the simulation view, there is the `ControlPanel` component. This control panel allows the user to switch between three simulation types: `cool`, `intricate`, and `complicated`. The `Cool` simulation is a simple case wherein there is a simple client-server architecture that is simulated by nodes. While the `Complciated` simulation is a full-on micro-service architecture that features several services.

Simulation Demonstration

Within the `ControlPanel` component, there is an option to either `run` or `reset` the simulation. If the user chooses `run`, then the simualtion will start. The simulation starts by picking any random node within the graph and it will effectively pain that node `red` as in faulty. This means that data flowing in/out of that node is effectivley rendered useless, and the data flow is demonstrated by a non-flowing red edge.

Simulation Demonstration cont.

In the case that a terminal node is selected as faulty, there is still a relatively high availability since the faulty node is terminal. However, as there are more connected nodes that may turn faulty, the data that is being passed is reduced. Once the user is done with the simulation, he or she can either click the `reset` button or refresh the entire page to clear out the simulation.

At the top of the page, there are links to the other parts of the website such as documentation and links to the Github.

Code Design

This entire project was built entirely using `React.js`, `Typescript`, `Next.js`, and `Material UI`. Other libraries such as `React Flow Renderer` were used in order to visualize the graph/nodes.

In addition, this project is hosted via `Vercel` which makes it accessible to all.

Pages

The web application split into three pages: `documentation`, `simulation`, and `navigation`.

The `documentation` page(s) include the `Code Documentation` (this document) which demonstrates how to run the simulation and how the code is structured. The `Simulation` page shows the simulation itself. It is equiped with a visualization of the nodes and a side panel from which users can run the simulation. Lastly, the `Navigation` page is the root index which allows the user to navigate around the site.

Components

A set of design components, located within the `/components` route were custom build using `CSS` and `Material UI`.

The non-presentational components such as the graph and the UI logic behind the graph were also custom build components, as well.

There are several

Code Structure

The code itself separates concerns very well. The `/components` directory only handles styles and custom components. This allows for cleaner code as the styles are discretely separated from the components that use the styles.

If a component do is stateful and contains heavy UI logic as in `graph.tsx` and `ControlPanel.tsx`. Then the only concern within these files are the UI actions and logic associated with them.

Within the `/simulations` directory, there exists three pre-build simulations which are essentially arrays of `FlowElements` (as defined in the `ReactFlowRenderer`) package.

Furthermore, the `/utils` folder abstracts some Javascript-specific helper functions that were useful in the manipulation of the graph.

Graph Algorithm

Although no specialzied algorithm was developed to modify the graph, there were a lot of UI logic that needed to be impleneted for the simulation to work correctly. In general, the heavy lifting for graph algorithms was done by the `ReactFlowRenderer` library.

In particular, there was a specific workflow/algorithm needed to paint and color the nodes of the specified fault node. The algorithm is found in `ControlPanel.tsx` and its code is demonstrated here:

```
// In ControlPanel.tsx
const handleClick = () => {
  // Generate a random node index
  const failedNodeIndex = toNumber(FindRandomNode(simulationType));

  // Make a copy of the failed node and its member values
  var failedNode = simulationElements[failedNodeIndex];

  // Paint the failed node red
  failedNode.style = {
    border: '2px solid red',
  };

  // Generate another copy of the new elements to be inserted into the DOM
  let newSimulationElements = [...simulationElements];
  newSimulationElements[failedNodeIndex] = failedNode;

  // Modify the edges
  ModifyNodes(newSimulationElements, failedNodeIndex);

  // Allow changes to take place.
  setTimeout(() => {
    setElements(newSimulationElements);
  }, 1000);
};
```

```
// In nodes.js
export const ModifyNodes = (newSimulationElements, failedNodeIndex) => {
  // Iterate through every single node
  newSimulationElements.forEach((element) => {
    // Prevents off-by-one error
    const adjustedFailedNodeIndex = failedNodeIndex + 1;

    // Paint the faulty nodes outgoing and ingoing edges as red
    if (
      element.target === adjustedFailedNodeIndex.toString() ||
      element.source === adjustedFailedNodeIndex.toString()
    ) {
      element.animated = false;
      element.style = {
        stroke: 'red',
      };
    }
  });
};
```