# Network Equilibria: An Applied Approach

Moya Chen Louis O'Bryan Victor Duan

December 10, 2013

CS 143 Communication Networks
Final Project, Fall Term 2013

# Contents

## 1 Introduction

Networking theory is an important part to the function of the Internet. Congestion control algorithms can heavily influence the speed of a connection.

In this paper, we describe the implementation of a network simulator including its architecture and design choices. We also include graphs of this simulator as run on a series of test cases. Lastly, we give a mathematical analysis for one of the congestion algorithms, as applied to one of the cases.

## 2 Simulator Description

Our network simulator is written in C++.

The network consists of several different types of network classes as well as a Controller to monitor the simulation.

We use event-based simulation, with a global event queue, to advance actions in our program.

For input, we take an XML file as input and use the pugixml C++ XML parser to read the file and create necessary network objects. The program uses these objects to simulate the network until all the flows finish.

The congestion control algorithms we implemented were TCP Reno and TCP Vegas.

As the simulation runs, we collect various statistics. At the end of the simulation, we use gnuplot to create figures based off of collected data.

### 2.1 Design Decisions

We chose to write the simulator in C++ for several reasons. C++ is a powerful and fast object oriented programming language which suited our architecture plan well. We wrote classes for different network objects (Link, Flow, etc.) as well as simulation objects (Controller, CongestionAlgorithm, etc.). By using C++s inheritance, we were able to group objects with similar behavior (ex. Host and Router) and create abstraction hierarchies (ex. Node, Packet). C++ allowed for writing these classes with encapsulation.

Because the network that we are simulating is based on discrete events, we chose to use event-based simulation. Continuous simulation did not make sense as it would be very CPU heavy and take a longer time to run on our computers. Because we knew that we would be spending a fair amount of time running code, we figured that event-based simulation would be the most efficient.

We chose the pugixml parser because it is lightweight and has a simple, open-source library. The lightweightness was particularly attractive because we did not want the simulator to be spending most of its runtime parsing input files. Furthermore, the pugixml library was easy to use. XML was the chosen input format because as a universally recognized format it was easy to make it fit our desired system parameters. Thus, we ended up using an XML DOM type input, as it could easily describe the objects we wanted for the simulation.

We chose two TCP congestion control algorithms: TCP Reno and TCP Vegas. We wanted to choose one loss-based congestion control algorithm (Reno) and one delay-based congestion control algorithm (Vegas). We chose Reno because among the loss-based control algorithms (Reno and Tahoe) because it included more advanced congestion control through the use of fast_recover. Further equations for the analysis are in the notes. Obviously, this section needs some work/deleting. Maybe a paragraph on why we chose gnuplot.

Architecture io.png Input/output structure of our system. Yellow denotes input/output files. Cyan denotes helper libraries. Blue rectangle denotes simulator objects. Blue diamond denotes simulator events. packetTransfer.png Packet transfer process between nodes (hosts, routers) and links. Arrows here represent packet transfer in the system. Arrowhead direction denotes passing of packets between functions. Arrow label denotes time offset added, as perceived by scheduler. (Time offset is zero when not labeled.) hierarchy.png An image of hierarchy of network object classes in our system. Arrows denote inheritance. Boxes inside of boxes denote a single instance of an object of a class being initiated as part of every instance of another class. (Ex. a link always contains 2 buffers)

The network design was split into the following groups of classes. Simulation, Network, and Input. The Simulation group contains the classes Event, Scheduler, and Controller. These classes are used to hold the network objects as well as manage the event-based simulation. The Network group contains the classes

Node, Host, Router, Link, Buffer, Router, RoutingTable, Packet, RoutingPacket, RouterRoutingPacket, HostRoutingPacket, DataPacket, AckPacket, Flow, CongestionAlgorithm, SLOW_START, TCP_TAHOE, TCP_RENO, Vegas, and Cubic. These classes are used to simulate network objects and their behavior. Their exact behavior and hierarchy are described in more detail below. Finally, the Input group of classes contains InputParser, SystemInfo, HostInfo, RouterInfo, LinkInfo, and FlowInfo. The InputParser is used to read in information from the input XML file and that information is used in Controller to create the objects.

Simulation Needs more work about finer details of events/schduler Classes in this category: Controller, Scheduler, Event The Simulation classes function to keep track of high-level system parameters as the simulation runs. The Controller class has a few main tasks. One instance of this class is created upon program start. (For simplicity, we will refer to this one instance of the Controller class as the controller.) It is the Controller to which the InputParser adds system objects. (See Input/Output below.) The controller is what keeps track of system time. It also contains an instance of Scheduler (hence forth called the scheduler). It is also the object that interfaces with network objects when the network objects want to make events. The controller also initializes the first routing update and system print events. In essence, the controller runs the entire simulation. The scheduler contains a priority queue, ordered by event time, which contains all of the systems events. In this scheduler, there is a doNext() function which is called by the controller until the system has finished running of all of its flows. This function enacts the next event in the queue. The Event class contains 3 things: a timestamp for when the event should be executed, a function pointer to the QQTQTQTQTQT It has a Scheduler, inputFile, and all the network objects. The Controller works by getting an input file to use, and initializing the system by using an InputParser instance to parse the file and return the information necessary to create the actual objects. The objects are created, and the network simulation begins. This starts with the initial router updates to get the routing update tables of all the routers. The network is designed to handle static routers and hosts. No changes can be made to the network topography. Thus, this initial routing update runs until the whole network is mapped. Then, while the flows are running, the network events will continue to be executed by the scheduler until there are no more flows. At this point, the plots are made and the simulation is completed.

Network The Network classes are used to hold information of the various network objects. They are split into a few general subgroups: Node, Link, Routing, Packet, and Flow. The Node subgroup consists of Node, Host, and Router. These were designed together because Hosts and Routers share many things in common and can be used interchangeably in some situations, such as the endpoints of a link or the destination of a packet.. As such, Node is an abstract class that contains the common values such as id, and the links it is connected to. Host and Router then inherit from Node and add in their unique attributes such as the flow that a host may be a source of. The Link subgroup consists of Link and Buffer. The Buffer class is used to simulate the buffers on the links. Each link has two buffers, one in each direction. The buffers are implemented as FIFO queues of Packets. The Link class uses the buffers and its own fields to manage the links activities. Specifically, links need to be able to schedule the events of transmitting the packets in its buffers and record stats for every interval. The Routing subgroup consists of Router and RoutingTable. Each router contains its own RoutingTable pointer which describes the behavior it will follow when routing packets. The RoutingTable itself is a map from Node pointers to pairs of weight and Link pointers. The key is the destination Node of the packet to route. It is a Node (as opposed to Host) because while all DataPackets must be sent to Hosts, RoutingPackets can also be sent to Routers. The weights are then the weight of that path and the Link is the next link to follow on that path. In addition to the routing table, the primary function of the Router class is to handle packets that it receives. Data and ack packets are simply forwarded to the next link in the path. Routing packets, on the other hand, are used to update the routers own routing table. This is done in the routing table update steps every routingTableUpdateTime. As far as the actual routing updates are done, we implement the Bellman-Ford algorithm. We chose to use this algorithm because it utilizes local knowledge of the graph as opposed to Dijkstras algorithm, which depends on global knowledge of the network. As such, it seems more natural to use Bellman-Ford for the routing table updates. Dynamic routing is calculated through a combination of buffer delay and propagation delay. Buffer weights are only calculated once per routing update period so that the routing table weights – stored as doubles – stabilize. This behavior mimics real routing table updates because otherwise the routing weights constantly change by slight amounts. The Packet subgroup consists of the large variety of Packet classes. All of these stem from the abstract Packet class. Packet contains core information of the packet such as id, size, source, and destination, and packet type. One level down, we have RoutingPacket, which adds the link the packet came from to use in the routing table updating. Inheriting from RoutingPacket, we have RouterRoutingPacket and HostRoutingPacket. HostRoutingPackets are merely used to tell Routers that the hosts still exist at that link. RouterRoutingPackets contain routing tables to tell the routers whether or not they have to update their routing tables. On the other end of the spectrum, we have DataPacket, which inherits from Packet. Data packets use extra information for the flow it is part of and the start time, which is used for calculating stats such as RTT and packet delay. Finally, we have AckPacket, which inherits from DataPacket. The main differences between a data packet and ack packet are their behavior when being handled by hosts and their size, so it is reasonable to inherit from DataPacket. The Flow subgroup consists of the Flow class as well as all of the congestion control algorithm classes, and is responsible for managing most of the work done in the simulations. The Flow class holds onto the information needed to handle the flow, including size, progress, outstanding packets, acks received, and relevant stat fields. Most importantly, every flow will have its own CongestionAlgorithm pointer. CongestionAlgorithm is the abstract base class used to simulate the congestion control algorithms. It holds onto fields necessary for congestion control such as window size and ssthresh. We then have several different congestion control algorithms that implement various types of TCP congestion control. The simplest of these is SLOW_START, which inherits from CongestionAlgorithm. SLOW_START simply implements slow start for that part of TCP, and then uses very basic congestion avoidance. While not in slow start, it increments cwnd by 1/cwnd every ack. When packets time out, it sets ssthresh to half cwnd and sets cwnd to 1. Next, we have TCP_TAHOE, which inherits from SLOW_START and, as expected, implements TCP_TAHOE congestion control. After this, we have TCP_RENO, which then inherits from TCP_TAHOE, and thus only needs to add fast_recovery. We also have Vegas, which inherits from TCP_RENO. This is useful because the Vegas algorithm is essentially the Reno algorithm with a modified congestion avoidance algorithm. Finally, we have Cubic, which inherits from SLOW_START. While not complete, it is a partial implementation of TCP Cubic, which was an interesting extra TCP algorithm we were interested in exploring.

### 2.2 InputParser

The InputParser class uses the pugixml C++ XML parsing ability to parse input files and store the information in lists of the various Info objects. One thing to note is that pugixml does not provide any method of checking the XML file against an XSD schema. As such, the expected input and some behavior is described in much greater detail in the networkSchema.txt file in the InputParser directory. The SystemInfo class is used as the parent class for the other Info classes. All it holds is the printing flag for the object. The other Info classes each contain values specific to the type of object it is designed to hold information of. For example, LinkInfo will hold members such as linkRate. Again, these are explained in more detail in the networkSchema.txt file. If the input file is formatted correctly, the InputParser can call its run function, which takes in references to the values it will set, to parse the file and set the fields. The fields it sets are snapshotTime, routingUpdateTime, hosts, routers, links, flows, and plotOptions. snapshotTime is the time interval between stat-collecting events. The smaller this is, the finer the resolution. routingUpdateTime is the time interval between updating the routing tables. hosts, routers, links, and flows are lists of their corresponding Info objects that will be used to create the objects later. plotOptions does stuff.

* Describe what you see for each test case and congestion control algorithm. Include time traces and commentary.

## 3 Mathematical Analysis

* Test Case 2 can be solved analytically. Set up the TCP equations that TCP Vegas / FAST-TCP are engineered to solve, and then find (and justify!) the equilibrium point for each phase of Test Case 2's operation. Compare your calculated results to your experimental results; if they are wildly different, discuss why this might be so.

In this section, we provide a mathematical analysis of TCP Vegas as applied to testcase 2 of our system.

By example 9 of chapter 3 (3.9), the equilibrium solution to Vegas is the same as the optimal minimization of $\sum_{i=0}^{n} \alpha_i \log(x_i)$ subject to the link capacity constraints $Rx <= c$. In the first time interval, there is only one flow, so this problem becomes

$$\frac{\alpha_1}{x_1} = \mu_1 + \mu_2 + \mu_3$$
$$\mu_1(x_1 - 10) = 0$$
$$\mu_2(x_1 - 10) = 0$$
$$\mu_3(x_1 - 10) = 0$$

Solving these equations gives the solutions $x_1 = 10$Mbps, and $\mu_1 = 0.055, \mu_2 = 0, \mu_3 = 0$ (although these may be chosen arbitrarily to satisfy nonnegativity and the top equation). This rate matches the realized rate of flow 1 during simulation after the initial slow start.

The propogation delay $d_1$ is 100ms because each of the 5 links along the path from S1 to T1 has the same delay of 10ms. The base RTT is approximately 110ms because of the queueing delay. By Little's law, the equilibrium queueing delay is $\alpha/x_1$, where in our simulation, $\alpha = 0.55 packets/ms$. This is equal to $\frac{0.55 packets/ms}{10Mbps} * \frac{1024B}{packet} * \frac{Mb}{10^6 b} * \frac{8b}{B} * \frac{10^3 ms}{s} * (110ms) = 50ms$. So the equilibrium RTT is 150 ms. This calculation also agrees with the round trip time in our simulation. The corresponding window size is $150ms * 10Mbps = 183 packets$.

When flow 2 enters, its base RTT includes the queueing delay at link 1. The queueing delay is approximately 50 ms by the above, assuming that the queueing delay is the same as the previous equilibrium queueing delay. In addition, the propogation delay is approximately 66 ms. The KKT conditions to the minimization become

$$\frac{\alpha_1}{x_1} = \mu_1 + \mu_2 + \mu_3$$
$$\frac{\alpha_2}{x_2} = \mu_1$$
$$\mu_1(x_1 + x_2 - 10) = 0$$
$$\mu_2(x_1 - 10) = 0$$
$$\mu_3(x_1 - 10) = 0$$

Solving these equations gives the solutions $x_1 = 5$Mbps, $x_2 = 5$ Mbps, and $\mu_1 = 0.110, \mu_2 = 0, \mu_3 = 0$. The simulation's flow rates of flow 1 and flow 2 converge toward this common value in the period before flow 3 is added, although they do not quite reach it before that time.

With the same base RTT for flow 1, the new equilibrium queueing delay is $\frac{0.55 packets/ms}{5Mbps} * \frac{1024B}{packet} * \frac{Mb}{10^6 b} * \frac{8b}{B} * \frac{10^3 ms}{s} * (110ms) = 100ms$. The new RTT of flow 1 is 200 ms, which agrees with the simulation's RTT. The equilibrium queueing delay for flow 2 is $\frac{0.55 packets/ms}{5Mbps} * \frac{1024B}{packet} * \frac{Mb}{10^6 b} * \frac{8b}{B} * \frac{10^3 ms}{s} * (66ms + 50ms) = 105ms$ since the base RTT is $66ms + 50ms$. So the equilibrium RTT for flow 2 is $105ms + 66ms = 171ms$. This RTT does not match the simulation exactly, which produces round trip times of about 150 ms, but it is within a reasonable margin of error. The corresponding window sizes for flows 1 and 2 are 122 packets and 104 packets. In turn, these measurements slightly differ from the simulation's values.

When flow 3 enters, its base RTT only depends on the propogation delay to T3 since the buffer at L3 is empty. So its base RTT is 66 ms, and its propogation delay is 60 ms. The KKT conditions become

$$\frac{\alpha_1}{x_1} = \mu_1 + \mu_2 + \mu_3$$
$$\frac{\alpha_2}{x_2} = \mu_1$$
$$\frac{\alpha_3}{x_3} = \mu_3$$
$$\mu_1(x_1 + x_2 - 10) = 0$$
$$\mu_2(x_1 - 10) = 0$$

$$\mu_3(x_1 + x_3 - 10) = 0$$

The solution to these equations is $x_1 = 3.33$ Mbps, $x_2 = 6.67$ Mbps, $x_3 = 6.67$ Mbps, and $\mu_1 = .0825, \mu_2 = 0, \mu_3 = .0825$. These rates approximately agree with the simulation.

The queueing delays are

$$q_1 = \frac{0.55 packets/ms}{3.33 Mbps} \frac{1024B}{packet} \frac{Mb}{10^6 b} \frac{8b}{B} \frac{10^3 ms}{s} (110ms) = 150ms$$

$$q_2 = \frac{0.55 packets/ms}{6.67 Mbps} \frac{1024B}{packet} \frac{Mb}{10^6 b} \frac{8b}{B} \frac{10^3 ms}{s} (66ms + 50ms) = 78ms$$

$$q_3 = \frac{0.55 packets/ms}{6.67 Mbps} \frac{1024B}{packet} \frac{Mb}{10^6 b} \frac{8b}{B} \frac{10^3 ms}{s} (66ms) = 45ms$$

and the corresponding window sizes are

$$w_1 = 102 packets$$

$$w_2 = 117 packets$$

$$w_3 = 90 packets$$

These correspond to RTT's of $T_1 = 250$ ms, $T_2 = 144$ ms, and $T_3 = 111$ ms. These results agree approximately with the simulation's results, however the simulation shows round trip times slightly lower. The windows sizes also approximately agree, however, they seem to converge to slightly lower results as well.

When flow 2 finishes, the new equilibrium equations are analogous to those given above for the case when only flow 1 and flow 2 are in the system; $x_1 = 5$ Mbps and $x_3 = 5$ Mbps, and $\mu_1 = 0, \mu_2 = 0, \mu_3 = .110$. The new queueing delays are

$$q_1 = \frac{0.55 packets/ms}{5 Mbps} \frac{1024B}{packet} \frac{Mb}{10^6 b} \frac{8b}{B} \frac{10^3 ms}{s} (110ms) = 100ms$$

$$q_1 = \frac{0.55 packets/ms}{5 Mbps} \frac{1024B}{packet} \frac{Mb}{10^6 b} \frac{8b}{B} \frac{10^3 ms}{s} (66ms) = 60ms$$

and the window sizes are

$$w_1 = 122 packets$$

$$w_2 = 77 packets$$

The new RTT's are $T_1 = 200$ ms and $T_2 = 126$ ms. Again, these do not exactly match the simulation's results, however they are not too far away from the results, which were approximately 225 ms and 140 ms. The simulation window sizes were about 135 packets for flow 1 and 60 packets for flow 2.

Finally, when flow 3 is the only flow left, the KKT equations become

$$\frac{\alpha_3}{x_3} = \mu_3$$

$$\mu_3(x_3 - 10) = 0$$

The solution is $x_3 = 10$ Mbps and $\mu_1 = 0, \mu_2 = 0, \mu_3 = .055$. The new queueing delay is

$$q_3 = \frac{0.55 packets/ms}{10 Mbps} \frac{1024B}{packet} \frac{Mb}{10^6 b} \frac{8b}{B} \frac{10^3 ms}{s} (66ms) = 30ms$$

and the new window size is

$$w_3 = 117 packets$$

* Talk about the project process and division of labor. What worked well? What would you have done differently?

WORKFLOW

weekly meetings in person debugin in person more productive when together and talking about stuff rather than separately

not so well focusing too much on one aspect at a time tendency to not test code not sure if things worked thanksgiving killed us

dol even for the most part better deadlines

version control lots of merge conflicts no branches useful for backtracking problematic as new git users

small group -¿ didnt benefit as much from ticketing/issue tracking