CSCI 4061 Programming Assignment 4

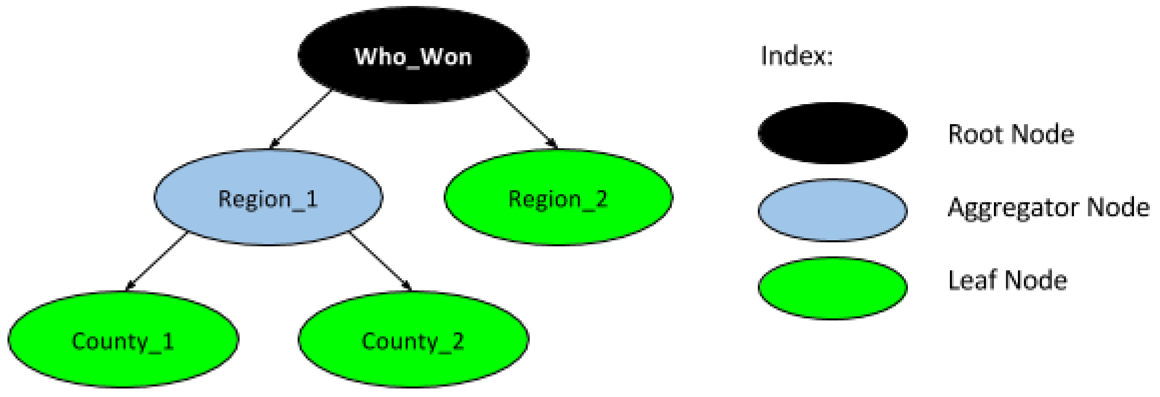
Due: Friday, April 27th
Weight: 100 points, 10% of total grade
Group Work: It is recommended to do the assignment in groups of no more than 2.
Topics: Sockets, TCP, network programming.

For this assignment, you will extend the vote counting idea we have been working on for the last three assignments to a networked vote counter. In the real world, votes have to be transmitted from where they are counted to more central locations to be properly summed and recorded. You will implement this application in a client-server model, meaning that you will have two programs, a client and a server, which will communicate using a protocol we define. Let's start with a review of the underlying data structure of our vote counter applications, the voting region DAG, and the differences in this assignment.

Don't let the 14 pages scare you, there's a lot of tables that take up a lot of space.

1) The DAG

As in previous assignments, we consider a DAG as in the below figure:



Recall from previous assignments, we have the root node, aggregator nodes, and leaf nodes. In previous assignments we have operated under the understanding that votes can only be added to leaf nodes, and are then aggregated until the final root node. We continue this pattern here, but add the concept of polls.

In real life, voting areas have polling places where votes are cast. These polling places are opened to begin voting, then closed at the end of voting. We add this concept here, where each node not only has the summed votes in it, but also a boolean flag indicating whether its polls are open or closed. Polls must be opened prior to adding votes, closed prior to the final count, and cannot be reopened after closing. Even though they cannot have votes added to them, we give aggregator nodes and the root node a flag for polls as well. **Any change made to the polls of a node with children should be reflected in its children, in that it should propagate down the tree recursively until the leaf nodes.**

To sum up, all nodes contain the current votes for each candidate and a flag indicating if the polls are open. Votes can only be added directly to leaf nodes, and the votes of an aggregator node are the sum of its children's votes. While we can open or close the polls of any node, they are affected hierarchically (opening an aggregator node's polls opens its childrens, closing all of an aggregator children's polls closes the aggregator's polls).

2) The Server

The server is responsible for reading a DAG input file (in the same format as in PA3, more info later), maintaining an accurate data representation of the DAG, and properly responding to all requests from clients, as well as printing out relevant information to STD_OUT. The server should be able to operate on any port passed to it from the command line, accept and maintain connections from 1 up to 5 clients.

The usage of your server program should be as follows. You must use the executable name "server".

```
./server <DAG FILE> <Server Port>
```

```
Where

<DAG FILE> is the file containing the specification of the DAG,
under any filename.

<Server Port> is any unsigned integer to be used as a port
number.
```

Again, your server is meant to load the DAG into memory, then wait for connections on the specified port. When it receives a connection, it should spawn a new thread to handle that connection, as long as you do not have the maximum number of connections currently active. The thread which handles the connection will be responsible for reading the client's commands from a socket set up between them, doing the necessary computation on the DAG (including summing votes, opening and closing polls), and sending responses to the client back through that socket.

Summing votes and propagating them up the tree will have to be done synchronously, and also done immediately after receiving the request (greedy execution, no waiting). The same applies for opening or closing polls.

The information the server should print out includes:

- `“Server listening on port <PORT_NUMBER>”`
- `“Connection initiated from client at <CLIENT IP>:<CLIENT PORT>”`
- `“Request received from client at <CLIENT IP>:<CLIENT PORT>, <REQUEST_CODE>,` **`<REQUEST_DATA>`**`”`
- `“Sending response to client at <CLIENT IP>:<CLIENT PORT>, <RESPONSE_CODE>` **`<RESPONSE_DATA>`**`”`
- `“Closed connection with client at <CLIENT IP>:<CLIENT PORT>”`

`<CLIENT_ID>` should be the IP of the client connected. Please follow the format of these prints closely.

3) The Client

The client is responsible for initiating a connection with the server on the IP and port given to it, reading a file containing requests to send to the server (more info later), sending those requests to the server, and printing out the relevant information to STD_OUT. A client only needs to connect to one server at a time.

The usage of your server program should be as follows. You must use the executable name “client”.

```
./client <REQ FILE> <Server IP> <Server Port>
```

```
Where

<REQ FILE> is the name of a file containing the requests to be
sent to the server.

<Server IP> is the IP of the server to connect to.

<Server Port> is the port of the server to connect to.
```

Again, your client should connect to the specified server, then send it the requests as defined by the <REQ FILE>, print out relevant information to STD_OUT, then close the connection. When the request involves adding votes to an area, the line in the <REQ FILE> will contain a filename containing the name of a file which contains the raw votes (candidate names, one vote per line). The client will have to sum the votes and send the total votes per candidate to the server.

The information the client should print out includes:

- `“Initiated connection with server at <SERVER IP>:<SERVER PORT>”`
- `“Sending request to server: <REQUEST_CODE>` **`<REQUEST_DATA>`**`”`
- `“Received response from server: <RESPONSE_CODE> <RESPONSE_DATA>”`
- `“Closed connection with server at <SERVER IP>:<SERVER PORT>”`

Please stick closely to this output format.

The overall operation should look like this. The communication between the server and client will be done using our application-layer communication protocol, below.

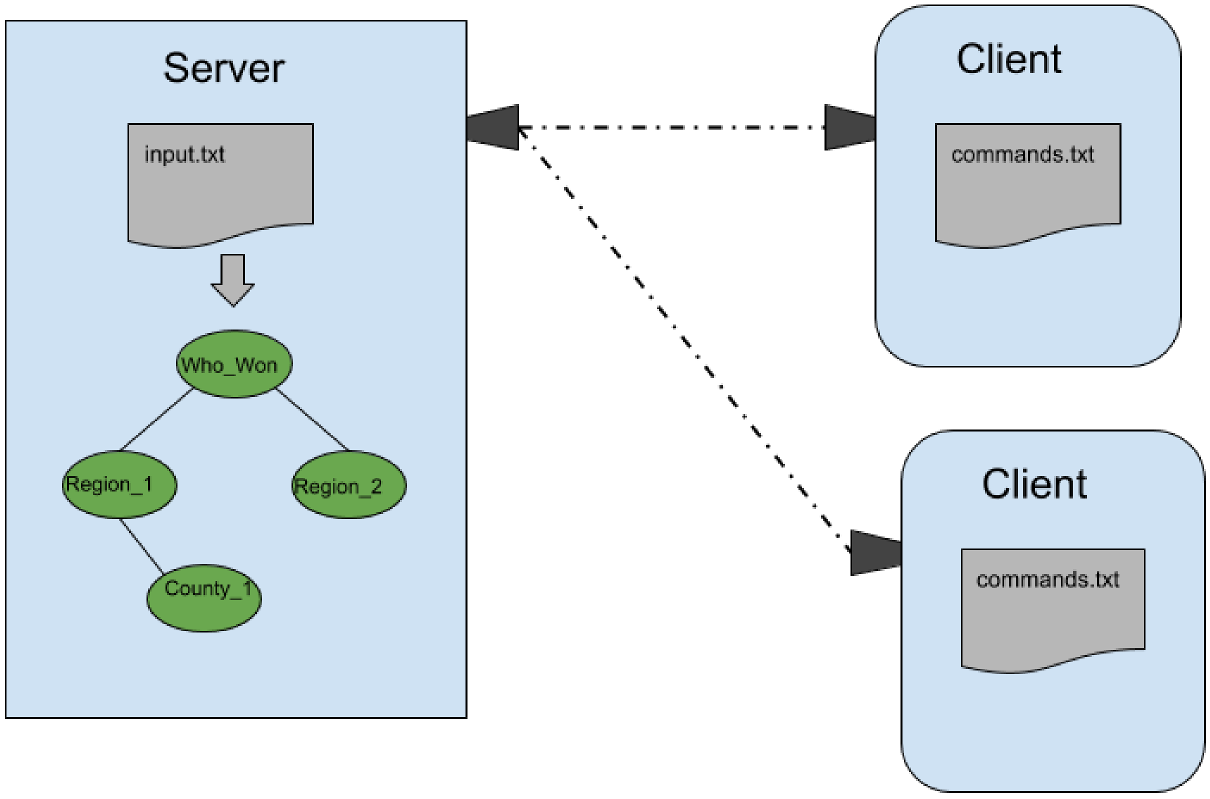


Figure 1: Example structure with DAG.

4) Communication Protocol

For this assignment, communication protocol is an application-layer protocol formed of requests and responses. Requests will be sent from the client, received by the server, and after the server does any necessary computation, responded to by the server. These requests and responses will be strings with the following structures.

4.1) Requests

The request structure is as follows, with each field being separated from the other by a semi-colon, and the entire string being delimited by the string terminator ‘\0’. This is the single ASCII character ‘\0’.

Table 1: Request Structure

Field Name	**Size (chars)**	**Purpose**
Request code	2	Specifies the request. See Table 2
Region Name	15	Specifies the region the command should operate on. Padded with spaces, if empty, fill with only spaces
Data	Arbitrary	Relevant data for the command (i.e. votes to add or remove). Not necessary for all commands, may simply be left empty.
Terminator	1	‘\0’, marks the end of the command packet.

Table 2: Request Codes

Request Name	**Request Code**	**Uses Region Name?**	**Uses Data?**
Return_Winner	RW	No	No
Count_Votes	CV	Yes	No
Open_Polls	OP	Yes	No
Add_Votes	AV	Yes	Yes
Remove_Votes	RV	Yes	Yes
Close_Polls	CP	Yes	No

Each request is below, along with the action it represents in relation to the DAG.

- Return_Winner
    - This request asks for the winner of the election, the candidate with the majority of the votes over all areas. Requires that the polls have been opened and closed in all areas.
- Count_Votes
    - This request asks for the candidate who has the most votes in the specific region given (and all sub-regions). This does not require that the polls be closed in this area, nor that they be opened, but should return correct data always. If there are no votes yet, return "No votes."
- Open_Polls
    - This request opens polls for voting in an area. MUST be called before changing votes or closing polls. Respond with a poll failure (PF) error if already open.
- Close_Polls
    - This request closes the polls for a given area. Must be called for an area for which polls have been opened. Respond with a poll failure (PF) error if already open.
- Add_Votes
    - This request adds votes to a given area. Polls must have been opened in this area, and not closed. Candidates that have not been seen before should be added, and not all candidates recorded in the region must be present in the command. Candidates who are neglected should be ignored.
- Remove_Votes
    - This request removes vote from a given area. Polls must have been opened in this area, and not closed. **If a candidate does not exist in the**

region, or the number of votes to be subtracted is greater than the total number of votes, this is an illegal subtraction and an error response is necessary.

Request format examples:

"RW;   ;\0" -- Return winner overall. No region or data is needed.
"OP;County_1;\0" -- Open polls for County_1. No more data besides the region name is needed.
"AV;Count_1;A:1,B:3,C:4,D:5\0" -- Add votes to County_1.

4.2) Responses

The form of the responses is below. Again, each field is separated by a semicolon, and the whole command is terminated by a '\0' character.

Table 3: Response Structure

Field Name	**Size (chars)**	**Purpose**
Response code	2	Specifies the response, see Table 4.
Data	Arbitrary	Returned data from the server.
Terminator	1	'\0', marks the end of the command packet.

Table 4:Response Codes

Response Code	**Meaning**	**Returns Data?**	**Data Returned**
SC	Success	Sometimes	Varies, See Table 5 below.
UE	Unhandled error	No	N/A
UC	Unhandled Command	Yes	Command
NR	Region does not exist	Yes	Name of region
RO	Region(s) not closed. Caused by Return_Winner requests with open regions.	Yes	Name of region
RC	Region(s) not opened. Caused by adding or removing votes with to a closed region.	Yes	Name of region
RR	Polls cannot be reopened. (already closed or votes have been added)	Yes	Name of region
PF	Poll failure. Either opening an already open region or closing a closed region.	Yes	Name of region and its current state, i.e. "County_1 open."
NL	Not leaf region, cannot add or remove votes.	Yes	Name of region
IS	Illegal subtraction	Yes	Name of candidate

Table 5:Data Returned on Success

Request Command Name	**Data Returned On Success**
Return_Winner	"Winner:A", A being the correct candidate name.
Count_Votes	Current count of votes for region in format: "A:12,B:15,C:20"
Open_Polls	No return.
Add_Votes	No return.
Remove_Votes	No return.
Close_Polls	No return.

Response format examples:

"SC;\0" -- Returns success, no data.
"SC;Winner:A\0" -- Returns success, with data.
"UE;\0" -- Returns general failure, no data.
"NR;Region_99\0" -- Returns specific failure code with related data.

This covers the communication protocol. Next we'll talk about input files.

5) Input Files

In this section, we will cover the input files for the server and client.

5.1) DAG Files (Server)

On the server side, we need to have a format for the DAG file. This will be the format used in Project Assignment 3, where every node is listed followed by its children, delimited by colons. An example, which matches the DAG shown in Figure 1 follows below:

Who_Won:Region_1:Region_2
Region_1:County_1

You can assume that all regions have unique names that are between 1 and 15 characters long, composed of letters, numbers, and underscores, and that a DAG file will always have a Who_Won file.

5.2) Command Files (Client)

On the client side of things, we need a format to define the commands we want the client to send to the server. These commands are stored in a commands.txt file, in the following format:

Each line is a command. The command starts with the name of the command (as in the above table, i.e. Add_Votes), then the relevant data. For all commands except Return_Winner, there is some relevant data. All other commands have a region name, and then Add_Votes and Remove_Votes have filenames for the vote changes after that. These fields are delimited by spaces.
An example follows below. In this example, the same DAG used in the previous examples has each area opened, adds votes to a few areas, gets an intermediate update using Count_Votes, removes votes from one area, then has the areas closed and gets the winner.

```
Open_Polls Who_Won
Add_Votes County_1 County_1.votes
Add_Votes Region_2 Region_2.votes
Count_Votes Region_1
Count_Votes County_1
Remove_Votes County_1 fake.votes
Count_Votes County_1
Count_Votes County_99
Close_Polls Who_Won
Open_Polls County_1
Return_Winner
```

Examples

The series of requests and responses produced by the commands above being sent to a server containing the DAG defined above by a client sending the commands defined above would look like this. Exact vote data may not be accurate.

Command In Command File	Request From Client	Response From Client
`Open_Polls Who_Won`	`OP;Who_Won;\0`	`SC;\0`
`Add_Votes County_1 County_1.votes`	`AV;County_1; A:5,B:1,C:5\0`	`SC;\0`
`Add_Votes Region_2 Region_2.votes`	`AV;Region_2; A:1,B:5,C:10,D:6\0`	`SC;\0`
`Count_Votes Region_1`	`CV;Region_1\0`	`SC;No votes.\0`
`Count_Votes County_1`	`CV;County_1\0`	`SC;A:1,B:5,C:10,D: 6\0`
`Remove_Votes County_1 fake.votes`	`RV;Count_1;A:1,C:2 \0`	`SC;\0`
`Count_Votes County_1`	`CV;County_1\0`	`SC;B:5,C:8,D:6\0`
`Count_Votes County_99`	`CV;County_99\0`	`NR;County_99\0`
`Close_Polls Who_Won`	`CP;Who_Won;\0`	`SC;\0`
`Open_Polls County_1`	`OP;Count_1\0`	`RR;County_1\0`
`Return_Winner`	`RW;;\0`	`SC;Winner:C\0`

Convenient Assumptions and Hints

- We will release synchronized multithreading code examples to help you. They will not be for copy-and paste, however.
- You should use TCP sockets, because the order for messages matter.
- A client only needs to connect to a single server at any time.
- A server needs to consider multiple clients at a time.
- Candidate names have an upper bound of 100 characters.
- Region names have an upper bound of 15 characters.
- A max of 100 candidates and 100 regions is permissible.
- You will need to keep your DAG in memory. There are no files to output.
- You will need to synchronize your DAG.
- To pick your port number, use the last four digits of your student number
- First get a single client and server pair working, then try multiple clients
- Start on local host, then later work on multiple machines.
- **The maximum message size is 256 bytes.**

Useful Functions

You will use the standard socket functions, including `socket()`, `bind()`, `connect()`, `listen()`, `accept()`, `send()`, `recv()`, `close()`, `shutdown()`. You will also need to use pthread functions, including synchronization structures such as mutex locks. It is important that you ensure synchronized behavior for the DAG so that you don't get incorrect vote sums.

Extra Credit

For the extra credit, you will also implement a command which adds a new region to the DAG on the fly. You can only add a region as a leaf (the child of a region, no children of its own). Once the region is added, you should print a message to STD OUT, and the region should function properly. To test this, use a DAG input file with only the Who_Won region, and add every other region using this command. You do not need to consider adding a node if there are no nodes, you can assume that there will at least always be a Who_Won region.

The command will be as follows:

Command Name	**Command Code**	**Uses Region?**	**Uses Data?**
Add_Region	AR	Yes	Yes

The region should be the region which will be the parent of the new region, and the Data will be the name of the new region. So, to add a Subcounty_1 to County_1, the following command should be used.

"AR;County_1   ;Subcounty_1\0"

Pad the first part, because it is a region name, but not the second, since it is the data of the command.

You can and should assume that Add_Region is called before any other command, so there's no need to worry about copying votes.

Submission

For your submission, please turn in all of the files needed to compile your client and server. Do not turn in the executables, turn in the source files. Also turn in a Makefile, and a README with the following format:

Names: <Student_1>, <Student_2>
X500's: <Student_1, <Student_2>
Lecture Section: Student 1 is in section 01, student 2 is in lection 10.
Extra Credit: <Put whether you are doing the extra credit here>
Additionally, put any other information about your submission here.

Files you should be turning in:

- Client source files
- Server source files
- Makefile
- README

Put all of your files in the root of your zip file. Save yourself points: unzip the folder and make sure everything is in the root, then try to compile and run it. Do not turn in any unnecessary files, do not turn in any test cases.

Again, only one member of your group should turn in the zip.

Grading

A complete rubric will be released at a later date. You should expect some points to depend on a proper submission, some for your makefile and readme, and then the rest of your points to be split evenly between the client and the server. Most of these points will be based on functionality testing, but some will be given for coding style and comments. The rubric will follow the form:

5 pts -- Submission format
5 pts -- ReadMe
5 pts -- Makefile

10 pts -- Client source file(s), style and comments.
10 pts -- Server source file(s), style and comments.

Single-Client Case
5 pts -- Client operates (Reads commands.txt,connects, sends and receives.)
5 pts -- Server operates (Reads DAG.txt, connects,receives and sends.)
5 pts -- Server handles error cases.

Multi-Client Case
5 pts -- Server maintains multiple client connections (Maintains order)
5 pts -- Server maintains synchronization.

Test Cases
40 pts -- Test cases, evenly weighted, variety of command orders and DAG complexities

10 pts -- Extra Credit

Possible 110 points.

A complete rubric, the testing server and client executables, and some test cases will be released at a later date. You should be able to use our server to test your client and vice versa.

Have fun!