Distributed Systems, Spring 2013
Assignment 3: Bootstrapping Capture-the-Flag

Introduction

The goal of this lab is to get us working on the capture the flag game.

1 Hand Out Instructions

My suggestion, as always, is that you meet early and organize yourselves and the tasks to be done. Don't forget to document all your meetings.

1.1 Capture the Flag

As part of this assignment you are to discuss in detail your interpretation of the game and develop a specification and game architecture. See `http://en.wikipedia.org/wiki/Capture_the_flag` for additional information. Each group will present their specification in a 10 to 15 minutes class presentation.

1.1.1 Specification

Your specification should at least include:

1. Types of maze cells and states that can be associated with a cell.
2. States and attributes that can be associated with each player.
3. The set of actions that a player can take.
4. For each action define associated rules governing the side affects associated with the action.
5. Conditions and Rules associated with winning the game.
6. Semantics and Rules for the flags, home bases, and jails and how players interact with each.

Create a 'doc' sub-directory and put your write up of the specification. Also place a copy of any additional material you plan to use to present your specification in this directory. Don't forget to add this directory to your repository using the 'git add' and 'git commit' commands.

1.1.2 Architecture

Discuss and define an architecture for you game. What roles and function the server and clients will take? How will you decompose the software (a diagram might be useful here)? Are the clients considered trusted? Eg. Will the client software be trusted to enforce the games rules and semantics? What considerations will impact the performance and scalability of the server?

Place the 1.5 - 2 page architecture white paper in your doc directory.

1.2 The Maze

Your task is to construct a simple server and client that let you play with the provided maze file. You will find a maze file here `http://www.cs.bu.edu/~jappavoo/Resources/451/daGame/daGame.map`. This is a simple ascii file that describes the default game maze. The maze is composed of 200 lines with each line containing 200 columns. Each possible character position represent a single cell of the maze. It is assumed that half of the maze's columns belongs to one team and the other to the second team – team one's portion of the maze is columns 0-99 and the other columns 100-199. The following defines what the character symbol's for a cell mean:

' ' A white space character indicates that a cell is floor cell.

'#' A pound character means that the cell is a wall cell.

'h','H' A home cell for team 1 and 2 respectively. Home cells are assumed to also be floor cells.

'j','J' A jail cell for team 1 and 2 respectively. Jail cells are assumed to also be floor cells.

Consider what states and operations will be associated with the maze and construct maze.c and maze.h files in your library. In these files define your data structures and operations on the maze. You might want to consider creating types/structs for: position, cell, and maze. Where maze is a structure that conglomerates all the elements needed to describe the maze. Remember in the long run you will need the server and clients to carry out various operations on the maze and performance will be a central requirement. How you define these data structures will influence the complexity and implementation of these various operations. Don't forget today's computers have considerable memory so it is feasible to use data structures that are large but permit order one operations.

You should at least include functions for:

load: given at least a file name creates and loads the data structures that you will use to describe and manage the maze. Note it is a good idea to have this code return back an error status if the maze could not be loaded. In this way the calling code can abort cleanly. It is also good practice to have this code create and return a handle (pointer) to the newly loaded maze so that future operations on the maze can be passed this handle.

dump: given a handle to the maze data structure print to standard out the state of the maze using the character definitions. You can use this output to verify that your maze data structure is correct by comparing it with the map file that the maze was loaded from.

Next you will construct a simple server that uses your library maze routines to load the maze and permit clients to connect and query basic facts about the maze state.

1.2.1 Server

The server should load the map file and wait for clients to connect and allow for clients to query the map for basic facts and also dump to it's standard out the maze if directed to do so. See the description of the client below to understand what facts you need to support.

1.2.2 Client

Construct a simple text based client that prints a prompt and supports the following:

connect <ip:port> connect to a server. Once connected this command should do nothing. On failure it should display an appropriate message

numhome <1 or 2> queries the server for the number of home cells for the specified team. Eg. 'daGame¿ home 1' would print the number of home cells that team 1 has. Appropriate error messages should be displayed if a bad argument is passed or other error occurs.

numjail <1 or 2> Same as previous but for jail cells.

numwall returns the number of wall cells.

numfloor returns the number of floor cells. Don't forget that home and jail cells are also floor cells.

dim returns the dimensions of the maze.

cinfo <x,y> returns information about maze cell at x,y. Where x and y are the column and row of the cell. Information displayed should include: What type of cell it is? What team it belongs too? Is it occupied?

dump Causes the server to dump the maze as ascii to it's standard out.

quit does the obvious.

2 Evaluation

I will expect to be able to look at your repo and find a a3 branch that I can evaluate.

I will look at your source, LOG file, and commit logs. I will review your specifications and architecture documents.

I will test your client and server to see that they work and conform to the requested function. Don't forget to document any meetings in which you discuss what you learnt from your readings and explorations.