Matthew Pallansch  
CSCI 4370 Simulations & Modeling  
12/13/15

Final Report

Goals:

For the topic of this project, I chose the first open ended project list: ‘Write your own distributed and parallel simulation executive’. I decided to write this simulation engine in JavaScript, using NodeJS for server-side JavaScript. My goals were as follows:

1. Create a server script to support multiple concurrent simulation processes

2. Create a JavaScript library which allows a web page to act as a simulation process and communicate with the backend server

3. Create an example air traffic simulation across several web pages

The overall goal was to create a framework in a very primitive way, to use as a starting point to be expanded upon later. For this reason, there is a lot of future work which can be done to improve upon the engine.

Accomplished:

I have accomplished all three of the goals that were specified above. The first goal can be seen in the server.js script under the /node folder of the project. This script takes as arguments, the domain of the simulation pages, and the port to listen for requests on. This script will then handle all communication for the local processes.

Server:

The first task this server does is to synchronize the start of a simulation. When the simulation is first run, the script has a variable set to false, called ‘started’. When the first local process registers in the system, it is deemed the admin process, and can then make the ‘start’ request to the server. The other local processes that register in the system periodically make requests to check the state of the ‘started’ variable, which is changed to true once the ‘start’ command is received by the admin.

Once the simulation is started, the local processes make three main requests: to check if it has received any events, to send an event, and to get GVT. The sending and receiving of events is handled very simplistically. There is an array of ‘Simulation’ objects maintained by the script. This object tracks all local processes when they register in the simulation. One property of the object is an array of events which have been sent to this process. When an event is sent to a process, it is simply appended to the array of events for that process. Each process will then periodically make requests to get the latest events sent to them, and the server will respond with that event array, and then clear the array. The other main request is for GVT. The GVT request is sent with the minimum local time stamp for that process, which is tracked in the ‘Simulation’ object for that process. When a GVT request is handled, the minimum time stamp from all local processes and all events waiting to be sent to local processes is used. This is an acceptable method of calculating GVT, because while the server script can *receive* requests in parallel, it cannot *process* requests in parallel (because JavaScript doesn’t support threading). So any event in the system is either in a local process or is stored in the server script at the time of a GVT calculation.

The last task is to handle the disconnecting of a process from the simulation. If a user attempts to leave a local process page, a request is sent to unregister them from the system. The corresponding ‘Simulation’ object is then deleted from the server script. If all of the simulations are unregistered, the ‘started’ variable changes back to false, and the server waits for a new admin to register and start a new simulation.

This file was written with JSDoc (<http://usejsdoc.org/>) documentation which is built into pages in the /documentation folder. These files contain detailed information about each method call. Additionally, there is thorough inline documentation of each method, to ensure that the server can be understood and later built upon by future developers.

Library:

The library allows a user to define a very minimalistic set of information about the process, and the handles the main body of executing a simulation. Basically, all that is defined for a process is the starting state, and the initial queue of events. Once this is defined the library automatically sends a register request to the server, and sets the page up to send an unregister request when the page is unloaded. The library then executes the main loop function which does the following:

1. Sends requests to compute the latest GVT and get the latest events. Although these requests may not return the information by the time the rest of the process is executed, they should be received by the next iteration of the main loop.

2. Calls the draw function with the current state

3. Saves the current state, and then processes the next event in the queue, saving anti-message copies of any messages sent during processing

4. Garbage collect events in the input queue with time stamp less than GVT

The loop executes at a specific increment so that the states are not transition between too fast for a user to understand what is going on. The execution of events happen based on order in the input queue, not based on the time stamp of the event relative to real time. In other words, events with every different time stamps will have the same execution delay as events with close time stamps.

This library requires that jQuery be referenced on the page. jQuery is used for all ajax calls to the server. Like most JavaScript libraries, I included a source file ‘simulation.js’ and a minified version ‘simulation.min.js’. The minified version is used in the sample project to increase efficiency and decrease load times. I used a free online tool called JSCompress (<http://jscompress.com/>) to minify the code. The source file has included JSDoc documentation, which was built into the pages in the /documentation folder. These pages have more detailed information about each method call in the library. Additionally, there is inline documentation included, so that the library can be understood and later built upon by future developers.

Sample:

The sample created is the very simplistic air traffic simulation that was discussed in the class notes. There are three types of events: departure, arrival, landing. These events edit three state variables: onTheGround, inTheAir, and runwayFree. Departure events subtract one from the number of planes on the ground, and schedule a landing event in the destination airport. Arrival events add one to the number of planes in the air, then if the runway is free schedule a landing event in the same airport. Landing events subtract one from the number of planes in the air and adds one to the number of planes on the ground. Then if there are more planes in the air waiting to land, it will schedule another landing event, otherwise it will set the runway back to free.

In this particular air traffic simulation, there are three airports: ATL, LAX, CHI. The state variables for a process are displayed to the user through images drawn on a canvas. The planes that are inTheAir are shown at the top right of the screen. The planes that are onTheGround are displayed at the bottom left of the screen. The runwayFree variable is indicated by the stoplight at the bottom right. And the time is indicated in the top left to show when there is a rollback. To run the example, follow the instructions in the ‘sample’ section of the API document provided in this project. You should see 5 planes take off from ATL, 4 planes take off from LAX, and 5 planes take off from CHI. You should then see 5 planes land in ATL, 5 planes land in LAX, and 4 planes land in CHI.

Future Work:

Because this engine was created as a starting point for a more robust framework, there are many possibilities for future improvements. Below I list the changes that I feel are most crucial.

Currently as the system is implemented, the local processes can send events to other processes. Each event has an execute function, which is defined as JavaScript. Because there is no system for validating processes in the simulation, and any JavaScript can be included in the execute function of an event, it is very possible that a hacker could connect to your server and send malicious JavaScript code to your browser. While this is a very low risk for the people using this system now (because someone would have to find the server which isn’t listed on search engines, figure out how the system works, and know when other users are connected to the server to send the malicious code), this should be fixed if it were ever to become a widely used framework. One solution could be to have the admin process define a password, so only authenticated processes can connect to a simulation. Another possibility is to define all execute functions in a simulation-specific library, and only pass the name of the function to be executed to the page. This may be limiting, but would severely impede a hacker’s ability to send malicious code.

Another addition that is critical is the implementation of a roster feature that would allow the admin process to see which processes are registered in the simulation. As it is implemented now, the system will not work if all of the processes are not connected, so it is important that the admin knows if all users are registered, and the simulation is okay to start. Once this feature is implemented, the modal window that is displayed when waiting for the simulation to start could act as a lobby, showing the names an id’s of the simulations connected. Additionally, it would be very beneficial to ensure that no duplicate simulations (simulation objects with the same name) are connected to the same simulation.

Another feature which is semi-important, but possibly very difficult to implement would be the ability for one server to host multiple simulations at one time. Currently, this could be accomplished by running the script on different ports, but that would restrict the user to a constant number of simulations, and each simulation would have to be coded differently to pass in the appropriate port number. A better solution would allow users when connecting to the simulation to create a new simulation or join an existing one, each having a unique name. Then each request would be passed along with the simulation name, allowing the server to keep simulation data separated. Once this is developed, multiple versions of the same simulation can be run concurrently, which is a very powerful addition.

Another addition which is less critical would be the ability to customize the execution of the simulation. This could be done through a configuration object as a parameter which could allow for certain flags to customize the simulation environment. For example, a simulation with infrequent rollbacks could be set to use infrequent state saving, while a simulation with frequent rollbacks could save state at every execution. Simulations with large states and small changes between event executions could specify that partial state saving be used as an optimization. These additional configuration options would greatly add to the power of this framework.

An additional framework improvement would be to allow more analytical capabilities of the simulation. Currently the primary use of simulations created with this framework would be to display the information to the user, however additional code could be written to track progression of variables in local processes or as a whole in the system. This would then allow for more analytical simulations to be created.

Also, unrelated to the expansion of the framework, there is a great possibility for future work in using this system to create simulations. This framework is very generic, and could support almost any parallel discrete event simulation. Using this system has many benefits: it can be run on any platform that supports an internet browser and JavaScript, it is lightweight and efficient, and easy to setup. For this reason, it can be used in modeling, gaming, or any other field of simulation where displaying information to the user is the primary function.

Word Count: 2034