PROJECT 4
ECE 6110 Spring 2015
CAD for Computer Networks

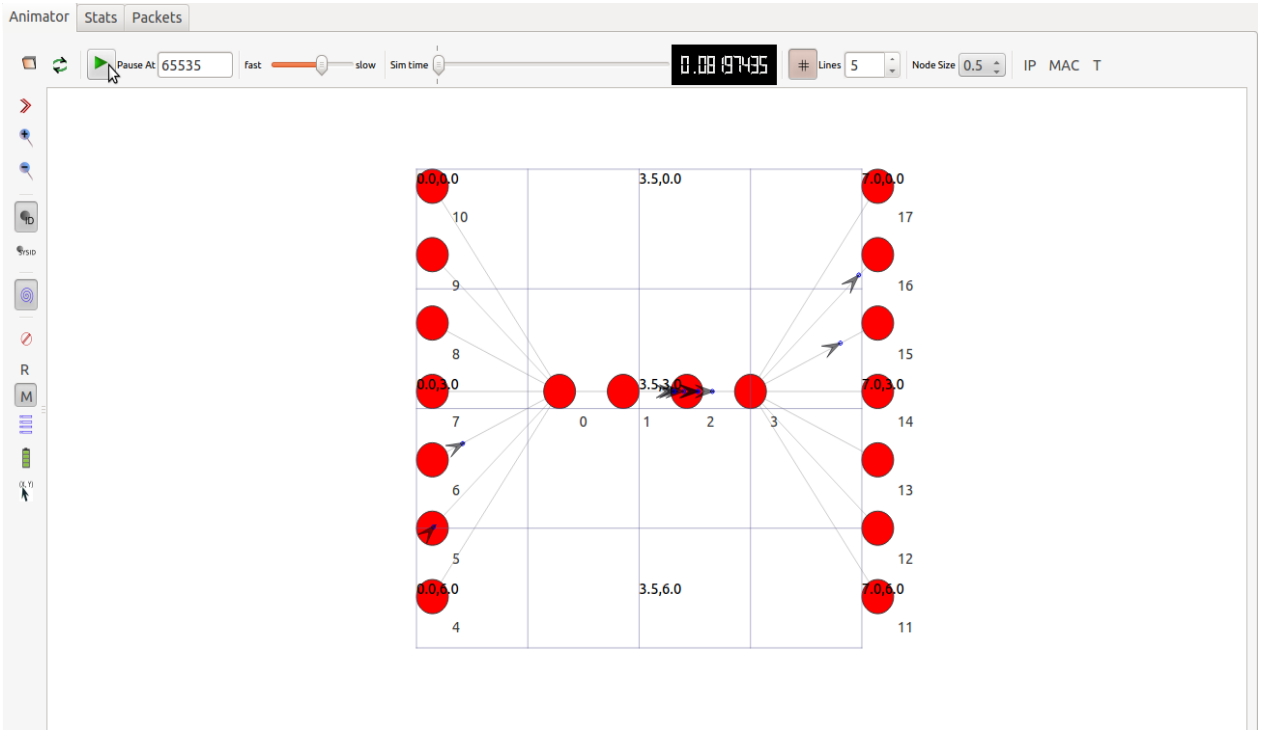
Tuning Red for Web Traffic

Members:
Amit Kulkarni (903038158)
Rashmi Mehere (903068492)
Rasika Subramanian (903037820)
Urvi Sharma (903044011)

Introduction:

In this project, we have recreated the Tuning Red experiment published by Jeffay et. al using ns-3 to see if the cumulative distributions measured by ns-3 match closely to those measured in the laboratory experiment performed by Jeffay. Web browsers and Web server Models were created for client, servers. We compare the performance of DropTail and RED queuing for web traffic. For the probability distributions model we implemented Jeffay's distribution model to simulate HTTP traffic (as seen in his paper 'Stochastic Models for Generating Synthetic HTTP Source Traffic').

Topology:



In our project we have created a topology in which, the rightNodes represent web servers and the leftNodes represent the clients. A total of 3000 web browsers are to be installed on the left and right node. Hence, each node represents 1 to 3000/7 web browsers according to Jeffay's parameters. An object of ApplicationContainer class has been created namely ServerApps that has been installed on the right nodes. Similarly, we have created ClientApps that has been installed on the left nodes. The nodes in the center are the routers. The screenshot of the working network topology is shown above.

Code:

The code takes an input from the user called ‘queueType’ which can be used to select the type of queuing used in the bottleneck link. Different link parameters are set depending on the topology set by Jeffay. For the topology, http client applications are installed on the left nodes and servers are installed on the right nodes. We assign the nodes their position using ConstantPositionMobilityModel for the animation. Web browser applications are installed on the left nodes depending on the user defined ‘*load’* value. At a *load* value of 100%, a total of 3000 browsers are installed by distributing them among the 7 nodes on the left. Therefore, at 100% load a total of 3000/7 = 428 browsers are installed on a single node to replicate the experiment performed by Jeffay. All the browsers begin at a random start time which are assigned with the help of a random variable.

When the simulator starts running, a connection is established between the client nodes and the servers before actual packets are transmitted. The http module is designed in such a way that it generates all the parameters like responseSize, thinkTime, serverDelay etc required for http operation.

We are printing values of sent and received times ie the for various browsers. We calculated response time by finding the difference between the two and plotted it.

Simulation of Web Browser:

The implementation of HTTP traffic generator composes four parts:

1. Client:
Class http::HttpClient
Implements functions for HTTPClient client and server applications. This application is installed on client nodes. Some of the important functions of this application are:

- *HttpClient::SetClientApplication*
  This function notifies about the completion of a connection.
- *HttpClient::ClientReceive (Ptr<Socket> socket)*
  It deals with web client receiving web pages parameter socket: the socket that sends the web page.

2. Server:
Class http::HttpServer
Functions to receive the connection from HTTP clients and establish the connection receive HTTP packets received by the HTTP client. The important functions of this applications are:

- *void StartApplication(void)*
  This function is called at a specified time to start the application.
- *void ServerReceive (Ptr<Socket> socket)*
  The server starts to receive the packets from the client from the socket which is used for the http connection

3. Controller:
Class http::HttpController
Centrally controls all the sending events of both client and server. Saves the Application Data Unit(ADU) containers for client and server.

- *void ClientSend (Ptr<socket> socket)*
  This function is used to call the first object to be sent by the client.
- *void ScheduleNextClientSend (Ptr<socket> socket)*
  This functions schedules the next server send through the socket.

4. Distribution Generation:
Class http::HttpDistribution
This class contains functions to take use of the distribution functions generated parameters. Some important functions defined in this class are.

- *double GetRequestGap (uint32_t page, uint32_t object)*
  This function is used to determine the time between first requests in consecutive sessions. The parameter 'page' is used to determine which page to work with and the 'object' parameter specifies the object within a webpage.

5. Random variable
Class http::HttpRandomVariableBase
This class contains distribution functions to generate the parameters used in the HTTP traffic model. The functions defined in this class return an average value for each distribution function.

Working of the model:

When the traffic generator starts, the http client initializes connections to the http server. After the connection is established, the client calls the distribution generation module and generates empirical parameters involved with http operation, some of which are request size, response size, page request gap time, object request gap time server delay time, total objects per page, total pages per web session. All these parameters will be saved in the runtime variable model, which can be accessed by both the http client and server. Based on the parameters saved in runtime variable, the client and server will generate http traffic which represents real-world http traffic.

This traffic generator is capable of generating both HTTP 1.0 traffic and HTTP 1.1 traffic. When HTTP 1.0 mode is used, each http request will wait until the previous http transaction is over, while in HTTP 1.1 has pipelining, which means that it sends multiple http requests without waiting for previous responses. However, parallel connections are not yet supported.

ADU defines the size and useful parameters in simulation. It can represent both request and response size ADUs. The ADUs are saved in ADU container and both the client and server have one copy of it. The request ADU and response ADU are saved in sequence inside the ADU container.

There are two working modes:

- Internet-like
- User-Defined

When internet-like mode is used, it generates the variables based on predefined distribution functions.

This module though, does not use the built in tracing module of ns-3, it generates its own files to record the parameters generated from distribution functions as well as the simulation results. The reason for this choice is that some of the metrics we use cannot be represented by the ns-3 built in tracing module. For example, the Web Page Delivery Delay needs to identify the time it takes for the whole HTTP object to be delivered to the destination. The build tracing module in ns-3 can only monitor the time it takes for a single transport segment to reach the destination. This is the reason we choose to use the text-based tracing of the module. Furthermore, the tracing module is carefully tested and validated.

To run the code, we have to add the newly created HTTP module in *ns-3.21/src/applications* and run the following commands on terminal to build the entire module in ns-3 *CXXFLAGS="-Wall" ./waf configure* and *./waf -vv*. After all the tests are built, we can install HTTP client and server applications on nodes and compile the code as usual.

DropTail Queue:
DropTail queueing method is a simple active queue management (AQM) technique. This method is independent of the network traffic. It will drop the packets if the queue is full. Hence, the goodput of DropTail queue varies heavily with increasing queue size and bandwidth at the bottleneck links.

RED Queue:
Random Early Detection or Random Early Discard is a more complicated queueing method compared to DropTail queue. It was observed in DropTail queue that, if the buffer was full, the packets were dropped. DropTail is independent of the transport protocol used. Due to this, the network remains under-utilized. RED was designed to address this issue.

Operation:
RED makes use of statistical probabilities to drop packets. This makes this method fairer as compared to DropTail. If the router buffer is empty or almost empty, the probability of dropping the packers is low and almost all incoming packets are accepted.
As the traffic increases and the buffer space decreases, the probability of dropping packets increases. When this probability reaches 1, all the packets are dropped and congestion takes place. The main parameters of RED are queueLimit, maxTh (Max queue threshold) and minTh (minimum threshold). maxTh parameter triggers forced drops and minTh parameter is a threshold for probabilistic drops. The goodput also depends on the load or the data rate of the sources.

<u>Parameters used by RED:</u>

Qlen, minth, maxth, wq, maxp

Qavg – average queue length
Qavg < minth, add the datagram
Qavg > maxth, discard the datagram – forced drop
minth < Qavg < maxth, random discard according to probability p – early drop test

RED can be optimised in the following ways:

- The minth should be large enough to ensure output link has high utilization
- If Qlen ≈ maxth it will be close to tail-drop behavior hence, maxth > 2 x minth is recommended
- p depends on current queue size but as traffic is bursty, that is why weighted queue size is used in practice. Hence, Early drop probability p = maxp x (avg – minth) / (maxth – minth), where avg depends on wq.

Jeffay's Paper:

In Jeffay's paper, we study the effects of RED on the performance of Web browsing with our work being the use of a user centric measure of performance that is, the response time for HTTP request-response pairs.

Even though Active Queue Management [AQM] is recommended for Internet congestion avoidance, RED is the best known AQM technique, has not been studied much for Web traffic, the dominant subset of TCP connections on the Internet in 2000. In the paper, the authors use response time, a user-centric performance metric, to study short-lived TCP connections that model HTTP 1.0. They model HTTP request-response pairs in a lab environment that simulates a large collection of browsing users. Artificial delays are added to a small lab testbed to approximate coast-to-coast US round trip times (RTT's). The authors use the effect of RED vs. DropTail FIFO on response time for HTTP 1.0 as the basis of comparison.

We empirically evaluated RED across a range of parameter settings and offered loads. Our results show that:

- Contrary to expectations, compared to a FIFO queue, RED has a minimal effect on HTTP response times for offered loads up to 90% of link capacity.
- Response times at loads in this range are not substantially affected by RED parameters,
- between 90% and 100% load, RED can be carefully tuned to yield performance somewhat superior to FIFO, however, response times are quite sensitive to the actual RED parameter values selected.
- In such heavily congested networks, RED parameters that provide the best link utilization produce poorer response times.

Hence, for links carrying only web traffic, RED queue management appears to provide no clear advantage over DropTail for end-user response times.

Mah's Distribution Model:

Mah's Distribution Model is an empirical model of network traffic produced by HTTP. So, Instead of relying on server or client logs, our approach is based on packet traces of HTTP conversations. Through traffic analysis, the statistics and distributions for higher-level quanti-ties such as the size of HTTP files, the number of files per "Web page", and user browsing behavior can be determined. These quanti-ties form a model can then be used by simulations to mimic World Wide Web network applications.

Jeffay's Distribution Model:

We used the distribution seen in the paper 'Stochastic models for generating synthetic HTTP source traffic' by Jeffay et. al. It is based on a large-scale empirical study of web traffic on 2 access links connecting Bell Labs and UNCC to the Internet. Their model is similar to Mah's as it is characterized by values of source variables such as time gaps between exchanges (analogous to think time), sizes of individual requests and responses (like HTTP Primary and Secondary request and response times and lengths seen in Mah's distribution) etc.

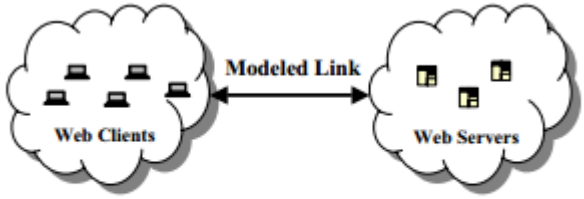
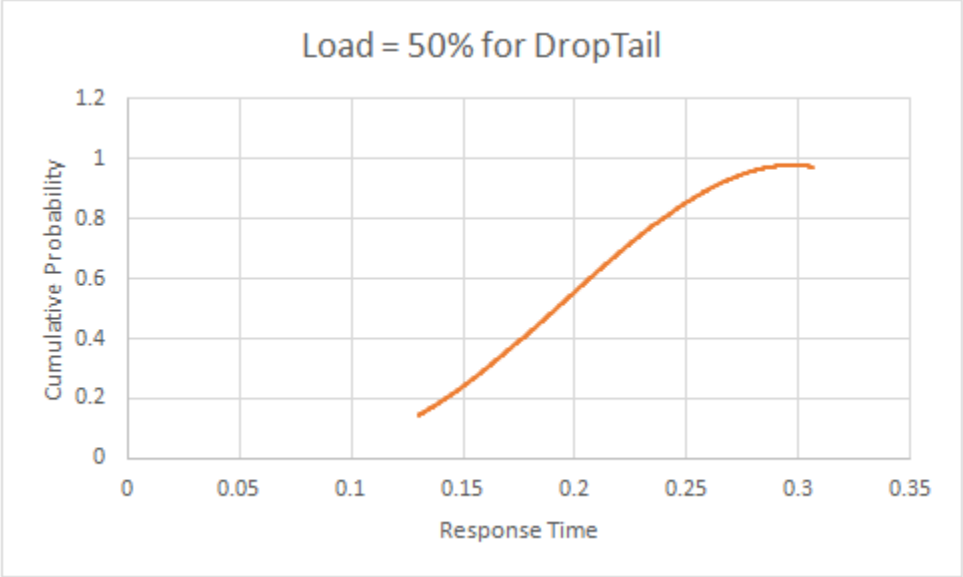


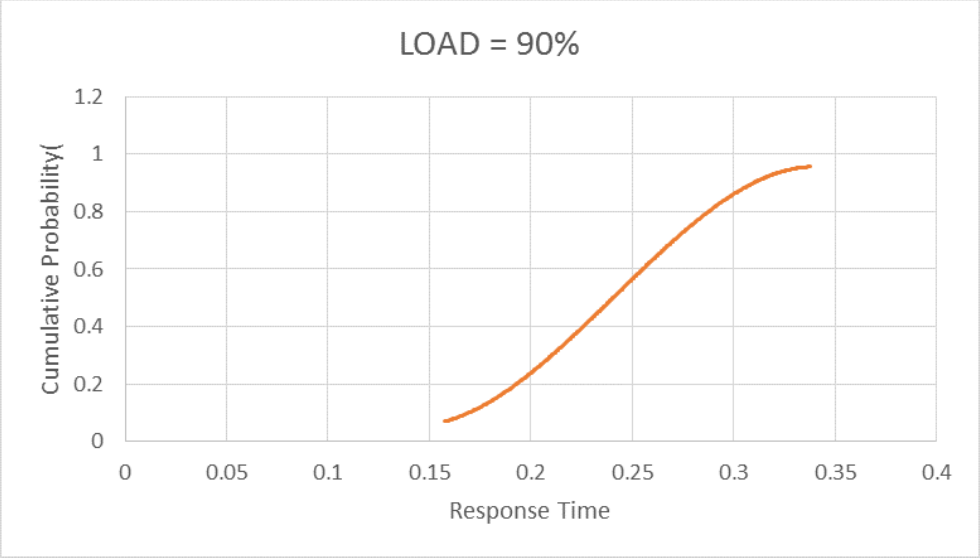
Fig. 1. Client cloud, server cloud, and a modeled network (a single link) carrying web traffic.

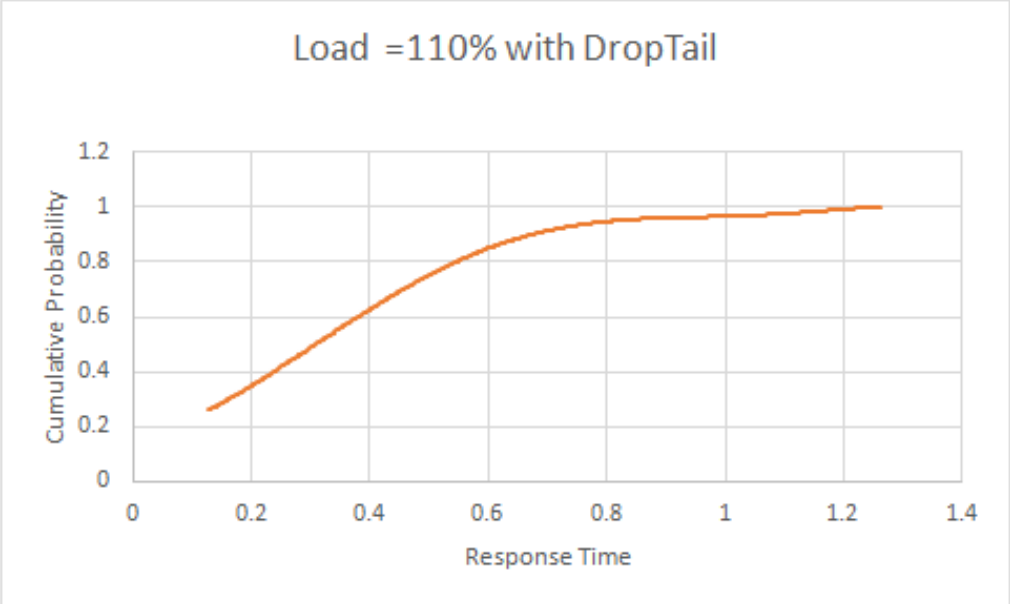
Results:

Note: All response times in milliseconds.

1) **FIFO for different loads:**

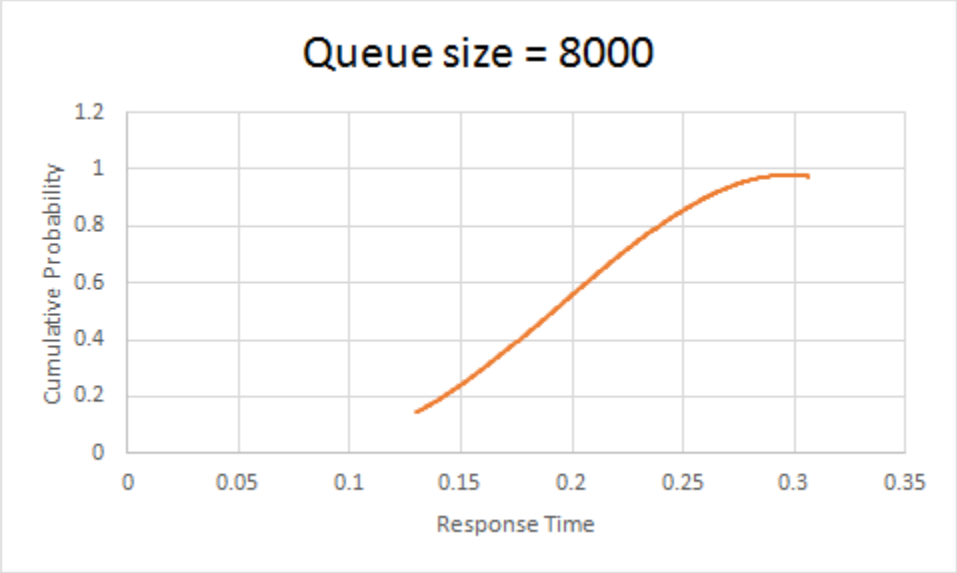


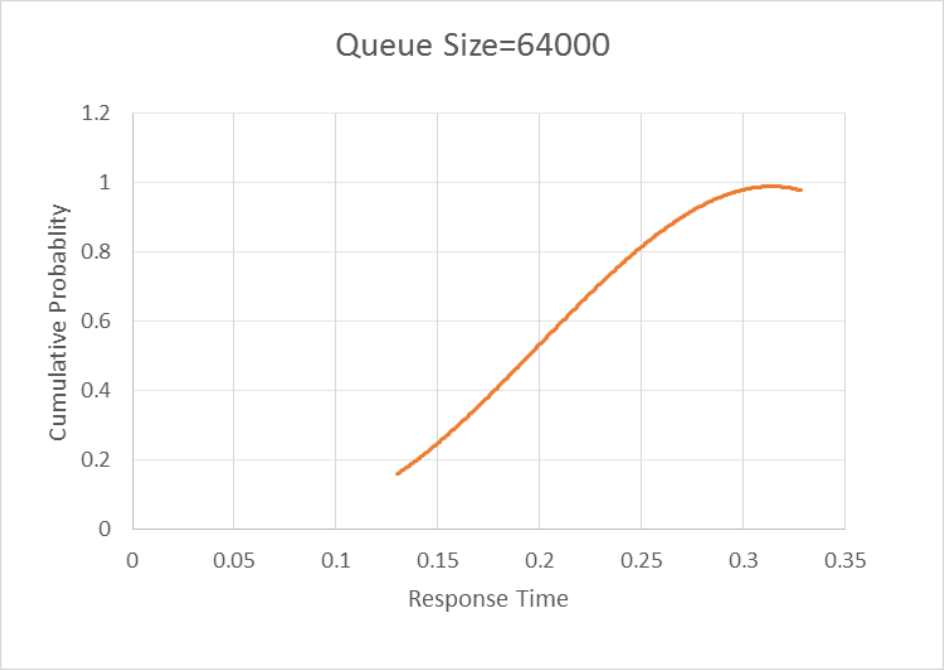




From the above graphs it can be inferred that for loads below 50% there is not much scope to improve the router queuing mechanism.
On carefully tuning the parameters further, it can be noticed that between 50% load and 90% load the performance improves.
When the load is increased to 110% which is above the link capacity of the topology, it is clear that it has a negative effect on the performance of the topology.

2) FIFO performance for different queue sizes:

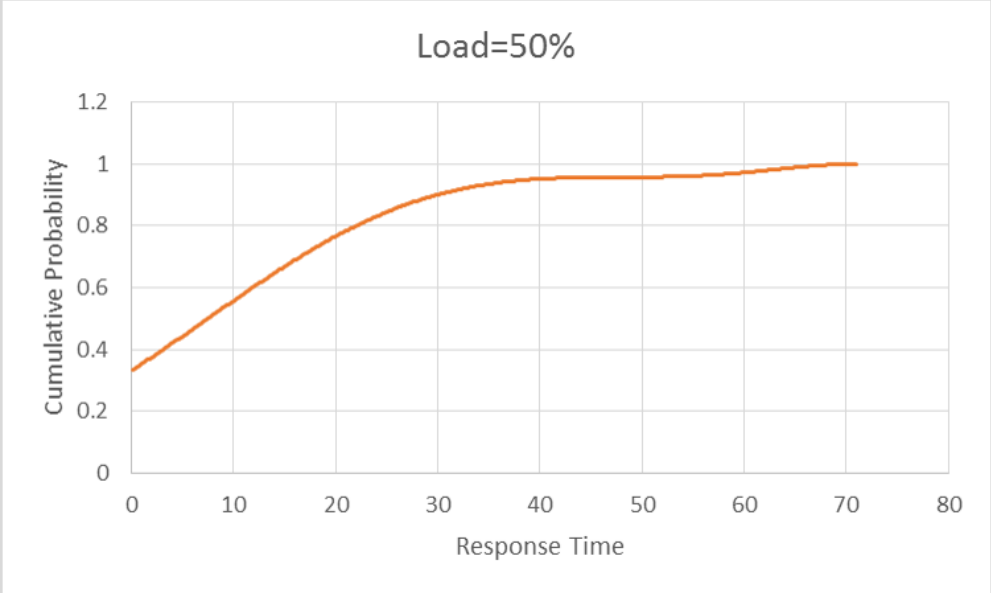


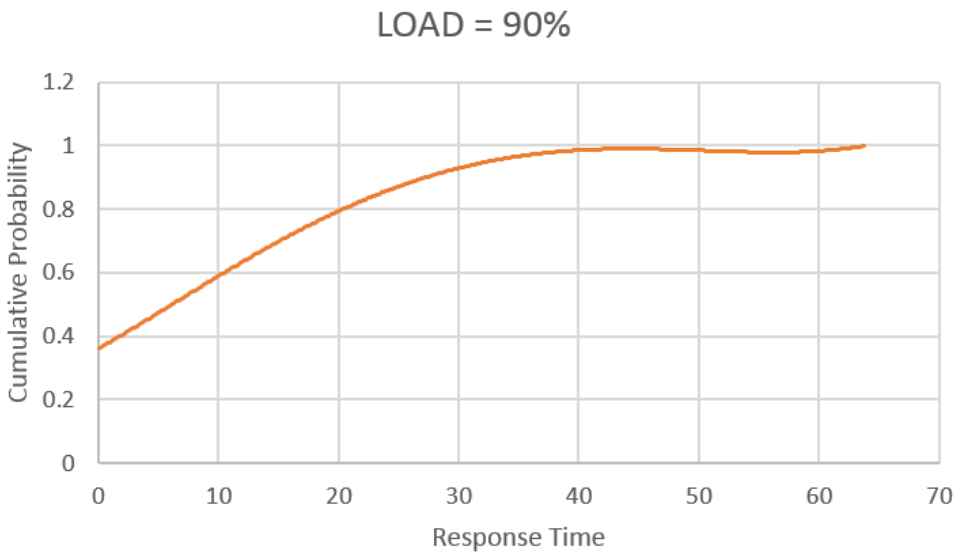


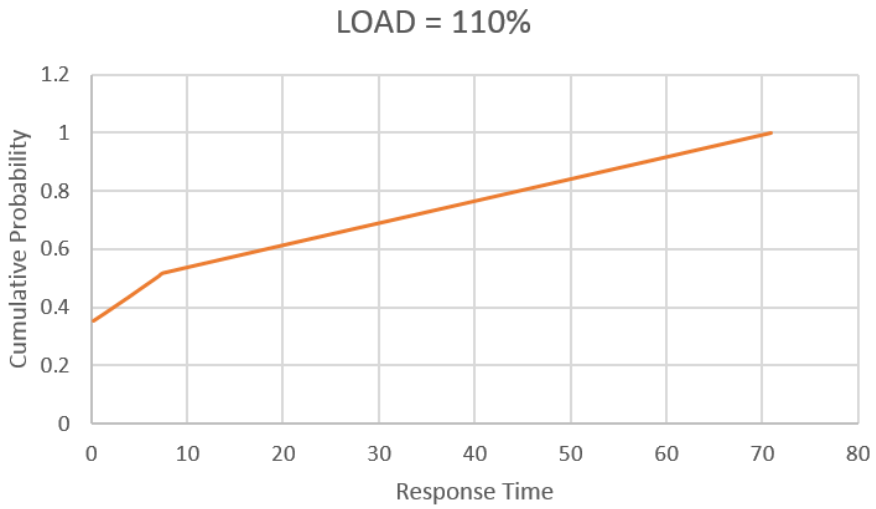
When the queue size of a FIFO queue is varied for fixed load sizes the following inferences can be made:
For lower load values (below 90%) there is no effect of increasing queue size on the response time.
Now, on increasing the load (above 90%) it can be seen that increasing the queue size has an effect on the response.
This effect though depends on the the size of the HTTP response data. Hence, for small requests, increasing the queue size has a negative effect on the response time.
On the other hand, for long requests, increasing the queue size improves the response time.

In this experiment using ns-3 , we could observe the same results as Jeffay had observed in his experiments.

3) RED performance for different loads



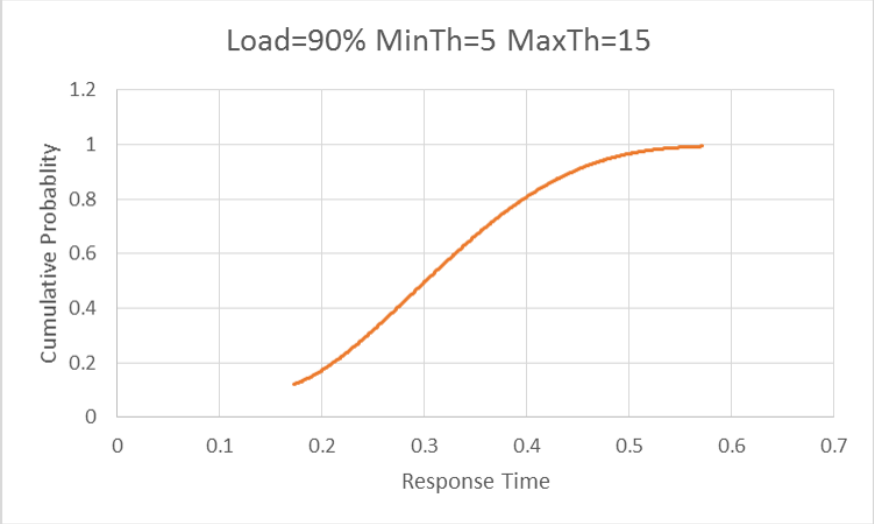


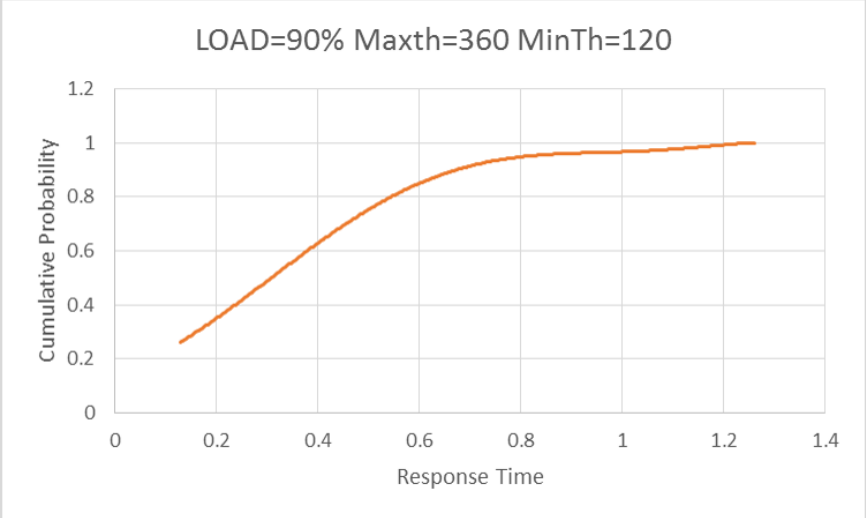


As in FIFO it is clear that for small requests, increasing the queue size has a negative effect on the response time. However, on the other hand for long requests, increasing the queue size improves the response time.

On increasing the load we can observe that at 50% load the number of dropped packets is very low which indicates that at loads of 50% and below, there is limited room for increasing the performance of the router queuing mechanism.
The performance changes significantly as the load is increased from 50% to 110%. Performance degradation only occurs at loads greater than 70%.
When loads exceed 70%, the performance decreases as the load increases. The most significant performance decrease occurs at load levels of 90-110%.

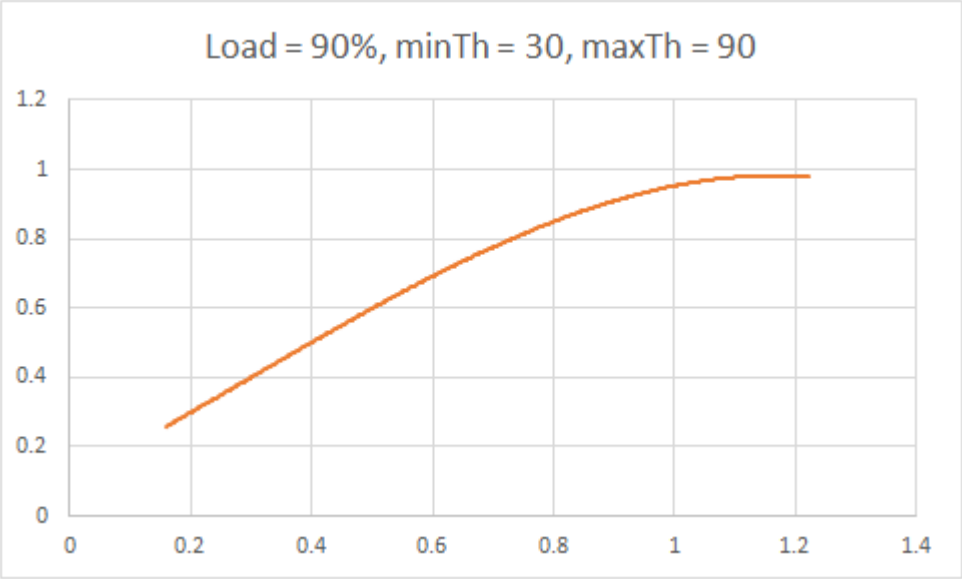
4) Effect of minTh and maxTh on RED:

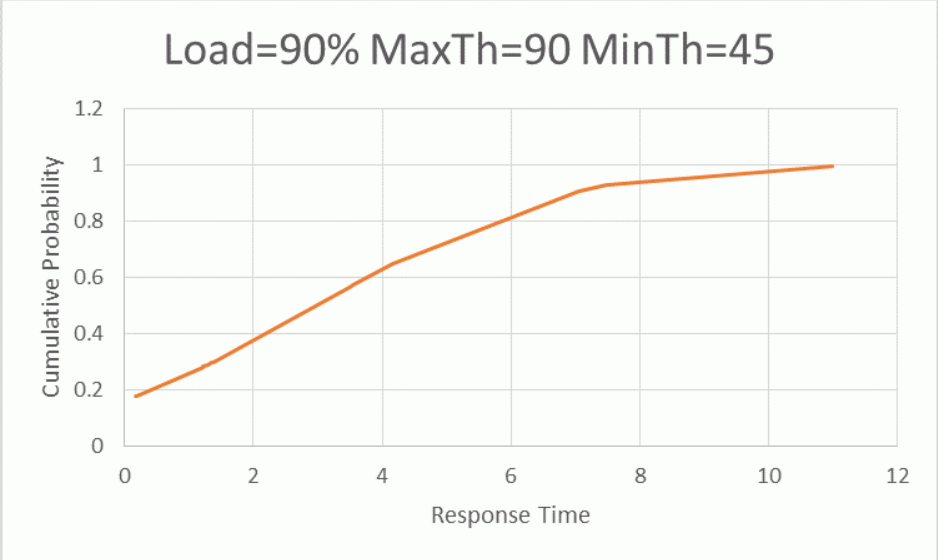




According to Jeffay's paper, the worst response is observed for minTh=5 and performance improves for ranges like (30, 90) and (60, 180). In our experiment we have observed that as delta = (maxTh - minTh) increases, the response time increases. This is inconsistent with Jeffay's findings.

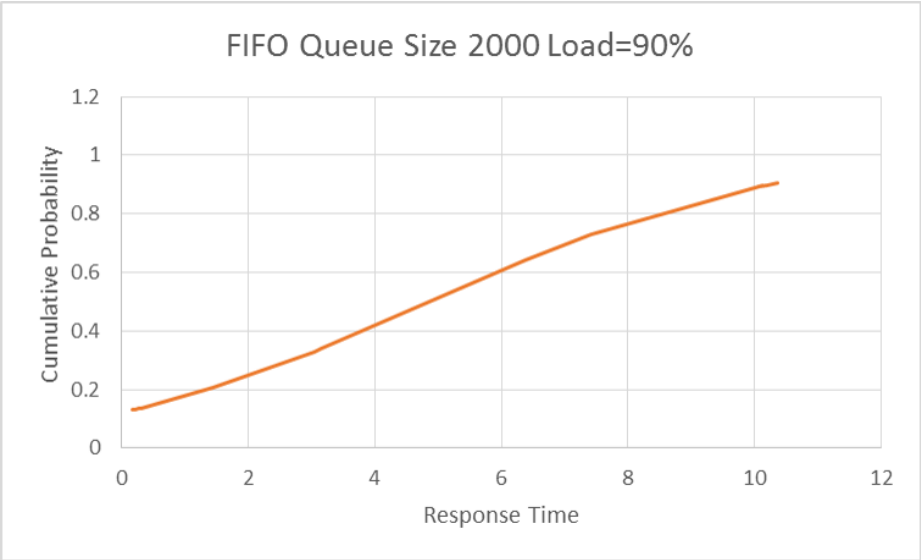
5) Varying only minTh:

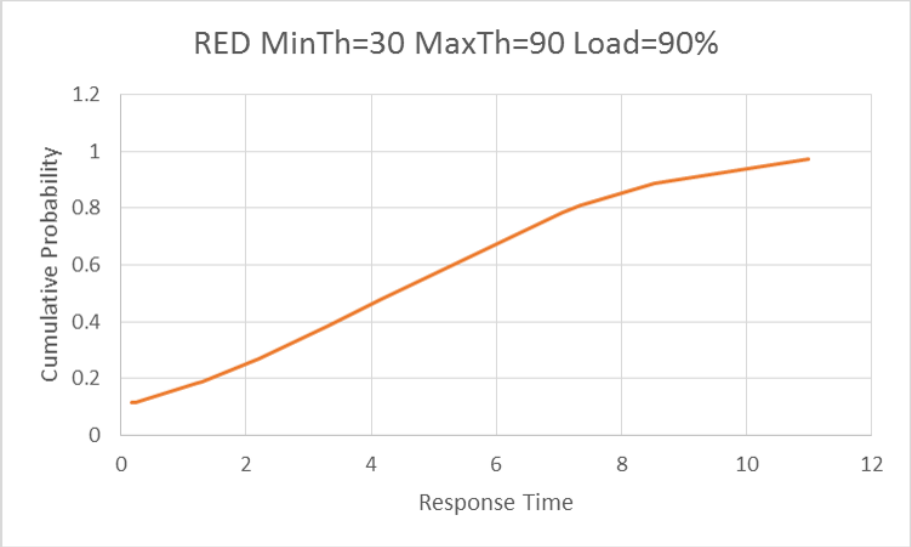




Jeffay observed that varying minTh does not yield marginally different results. In our case we observed that increasing minTh from 30 to 45 while keeping maxTh constant at 90% load also increases the response time significantly.

6) Comparing RED & DropTail:





For the comparison of RED vs. FIFO we have compared the best configurations of RED and FIFO with constant load. We ran a simulation for FIFO with the Queue size as 2000 and RED with a minTh as 30 and maxTh as 90.

It can be inferred in the above graphs that FIFO performs slightly better than RED for the above configurations.

But it should be noted that the performance of RED and FIFO could be similar if RED was carefully tuned (98% load) where response times for shorter responses are improved.

Conclusion:

We were able to match Jeffay's results in case of FIFO performance for different loads, RED performance for different loads but not for RED performance for variations in minTh and maxTh. These differences could be due to the different distribution used by us. Also, the differences may be because the ns3 simulation has conditions different than that of a laboratory or of real world traffic.

We would have liked to use a larger number of browsers for a longer period to get more accurate results.

Lessons Learnt:

1. Implementing a new model requires planning and vision. There are many different ways of implementing the same thing.
2. How web clients and servers work.
3. Understanding and editing an existing open source code without the help of documentation.

References:

[1] Tuning RED for Web traffic - http://users.ece.gatech.edu/~riley/ece6110/handouts/TuningRedLong.pdf
[2] An Empirical Model of HTTP Network Traffic - http://users.ece.gatech.edu/~riley/ece6110/handouts/MahHttp.pdf
[3] http model - https://codereview.appspot.com/4940041
[5] Jeffay's paper on HTTP distribution - http://www.cs.odu.edu/~mweigle/papers/INFOCOM04.pdf
[6] Paper by Floyd and Jacobson: http://users.ece.gatech.edu/~riley/ece6110/handouts/Floyd-Red.pdf
[5] http://en.wikipedia.org/wiki/Random_early_detection