

CSE344 – System Programming - Final Project Report

Ahmed Semih Özmekik, 171044039
June 26, 2020

Abstract

Demonstration of solutions for problems encountered in the final project.

1 Problem

In the context of Processes, Sockets, Threads, IPC, Synchronization, we have a lot of problems to solve in this project.

But we will face more serious problems especially in the context of socket and sync. Let us explain step by step how we satisfy these demands by defining the requests expected from us in the project report in each step and showing them on the code.

2 Becoming a Daemon

The entire server process needs to run as a daemon. For this, we have to perform the routine of being a few daemons that we have to go through. In addition, this daemon should not have two instantaneous. In other words, there cannot be two processes from the server program. First, let's take a precaution to prevent two instances. There may be more than one and different solutions to this prevention. My solution is to create a named-semaphore. Let's explain briefly with the code below:

```
1 void create_sem()
2 {
3     if ((single_instance_sem = sem_open(SEM_SINGLE_INSTANCE_NAME, O_CREAT | O_EXCL, 0666, 0)) ==
4         SEM_FAILED)
5     {
6         if (errno == EEXIST)
7         {
8             printf("can't create a second instance of the server daemon!\n");
9             printf("%s\n", SEM_SINGLE_INSTANCE_NAME);
10            exit(EXIT_FAILURE);
11        }
12        xerror(__func__, "creating semaphore");
13    }
```

With this function on the first line in Main, the program is trying to open this semaphore without going any step further. If it has already been opened. With this, we understand that this program is already running, and in this way we prevent 2 instances. And as expected, when the program receives **SIGINT**, that is, terminates properly, it will delete this semaphore. So that it can work next time.

we have done the other routines required to become a daemon, namely making sure the server process has no controlling terminal and closing all of its inherited open files, as follows.

```
1 void become_daemon()
2 {
3     // @see main to prevent prevent double instantiation.
4
5     // get rid of controlling terminals.
6     switch (fork())
7     {
8     case -1:
9         xerror(__func__, "first fork");
10    case 0:
11        break;
12    default:
13        exit(EXIT_SUCCESS);
```

```

14 }
15
16 if (setuid() == -1)
17     xerror(__func__, "setuid");
18
19 signal(SIGCHLD, SIG_IGN);
20 signal(SIGHUP, SIG_IGN);
21
22 switch (fork())
23 {
24 case -1:
25     xerror(__func__, "second fork");
26 case 0:
27     break;
28 default:
29     exit(EXIT_SUCCESS);
30 }
31
32 // flushing file options.
33 umask(0);
34
35 // close all inherited open files.
36 int x;
37 for (x = sysconf(_SC_OPEN_MAX); x >= 0; x--)
38     if (x != args.infd && x != args.outfd)
39         close(x);
40 }

```

There was an important detail, which was: **If it fails to access it for any reason then make sure the user is notified through the terminal before the server detaches from the associated terminal.** So just before the daemon, we parsed the arguments given to the program and made sure they were exactly as they should be. In other words, we have provided the necessary information to the terminal if the arguments are wrong, before daemon has been detached from the terminal.

So far this is our main function:

```

1 int main(int argc, char *argv[])
2 {
3     create_sem(); /* prevent double instantiation */
4     parse_args(argc, argv, &args);
5     become_daemon();
6     ...
7
8     return 0;
9 }

```

After that, it is the part of putting our data into the model by reading the file.

3 Synchronization

Consider the threads on the synchronization side and their roles, let's get to know our problem well. There are three different thread types: Main thread (connection listener thread), connection handler threads, and resizer thread.

First, thread listening on an address (127.0.0.1 for our case, local host) and on a specified port. This thread expects only clients with the **accept()** function in an infinite loop. And as soon as a client connects, this link needs to redirect the handler threads. In the second thread, it takes the file descriptor, and read the indices sent and send the path (or database) to the client. The task of the third type is to increase the number of threads of the second type, if necessary, according to the number of threads running in certain situations. There will be only one thread of the first and third types, and the second type will change between the min and max values.

In other words, one thread from the second type thread needs to be awakened only when a client arrives. And the third type of thread needs to be awakened only when threads need to increase. So a few simple semaphore will

solve all our sync problem. First of all, let's look at the main thread, the connection listener thread function.

```

1 void connection_listener()
2 {
3     int sockfd, clientfd;
4     struct sockaddr_in host_addr, client_addr;
5
6     if ((sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_IP)) == -1)
7         xerror(__func__, "socket");
8
9     int optval = 1;
10    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(int)) == -1)
11        xerror(__func__, "setsockopt");
12
13    host_addr.sin_family = AF_INET;
14    host_addr.sin_port = htons(args.port);
15    host_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
16    memset(&(host_addr.sin_zero), '\0', 8); // zero the rest.
17
18    if (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr)) == -1)
19        xerror(__func__, "bind");
20
21    if (listen(sockfd, 20) == -1)
22        xerror(__func__, "listen");
23
24    while (TRUE)
25    {
26        socklen_t len = sizeof(struct sockaddr_in);
27        if ((clientfd = accept(sockfd, (struct sockaddr *)&client_addr, &len)) == -1)
28            xerror(__func__, "accept");
29
30        xsem_wait(conr->graph_mutex);
31        enqueue(&conr->client_queue, clientfd);
32        xsem_post(conr->graph_mutex);
33
34        /* forward connection */
35        xsem_post(conr->handler_sem);
36    }
37    exit(EXIT_SUCCESS);
38 }

```

As you can see, most of it is setting up sockets. After completing these procedures, it listens in an infinite loop and, when a client arrives, throws the file descriptor into the client queue and immediately posts the semaphore that the second type threads expect.

Let's look at the function of the second type of threads.

```

1 void *connection_handler(void *p)
2 {
3
4
5     int packet_len = sizeof(struct Packet);
6     char *recv_packet = xmalloc(packet_len);
7
8     int *nth = (int *)p;
9
10    while (TRUE)
11    {
12        dprintf(args.outfd, "Thread #%d: waiting for connection \n", *nth);
13        xsem_wait(conr->handler_sem);
14
15        if (is_finished())
16        {
17            xsem_post(conr->handler_sem);
18            break;
19        }
20
21        xsem_wait(dynr->load_mutex);

```

```

22     dynr->handler_count++;
23     float load = dynr->handler_count / (float)(dynr->n - 1);
24     xsem_post(dynr->load_mutex);
25
26     if (need_resize(load))
27     {
28         xsem_wait(dynr->load_mutex);
29         if (!dynr->resize)
30         {
31             dynr->resize = TRUE;
32
33             xsem_post(dynr->pooler_sem); // wake up pooler.
34         }
35         xsem_post(dynr->load_mutex);
36     }
37
38     dprintf(args.outfd, "A connection has been delegated to thread id %d, system load %.1f
39     %%\n",
40         *nth, 100 * get_load());
41
42     /* main job */
43     ...
44     /* main job */
45     xsem_wait(dynr->load_mutex);
46     dynr->handler_count--;
47     xsem_post(dynr->load_mutex);
48 }
49 free(recv_packet);
50 free(nth);
51 pthread_exit(NULL);
52 return NULL;
53 }

```

I have not put the main task of these threads in the code for now. Before coming to that part, here, the semaphore posted in the main thread is waited first. If posted, it means that one more of the connection handler threads is becoming busy. Therefore, before the path is found, if the load reaches 75%, the signal that the resizer thread is waiting is given, that is, the semaphore is posted. Then, the resizer thread increases the number of threads. Since the sync part here is simple, I didn't need to share it in my report.

After solving this sync problem, we can say that we came to the most serious sync problem in the project. This is the reader / write problem. Threads of the second type represent both readers and writers.

```

1 void *connection_handler(void *p)
2 {
3
4     while (TRUE)
5     {
6         /* main job */
7
8         xsem_wait(conr->graph_mutex);
9         int clientfd = dequeue(conr->client_queue);
10        xsem_post(conr->graph_mutex);
11
12        // get the indexes.
13        xread(clientfd, recv_packet, packet_len);
14        struct Packet *indices = (struct Packet *)recv_packet;
15
16        dprintf(args.outfd, "Thread %d: searching database for a path from node %d to node %d\n",
17            *nth, indices->i1, indices->i2);
18        long in_cache = read_database(indices->i1, indices->i2);
19        char *path;
20        if (in_cache)
21        {
22            path = (char *)in_cache;
23            dprintf(args.outfd, "Thread %d: path found in database: %s\n", *nth, path);

```

```

24     }
25     else
26     {
27         // find the path.
28         dprintf(args.outfd, "Thread #%d: no path in database, calculating %d->%d\n",
29             *nth, indices->i1, indices->i2);
30         xsem_wait(conr->graph_mutex);
31         struct Queue *bfs = BFS(conr->graph, indices->i1, indices->i2);
32         xsem_post(conr->graph_mutex);
33
34         path = prepare_packet(bfs);
35
36         if (bfs == NULL)
37             dprintf(args.outfd, "Thread #%d: %s from node %d to %d.\n", *nth, path, indices->
38 i1, indices->i2);
39         else
40         {
41             destroy_queue(bfs);
42             dprintf(args.outfd, "Thread #%d: path calculated: %s\n", *nth, path);
43         }
44
45         write_database(path, indices->i1, indices->i2);
46         dprintf(args.outfd, "Thread #%d: responding to client and adding path to database\n",
47 *nth);
48     }
49
50     int len = strlen(path);
51     xwrite(clientfd, path, len);
52     close(clientfd);
53     /* main job */
54 }

```

The reader / writer problem was also solved according to the classical solution in the reader / writer paradigm. For solution details, you can examine the **read_database()** and **write_database()** functions in detail.

4 Data Structures and Design

We have a big dataset. And from the very beginning, we need to determine the data structures and data models we will use according to a problem that appears in the context of the server-client problem. Our problem is this: We will find a path in graphs with the bfs algorithm. And since this solution is suboptimal, we need to choose a good graph implementation and ease the job in the bfs. Thus, the small threads of the server will finish the calculation immediately and give the paths to the clients as fast as possible.

I left this problem to an end, because when my system is fully ready, I can run tests across multiple graph implementations, and I can ultimately choose the most successful.

We need 3 different data structures. The first is a graph implementation that we needed for to model the graph, one to model the cache structure, and a queue, since we need in many places throughout the project (bfs algorithm, client file descriptors, keep path).

4.1 Cache

For Cache, I chose a data structure that is similar to graph's adjacency list implementation but is not exactly equivalent in the literature. There is a linked-list structure for each vertex in the array. For an edge, paths starting from that edge keep it as a linked list.

The paths were also kept as strings, ready to be sent directly to clients. Let's say that for a node, "n" paths were found, where that node is the starting node. It is $O(n)$ in the retrieval time worst case from the database.

```

1 #ifndef CACHE_H
2 #define CACHE_H
3 #include "utils.h"
4

```

```

5 struct CacheNode
6 {
7     int vertex;
8     char *path;
9     struct CacheNode *next;
10 };
11
12 struct Cache
13 {
14     int V;
15     struct CacheNode **list;
16 };
17
18 struct CacheNode *create_cache_node(int v, char *path)
19 {
20     struct CacheNode *node = (struct CacheNode *)xmalloc(sizeof(struct CacheNode));
21     node->vertex = v;
22     node->next = NULL;
23     node->path = path;
24     return node;
25 }
26
27 struct Cache *create_cache(int V)
28 {
29     struct Cache *cache = (struct Cache *)xmalloc(sizeof(struct Cache));
30     cache->V = V;
31     cache->list = (struct CacheNode **)xmalloc(V * sizeof(struct CacheNode *));
32
33     for (int i = 0; i < V; i++)
34         cache->list[i] = NULL;
35
36     return cache;
37 }
38
39 void to_cache(struct Cache *cache, int i, int j, char *path)
40 {
41     struct CacheNode *node = create_cache_node(j, path);
42
43     /* connect new node to rest of the neighbours */
44     node->next = cache->list[i];
45     cache->list[i] = node;
46 }
47
48 long get_cache(struct Cache *cache, int i, int j)
49 {
50     struct CacheNode *node = cache->list[i];
51     while (node != NULL)
52     {
53         if (node->vertex == j)
54             return (long)node->path;
55         node = node->next;
56     }
57
58     return FALSE;
59 }
60
61 void delete_cache_node(struct CacheNode *node)
62 {
63     while (node != NULL)
64     {
65         delete_cache_node(node->next);
66         free(node->path);
67         free(node);
68     }
69 }
70
71 void destroy_cache(struct Cache *cache)
72 {

```

```

73     for (int i = 0; i < cache->V; i++)
74         delete_cache_node(cache->list[i]);
75     free(cache->list);
76 }
77
78 #endif

```

4.2 Graph

I wrote two implementations for Graph: one with the adjacency list and the other with the adjacency matrix. Before I discuss the typical pros and cons of these, the one I guessed was the most expensive job for BFS was the loop in which neighbors are iterated for each node. At any given time in the code, we are not concerned with which node is connected with which node. So we will not use the constant complexity advantage of adjacency matrix here. Since we need Graph for a single bfs algorithm, we need the advantage of adjacency list implementation. Thus, it would make sense to prefer $O(E)$ instead of $O(V)$ while visiting the neighbors of each node in bfs. So in theory, the adjacency list seems logical. My idea of Graph's creation phase was that the two would be almost the same.

You will see two different implementations below. Since the interface is the same, nothing will change in the system.

```

1
2 #ifndef GRAPH_H
3 #define GRAPH_H
4 #include "utils.h"
5 #include "queue.h"
6
7 /**
8  * graph.h
9  * represent the graph implementation with adjacency matrix.
10  * @see server.c
11  */
12
13 struct Graph
14 {
15     int V;
16     long **mat; // adjacency matrix: [V][V] for nodes.
17 };
18
19 struct Graph *create_graph(int V)
20 {
21     struct Graph *graph = (struct Graph *)xmalloc(sizeof(struct Graph));
22
23     graph->V = V;
24     graph->mat = (long **)xmalloc(sizeof(long *) * V);
25     for (int i = 0; i < V; i++)
26     {
27         graph->mat[i] = (long *)xmalloc(sizeof(long) * V);
28         for (int j = 0; j < V; j++)
29             graph->mat[i][j] = FALSE;
30     }
31     return graph;
32 }
33
34 // adds an edge from node i to node j.
35 void add_edge(struct Graph *graph, int i, int j)
36 {
37     assert(!graph->mat[i][j]);
38     graph->mat[i][j] = TRUE;
39 }
40
41 // removes the edge from node i to node j.
42 void remove_edge(struct Graph *graph, int i, int j)
43 {
44     assert(graph->mat[i][j]);
45     graph->mat[i][j] = FALSE;

```

```

46 }
47
48 // checks if there is an edge from node i to node j.
49 long edge(struct Graph *graph, int i, int j)
50 {
51     return graph->mat[i][j];
52 }
53
54 void destroy_graph(struct Graph *graph)
55 {
56     for (int i = 0; i < graph->V; i++)
57         free(graph->mat[i]);
58     free(graph->mat);
59 }
60
61 struct Queue *BFS(struct Graph *graph, int start, int end)
62 {
63     if (start == end) // source and dest given as the same.
64     {
65         struct Queue *result = create_queue(graph->V);
66         enqueue(&result, start);
67         return result;
68     }
69     if (edge(graph, start, end)) // there is an edge from source to dest.
70     {
71         struct Queue *result = create_queue(graph->V);
72         enqueue(&result, start);
73         enqueue(&result, end);
74         return result;
75     }
76
77     /* finding path with bfs algorithm */
78     struct Queue *paths = create_queue(1024);
79     struct Queue *path = create_queue(16);
80     int *visited = (int *)xmalloc(sizeof(int) * graph->V);
81     for (int i = 0; i < graph->V; i++)
82         visited[i] = FALSE;
83     enqueue(&path, start);
84     enqueue(&paths, (long)path);
85
86     int found = FALSE;
87     while (!is_empty(paths))
88     {
89         path = (struct Queue *)dequeue(paths);
90         int node = back(path);
91         if (node == end)
92         {
93             found = TRUE;
94             break;
95         }
96         visited[node] = TRUE;
97         for (int i = 0; i < graph->V; i++)
98         {
99             if (edge(graph, node, i) && !visited[i])
100             {
101                 struct Queue *new_path = create_queue(16);
102                 copy(path, new_path);
103                 enqueue(&new_path, i);
104                 enqueue(&paths, (long)new_path);
105             }
106         }
107
108         destroy_queue(path);
109         free(path);
110     }
111
112     // clean up resources.
113     while (!is_empty(paths))

```



```

114 {
115     struct Queue *p = (struct Queue *)dequeue(paths);
116     destroy_queue(p);
117     free(p);
118 }
119 destroy_queue(paths);
120 free(paths);
121
122 return found ? path : NULL;
123 }
124
125 #endif

```

```

1
2 #ifndef GRAPH_H
3 #define GRAPH_H
4 #include "utils.h"
5 #include "queue.h"
6
7 /**
8  * graph.h
9  * represent the graph implementation with adjacency list.
10  * @see server.c
11  */
12
13 struct AdjacencyNode
14 {
15     int vertex;
16     struct AdjacencyNode *next;
17 };
18
19 struct Graph
20 {
21     int V;
22     struct AdjacencyNode **list;
23     int *visited;
24 };
25
26 struct AdjacencyNode *create_graph_node(int v)
27 {
28     struct AdjacencyNode *node = (struct AdjacencyNode *)xmalloc(sizeof(struct AdjacencyNode));
29     node->vertex = v;
30     node->next = NULL;
31     return node;
32 }
33
34 struct Graph *create_graph(int V)
35 {
36     struct Graph *graph = (struct Graph *)xmalloc(sizeof(struct Graph));
37     graph->V = V;
38     graph->list = (struct AdjacencyNode **)xmalloc(V * sizeof(struct AdjacencyNode *));
39     graph->visited = (int *)xmalloc(sizeof(int) * V);
40
41     for (int i = 0; i < V; i++)
42     {
43         graph->list[i] = NULL;
44         graph->visited[i] = 0;
45     }
46
47     return graph;
48 }
49
50 void add_edge(struct Graph *graph, int i, int j)
51 {
52     struct AdjacencyNode *node = create_graph_node(j);
53
54     /* connect new node to rest of the neighbours */
55     node->next = graph->list[i];

```

```

56     graph->list[i] = node;
57 }
58
59 void delete_graph_node(struct AdjacencyNode *node)
60 {
61     while (node != NULL)
62     {
63         delete_graph_node(node->next);
64         free(node);
65     }
66 }
67
68 void destroy_graph(struct Graph *graph)
69 {
70     for (int i = 0; i < graph->V; i++)
71         delete_graph_node(graph->list[i]);
72     free(graph->visited);
73     free(graph->list);
74 }
75
76 int edge(struct Graph *graph, int i, int j)
77 {
78     struct AdjacencyNode *node = graph->list[i];
79     while (node != NULL)
80     {
81         if (node->vertex == j)
82             return TRUE;
83         node = node->next;
84     }
85     return FALSE;
86 }
87
88 struct Queue *BFS(struct Graph *graph, int start, int end)
89 {
90     if (start == end) // source and dest given as the same.
91     {
92         struct Queue *result = create_queue(graph->V);
93         enqueue(&result, start);
94         return result;
95     }
96     if (edge(graph, start, end)) // there is an edge from source to dest.
97     {
98         struct Queue *result = create_queue(graph->V);
99         enqueue(&result, start);
100         enqueue(&result, end);
101         return result;
102     }
103
104     /* finding path with bfs algorithm */
105     struct Queue *paths = create_queue(1024);
106     struct Queue *path = create_queue(16);
107     int *visited = xmalloc(sizeof(int) * graph->V);
108     for (int i = 0; i < graph->V; i++)
109         visited[i] = FALSE;
110
111     enqueue(&path, start);
112     enqueue(&paths, (long)path);
113
114     int found = FALSE;
115     while (!is_empty(paths))
116     {
117         path = (struct Queue *)dequeue(paths);
118         int node = back(path);
119         if (node == end)
120         {
121             found = TRUE;
122             break;
123         }

```

```

124
125     struct AdjacencyNode *adj = graph->list[node];
126     visited[node] = TRUE;
127     while (adj != NULL)
128     {
129         if (visited[adj->vertex])
130             adj = adj->next;
131         else
132         {
133             struct Queue *new_path = create_queue(16);
134             copy(path, new_path);
135             enqueue(&new_path, adj->vertex);
136             enqueue(&paths, (long)new_path);
137             adj = adj->next;
138         }
139     }
140
141     destroy_queue(path);
142     free(path);
143 }
144
145 // clean up resources.
146 while (!is_empty(paths))
147 {
148     struct Queue *p = (struct Queue *)dequeue(paths);
149     destroy_queue(p);
150     free(p);
151 }
152 destroy_queue(paths);
153 free(paths);
154
155 return found ? path : NULL;
156 }
157
158 #endif

```

Let us now examine how many different path searches take in these different implementations.

4.2.1 Adjacency matrix implementation benchmark

Server Side (logfile)

```

    Executing with parameters:
-i nodes.txt
-p 5000
-o log.txt
-s 4
-x 24
Loading graph...
Graph loaded in 0.233916 seconds with 6301 nodes and 20777 edges.
Thread #1: waiting for connection
Thread #2: waiting for connection
Thread #3: waiting for connection
Thread #4: waiting for connection
A pool of 4 threads have been created
A connection has been delegated to thread id #1, system load 25.0%
Thread #1: searching database for a path from node 0 to node 9999
Thread #1: no path in database, calculating 0->9999
Thread #1: path not possible. from node 0 to 9999.
Thread #1: responding to client and adding path to database
Thread #1: waiting for connection
A connection has been delegated to thread id #2, system load 25.0%

```

Thread #2: searching database for a path from node 0 to node 734
 Thread #2: no path in database, calculating 0->734
 Thread #2: path calculated: 0->4->144->146->5171->1338->3179->728->734.
 Thread #2: responding to client and adding path to database
 Thread #2: waiting for connection
 A connection has been delegated to thread id #3, system load 25.0%
 Thread #3: searching database for a path from node 0 to node 1000
 Thread #3: no path in database, calculating 0->1000
 Thread #3: path not possible. from node 0 to 1000.
 Thread #3: responding to client and adding path to database
 Thread #3: waiting for connection
 Termination signal received, waiting for ongoing threads to complete.
 All threads have terminated, server shutting down.

Client Side:

```
drh0use@wife: ~/Downloads/sys/final/source
drh0use@wife:~/Downloads/sys/final/source$ ./client -a 127.0.0.1 -p 5000 -s 0 -d
9999
[Fri Jun 26 19:56:23 2020] Client (16306) connecting to 127.0.0.1:5000
[Fri Jun 26 19:56:23 2020] Client (16306) connected and requesting path from nod
e 0 to 9999
[Fri Jun 26 19:56:24 2020] Server's response to (16306): path not possible, arri
ved in 1.020667 seconds. shutting down.
drh0use@wife:~/Downloads/sys/final/source$ ./client -a 127.0.0.1 -p 5000 -s 0 -d
734
[Fri Jun 26 19:56:29 2020] Client (16307) connecting to 127.0.0.1:5000
[Fri Jun 26 19:56:29 2020] Client (16307) connected and requesting path from nod
e 0 to 734
[Fri Jun 26 19:56:30 2020] Server's response to (16307): 0->4->144->146->5171->1
338->3179->728->734, arrived in 0.688194 seconds. shutting down.
drh0use@wife:~/Downloads/sys/final/source$ ./client -a 127.0.0.1 -p 5000 -s 0 -d
1000
[Fri Jun 26 19:56:34 2020] Client (16308) connecting to 127.0.0.1:5000
[Fri Jun 26 19:56:34 2020] Client (16308) connected and requesting path from nod
e 0 to 1000
[Fri Jun 26 19:56:35 2020] Server's response to (16308): path not possible, arri
ved in 1.032758 seconds. shutting down.
drh0use@wife:~/Downloads/sys/final/source$
```

Figure 1

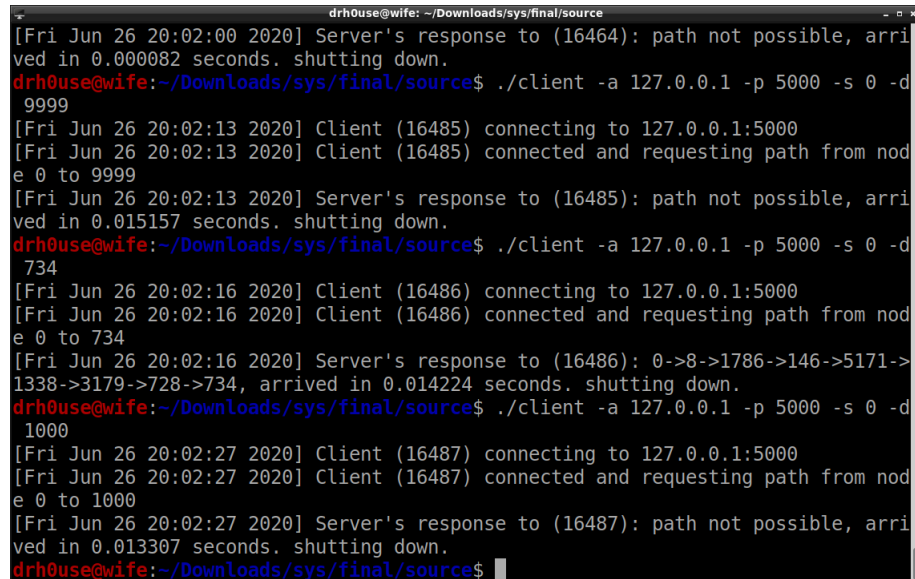
4.2.2 Adjacency list implementation benchmark

Server Side (logfile)

Executing with parameters:
 -i nodes.txt
 -p 5000
 -o log.txt
 -s 4
 -x 24
 Loading graph...
 Graph loaded in 0.005833 seconds with 6301 nodes and 20777 edges.
 Thread #1: waiting for connection
 Thread #2: waiting for connection
 Thread #3: waiting for connection
 A pool of 4 threads have been created
 Thread #4: waiting for connection

A connection has been delegated to thread id #1, system load 25.0%
 Thread #1: searching database for a path from node 0 to node 9999
 Thread #1: no path in database, calculating 0->9999
 Thread #1: path not possible. from node 0 to 9999.
 Thread #1: responding to client and adding path to database
 Thread #1: waiting for connection
 A connection has been delegated to thread id #2, system load 25.0%
 Thread #2: searching database for a path from node 0 to node 734
 Thread #2: no path in database, calculating 0->734
 Thread #2: path calculated: 0->8->1786->146->5171->1338->3179->728->734.
 Thread #2: responding to client and adding path to database
 Thread #2: waiting for connection
 A connection has been delegated to thread id #3, system load 25.0%
 Thread #3: searching database for a path from node 0 to node 1000
 Thread #3: no path in database, calculating 0->1000
 Thread #3: path not possible. from node 0 to 1000.
 Thread #3: responding to client and adding path to database
 Thread #3: waiting for connection
 Termination signal received, waiting for ongoing threads to complete.
 All threads have terminated, server shutting down.

Client Side:



```

drh0use@wife: ~/Downloads/sys/final/source
[Fri Jun 26 20:02:00 2020] Server's response to (16464): path not possible, arrived in 0.000082 seconds. shutting down.
drh0use@wife:~/Downloads/sys/final/source$ ./client -a 127.0.0.1 -p 5000 -s 0 -d 9999
[Fri Jun 26 20:02:13 2020] Client (16485) connecting to 127.0.0.1:5000
[Fri Jun 26 20:02:13 2020] Client (16485) connected and requesting path from node 0 to 9999
[Fri Jun 26 20:02:13 2020] Server's response to (16485): path not possible, arrived in 0.015157 seconds. shutting down.
drh0use@wife:~/Downloads/sys/final/source$ ./client -a 127.0.0.1 -p 5000 -s 0 -d 734
[Fri Jun 26 20:02:16 2020] Client (16486) connecting to 127.0.0.1:5000
[Fri Jun 26 20:02:16 2020] Client (16486) connected and requesting path from node 0 to 734
[Fri Jun 26 20:02:16 2020] Server's response to (16486): 0->8->1786->146->5171->1338->3179->728->734, arrived in 0.014224 seconds. shutting down.
drh0use@wife:~/Downloads/sys/final/source$ ./client -a 127.0.0.1 -p 5000 -s 0 -d 1000
[Fri Jun 26 20:02:27 2020] Client (16487) connecting to 127.0.0.1:5000
[Fri Jun 26 20:02:27 2020] Client (16487) connected and requesting path from node 0 to 1000
[Fri Jun 26 20:02:27 2020] Server's response to (16487): path not possible, arrived in 0.013307 seconds. shutting down.
drh0use@wife:~/Downloads/sys/final/source$
  
```

Figure 2

As you can see, with the adjacency list implementation response time is much shorter. That's why we choose the adjacency list implementation.