***Delta Team***

**Alternative Internet Name Service**

Members: Karthika Krishnamurthy (546810), Nguyen Van Tan (547301), Rohan Krishnakumar (546807), Bijay Basnet (545633)

# 1. Overview and overall architecture

**Overview:**

Alternate Internet Name Service (AINS) is a distributed service that maps names into some other information. The result information could be phone numbers, other contact details, IP addresses or something else [1]. The following section explains Alternate Internet Name Service (AINS) implementation done in this project. AINS is implemented similar to Domain Name System (DNS). DNS is a naming system for resources that are connected to the Internet. It translates domain name of each resource into IP address. Whenever a client types a domain name in a browser, a query is initiated from client to the DNS server which in turn resolves the domain name to IP address. Domain names are used for user friendly purpose. It is necessary to convert domain names to IP address to route the packet in a network. Routers understand only the IP address [2]. In this project, AINS is implemented very similar to DNS. Unlike DNS, it does not maintain a hierarchical naming system instead a single server has all domain names to respond to DNS queries. If the requested domain name is not present, then server responds client with a message "No entries found".

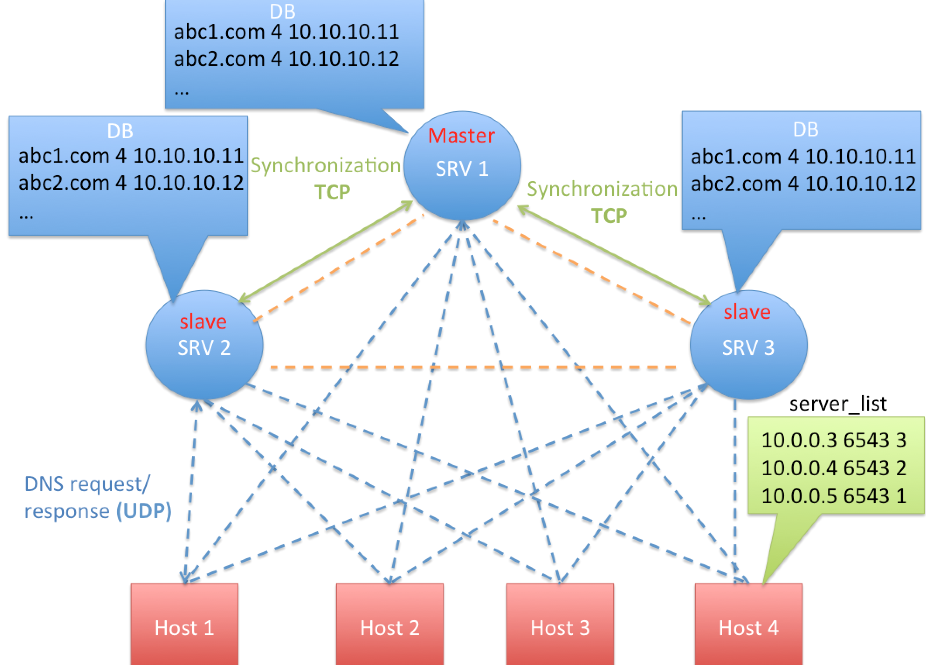
****

Figure 1 - Overall architecture

**Overall Architecture:**

**AINS database:**

AINS database has the mapping of domain name to IP address in a text file. It has domain name, IP version and IP address in a single line. AINS database can be manually updated anytime on the master server.

**Server:**

As part of this project, three AINS servers will used for implementation. Each server will maintain identical AINS database. Each server is capable of responding to client request independently. To each AINS query, servers responds with AINS reply which contains IP address to the requested domain name.

**Client:**

Each client sends query which contains a domain address to be resolved by the DNS server. Any client can query any DNS server. Each client maintains the list of DNS servers. List of DNS servers will be statically updated in each and every client. The result of DNS is used to forward packets. Forwarding packets is out of scope. Timeout happens in the client if it does not get reply from server any of the servers.

**Master/Slave server synchronization:**

This AINS implementation has three DNS servers. Out of the three DNS servers, one server is configured as master server and rest of the servers are configured as slave servers. Database can be manually updated only in the master server. As soon as the master server is updated, it updates rest of all the slave servers. The same design can be extended to any number of servers. Time delay in updating the slave servers is kept minimum.

**Load balancing:**

Capacity of server is defined as the maximum number of queries that server can respond to. If server is loaded with more queries than its capacity, then server will start dropping the queries. To avoid this situation, it is necessary to balance the load between the servers. It is done in following way.

**Communication protocol:**

Client query and server reply will be implemented using UDP protocol and database synchronization will be implemented using TCP protocol. This project supports both IPv4 and IPv6 addressing.

**Server logging:**

Server logging is important to debug server behavior. Each server log messages are stored in a ring buffer. Server copies log messages from ring buffer to log files using a separate thread.

# 2. Requirements description

**Functions to be implemented:**

**Client:**

- Client includes following functionality.

- sending DNS query to server.

- processing received DNS response.

- implementing timeout mechanism.

- preliminary load balancing.

**Server:**

- Server includes following functionality.

- processing DNS query.

- searching database with given query.

- sending response to client.

- allowing manual update of database.

- sending update to slave servers after timer expiry.

- processing database update request from master server.

- allowing to update database through synchronization.

- logging of error and debug messages.

**How functionality is distributed between different components:**

* **Client-Server:** AINS query and response takes place between the server and the client. It is implemented using UDP protocol.
* **Server-server:** Database synchronization takes place between master and slave servers. It is implemented using TCP protocol.
* **Logging:** Server logging in all servers.
* **Load balancing:** Client implements load balancing logic.

**How software is used:**

* Client initiates a query to the AINS server with a domain name.
* Server after receiving request from client, searches its database and sends response back to client.
* Client has a timer. Query packet is considered lost if response is not received before the timer expiry.
* Since there are more than one servers, database in all servers must be identical. The master server sends update to the all slave servers after a regular timer expiry. Database in the slave servers cannot be manually updated.
* Slave servers run even if master server fails but database cannot be updated.

**Programming language:** C

**Operating platform:** Linux

**Library dependencies:** No, except default C and Linux libraries.

**nwprog\* hosts:** Program any run in any Linux hosts.

# 3. Instructions

**How to build**

“**Makefile**” is present inside the “**src**” folder. Separate compilation commands are need for master server, slave server and client.

**Compilation command for client:**

make mode=client

**Compilation command for master server:**

make mode=server\_master

**Compilation command for slave server:**

make mode=server\_slave

**Command to clean all object (.o) files**

make clean

Binary files namely **client**, **server\_master** and **server\_slave** will be generated after successful compilation of client, master server and master slave inside **bin** folder. **bin** folder is present inside **src** folder.

**How to use:**

AINS service can be tested in the following way.

**Step 1:** All files and folders inside a **src** directory are necessary to run the test. Ensure “**server\_list**” file is present inside src directory and “**DB**” file is present inside “**src/server**” directory

**Step 2:** “server\_list” contains list of the server IP address , UDP port numbers and priority for each server. Higher value is considered as higher priority. Master server must run in the IP address with highest priority.

**Step 3**:Execute all compilation commands. Binaries are generated inside “src/bin” folder. Use “make clean” before generating each and every binary file. For example,

> make clean

> make mode=server\_master

> make clean

> make mode=server\_slave

> make clean

> make mode=client

**Note**: **make clean is needed in between different builds.**

**Step 4**:First, server\_master must be executed. Master server must be up and running for the slave servers to get DB update.

**Step 5**:Next, server\_slave binary must be executed. After successful connection, slave\_DB will be created inside **“src/server”** folder. Generally slave DB file name starts with prefix “**slave\_DB\_** “.

**Step 6:**Use “cat” file names to check whether all DB have identical data.

**Step 7:**Client binary can be executed with any “domain name” as command line argument.

**Step 8:**Client must receive reply from server with IP address corresponding to the given domain name from any running server. Client chooses server randomly to load balance between servers. Example for query:home.flash.com

**Step 9:** After testing, kill daemon process using “kill -USR1 <pid>” or “kill -USR2 <pid>. If process is not killed using above commands, then memory will not be freed properly, logs will not be collected properly and slave server DB will not be deleted (though it will not used later).

**Step 10:** Logs can be found inside “**src/bin**” folder and logs will be deleted once program starts execution again.

**Must work in Aalto login servers, or in course test servers:**

Code is tested and shows similar behaviour in all below servers/networks.

1) Aalto login server

2) Course test servers

3) Mininet and

4) Local machines.

**Is any configuration needed:**

- No additional configuration or libraries are needed.

- But other than source code, software testing needs,

- DB file which has domain name to IP address mapping.

- File with available server IP addresses.

# 4. Communication protocol

**Describe the protocol messages and message interaction. Diagrams are likely useful**

AINS service uses two communication protocols

1) UDP: AINS uses UDP protocol to query server and servers send a UDP reply to those queries.

2) TCP: AINS uses TCP protocol to synchronize data bases between AINS servers. All slave servers will receive update from master server.

**UDP: AINS query and response**

Each client will resolve domain name to IP address before accessing the Internet. To resolve domain name, client sends a query to AINS server. Thus all AINS servers must be up and running to respond to any client request.

- Server creates a thread during initialization which receives the UDP queries. Server uses a fixed port say 6543 to listen to the UDP queries. Server runs a loop such that it can receive and process UDP request all time.

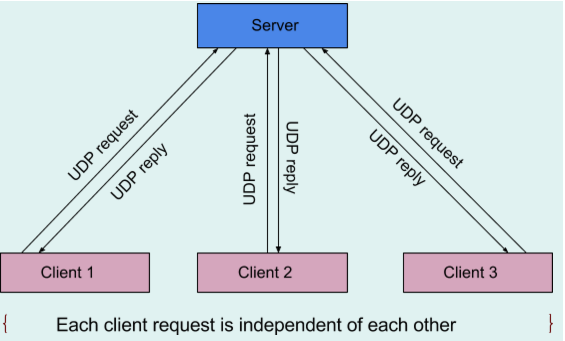


Figure 2 - UDP query and response

- Client sends domain name [eg. home.flash.com] in the message to server's fixed port say 6543. Message to server consists of only domain address in the message.

- Client chooses server from list of available servers using random logic.

- Server as soon as receives a UDP query, searches AINS database and finds the IP address corresponding to domain address and sends response to client.

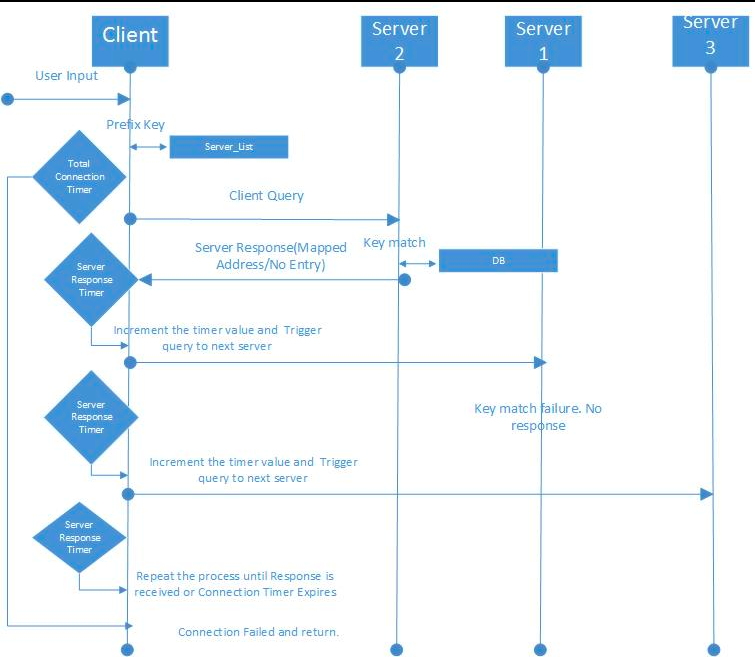


Figure 3 - Client-Server communication

- Since server and client uses UDP, if a UDP request from client or UDP reply from server is lost or if server is down, it is not possible to identify the packet loss. To avoid this problem, two solutions were implemented. Client starts running a timer as soon as it sends a query to server. If reply is not received before timer expiry, then UDP request is considered lost and UDP query is sent once again to different server. DNS reply received after timeout is considered lost.

**TCP: AINS server database synchronization**

Servers need database (DB) files to respond to all AINS queries. Practically it is not possible to manually configure/ update DB present in all AINS servers. To overcome this problem, out of available AINS servers, one server is configured as master server and rest of the servers are configured as slave servers. A server\_list is present in all slave servers. Servers with highest priority is chosen as master and rest of the servers will act as slave servers. It is enough to update DB of master server alone and master sends update to all slave servers.

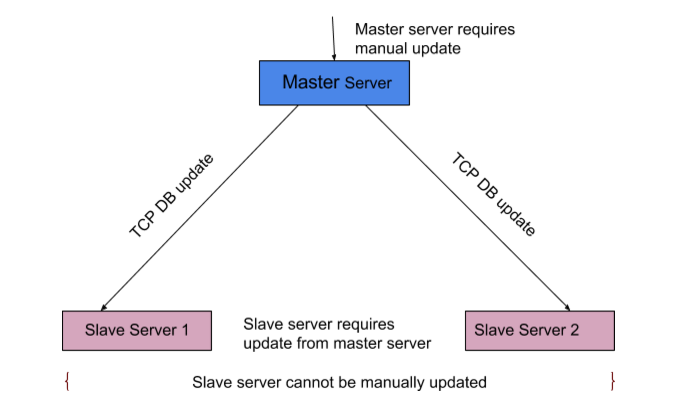


Figure 4 - TCP server DB synchronization

- As master server boots up, it creates a new thread and accepts incoming TCP connection from other servers. Server does not accept connections from servers which does not know the key with which is shared with the master server.

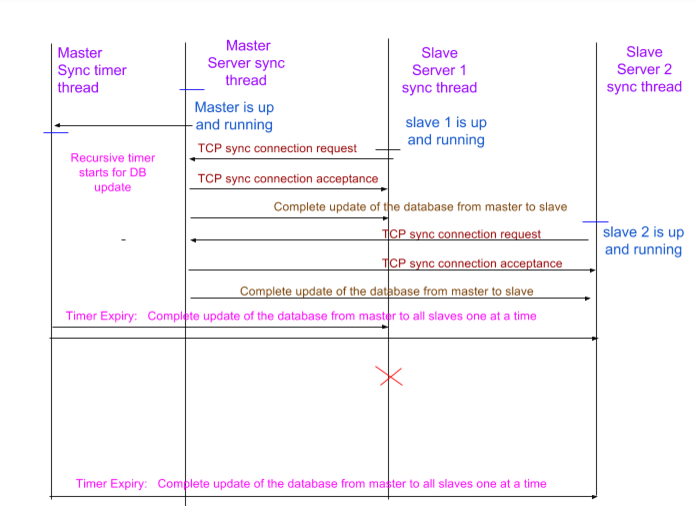


Figure 5 - Server synchronization sequence

- Master server creates another thread which runs the timer to send database update to slave servers.

- Upon timer expiry master servers sends complete database to all connected slave servers.

- Even if master fails, slave servers responds to queries using old database.

- Slave server deletes it database during termination.

- As DB is shared between multiple threads, mutex is used to access DB files.

**Server logging:**

- Server logging is necessary to observer server behaviour.

- Server logging is implemented in the following way.

- During server boot-up a 2MB is reserved for logging.

- A separate thread is created which runs a 6 sec timer.

- All log messages from main thread, master server sync thread, master sync timer thread, UDP responder thread, slave server sync thread are sent to a ring buffer which can hold to a maximum of 100 messages.

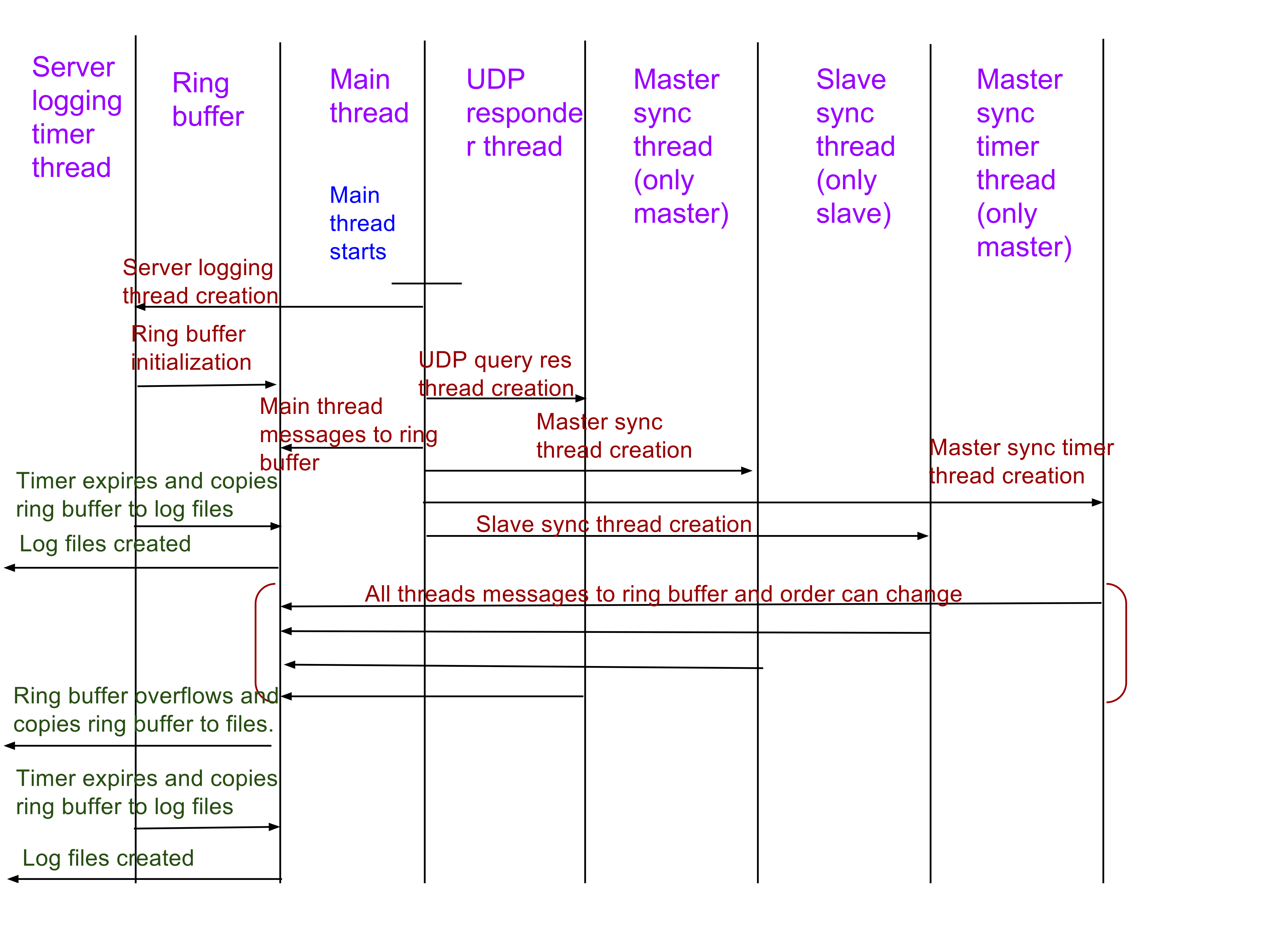
****

Figure 6 - Server logging sequence

- After every 6s, server logging thread copy all data from ring buffer to a log file. If ring buffers overflows within 6s, then all ring buffer data will be automatically copied to log file.

- If log memory overflows, the oldest log files will be deleted first.

- If logging thread creation fails during boot-up, then server initialization will not fail as logging is a secondary functionality.

- Logging thread also use mutex as multiple threads use ring buffer.

- During termination, all logs left over in the ring buffer is copied to log files.

**Signal handling:**

**-** Signal handling is done for various signals like SIGHUP, SIGPIPE, SIGUSR1, SIGUSR2 and SIGINT.

- Rest of the signals are ignored.

- Upon signal capture, global structure memories will be freed, logs will be copied to files and slave DB will be deleted.

**Daemon process:**

- Daemon process is implemented for both master and slave server. Parent process dumps the process id of the child process and terminates.

**What is done in terms of reliability? How to avoid congestion?**

- Synchronization messages are important as they are basis for AINS functionality. Thus it is implemented using TCP for reliability.

- AINS queries requires very less processing time. AINS queries will never cause storm or congestion as only one query packet is sent by each client application.

- Usually DNS query packets have higher L3 priority than normal user traffic. This project involves only application programming. Prioritizing the traffic is out of scope.

- Assuming a link with 100Mbps link and 64B DNS packets, approximately needs 16384 packets per second. This much DNS traffic is not expected in real-time scenario.

- But congestion is tested with other user traffic which has very low priority. Most of the DNS queries are successful.

# 5. Implementation description

**Software modules:**

**UDP Responder:**

UDP query responder listens on UDP port 6543 and create a new thread to accept all incoming datagrams from clients after verifying the key. It queries the DB and sends result (a corresponding IP address or "No entries found" message to clients). UDP has a initialization function **server\_query\_responder\_init()** which creates a sockets and then creates a new thread which waits in a loop to responds to all UDP queries. When server receives UDP query **server\_query\_responder\_handle\_request()** function searches DB

and sends response to client.

**Master server synchronization:**

Software modules for the server master are the initialization/de-initialization modules, DB querier and TCP handler. Initialization **server\_master\_sync\_init()** / de-initialization functions  **server\_master\_sync\_deinit()** are responsible for initializing the global structure used by server master, creating new threads to **(server\_sync\_master\_query\_db())** query DB and handle slave server connections.

DB querier implements a periodic timer in **server\_sync\_master\_handle\_db\_timer()** which queries the DB and sends updates to all the slave servers connected to master.TCP handler listens on a pre-defined socket and accepts incoming slave server connections after verifying the key. It adds the socket descriptor to the global structure and is used by the DB querier to send out updates.

**Slave server synchronization:**

Software modules for the slave server are the initialization module server selection module, DB update module.

Initialization function **server\_slave\_sync\_init()** creates a socket with results from select server function **server\_select\_master()** and then connects to the server. Once it receives update from master server, DB update function **server\_sync\_slave\_handle\_db\_update()** sends all updates to DB file.

**Server logging:**

Server logging has a timer thread and a ring buffer module. During server logging initialization **server\_logging\_init(),** a new timer thread is created.All log messages are sent to ring buffer. Server logging timer thread copies log messages from ring buffer to log file every 6s.

**Daemon process:**

Daemon process module is the first function called in main thread. It creates a child process after which the main process terminates. This daemon process help to detach the server process from the shell and run as background process.

**Signal handler:**

Signal handler function **signal\_handler()** cleans-up all memory and user defined termination actions during during the normal/abnormal termination of the process.

**Client:**

When the client is run, it requests the client to enter the query. The query is prefixed with a key so that servers can identify the client is not a malicious client. However, the user is not notified about the prefix.

As per our design, there will be one master server and two slave servers. The IP address of the servers is written in the server list file. Client makes use of the server\_list file in order to obtain the Ip address and the port number which it wants to connect. The third parameter of the file is not needed for client.

The client picks one of the servers in a random manner using **select\_random\_server()** . The randomization is added to avoid the congestion concentration to a single server when multiple clients are run and multiple queries are made. However, random selection of server is done only once at the beginning. At each retry, the next server is selected. This design is adopted so that if any server is down, the client can query to the next available server from the list and make sure every server is attempted before final failure is declared.

*CONNECTION*\_TIME is the total time until which a client attempts to get response for the user query. This value is set 10s (can be adjust based on performance). During this time, the client attempts multiple queries to all the servers, until the response is received. However, if no response is received until the connection timer expires, the query is deemed as failed to get the response and same is notified to the user at the terminal.

After starting the connection timer, the client reads the server info from the server\_list file. Based on the string lengths of the IP address, it determines which socket descriptor and connection to use i.e. either Ipv4 or Ipv6. **read\_server\_info()** is used for this purpose.

The connection between the clients and a server uses UDP protocol. The client sends the query to the selected server. The time at which the query was sent is noted down. This information is used to calculate the round trip time taken for query-response as one of the performance metrics.

A *g\_EACH\_SERVER\_RESPONSE\_WAIT\_TIME* is started and the client waits for the response from the server. The each server wait timer is initially set to 1s and can be adjusted, if required. If response is received within this timer, the time at which the response was received is noted down. Subtracting the query sent time from response received time, we can get the round trip time. This value is printed in microsecond for better time resolution.

However, if the response is not received within each server wait timer, the socket is closed, *g\_EACH\_SERVER\_RESPONSE\_WAIT\_TIME* is increased by 1s. This design is adopted considering the delays in the server side to respond e.g., because of huge DB file, server overload etc. Also, each server wait timer is not kept larger value as it can impact the performance e.g. if one server is down, the client unnecessarily has to wait until that long before attempting next query. The next server is selected from the list and its information is read from the file same as before. The client makes new connection and sends the query again, however the query sent time is not updated for retries. This way we can actually calculate the performance from original query sent to actual response received. This process is continued until either response is received or the connection timer expires.

After response is received, corresponding mapped address is printed in the terminal. The round trip time is calculated for performance metrics. If no response is received at all, failure message is printed in the terminal. In case of multiple repeated queries, the total no. of failure response is stored to calculate the number of packet loss (or failure ratio) as a performance metrics.

**Important interfaces:**

**-** Server list file is used by clients, slave servers.

- DB file is used master server and slave servers.

- UDP port number is 6543

- TCP port number is 5000

These two files are important interface between all modules.

**Libraries used:**

Standard C libraries are used. For instance, POSIX thread library for creating and handling threads. C socket libraries and timer.

# 6. Testing

**Describe performance indicators/metrics you use to analyze your code**

- To measure minimum, average and maximum delay (RTT) for query and response.

- Rate of success of queries.

**Test:**

**Scenario 1 - From wireless network (e.g. Aalto Open) to a fixed server**

**Preconditions:**

- 4 hosts running on wireless network (Aalto Open)

- 3 servers running on course test servers: nwprog1.netlab.hut.fi, nwprog2.netlab.hut.fi and nwprog3.netlab.hut.fi

- IPv4 addresses

* **Test case 1\_Name service query - Found entries\_1 client**

|  |  |  |
| --- | --- | --- |
| Steps | Expected result | Actual result |
| Start Master and 2 slave servers on test servers.  Run client on 1 host | Program starts successfully | Program started successfully |
| Input query: home.flash.com | Receives IP address 192.168.2.1 | RTT: 9108 us |
| Input query: lab.flash.com | Receives IP address 1.1.1.2 | RTT: 6001 us |
| Input query: office.flash.com | Receives IP address 192.168.2.3 | RTT: 3210 us |
| Input query: comsci.flash.com | Receives IP address 10.1.1.4 | RTT: 901 us |

* **Test case 2\_Name service query - No entries found\_1 client**

|  |  |  |
| --- | --- | --- |
| Steps | Expected result | Actual result |
| Run ./client on 1 host | Program starts successfully | Program started successfully |
| Input query: h | Receives "No entries found" | RTT: 2516 us |
| Input query: home.flash.com.abc | Receives "No entries found" | RTT: 4193 us |
| Input query: office.com | Receives "No entries found" | RTT: 6128 us |
| Input query: qwerty | Receives "No entries found" | RTT: 803 us |

* **Test case 3\_Name service query - Found entries\_4 clients**

|  |  |  |
| --- | --- | --- |
| Steps | Expected result | Actual result |
| Run ./client on 4 hosts | Program starts successfully |  |
| Input query: rohan.flash.com from host 1 | Receives IP address 10.1.1.6 | RTT: 6790 us |
| Input query: karthika.flash.com from host 2 | Receives IP address 10.1.1.8 | RTT: 57511 us |
| Input query: bijay.flash.com from host 3 | Receives IP address 10.1.1.9 | RTT: 2901 us |
| Input query: tan.flash.com from host 4 | Receives IP address 10.1.1.10 | RTT: 8219 us |
| Verify load balancing in client side by reading the output in client | Clients select different servers randomly | Clients select different servers randomly:  host 1 - 195.148.124.163 host 2 - 195.148.124.77 host 3 - 195.148.124.76 host 4 - 195.148.124.77 |

* **Test case 4\_Name service query - No entries found\_4 clients**

|  |  |  |
| --- | --- | --- |
| Steps | Expected result | Actual result |
| Run ./client on 4 hosts | Program starts successfully |  |
| Input query: h from host 1 | Receives "No entries found" | RTT: 1510 us |
| Input query: home.flash.com.abc from host 2 | Receives "No entries found" | RTT: 3926 us |
| Input query: office.com from host 3 | Receives "No entries found" | RTT: 31132 us |
| Input query: qwerty from host 4 | Receives "No entries found" | RTT: 5167 us |
| Verify load balancing in client side by reading the output in client | Clients select different servers randomly | Clients select different servers randomly:  host 1 -195.148.124.77 host 2 -195.148.124.77 host 3 -195.148.124.163 host 4 -195.148.124.76 |

* **Test case 5\_Synchronization database - Add new records**

|  |  |  |
| --- | --- | --- |
| Steps | Expected result | Actual result |
| Stop the server | Updated DB must be sent to slave servers |  |
| Delete DB records |  | DB is updated in slave servers |

* **Test case 6\_Synchronization database - Delete record**

|  |  |  |
| --- | --- | --- |
| Steps | Expected result | Actual result |
| Stop the server |  |  |
| Add new DB records | Updated DB must be sent to slave servers | DB is updated in slave servers. |

* **Test case 7\_** **Synchronization database - Processing queries dependency**

|  |  |  |
| --- | --- | --- |
| Steps | Expected result | Actual result |
| Precondition: makes synchronization longer than the timer on client |  |  |
| Send one query from client | Connection timeout, second query must get proper reply |  |

* **Test case 8\_Load balancing\_One server is down**

|  |  |  |
| --- | --- | --- |
| Steps | Expected result | Actual result |
| Stop one slave server:  kill -USR1 process\_id | Slave running on **nwprog3.netlab.hut.fi** was down | Slave running on **195.148.124.163** was down |
| Send one query from client | Client selects another server when the first server is not available. Client then receives response from next server | Client switched to 195.148.124.77 RTT: 2000 us |
| Stop one slave server:  kill -USR1 process\_id | Slave running on **nwprog3.netlab.hut.fi** was down | Slave running on **195.148.124.163** was down |

* **Test case 9\_Load balancing\_All servers are down**

|  |  |  |
| --- | --- | --- |
| Steps | Expected result | Actual result |
| Stop all servers: kill -USR1 process\_id | All servers was killed | Master and slave servers were down |
| Send one query from client | Client tries to send request to all servers until the timer expires | Client switched among 195.148.124.76, 195.148.124.77, 195.148.124.163 several times until timer expires (30 seconds) |
| Stop all servers: kill -USR1 process\_id | All servers was killed | Master and slave servers were down |

**Scenario 2 - IPv6 compatibility**

**Preconditions:**

- Run on course test servers with IPv6 addresses. Update the server\_list:

2001:708:40:2001::38:3610:1 6543 3

2001:708:40:2001::38:3610:2 6543 2

2001:708:40:1001::38:3610:3 6543 1

* **Test case 1\_IPv6 at hosts and servers**

|  |  |  |
| --- | --- | --- |
| Steps | Expected result | Actual result |
| Update the server\_list file with following IPv6 addresses and port 2001:708:40:2001::38:3610:1 6543 2001:708:40:2001::38:3610:2 6543 2001:708:40:1001::38:3610:3 6543 | server\_list file is updated successfully | server\_list was updated |
| Running server\_master in server 1 and server\_slave in server 2 & 3. Running client in server 2 | Programs run successfully | Programs ran successfully |
| Input query "home.flash.com" | Receives IP address 192.168.2.1 | RTT: 798 |
| Input query "class.flash.com" | Receives IP address 192.168.2.4 | RTT: 240 |
| Input query "comsci.flash.com" | Receives IP address 10.1.1.4 | RTT: 8282 |
| Input query "karthika.flash.com" | Receives IP address 10.1.1.8 | RTT: 919 |
| Input query "lab.flash.com" | Receives IP address 1.1.1.2 | RTT: 871 |

* **Test case 2\_IPv6\_Synchronization database\_Add new records**

|  |  |  |
| --- | --- | --- |
| Steps | Expected result | Actual result |
| Update the server\_list file with following IPv6 addresses and port 2001:708:40:2001::38:3610:1 6543 2001:708:40:2001::38:3610:2 6543 2001:708:40:1001::38:3610:3 6543 | server\_list file is updated successfully | server\_list file is updated successfully |
| Running server\_master in server 1 and server\_slave in server 2 & 3. Running client in server 2 | Programs run successfully | Programs run successfully |
| Modify domain name in DB file in master server at location … | DB file is updated successfully | DB file is updated successfully |
| Verify DB file of 2 slave servers | DB files in slave servers should be updated accordingly to the changes from DB of master server |  |
| Send one query with updated domain name | Receives IP address successully which mapped to the updated domain name from server | Time to complete 1 query |

**Mininet scenarios:**

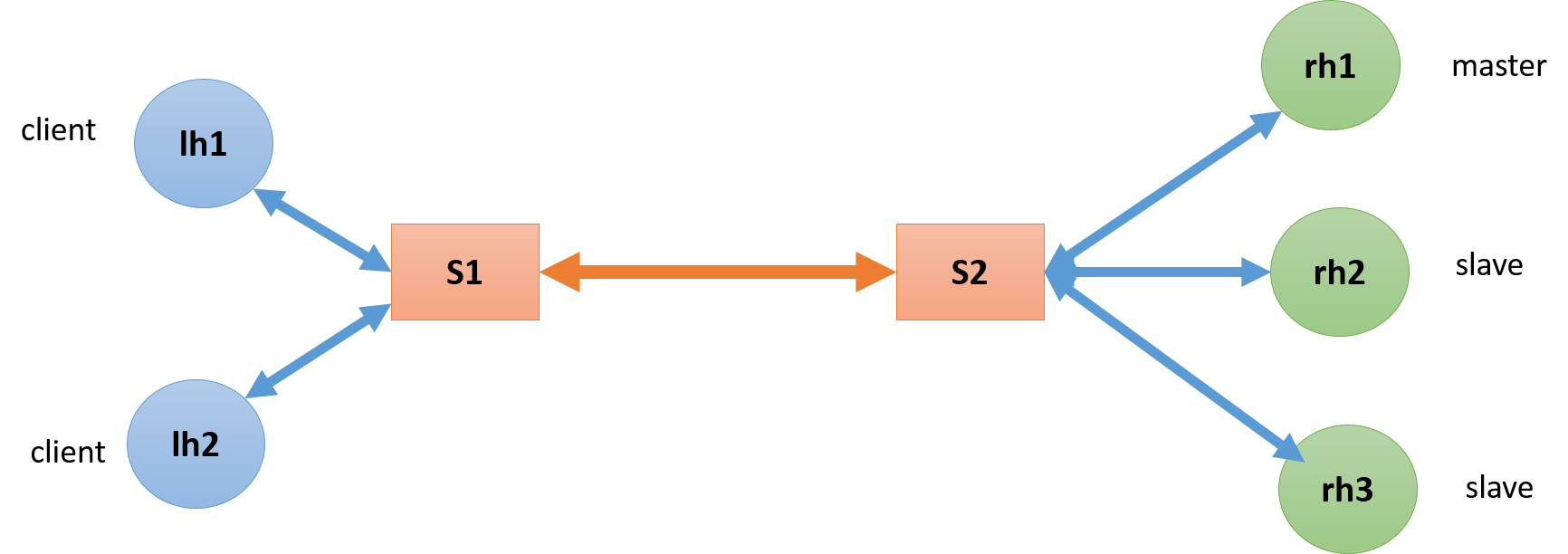


Figure 7 - Mininet Topology

**Scenario 3 - High-latency mininet scenario ("latency")**

**Preconditions:**

- Use **topologies.py** script to create topology with name “**latency**” (bw=10, delay='400ms') in mininet: sudo mn -x --custom topologies.py --link=tc --topo latency

1. **Test case 1\_Latency\_Multiple queries**

|  |  |  |
| --- | --- | --- |
| Steps | Expected result | Actual result |
| Pre-condition: create latency topology in mininet |  | Pass rate: 100% Average rtt: 820 ms Max rtt: 1580 ms Min rtt: 801ms |
| Run client from 1 host | Program runs successfully |
| Client reads the domain names from input file and then sends to server (100 domain names) | Receives IP address successfully from the server |
| Get the roundtrip time for each query to output file | Response after 400 ms and before 30 s |

**Scenario 4 - Slow mininet scenario ("slow")**

**Preconditions:**

- Use **topologies.py** script to create topology with name “**slow**” (max\_queue\_size=5, bw=0.1, delay='200ms') in mininet:

sudo mn -x --custom topologies.py --link=tc --topo slow

1. **Test case 1\_Client\_Long Query Timer**

|  |  |  |
| --- | --- | --- |
| Steps | Expected result | Actual result |
| Pre-condition: create slow topology in mininet |  | Pass rate: 100% Average rtt: 421 ms Max rtt: 992 ms Min rtt: 403 ms |
| Run client from 1 host | Program runs successfully |
| Client reads the domain names from input file and then sends to server (100 domain names) | Receives IP address successfully from the server |
| Get the roundtrip time for each query to output file | Response after 200 ms and before 30 s |

**Scenario 5 - Over-buffered mininet scenario ("buffers")**

**Preconditions:**

- Use **topologies.py** script to create topology with name “**buffers**” (max\_queue\_size=200, bw=0.1, delay='200ms') in mininet:

sudo mn -x --custom topologies.py --link=tc --topo buffers

1. **Test case 1\_Buffers\_Number of packets is greater than max queue size**

|  |  |  |
| --- | --- | --- |
| Steps | Expected result | Actual result |
| Run Iperf server in master server and Iperf client in client side: - Server: iperf -s -u - Client: iperf -c 10.0.0.3 -u | Iperf runs successfully | Pass rate: 98% Average rtt: 212 ms Max rtt: 12682 Min rtt: 44 ms |
| Run client from 1 host | Program runs successfully |
| Client reads the domain names from input file and then sends to server (100 domain names) | Receives IP address successfully from the server |
| Get the roundtrip time for each query to output file | Response after 200 ms and before 30 s |

**Scenario 6 - Lossy mininet scenario ("lossy")**

**Preconditions:**

- Use **topologies.py** script to create topology with name “**lossy**” (loss=10, bw=10, delay='20ms') in mininet:

sudo mn -x --custom topologies.py --link=tc --topo lossy

1. **Test case 1\_Client\_Query Timer\_Greater than delay**

Test multiple times

|  |  |  |
| --- | --- | --- |
| Steps | Expected result | Actual result |
| Pre-condition: client running in a loop. Ouput to a file |  | Pass rate: 94% Average rtt: 499 ms Max rtt: 8908 ms Min rtt: 42 ms |
| Run client from 1 host | Program runs successfully |
| Client reads the domain names from input file and then sends to server (100 domain names) | Receives IP address successfully from the server |
| Get the roundtrip time for each query to output file | Response after 20 ms and before 30 s |

**Scenario 7 – SIGNALS handling**

|  |  |  |
| --- | --- | --- |
| Steps | Expected result | Actual result |
| Capturing signal SIGHUP, SIGPIPE, SIGUSR1, SIGUSR2 and SIGINT signals must be captured. | All user defined clean-up must be done after capturing the signal. | All user defined clean-up must be done after capturing the signal. |

**Test results:**

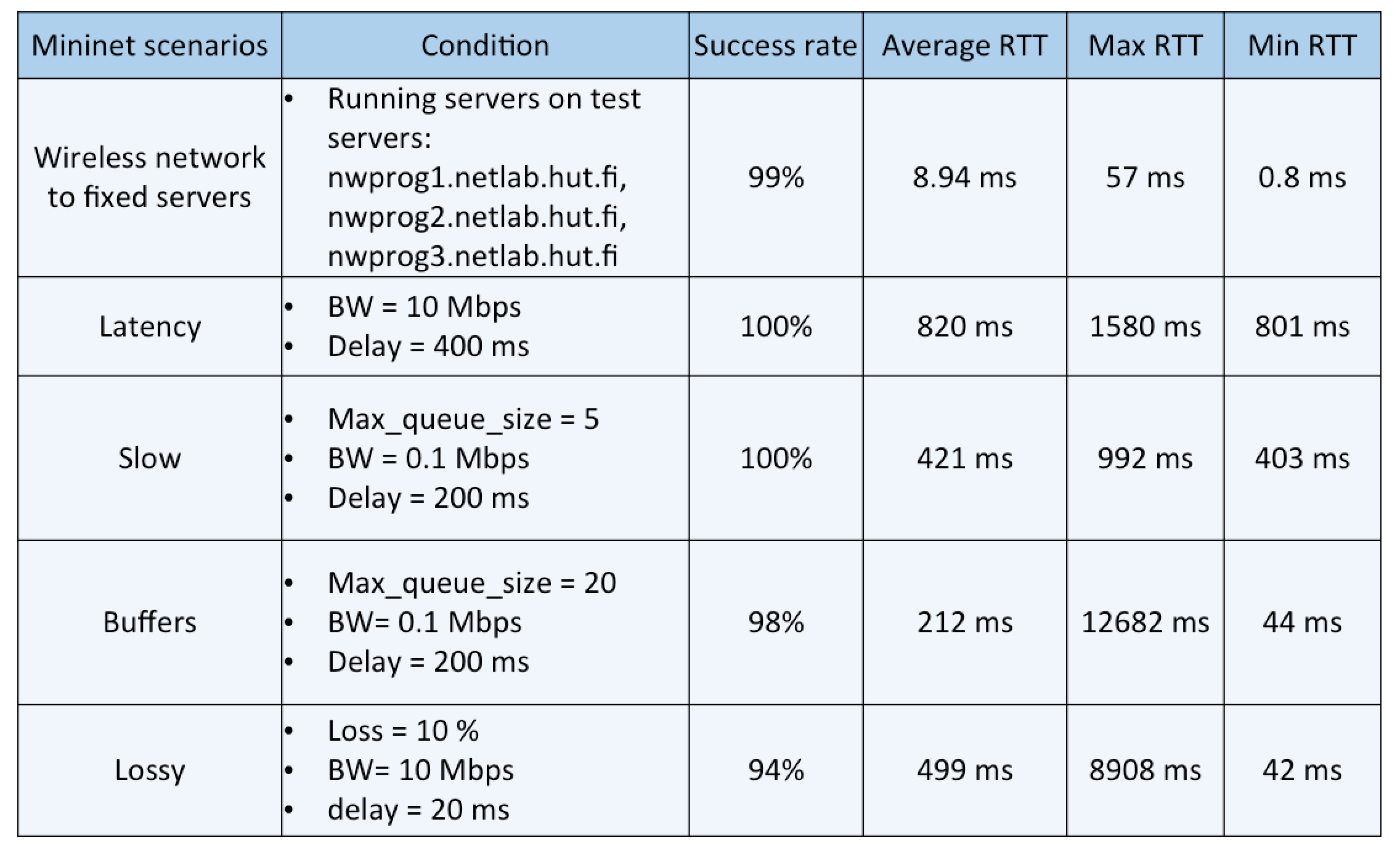


Table 1 - Performance test result

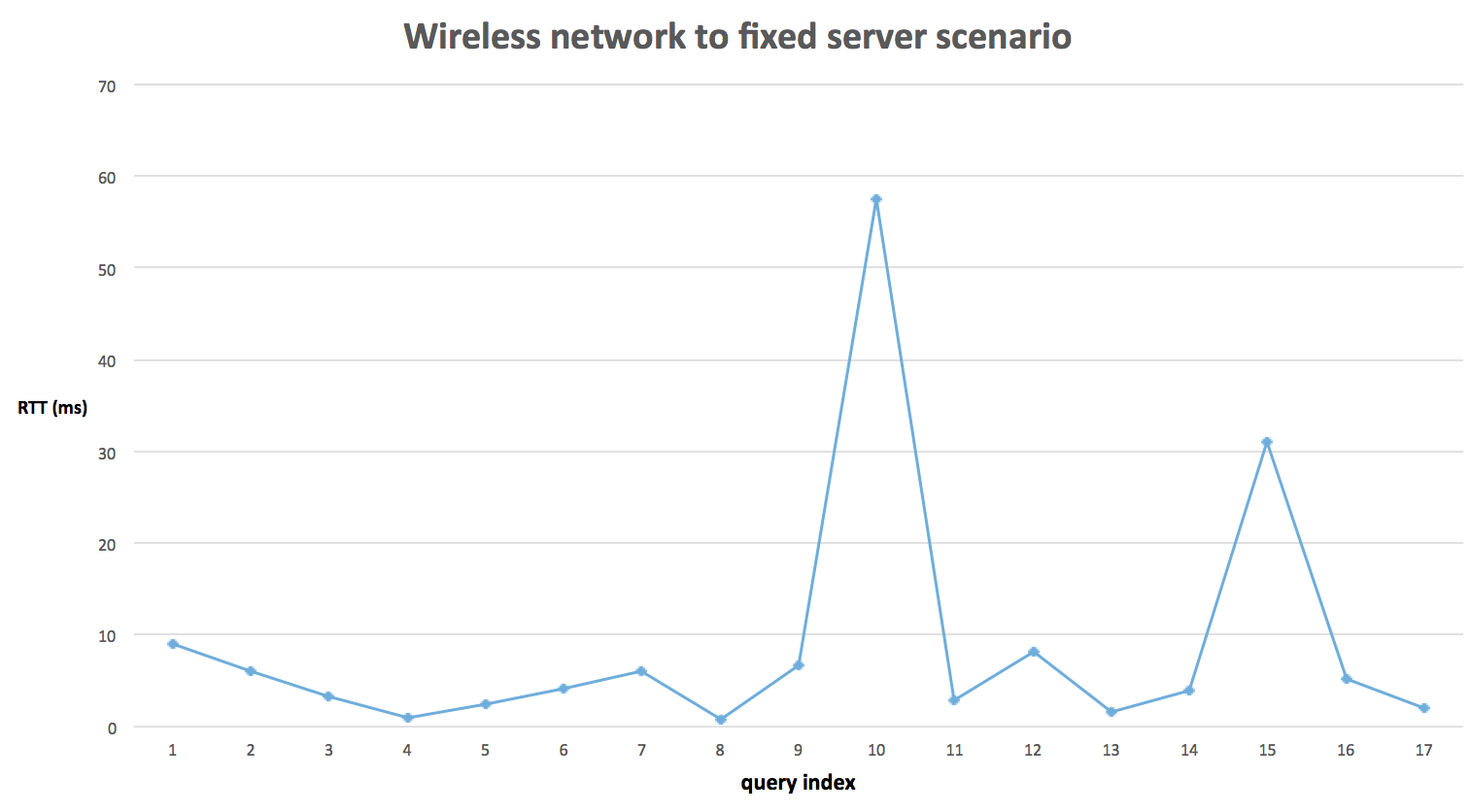


Figure 8 - RTT of quries in wireless network to fixed server scenario

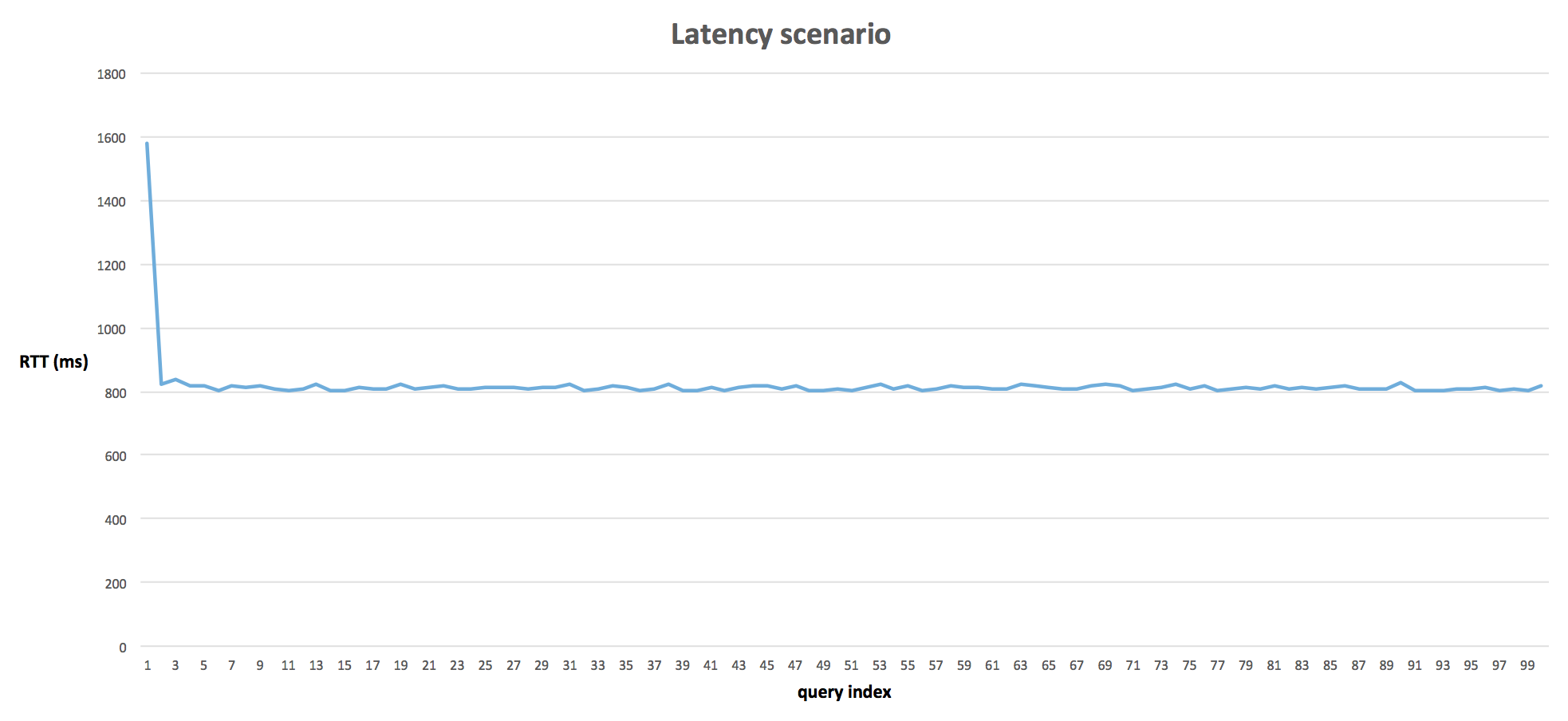


Figure 9 - RTT of queries in Latency scenario

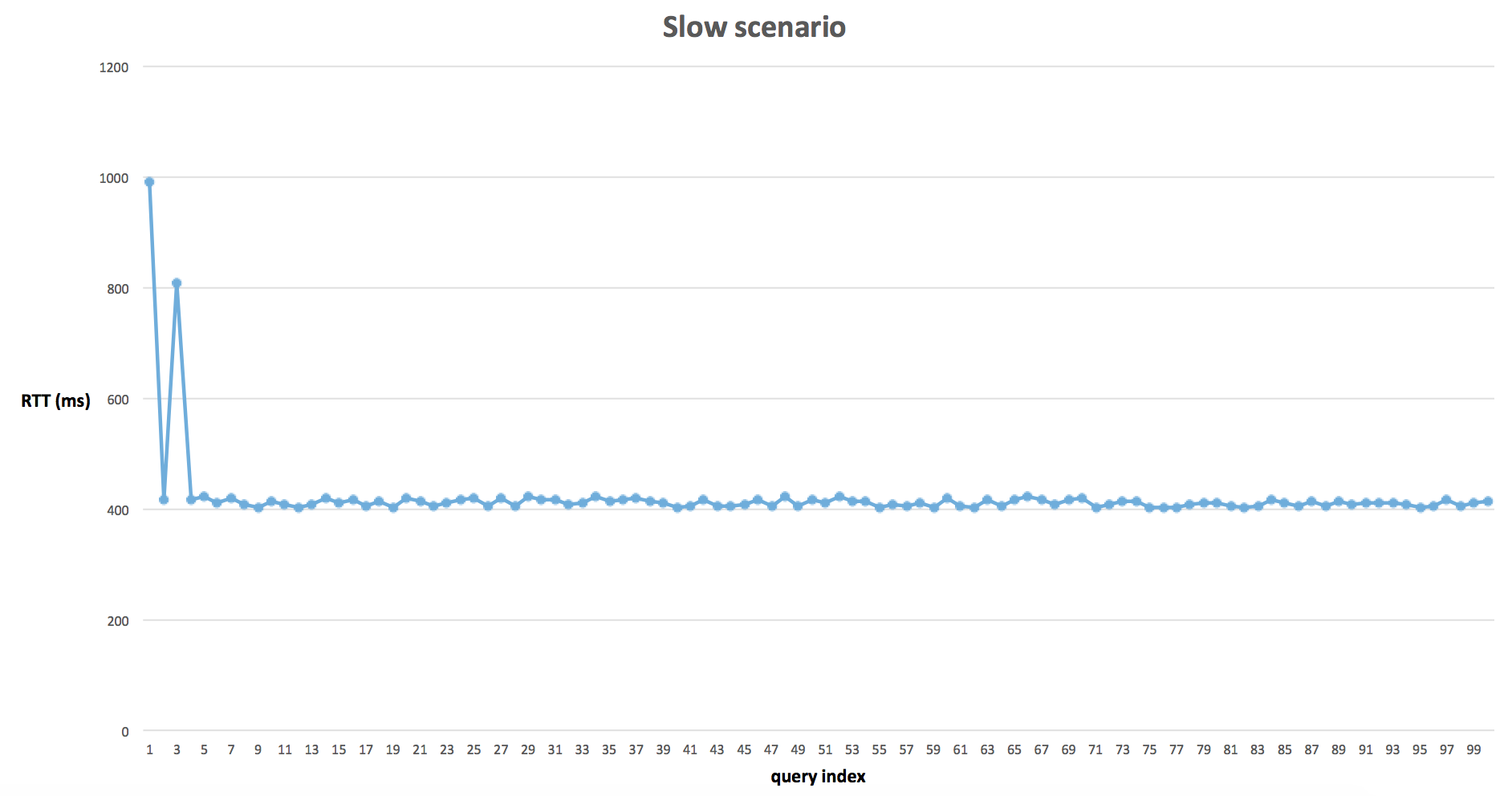


Figure 10 - RTT of queries in Slow scenario

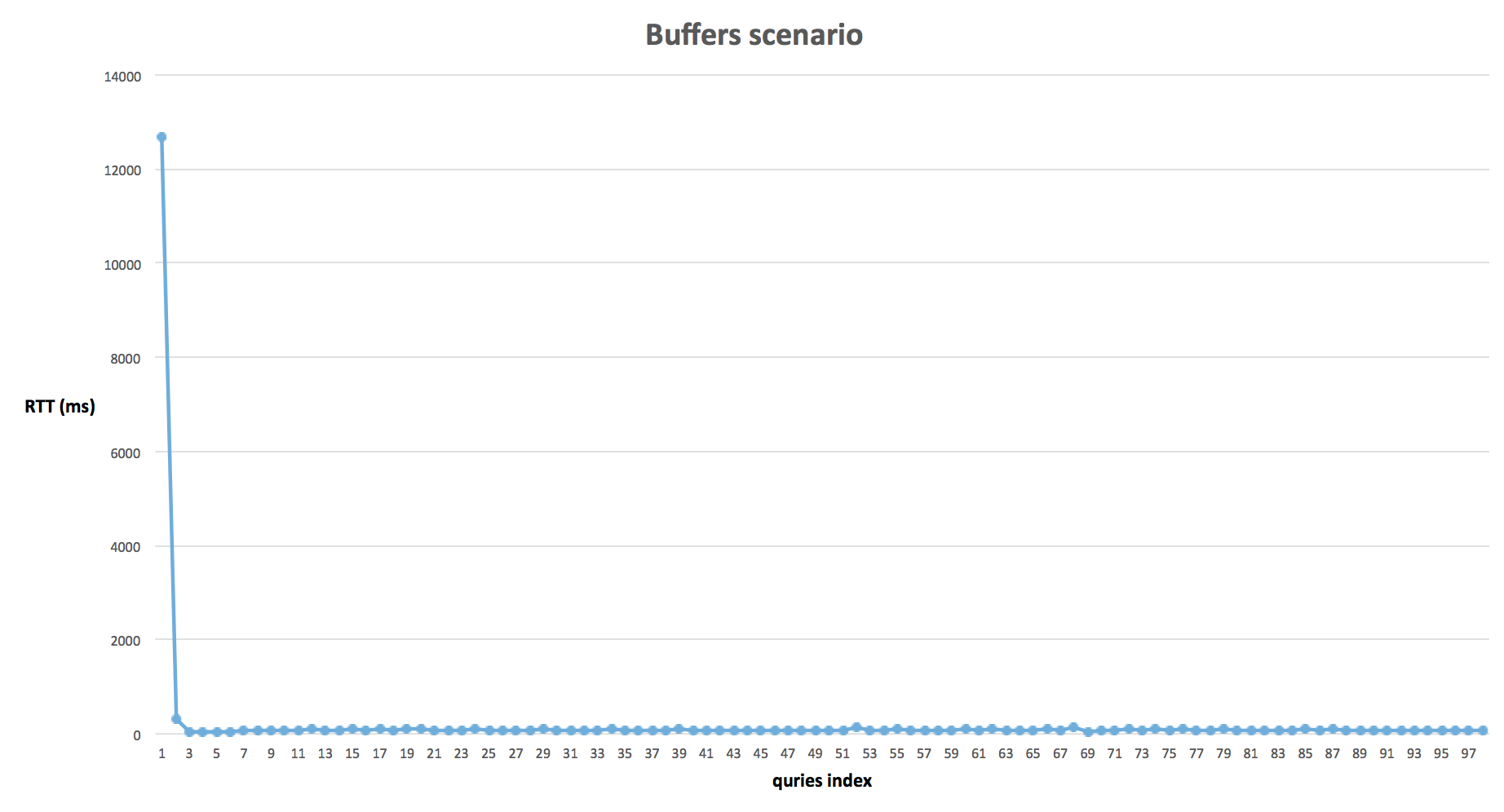


Figure 11 - RTT of queries in Buffers scenario

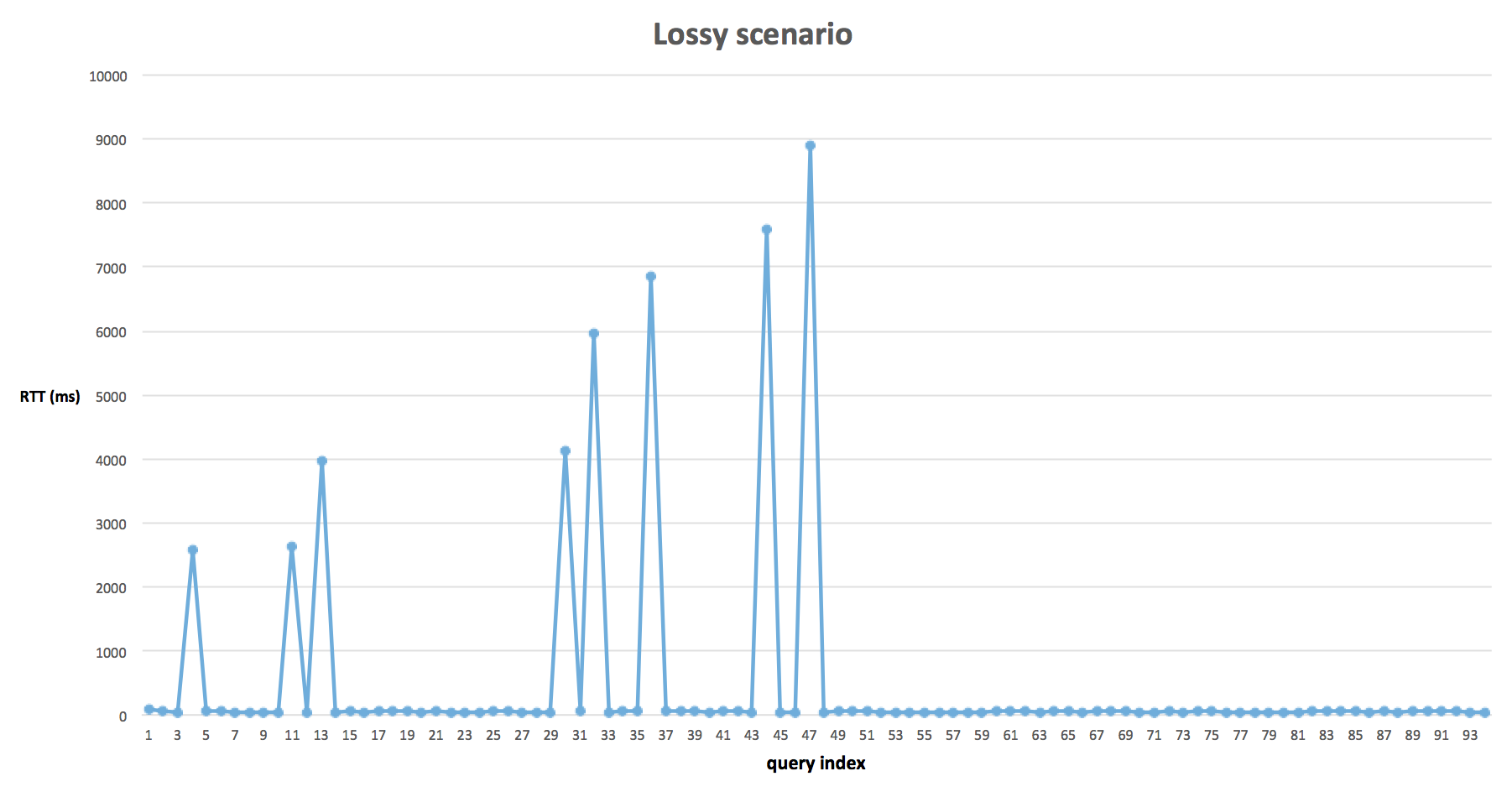


Figure 12 - RTT of queries in Lossy scenario

# 7. Known defects and other shortcomings

**Known defect:**

Run time update of master DB is not properly reflected to the slave servers. This issue is not seen consistently. It happens once in a while.

**Future enhancements:**

- Introducing multiple DB files.

- Advanced security measures.

- Having multiple master servers.

- Dynamically getting the server\_list from any other server during boot-up.

- Having hierarchical database system.

# 8. Distribution of work

Total project is internally divided different stages:

1) Design stage – Phase 1

2) Implementation stage – Phase 1 & Phase 2

2) Testing/Bug fixing stage – Phase 2 & Phase 3

**Phase – 1 Tasks**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.No** | **Task** | **Task Owner** | **Duration** | **Phase** |
| 1 | Design phase (Everyone studied the project before design discussion) | All Team members | 2 hours | Phase 1 |
| 2 | Documentation for  phase 1 | Tan & Karthika | 8 hours | Phase 1 |
| 3 | Client: UDP to send DNS request and receive response, randomly select DNS server | Bijay | 10 hours | Phase 1 |
| 4 | Server: UDP to receive DNS request, read data and send response | Tan | 10 hours | Phase 1 |
| 6 | TCP synchronization for master server | Rohan | 10 hours | Phase 1 |
| 7 | TCP synchronization for slave server | Karthika | 10 hours | Phase 1 |

**Phase – 2 Tasks**

| **S.No** | **Task** | **Task Owner** | **Duration** | **Phase** |
| --- | --- | --- | --- | --- |
| 1 | Client: UDP to send DNS request and receive response, randomly select DNS server | Bijay | 10 hours | Phase 2 |
| 2 | TCP synchronization for master server | Rohan | 10 hours | Phase 2 |
| 3 | TCP synchronization for slave server | Karthika | 20 hours | Phase 2 |
| 4 | Makefile | Bijay and Rohan | 4 hours | Phase 2 |
| 5 | Load balancing on client side | Tan & Bijay | 4 hours | Phase 2 |
| 6 | Mutex lock | Karthika | 6 hours | Phase 2 |
| 7 | Documentation for  phase 2 | Karthika | 8 hours | Phase 2 |
| 8 | High level test plan | Tan & Bijay | 15 hours | Phase 2 |
| 9 | Integration of code | Rohan | 15 hours | Phase 2 |
| 10 | Timeout in client | Bijay | 5 hours | Phase 2 |

**Phase – 3 Tasks**

| **S.No** | **Task** | **Task Owner** | **Duration** | **Phase** |
| --- | --- | --- | --- | --- |
| 1 | Integration Testing | Tan & Karthika | 20 hours | Phase 3 |
| 2 | Mininet Testing | Tan | 30 hours | Phase 3 |
| 3 | Bug fixing | Karthika | 40 hours | Phase 3 |
| 5 | Bug fixing | Rohan | 15 hours | Phase 3 |
| 6 | Bug fixing | Bijay | 25 hours | Phase 3 |
| 7 | Daemon process | Tan | 6 hours | Phase 3 |
| 8 | Signal handling | Karthika | 6 hours | Phase 3 |
| 9 | Valgrind | Rohan | 6 hours | Phase 3 |
| 10 | Documentation for  phase 3 | Karthika | 10 hours | Phase 3 |

**Meeting schedule:**

**Meeting – 1:**

**Date:** 10/02/2016

**Location:** I346, Otakaari 5.

**Time:** 12 – 2 PM

**Agenda:** Project kick-off

**Participants:** Karthika, Tan, Rohan, Bijay

**Minutes of the meeting:**

1) Project manager “Karthika”, Communication channel (telegram / facebook messenger), Programming language “C”

2) Understood the requirements of the project.

**Meeting – 2:**

**Date:** 19/02/2016

**Location:** I346, Otakaari 5.

**Time:** 2 – 4 PM

**Agenda:** Design clarifications from advisor.

**Participants:** Karthika, Tan, Bijay, advisor (Arseny)

**Minutes of the meeting:**

1) Discussed possible designs for the project. 2) Clarified queries from advisor.

**Meeting – 3:**

**Date:** 24/02/2016

**Location:** Table in front of I346, Otakaari 5.

**Time:** 12 – 2 PM

**Agenda:** Freeze the design of the project.

**Participants:** Karthika, Tan, Rohan, Bijay

**Minutes of the meeting:**

1) Design for DNS implementation project was finalized. 2) Project implementation was divided into various phases. 3) Tasks of the first phase must be completed before/on march 4th 2016. 4) Next meeting is scheduled for next week (before March 4th class). 5) Before next meeting, each project member have to list all sub-tasks from assigned high level task.

**Meeting – 4:**

**Date:** 02/03/2016

**Location:** Table in front of I346, Otakaari 5.

**Time:** 1 – 2 PM

**Agenda:** To check the status of the project and deliverables of phase-1.

**Participants:** Karthika, Tan, Rohan, Bijay

**Minutes of the meeting:**

1) Status of the phase – 1 deliverable is checked. 2) Status of coding on various sub-tasks were checked. 3) Deadlines for the sub-tasks were fixed.

**Meeting – 5:**

**Date:**

10/03/2016

**Location:** Table in front of I346, Otakaari 5.

**Time:** 12 – 3 PM

**Agenda:** To check the status of the project and deliverables after phase-1. Checked out few new tasks.

**Participants:** Karthika, Tan, Rohan. Bijay

**Minutes of the meeting:**  
1) Integrator - Rohan HLTP (High Level Test Plan) - Tan, Bijay, Document - Karthika.  
2) Server synchronization must be implemented in phase-2  
3) Threads implementation must be done in phase-2  
4) Daemon process - Rohan & Tan.  
5) Lock implementation - Karthika  
6) Enhancements to be done in phase-3

**Meeting – 6:**

**Date:**

14/03/2016

**Location:** Table in front of I346, Otakaari 5.

**Time:** 11 AM – 12 PM

**Agenda:** To check the status of the project and deliverables of phase-2.

**Participants:** Karthika, Tan, Rohan

**Minutes of the meeting:** 1) Discussed issues in integration. 2) Status of tasks.

**Meeting – 7:**

**Date:**

15/03/2016

**Location:** Maari Building

**Time:** 6 PM – 9 PM

**Agenda:** High Level Test Plan **(**HLTP) preparation and Sync testing

**Participants:** Karthika, Tan, Rohan, Bijay

**Minutes of the meeting:** 1) HLTP completion

2) Sync basic functionality

**Meeting – 8:**

**Date:**

22/03/2016

**Location:** Second floor, Otakaari 5.

**Time:** 11 AM – 12 PM

**Agenda:** Discussion with advisor

**Participants:** Karthika, Tan, Rohan, Arseny

**Minutes of the meeting:** 1) Status of project

2) Clarified queries from Arseny.

**Meeting – 9:**

**Date:**

30/03/2016

**Location:** Table in front of I346, Otakaari 5.

**Time: 1**2 PM – 3 PM

**Agenda:** Bug fixing

**Participants:** Karthika, Tan, Bijay

**Minutes of the meeting:** 1) Bug fixing deadlines

**Meeting – 10:**

**Date:**

04/04/2016

**Location:** Table in front of I346, Otakaari 5.

**Time:** 2 PM – 6 PM

**Agenda:** Bug fixing

**Participants:** Karthika, Tan, Rohan, Bijay

**Minutes of the meeting:** 1) Bug fixing deadlines

**Meeting – 11:**

**Date:**

05/04/2016

**Location:** Table in front of I346, Otakaari 5.

**Time:** 2 PM – 6 PM

**Agenda:** To prepare for exam and bug fixing

**Participants:** Karthika, Tan, Rohan, Bijay

**Minutes of the meeting:** 1) Bug fixing deadlines

**Meeting – 12:**

**Date:**

07/04/2016

**Location:** Library meeting room ELEC Building.

**Time:** 10 AM – 10 PM

**Agenda:** Bug fixing and testing

**Participants:** Karthika, Tan, Rohan, Bijay

**Minutes of the meeting:** 1) Bug fixing

2) Integration testing

3) Mininet testing.

**Meeting – 13:**

**Date:**

08/04/2016

**Location:** Library meeting room ELEC Building.

**Time:** 10:30 AM – 2 PM

**Agenda:** To do regression testing before demo.

**Participants:** Karthika, Tan, Rohan, Bijay

**Minutes of the meeting:** 1) Code is proper for demo.

2) Document has to be completed before midnight.

# 9. References

[1] <https://src.aalto.fi/ip/instructions/index.html#03_topics>

[2] <http://compnetworking.about.com/cs/domainnamesystem/g/bldef_dns.htm>/

[3] <http://www.programminglogic.com/sockets-programming-in-c-using-udp-datagrams/>

[4] <http://www.binarytides.com/dns-query-code-in-c-with-linux-sockets>/

[5] <https://training.github.com/kit/downloads/github-git-cheat-sheet.pdf>

[6] <https://www.git-tower.com/blog/git-cheat-sheet/>

[7] <https://tools.ietf.org/html/rfc1034>

[8] <http://tools.ietf.org/html/rfc3986>

[9] <http://stackoverflow.com/questions/4658409/whats-the-most-efficient-file-logging-in-a-server-written-in-c>

[10] http://www.netzmafia.de/skripten/unix/linux-daemon-howto.html

**Appendix:**

**Valgrind report:**

**Master server:**

==1665== LEAK SUMMARY:

==1665== definitely lost: 0 bytes in 0 blocks

==1665== indirectly lost: 0 bytes in 0 blocks

==1665== possibly lost: 1,088 bytes in 4 blocks

==1665== still reachable: 834 bytes in 5 blocks

==1665== suppressed: 0 bytes in 0 blocks

==1665==

==1665== For counts of detected and suppressed errors, rerun with: -v

==1665== ERROR SUMMARY: 26 errors from 9 contexts (suppressed: 0 from 0)

**Slave server:**

==1711== LEAK SUMMARY:

==1711== definitely lost: 0 bytes in 0 blocks

==1711== indirectly lost: 0 bytes in 0 blocks

==1711== possibly lost: 544 bytes in 2 blocks

==1711== still reachable: 1,638 bytes in 10 blocks

==1711== suppressed: 0 bytes in 0 blocks

==1711==

==1711== For counts of detected and suppressed errors, rerun with: -v

==1711== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)