CSCI415—Systems Programming

Spring 2017

Programming Assignment 4

1 Distributed Tic-Tac-Toe

The old paper-and-pencil game called tic-tac-toe serves as the basis of a distributed program in this assignment. The intent is to create a facility by which individuals anywhere on the internet can rendezvous to play a game of tic-tac-toe across the net. This assignment will require that you use both datagram and virtual circuit communication. Even though the application is a toy, with some requirements made for pedagogical reasons, the programming techniques are the same as you would use in the construction of a production networked application.

You must design and implement two programs for this assignment:

- `TTT`: is a server program which is configured to run as a non-terminating daemon. It provides a facility by which potential players can arrange a match; once a match has been arranged, it manages that *single* match (one-match-at-a-time service).
- `ttt`: is the client program which is used by a potential player to get an opponent. It can also be used to simply query the server to see who, if anyone, is engaged in a match, and who might be waiting to get an opponent. If `ttt` is used to get an opponent, it handles the play of the match until a win, loss, or draw is eventually achieved.

The operation of the two programs is detailed below.

2 The Client

The `ttt` program is the client in this system. If invoked as

```
% ttt -q
```

it is operating in query mode. It should send a datagram to the server assumed to be running at a well-known port on a well-known system (see Notes below). It expects to receive a datagram from the server in reply to its query. That reply should indicate whether or not there is a game in progress. If so, it should give the handles (see below) of the the players. If there is no game, but there is a request to play pending, the server should state so and give the name of the player who is waiting for an opponent.

If the client is invoked simply as

```
% ttt
```

it is an attempt, on the part of the invoker, to play a game of tic-tac-toe. In operational terms, `ttt` should attempt to establish a connection to the server, again assumed to be running at a well-known port on a well-known system (see Notes below). When the server accepts the connection, the `ttt` client will serve as an interface to enable play of the game. The client may be invoked as

`% ttt -t` *timeout*

where *timeout* must be an integer. If there is no connection within *timeout* seconds of the time that `ttt` is invoked, the client should abort. Even if there is a connection, if there is no match arranged within *timeout* seconds, the client should abort.

Assuming that a connection is established between a client and the server, the server will ask the client for a *handle*, a string used to identify the player. If an opponent registers with the server, the server will eventually tell the client the handle of the opponent and whether the player is X or O. The server will then ask the client for a move; as part of that request, the server will send information which shows the current configuration of the board. A move is specified by a single integer from 1 through 9 inclusive, corresponding to a square on the board as shown in figure 1. The client should do simple sanity checking, not allowing a move in a previously occupied square, and should send the move to the server. Eventually, the server will send a message that states the final outcome of the game (win, loss, or draw), and also includes the final configuration. The client is responsible for maintaining a display of the state of the board in keeping with information sent it by the server. When a game is complete, both clients should terminate cleanly.

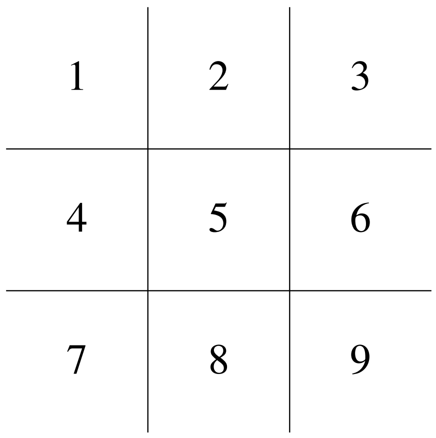

Figure 1: The Board Layout

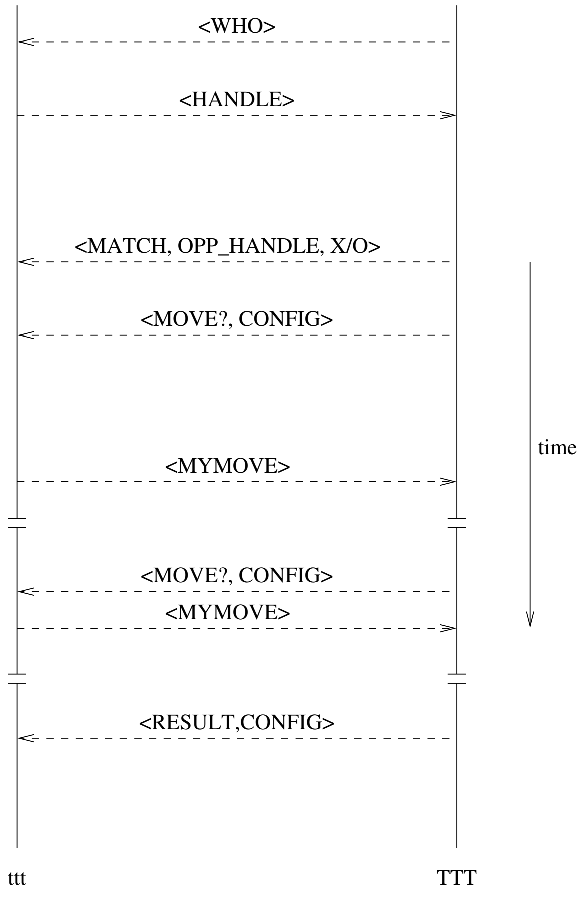

Figure 2: Client/Server Interaction

A diagram of the interaction between a client and a server is shown in figure 2. The form of the data is up to you; labels on the figure are for informational purposes exclusively.

3 The Server

The server must allow two modes of interaction with clients. In query mode, a client uses datagram communication to get information from the server (the form of that information is described above). Clients will also attempt to establish virtual circuits to the server in order to engage in a match. The server will support a single match at a time; hence, the server needs to deal with a maximum of two virtual circuits at any point in time. When the server allows creation of a virtual circuit, it asks for a string handle of the player. Once two players have established virtual circuits, the server chooses one to be X, one to be O, informs each client of that choice, and then asks X for a move, sending the current board configuration. When the player replies, the server records the move and engages in similar dialog with the other client. Players alternate turns until some player wins by getting three of her symbols in a row (horizontally, vertically, or diagonally). If all squares of the board are filled without either player getting three symbols in a row, the game ends in a draw. After reporting the result of the game to both players, the server should close its end of the virtual circuits which supported the completed game and be in a position to accept new connections.

4 Notes

Some incidental bits of information:

1. An implicit assumption in the structure of this system is that the server can be reached by any client at a "well-known address." If you had root access to the system, this wouldn't be a problem (you could dedicate two of the first 1k ports to this service). Since you don't have root access, I will allow you a single file (shared across all systems in the department) in which you may write things which allow readers and writers the illusion of a well-known address for the server (for example, a symbolic host name and two port numbers). If this file does not exist when clients try to access the server, they should sleep for 60 seconds, retry accessing the file, and then they should terminate if the file remains inaccessible.

   Note that this file is used exclusively to locate the server cannot be used to store any state information related to the server.

2. Queries should be honored by the server whenever it can. A game in progress is no reason to postpone honoring queries. From a client's perspective, if it doesn't get a reply to a query within 10 seconds, it should terminate with an appropriate message.

3. The server should be bulletproofed against bad behavior on the part of clients. In particular, a client timeout should not cause the server to function incorrectly. The death of a client (crash of a client's machine) should cause the server to cancel the game. An unexpected response from a client should cause the server to cancel the game. After canceling a game, whatever the reason, the server should revert to normal mode of operation.

4. Your server only supports interaction with six clients who want to play a game: two in game play and four waiting. Beyond these six, you should discard clients.

5. Be careful to kill any processes you may fire up on departmental systems.

6. Figure 3 shows a snapshot of a running system consisting of the server, two clients engaging in a match, a client querying the server, and a client awaiting a match. Here's the socket inventory:

   - The socket labeled 1 is a `SOCK_STREAM` on which the server has invoked `listen()`. The server fields client connections on this socket.
   - The socket labeled 2 is a `SOCK_DGRAM` on which the server receives queries.
   - Sockets 3 and 4 are a connection between a client and the server as are sockets 5 and 6. The clients on these two connections are engaged in a game.
   - Sockets 7 and 8 are a connection between a client and the server. This client wants to play a game, but the server is already managing a game. This client must wait (both for an opponent and for the current game to end).
   - Socket 9 is a `SOCK_DGRAM` associated with a querying client.

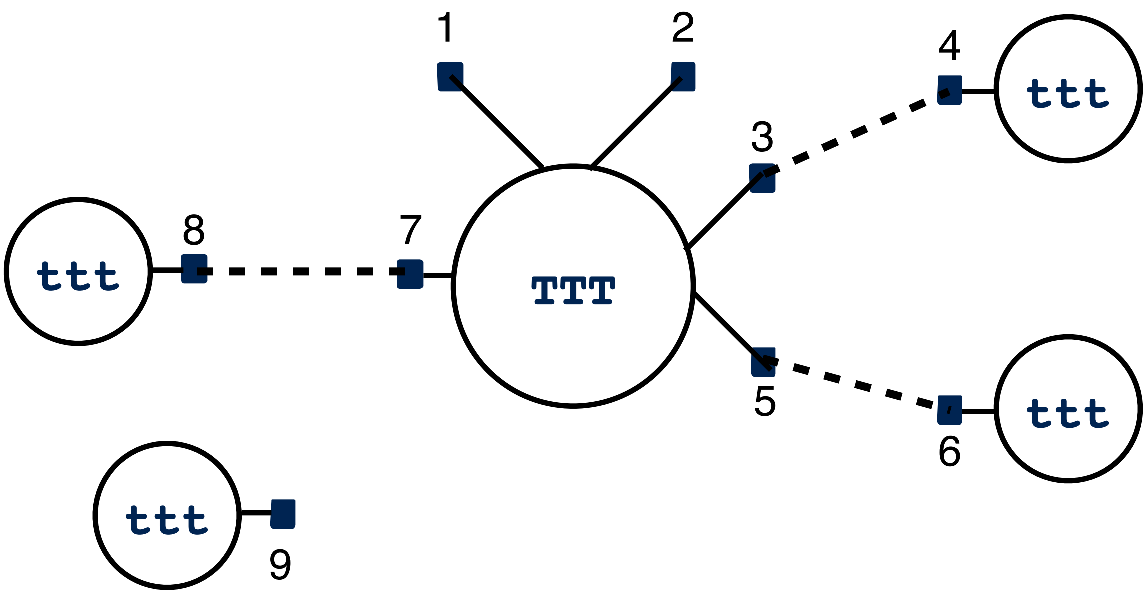

Figure 3: A Working System

5 Submitting

How you structure your code is up to you. You must prepare a `Makefile` and all necessary source files so that the TA can simply do a make and build `ttt` and `TTT`. To that end, create a directory called `ttt.d` in which your `Makefile` and all required source files will reside. Make sure that `ttt.d` is protected 0700 to prevent others from looking at your source. Make the *parent* of `ttt.d` your working directory and do the following:

```
- tar zcvf ttt.tgz ttt.d

- gpg -r CSCI415 --sign -e ttt.tgz

- mv ttt.tgz.pgp ~ ; chmod 0644 ~/ttt.tgz.pgp
```

See the web site for the submission deadline.