COMP 8005

Assignment 2 Design Document

Scalable Epoll-based Client Server

By: Derek Wong

Instructor: Aman Abdulla

Due: 12 PM on March 1st, 2021

Table of Contents

Overview

To compare the scalability and performance of an epoll-based client-server implementation.

Requirements

Design and implement two separate applications:

1. A client application using an architecture that utilizes any API library of your choice.
2. An epoll (edge-triggered) asynchronous server.

The server has the following design and constraints:

1. Server will be written in C.
2. Designed to handle multiple concurrent client connections and transfer a specified amount of data back to a connected client.
3. Implemented as an extended echo server using edge triggered readiness notification, for data exchanges between the server and client.
4. OpenMP will be used to parallelize the program instructions.
5. Server will record each client connection (host names) with total bytes generated and the time the server received the data transfer.

The client has the following design and constraints:

1. Client will be written in C.
2. Designed and implemented to be no more complex than a simple echo client that will connect to a server to send variable length text strings to the server and number of times to send the strings.
3. User will be able to vary the number of concurrent client connections along with how much data each will load on to the server.
4. Client will record statistics during its connection with the server, specifically, the number of requests generated, amount of total data transferred, total elapsed time for data transfer and average transfer per request.

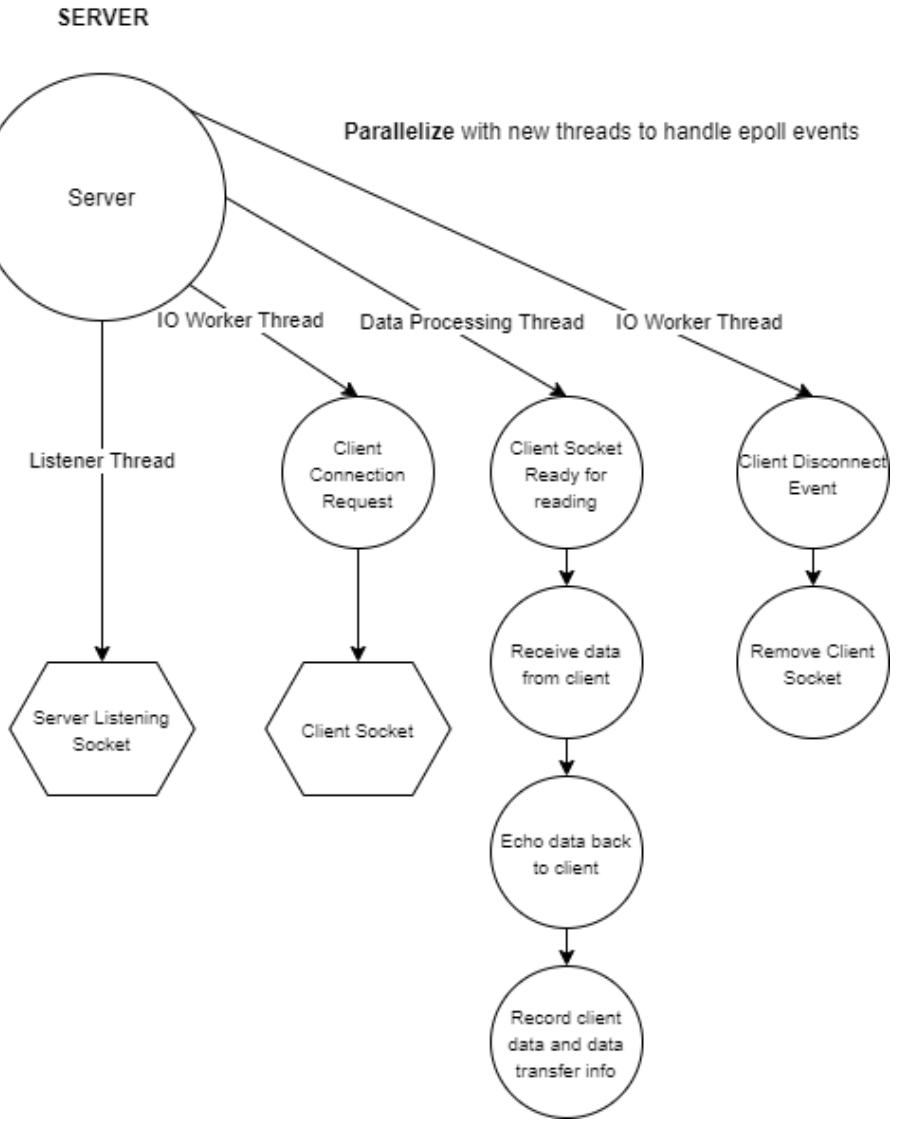
Overall, we want to design a server application that can handle as many connections (scalability) as possible and handle data transfer requests as fast as possible (performance) back to the clients.

Program Design

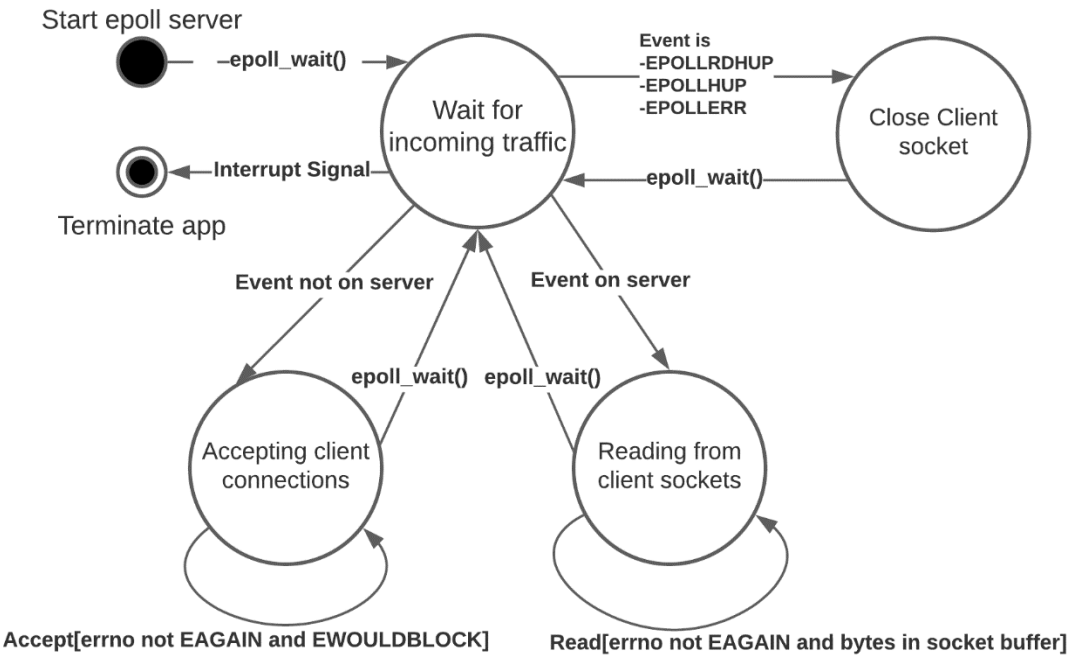
Epoll Server Architecture

The following illustrates the architecture for the networking portion of the application. This architecture allows for almost unlimited scalability of an application on single and multi-processor systems.

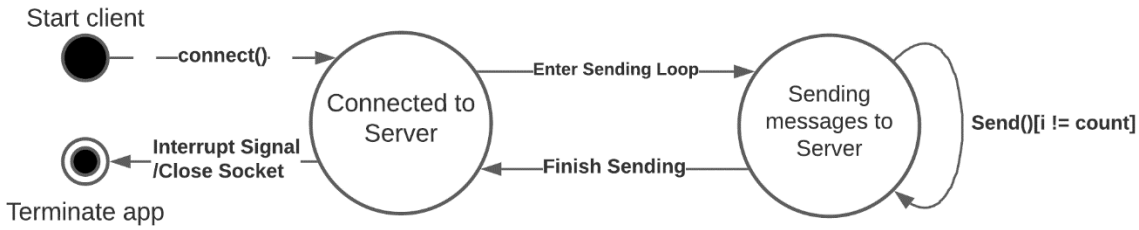
1. **Listener** – Thread that performs bind() and listen() calls and waits for incoming connections. The new connection arrives, the thread executes accept on listening sockets and sends accepted connection socket to one of the I/O workers.
2. **I/O Workers** – One or more threads to receive connections from **listener** and add them to epoll.
3. **Data Processing Workers** – One or more threads to receive data from and send data to **I/O workers** to perform data processing



Epoll Server State Diagram



Client State Diagram



Pseudo Code

Epoll Server

```
Initialize Socket {
        Create stream socket for listening
        Set listening socket to address reusable
        Set listening socket to non-blocking
        Bind address to listening socket
        Listen on listening socket
}

Setup Epoll {
        Create epoll file descriptor with epoll queue length
        Set events to EPOLLIN | EPOLLERR | EPOLLHUP | EPOLLRDHUP | EPOLLET
        Set event data file descriptor to listening socket
}

Event loop {
   While(true)
      epoll_wait() call, block and wait for activity, store events to events on unblock;
      OMP Run in parallel over loop
      Loop through events in events up to number of fd return from epoll_wait
         if new event is EPOLLRDHUP
            Close the client socket;
            continue;

         if new event is EPOLLERROR  or EPOLLHUP
            Print error to stderr and close the socket
            continue;

         if new event's fd == fd_server
            Create flag for indicating EAGAIN error(EAGAIN_REACHED), and
            initialize it to false;
            Run in parallel
               while (!EAGAIN_REACHED)
                  fd_new = accept() the new connection;
                  if fd_new == -1, check errno
                     if errno is either EAGAIN or EWOULDBLOCK
                        flip the EAGAIN_REACHED flag to true;
                        break;
                     } else {
                        Error occurred, print the error message to stderr
                        break;
                  Set the socket fd_new to non blocking;
                  Add the new socket descriptor to the epoll loop;
            continue;
  ClearSocket(new events fd)
}
```

```
Clearsocket(fd) {
   Create flag (done) for indicating read complete
      while(TRUE)
         count = read()
         If count == -1
            If errno != EAGAIN
               done = true
            break
         else if count == 0
            done = true
            break

         Send reply back to client on fd

      if done
         close socket fd
}

main() {
   Initialize socket
   Setup epoll
   Event loop
}
```

Client

```
Initialize socket {
   Create a stream socket for sending(fd_server);
   Connect to server, block wait for server reply to complete three way
   handshake
}

main() {
   Initialize socket
   for number of times to send
      send() message to server
      recv(), block and wait for server reply
      Record time wait before server reply
      Delay some time before next iteration

   Calculate number of byte transfer, response time and average response time
   Print the result to stdin
   Close socket
}
```