**EE122: Communication Networks**

**Optimized Edge Routing Protocol**

**by Dasun Wang and Thomas Yun[[1]](#footnote-1)**

**Method (Code explanation):**

The program is coded in JAVA and consists of 3 JAVA class files; source.java, destination.java, and OER.java. OER.java contains global variables which the source and destination both use to calculate times pertinent to the optimized edge routing protocol. Source.java contains the code for the source/client including both cases of sending and receiving as well as the main method that starts the whole program. Destination.java contains code for both the queue and destination. The program can be started by running source.java in Eclipse. The parameter for the average rtt (round trip time) variable q can be set inside OER.java.

The main method in source.java initializes a new source and starts the program. The source constructor creates a socket that connects to the localhost on a port number, which must be hard coded by the user, and a call to initialize the destination. The destination constructor creates a socket at the localhost with the port number specified by the user in Destination.java and starts the threads for both the queue and destination listening.

When the program is started, the run method in Source.java is started. The method creates a thread that sends a packet every 1 second (1000 milliseconds) and adds the time sent into a list at index equal to the packet ID. The run method also creates a thread that is constantly trying to receive a packet from the destination and a call to the queue's accept method so that the packet will be added into the queue.

The queue thread, which is running continuously, waits a certain amount of time (calculated in Qn method by random exponential picking) before forwarding to the destination. A new queue time is picked randomly after a packet is forwarded. After the packet leaves the queue, it enters the destination thread (also running continuously). The destination only tries to receive the packet if the queue has forwarded the packet to it. Otherwise, it just loops infinitely through nothing. It then simply sends an ACK packet back to the source, which reads the source port number and source address from the packet. When the source receives the ACK from the destination, it calculates the round trip time by subtracting the time it was sent from the current time. It then uses that to calculate the average round trip time using the algorithm and checks the round trip time against the timeout value which is 2\*avgrtt. All of this is run in 5 different threads which all run continuously and concurrently.

**Results: Average delay (T milliseconds) vs. Queue's servicing rate (Qn milliseconds)**

This graph was constructed by sampling 30 packets' average delays and queue servicing rates. These 30 packets were sent across the same local machine in one second intervals. Ran with A=10.

The graph shows a linear correlation because the average delay on a local machine will mostly be based upon the queue's servicing delay since sending data across a local medium is practically instantaneous and lossless. To prevent the threads from timing out, a 1 millisecond difference is created, which can be observed between the average delay and queue's servicing rate. The outlier shown in the graph can be attributed to JAVA overhead.

**Results: Average delay (T milliseconds) vs. Exponential averaging (A constant)**

This graph was constructed by sampling 30 packets' delays for 5 different A values. The average delay was then calculated by doing a mathematical average of the packet delays.

Since the average delay is linearly correlated with the queue's servicing rate, the average delay will also be between the bounds 0 -> 2A-1. Because the exponential selector chooses randomly between the bounds, the average delay shows a value close to (2A-1)/2. When A increases, the average delay will exponentially increase by 2, which is shown in the graph.

**Results: Queue Dynamics vs. Queue's servicing rate**

In our project, we defined a(t) as the parameter for the exponential averaging. For these graphs, we plotted the queue length as the result of varying queue servicing rates by adjusting the values of the constant A. Here, each queue length is polled for 30 seconds.

These graphs indicate that at A = 10 the queue is able to service all of its packets before a new packet arrives. Also, at A=12 the queue isn't servicing its packets quite fast enough, which leads to a steady queue length over time. Finally, at A=15, the queue length keeps growing because the queue servicing delays are now multiple seconds.

**Conclusion**

These results indicate that the protocol runs ideally on a local machine, since there were never any delays or problems from the connection. Additionally, the average delay increases exponentially as the parameter for exponential averaging increases, and the ideal value for servicing packets while avoiding the most collisions possible occurs when A = 10.

Although this project creates a simple version of the optimized edge routing protocol, it could be improved by implementing the project in another language such as C. This would help prevent any overhead errors and timings caused by JAVA, such as garbage collection. Furthermore, the project could have implemented the retransmission of packets for failed ACK's, this would emphasize students' ability to thread properly.

1. **Contributions:**

   Dasun Wang: Program setup (initialization), and send/receive for source, queue, and destination.

   Thomas Yun: MultiThreading for source, queue, and destination, and queuing algorithm. [↑](#footnote-ref-1)