Design Document

Scalable Server

COMP8005 BCIT 2016

February 5, 2016

Authored by: Marc Vouve

DESIGN DOCUMENT

SCALABLE SERVER

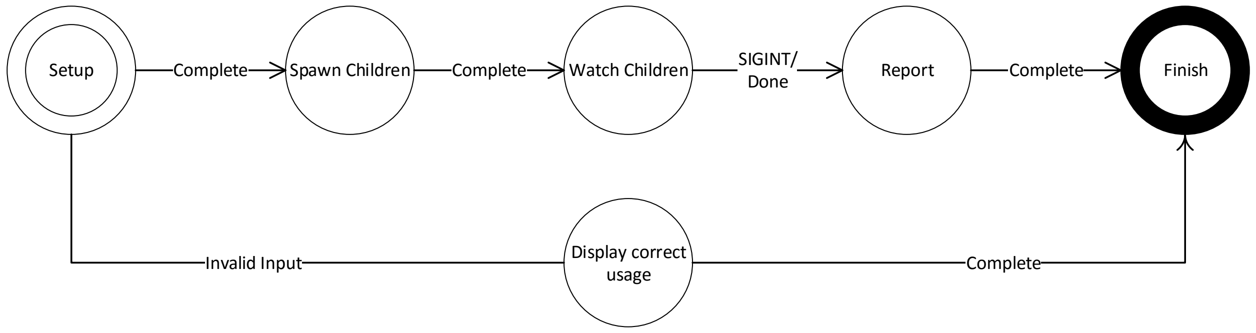
Contents

Client – Multi-threaded

The client is intended to be used to test the various scalable servers. It does this by creating multiple child clients which connect to the server and report information about the connection upon the connections closing.

Pseudo-code and Finite State machines shown below.

Finite State Machine: Parent Observer



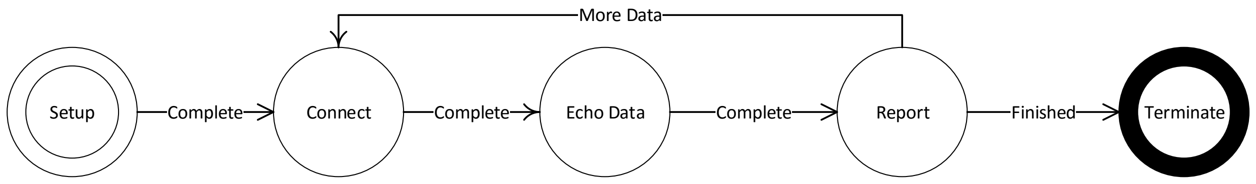
State	**Description**	**Events**
Setup	Setup parses user input and initializes some required data structures to be used later in the program	**Complete:** Completed state successfully **Invalid Input**: Users input was not valid
Display Correct Usage	Display how the command should be invoked.	**Complete:** Finished displaying proper input
Spawn Children	Create the children and start connecting to the server	**Complete:** All children have been created.
Watch Children	Save data passed by the children to be reported when the program exits.	**SIGINT/Done:** If either SIGINT is received or workers have all returned
Report	Generate a report based on data passed by the child routines.	**Complete:** File has finished generating

Pseudocode: Parent Observer

The parent thread also acts as an observer; it manages data about child threads as they close.

```
1. Validate/parse user input
2. If user is invalid inform user, and terminate the program
3. Create data transfer mechanism which is capable of passing data about connections
4. Spawn child threads based on user inputs
5. Read and save information from child threads
6. Upon program termination, or all threads being finished build a report based off connection
   information.
```

Finite State Machine: Child Worker



State	Description	Events
Setup	Prepare for sending data	**Complete:** Finished sending data
Connect	Make a connection to the server.	**Complete:** When the connection has been successfully established with the server.
Echo Data	A user specified number of times, transfer data to the server and read the data back. Store the round trip time and calculate an estimate of the average RTT.	**Complete:** Finished echoing to server for the desired iterations
Report	Send information about the connection to the parent, specifically the number of iterations completed and the average RTT	**Finished:** The worker has no more data to send. **More Data:** The user specified more iterations be made to the server than have already been completed
Terminate	Finished, exit.	

Pseudocode: Child Worker

The child worker threads connect to the server based on specifications provided at runtime.

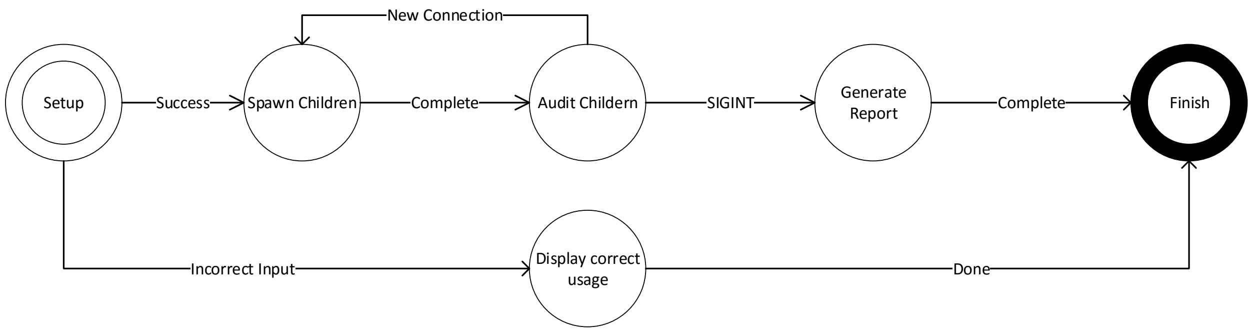
1. Connect to the server
2. Get current time for logging
3. Send Data to server
4. Read Data from server
5. Repeat steps 3 and 4 based on user specified inputs
6. Calculate time taken per iteration and send to communication channel

Server Multi-Threaded

The multi-threaded scalable server is both conceptually and by design the simplest one. For each new connection received the server spawns a child process which interacts with the new client.

Below is the pseudo-code and finite state machines for the multi-threaded server

Finite State Machine: Parent Observer

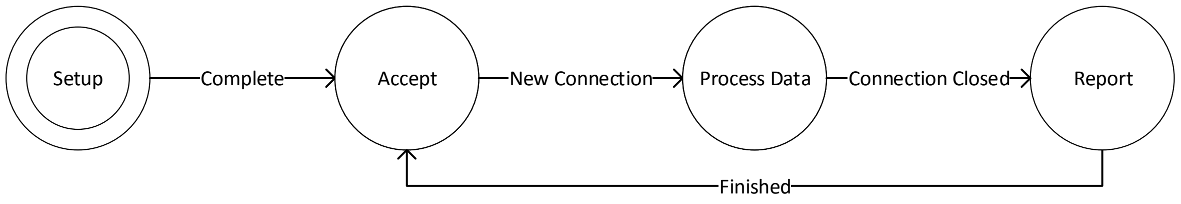


State	Description	Events
Setup	Setup parses user input and prepares the basic program	**Success:** User inputs were valid and the main section of the program may start **Incorrect Input:** User inputs were bad.
Display Correct Usage	This state simply displays the correct program usage before terminating	**Done:** Program should terminate
Spawn Children	This state spawns child processes to handle client connections	**Complete:** Once the children have completed spawning move on to the next state
Audit Children	Store information gathered by/about children.	**New Connection:** A child reports a new connection. If there aren't enough idle worker threads spawn more. **SIGINT:** OS signaled that the program should terminate
Generate Report	This state generates a report about all the connections made to the server while it was running and outputs them to a file for later reference.	**Complete:** The report has finished generating and has saved.
Finish	Program exits	

Pseudocode: Parent Observer

```
1. Parse/Validate User input
2. If user input is invalid display proper usage to user and exit
3. Spawn children to handle client data
4. Audit and save data from children
5. When the program closes, save data to a file.
```

Finite State Machine: Child



State	Description	Events
Setup	Prepare worker to start reading.	**Complete:** Completed initial setup
Accept	Wait for a client to connect to the server.	**New Connection:** A client connected to the server
Process Data	Inform parent that there is a connection and begin to process data sent by the client to the server.	**Connection Closed:** The client has closed the connection signifying that it has finished sending data to the server.
Report	Before waiting for another connection, send a report to the parent thread. This will also show that there is an available worker	**Finished:** Finished sending data to parent.

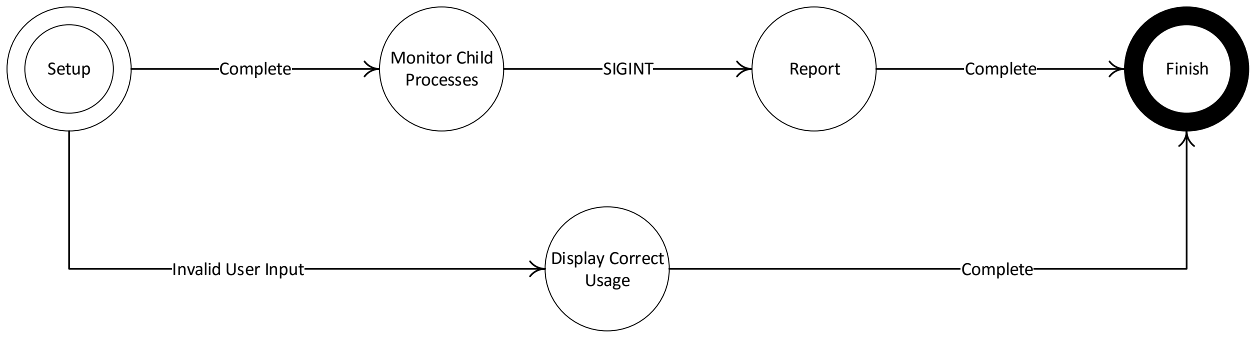
Pseudocode: Child

```
1. Listen on servers listening port for a new connection.
2. If a new connection is received, inform the parent that this process is now working
3. Read for data on the connection
4. If the connection is closed, close the socket and send statistics on the connection to the parent
   process. Return to step 1
5. When data is received echo the data in response
6. Save the amount of data that was sent, and increment a counter for how much data this
   connection is sending
7. Return to step 4
```

EPoll/Select Server

The EPoll and select servers follow a similar path of execution. The general design is the same for both. The primary purpose of them is to multiplex incoming connections.

Finite State Machine: Parent

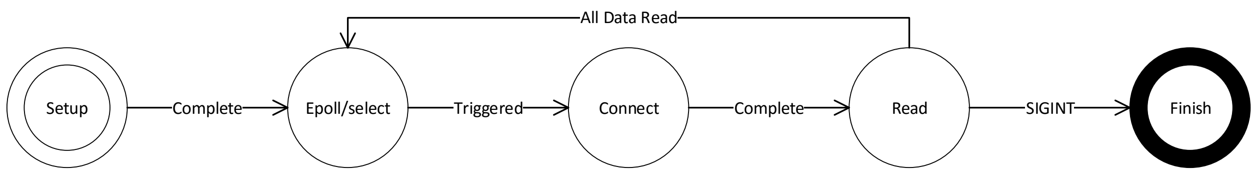


State	Description	Events
Setup	Setup parses and validates the user input then creates the necessary IPC and worker threads for the program to execute.	**Complete:** The setup has completed successfully **Invalid User Input:** If the users input was invalid and the server could not parse it.
Display Correct Usage	Display the correct way to evoke the program.	**Complete:** Correctly displayed user input
Monitor child Processes	Monitor and store data gathered from the child processes	**SIGINT:** Process received signal from the server to terminate.
Report	Generate a document from the data gathered from the child processes	**Complete:** Report is finished Generating
Complete	Process has successfully terminated.	

Pseudo-code: Parent

```
1. Parse user input, if user input is invalid display proper input and exit the program
2. Bind listening socket
3. Spawn child routines
4. Monitor child routines for connection details such as number or requests, amount of data and
   hostname
5. If SIGINT is received, generate a report based on connection info collected from clients
6. Exit
```

Finite State Machine: Child



State	Description	Events
Setup	Prepare necessary structures for multiplexed connections. Add the listening descriptor from the parent routine to the set of watch file descriptors	**Complete:** Routine is ready to accept connections

Epoll/Select	The main state for epoll and select, routine will block here until data is received on one of the receiving descriptors.	**Triggered:** Data ready in one of the watched file descriptors
Connect	If the listen file descriptor has any new connections, connect to them and add them to the block of watched descriptors.	**Complete:** Finished reading from parent
Read	Read from other file descriptors with data being watched, write the data back to the client. If the client has closed the connection, close the connection.	**SIGINT:** If the program terminates exit the loop **All Data Read:** All the events have been processed
Finish	Server has completed its operations exit	

Pseudo Code: Child

```
1. Initialize required structures for epoll or select. Add the listening file descriptor to the set
2. Wait on epoll or select until a descriptor is ready to be read
3. if the listening file descriptor has data, connect to the new client that is trying to connect.
4. Read data from other file descriptors and echo the data back to the clients if the clients have
   closed the connection close the connection on the server end and make sure it has been removed
   from the listening set.
5. Return to step 2
```