|  |
| --- |
| University of southampton |
| Contagion – Developer Guide |
|  |
| **Lewis Hill** |
| **10/22/2019** |

Table of Contents

[Introduction 3](#_Toc24107373)

[Setup (Server - Local) 4](#_Toc24107374)

[Setup (Server- Remote) 4](#_Toc24107375)

[Initial Setup: 4](#_Toc24107376)

[Subsequent Setup: 4](#_Toc24107377)

[Setup (Client - Local) 5](#_Toc24107378)

[Setup (Client – Remote) 5](#_Toc24107379)

[Setup (Database – Remote Only) 6](#_Toc24107380)

[Initial Setup (Method 1, cmd): 6](#_Toc24107381)

[Subsequent Times 6](#_Toc24107382)

[Initial Setup (Method 2): 6](#_Toc24107383)

[Accessing Data (Method 2): 6](#_Toc24107384)

[Updating Server-Client Communications 7](#_Toc24107385)

[Other Useful Information 8](#_Toc24107386)

[Implementing new AIs 8](#_Toc24107387)

[Predetermined Moves (for trials) 9](#_Toc24107388)

[Adding New Network Configurations 9](#_Toc24107389)

[Server Variables 9](#_Toc24107390)

[Debugging 9](#_Toc24107391)

[Important Files 10](#_Toc24107392)

# Introduction

This guide is designed to provide a starting point for other people to tackle any bugs or implement new features for the Contagion game in the Influence Maximisation project. Please contact me at [lh9g14@soton.ac.uk](mailto:lh9g14@soton.ac.uk) for details not covered in this document.

I have tried to capture the dependencies in a prerequisites section – but if any are missing, please let me know. These should be simple to install.

The credentials are:

Gmail: contagiongamesoton@gmail.com / southampt0N

Heroku: [email] / southampt0N!

# Setup (Server - Local)

Prerequisites: Node.js (There may be some packages like Express that may require separate installation. Please contact me if this is an issue, and I can update the guide).

To run the server, simply change *Server.LocalMode* and *Simulations.LocalMode* to true and run *node server.js* in the command prompt in the ContagionServer directory. If you’re anticipating having to restart the server frequently, you may wish to call *servercode.bat*, which calls the above command after the previous instance is killed many times.

However, if you’re planning on doing any non-trivial debugging, I strongly suggest using Visual Studio Code, and following the guide here: <https://itnext.io/the-absolute-easiest-way-to-debug-node-js-with-vscode-2e02ef5b1bad>

Running the server locally allows you to test out any changes without having to go through the lengthier process of uploading your data remotely, and almost always behaves in the same way as the remote server. Any differences are controlled in the code by Server.LocalMode and Simulations.LocalMode – make sure these are set to true when running locally!

# Setup (Server- Remote)

## Initial Setup:

(Remember to change Server.LocalMode to false)

1. Install the Heroku CLI (works well with Git Bash if you have it) at <https://devcenter.heroku.com/articles/heroku-cli>
2. Navigate to the ContagionServer folder in the command line
3. Enter *heroku login* and login with the credentials listed above
4. Enter: *git remote add herokugit@heroku.com:stark-atoll-77422.git*
5. Enter: *git logs –tail* to check the status of the app

## Subsequently Accessing Logs:

1. Navigate to the ContagionServer folder
2. Enter: *git logs –tail*

## Restarting the Server:

The server will restart itself whenever you push something new to it. However, to restart the server without pushing new code, do:

1. Navigate to the ContagionServer folder
2. Enter: *Heroku restart -a stark-atoll-77422*

# Setup (Client - Local)

Once the server is setup (local or remote), starting the client is as simple as running *index.html* from the *ContagionClient* folder. This should work in all modern browsers, but has been tested in Firefox and Chrome. Make sure *LocalMode* in Simulations.js is set to correspond with your required server type (local/remote).

# Setup (Client – Remote)

Note: You may wish to use your own hosting platform for this. The steps below are for use with the free hosting platform I’ve been using during development, trials, conferences, etc.

## Initial Setup:

1. Create an account as itch.io (I have been using a personal account) at <https://itch.io/register>
2. Navigate to the Creator Dashboard at <https://itch.io/dashboard> and click Create New Project
3. Fill in the details, such as title, desired URL, etc.
4. Zip the ContagionClient folder and add it to the uploads section.
5. Click save & view page. The game will be ready to play within a minute (assuming the server is correctly setup).

## Updating the App:

1. Once logged in, navigate back to the dashboard at <https://itch.io/dashboard>
2. Click edit on the game you wish to change.
3. Zip the ContagionClient folder, add it to uploads and save as before.

Note: You may wish to upload several versions of the game for various reasons (e.g. one test build and one trial build for others to play). This is supported by the server! As long as you create some way to identify which version you are using (e.g. prepend some unique string to the player ID), and all versions use some logic supported by the server, this will work as expected.

# Setup (Database – Remote Only)

## Setup & Data Access (Method 1):

(NOTE: This is a more convenient command line method that involves less setup, but not one that I have personally tested.)

1. Login to Heroku using the above credentials
2. Go here: <https://dashboard.heroku.com/apps/stark-atoll-77422> and click "Heroku Postgress"
3. Click Settings -> View Credentials and note the details
4. Setup postgres on your machine by following <https://devcenter.heroku.com/articles/heroku-postgresql#set-up-postgres-on-mac>
5. In the ContagionServer folder (in command line), enter heroku pg:psql
6. The command line should prompt you for some details that match what you saw in step 3.
7. For overall game data, enter the query "SELECT \* FROM master\_games\_table;" or individual step data, "SELECT \* FROM player\_actions\_table;" (without quotes)
8. For any more complex queries (like get a game and associated moves), I can update the guide with this info.

Subsequent Times:

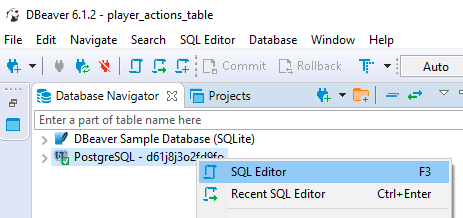
Follow steps 1-3, 5 & 7 (or just 5 & 7 if your command line interface remembers your details)

## Initial Setup (Method 2):

This is easier to get to grips with than method 1, but slightly more cumbersome.

1. Follow steps 1-3 of the above method to get the database details
2. Install DBeaver (any SQL client should work, but I found this easy to use – the following steps specifically use this software)
3. Click the ‘New Database Connection’ button at the top left
4. Select PostgreSQL, and fill in the sections using the information from step 1.

## Accessing Data (Method 2):

From there, you should be able to use PostgreSQL commands to access the data. Right click the database name in the navigator to the left and open the SQL editor. Enter your commands, and press Ctrl+Enter to execute. The tables used are *master\_games\_table, player\_actions\_table* and *player\_clicks\_tab*le*.*

# Updating Server-Client Communications

Overview

As this is a somewhat involved task, spread across the client and server files, a brief demonstration is included here.

Server -> Client communication is handled by *Server.sendClientMessage* in *server.js* in ContagionServer (outgoing) and *parseEvent* in S*imulations.js* in ContagionClient/js/sim (incoming)

Client -> Server communication is handled by the *sendServerMessage* function in S*imulations.js* (outgoing) and *Server.ParseMessage* function in *server.js* (incoming)

Both of these use the Message.js class, a very simple wrapper whose wrapper takes a *payload* (data to be sent) and *status* (identifier for the type of message). For instance:



Sends a NEW\_GAME\_TOKEN type message with a payload containing the client’s username.

The types of message that the clients and servers can send and receive are mutually exclusive. E.g. clients can only send (not receive) NEW\_GAME\_TOKEN messages, and servers can only receive (not send) these messages. This is not strictly enforced, but keeps things simple.

Implementing New Messages

For both client -> server and server -> client messages, implementing new types is a simple three step process.

1. Once you have the data you wish to send, put it in a list by enclosing the comma-separated data within square brackets (if there is more than one element to send), e.g.:

*var payload = [user.Username, user.Score, user.Fun];*

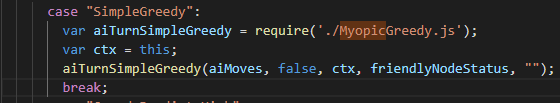
1. Then choose a descriptive name for your message type, e.g. USER\_RESULTS\_TOKEN, and create a new message & pass it to the dedicated communications function as shown above.
2. If you did this on the server, change to the client (and vice versa). Then, in the *Server.ParseMessage* or *parseEvent* function, change the switch function to include a new case for your chosen message type (e.g. USER\_RESULTS\_TOKEN), and implement the logic for how you want to handle the transmitted data from there. The communication function detects the message type and allows you to run whatever you like from there with the payload of the message.

# Other Useful Information

## Implementing new AIs

This requires some changes in a couple of different places.

In order to insert the AI into the existing infrastructure, navigate to *GameState.prototype.aiTurn* in *server.js*, and add a case to the switch statement.



For complex strategies, you may wish to put it in its own file like *MyopicGreedy.js*. This file must contain the line *module.exports = [name of main function of the strategy];* and the *require* function must be called in *aiTurn* as above, substituting the name of your new file. You will also need to pass *this* as a parameter if your strategy requires knowledge of game state (likely).

You will also need to pass parameters such as:

*aiMoves* – The chosen moves will be added to this. See existing AI implementations for how to do this, it is simple enough to copy and paste into your new strategy.

*friendlyNodeStatus* – as a result of implementing AI vs AI experiments entirely on the server-side, these strategies need to be able to work for both players, so hardcoding this part is not possible. However, this will have already be calculated for you and passed into *aiTurn*, so you simply need to pass this on.

*removeToken* – If you wish to re-implement players being able to remove tokens between rounds, most strategies will require you to find a token with the worst ‘fitness’ (i.e. the token that is contributing the least to victory). Passing this as *true* should remove a token and replace it with a token at the highest fitness node.

*Anticipating* – This is a simple augmentation to any strategy, where we add a dummy token to the opponent’s list of moves, for our strategy to react against. Be warned – this can sometimes have unexpected side effects. For instance, when myopic greedy naively predicts the highest degree node being taken by the enemy, it will place a token elsewhere – meaning that the opponent gets the highest degree node at turn one 100% of the time, instead of 50% without anticipatory augmentation. This makes the strategy perform much worse, as gaining the highest degree node at turn one is a strong indicator of final success.

In addition, if you wish to use this in AI vs AI experiments, add it to the *strategyNames* list in *ExperimentalAi.js*.

To change the strategy that the server is currently using, change the *Server.AiStrategy* variable in *server.js.*

## Predetermined Moves (for trials)

Every network topology (NOT layouts, which are alternative visual representations of the same topology) should have a list of 10 predetermined moves for use in trials where we want to minimise the impact of the AI (the random number generator also has a fixed seed at this point). These moves are found in *Server.TestMoves* and are the same for each topology every time the server is reset (but may differ which topology gets which moves if new lists are added). To use this with more topologies in future, just add an extra list.

## Adding New Network Configurations

For each new topology you wish to add, you need the following files:

1. *edges.csv* – a file containing one line for each connection between nodes. These lines are in the format *x,y* where x and y are the indices of the nodes to be connected.
2. *positions\_N.csv* – N files containing K lines, one for each node (the game is written with 20 nodes in mind – changing this will require some effort). This has the x/y coordinates of each node, scaled between 0 and 1. There can be N of these files, one for each layout of that particular topology.

NOTE: For node positions, (0,0) is the top left corner. (0,1) is the bottom left corner, (1,0) is the top right corner, and (1,1) is the bottom right corner.

These files should be in a folder, named as the title of the topology. This folder then goes into *Config\_Files* in *ContagionServer*, where it will be picked up by the server upon restart.

The *NetworkConfigurations.js* file shows how these files are processed by the server, and how elements such as the Laplacian are linked to game states. There is a lot to unpack in this area, though it is all handled for you, and is unlikely to need any further changes. However, if you would like a more detailed explanation, please contact me and I can expand upon this section.

## Server Variables

Much of the server behaviour (such as whether it’s running on local mode) that you might like to change frequently is parameterised. Check the top of each file to see a list of these (there is also another section within *server.js* for class variables for the *GameState* class). These variables are typed in camelCase, but with the first letter capitalised to distinguish them from other variables.

These can also be found in *ExperimentalAI.js* and *Simulations.js* (in *ContagionClient*).

## Debugging

For the client side, I recommend using the Chrome debugger tools, which are easy to use, and simple tutorials can be easily found online.

For the server side, I strongly recommend using Visual Studio Code with its debugger. A great guide to getting this setup can be found at <https://itnext.io/the-absolute-easiest-way-to-debug-node-js-with-vscode-2e02ef5b1bad>

This will allow you to debug like in any other powerful IDE. However, if you need a demonstration on how this works, please let me know.

# Important Files

I strongly recommend using Agent Ransack (<https://www.mythicsoft.com/agentransack/>) for searching keywords – it is a powerful tool for searching the inside of files & has excellent filtering functionality (usually you’ll want to use \*.js for the file name).

The files I have created/made significant changes to over the course of this project are as following (note - this does not count library files, deletions from the original project, or files you are very unlikely to need to change):

In *ContagionClient/js:*

*Server/Message.js* (client-server comms)

*Chapters/A\_Preloader.js (to change the initial splash screen)*

*Chapters/Strategy* (container for the game, things like where the Next Round button is placed)

*Chapters/Tutorial*

*Sim/Connection* (Connections & Interactions between peeps)

*Sim/ConnectorCutter* (Adding & removing tokens)

*Sim/Peep* (Anything else to do with nodes, e.g. what infection status, ID, current tokens, colour)

*Sim/Simulations* (All other client-side logic is handled here, including cookies, server connections, score, timers, integration of other classes into main game – think of this as the main file)

*Slideshow/\** (Infrastructure-related items, such as pop-up boxes, music)

In *ContagionServer:*

*Server.js*

*NetworkConfigurations.js* (handles importing configs)

*MyopicGreedy.js*

*ExperimentalAi.js* (handles AI vs AI experiments)

*Message.js* (Identical to ContagionClient version)