GENESIS support for ns and SSFNet

Anand Sastry
Department of Computer Science, RPI, Tory, NY 12180

February 8, 2003

Abstract

The complexity and dynamics of the Internet is driving the demand for scalable and efficient network simulation. Yet, parallelizing network simulation at packet level does not work efficiently and therefore do not scale to large number of processors because of tight synchronization between network components. To overcome this problem we designed a method in which a large network is decomposed into parts and each part is simulated independently and concurrently with the others. These parts exchange information periodically about the packet delays and losses along the paths within each part. Each part iterates over the selected simulated time interval until the exchanged information changes less than the prescribed tolerance.

Each decomposed part may represent a subnet or a subdomain of the entire network, thereby mirroring the network structure in the simulation design. The proposed method is independent of the specific simulator technique employed to run simulators of the parts of the decomposed network. Hence, it is a general method for efficient parallelization of network simulation based on convergence to the fixed point solution of inter-part traffic. The method can be used in all applications in which the speed of the simulation is of essence, such as: on-line network simulation, network management, ad-hoc network design, emergency network planning, large network simulation or network protocol verification under extreme conditions (large flows).

The paper describes this method we call Genesis(General Network Simulation Integration System), its implementation based on ns and SSFNet simulator, and its performance for sample communication networks. We also describe how Genesis has been ported to parallelize BGP (Border Gateway Protocol), with a goal to provide a novel outbound load-balancing technique using BGP LOCAL_PREF settings and aided by online simulation

1 Introduction

The major difficulty in simulating large networks at the packet level is the enormous computational power needed to execute all events that packets undergo in the network [12]. The usual approach to providing required vast computational resources relies on parallelization of an application to take advantage of a large number of processors running concurrently. Such parallelization does not work efficiently for network simulations at packet level because of tight synchronization between network components [11]. To overcome this difficulty, we designed a method described in this paper, in which a large network is decomposed into parts and each part is simulated independently and simultaneously with the others. Each part represents a subnet or a subdomain of the entire network. These parts are connected to each other through edges that represent communication links existing in the simulated network. In addition, we partition the total simulation time into separate simulation time intervals selected in such a way that the traffic characteristics change little during each time interval.

In the initial (zero) iteration of the simulation process, each part assumes on its external in-links either no traffic, if this the the first simulated interval (alternatively, the initial external traffic may be defined by the real-time measurements of the simulated network), or the traffic defined by the packet delays and drop rate defined in the previous simulation time interval for external domains. Then, each part simulates its internal traffic, and computes the resulting outflow of packets through its out-links.

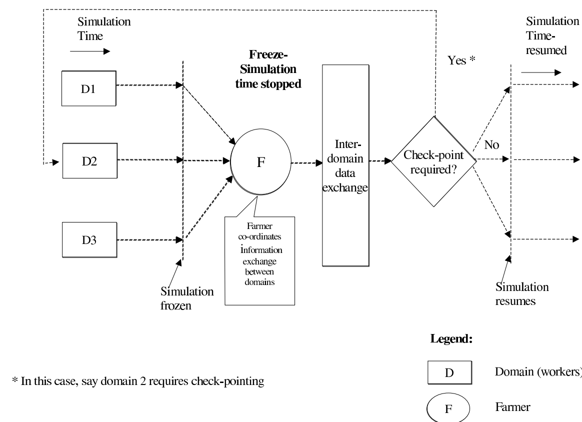


Figure 1: Progress of the Simulation Execution

In the subsequent $k > 0$ iteration, the inflow into each part from the other parts will be generated based on the outflows measured by each part in the iteration $k - 1$. Once the inflows to each part in iteration k are close enough to their counterparts in the iteration $k - 1$, the iteration stops and the simulation either progresses to the next simulation time interval or completes execution and produces the final results (see Figure 1).

More formally, consider a network $\Gamma = (N, L)$, where N is a set of nodes and L (a subset of Cartesian product $N \times N$), is a set of unidirectional links connecting them (bidirectional links are simply represented as a pair of unidirectional links). Let $(N_1, ..., N_q)$ be a disjoint partitioning of the nodes, each partition modeled by a simulator. For each subset N_i, we can define a set of external out-links as $O_i = L\&N_i \times (N - N_i)$, in-links as $I_i = L\&(N - N_i) \times N_i$, and local links as $L_i = L\&N_i \times N_i$.

The purpose of a simulator S_i, that models partition N_i of the network, is to characterize traffic on the links in its partition in terms of a few parameters changing slowly compared to the simulation time interval. In the implementation presented in this paper, we characterize each traffic as an aggregation of the flows, and each flow is represented by the activity of its source and the packet delays and losses on the path from its source to the boundary of that part. Since the dynamics of the source can be faithfully represented by the copy of the source replicated to the boundary, the traffic is characterized by the packet delays and losses on the relevant paths. Thanks to queuing at the routers and the aggregated effect of many flows on the size of the queues, the path delays and packet drop rates change more slowly than the traffic itself.

It should be noted that we are also experimenting with the direct method of representing the traffic on the external links as a self-similar traffic defined by a few parameters. These parameters can be used to generate the equivalent traffic using on-line traffic generator described in [17]. No matter which characterization is chosen, based on such characterization, the simulator can find the overall characterization of the traffic through the nodes of its subnet. Let $\xi_k(M)$ be a vector of traffic characterization of the links in set M in k-th iteration. Each simulator can be thought of as defining a pair of functions:

$$\xi_k(O_i) = f_i(\xi_{k-1}(I_i)),\ \xi_k(L_i) = g_i(\xi_{k-1}(I_i))$$

(or, symmetrically, $\xi_k(I_i),\ \xi_k(L_i)$ can be defined in terms of $\xi_{k-1}(O_i)$).

Each simulator can then be run independently of others, using the measured or predicted values of $\xi_k(I_i)$ to compute its traffic. However, when the simulators are

linked together, then of course $\bigcup_{i=1}^{q} \xi_k(I_i) = \bigcup_{i=1}^{q} \xi_k(O_i) = \bigcup_{i=1}^{q} f_i(\xi_{k-1}(I_i))$, so the global traffic characterization and its flow is defined by the fixed point solution of the equation.

$$\bigcup_{i=1}^{q} \xi_k(I_i) = F(\bigcup_{i=1}^{q}(\xi_{k-1}(I_i)), \tag{1}$$

where $F(\bigcup_{i=1}^{q}(\xi_{k-1}(I_i))$ is defined as $\bigcup_{i=1}^{q} f_i(\xi_{k-1}(I_i))$. The solution can be found iteratively starting with some initial vector $\xi_0(I_i)$, which can be found by measuring the current traffic in the network.

We believe that communication networks simulated that way will converge thanks to monotonicity of the path delay and packet drop probabilities as the function of the traffic intensity (congestion). For example, if in an iteration k a part N_i of the network receives more packets than the fixed point solution would deliver, then this part will produce fewer packets than the fixed point solution would. These packets will create inflows in the iteration $k+1$. Clearly then, the fixed point solution will deliver the number of packets that is bounded from above and below by the numbers of packets generated in two subsequent iterations I_k and I_{k+1}. Hence, in general, iterations will produce alternately too few and too many packets in the inflows providing the bounds for the number of packets in the fixed point solution. By selecting the middle of each bound, the number of steps needed to convergence can be limited to the order of logarithm of the needed accuracy, so convergence is expected to be fast. In the initial implementations of the method, the convergence for UDP traffic and small networks was achieved in 2 to 3 iterations.

It should be noted that the similar method has been used for implementation of the flow of imports-exports between countries in the project Link [14] led by the economics Noble Laureate, Lawrence Klein. The implementation [15] included distributed network of processors located in each simulated country and it used global convergence criteria for termination [16].

One issue of great importance for efficiency of the described method is frequency of synchronization between simulators of parts of the decompose network. Shorter synchronization time limits parallelism but decreases also the number of iterations necessary for convergence to the solution because changes to the path delays are smaller. Variance of the path delay of each flow can be used to adaptively define the time of the synchronization for the subsequent iteration or the simulation step.

The efficiency of our approach is based on the following property of network sim-

ulation:

The simulation time of a network grows faster than linearly with the size of the network.

Theoretical analysis indicates that for the network size of order $O(n)$, the simulation time contains terms which are of order $O(n * \log(n))$, that correspond to sorting event queue, of order $O(n^2)$, that result from packet routing, and even of order $O(n^3)$, that are incurred while building routing tables. Some of our measurements [13] indicate that the dominant term is of order $O(n^2)$ even for small networks. Using the least squared method to fit the measurements of execution time for the different network sizes, we got the following approximate formula for star-interconnected networks:

$$T(n) = 3.49 + 0.8174 \times n + 0.0046 \times n^2 \qquad (2)$$

where T is the execution time of the simulation, and n is the number of nodes in the simulation. From the above, we can see that the execution time of a network simulation may hold a quadratic relationship with the network size. Therefore, it is possible to speed up the network simulation more than linearly by splitting a large simulation into smaller pieces and parallelizing the execution of these pieces.

As we demonstrate later in the measurement section, a network decomposed into 16 parts will require less than 1/16 of the time of the entire sequential network simulation (so also less computational power, because there are 16 parts each needing less than 1/16 of the computational power of the sequential simulator), despite the overhead introduced by external network traffic sources added to each part (as explained below) and synchronization and exchange of data between parts. Hence, with modest number of iterations the total execution time can be cut an order of magnitude or more.

Another advantage of the proposed method is that it is independent of the specific simulator technique employed to run simulators of the parts of the decomposed network. Rather, it is a scheme for efficient parallelization based on convergence to the fixed point solution of inter-part traffic which is measured by a set of parameters necessary to characterize this traffic rather then flow of packets. Our primary application is the use of the on-line simulation for network management [13] to which the presented method fits very well and can be combined with the on-line network monitoring. The simulations in this application predicts changes in the network performance caused by tuning network parameters. Hence, the fixed point

solution found by our method is with high probability the point into which the real network will evolve. However, this is a still an open issue under what conditions we can guarantee that the fixed point solution is unique, and if it is not, when the solution found by the method is the same as the point that the real network reaches.

The method can be used in all applications in which the speed of the simulation is of essence, such as:

- on-line network simulation,
- ad-hoc network design,
- emergency network planning,
- large network simulation,
- network protocol verification under extreme conditions (large flows).

2 Implementation

Our current simulation platform is the *ns* network simulator [3]. A simulations is defined in ns by Tcl scripts which can also be used to interface the core of the simulator. The kernel of the simulation system is written in C++. The ease of adding extensions and rich suite of the network protocols made ns a popular and common, albeit not too efficient, platform for research in networking. Hence, we believe that implementing our method within ns will enable others to experiment with our system.

Our extensions to ns enable collaboration among individual parts into which the simulated network is divided. Since network domains are convenient granules for such partitioning, we will refer to these parts as *simulations domains* or *domains* in short. Each domain is simulated by a separate copy of ns running on a unique processor. The implementation specifics are described in the sections below.

2.1 New Features Added to ns

To accomplish per processor based domain simulation the following extension were added to ns.

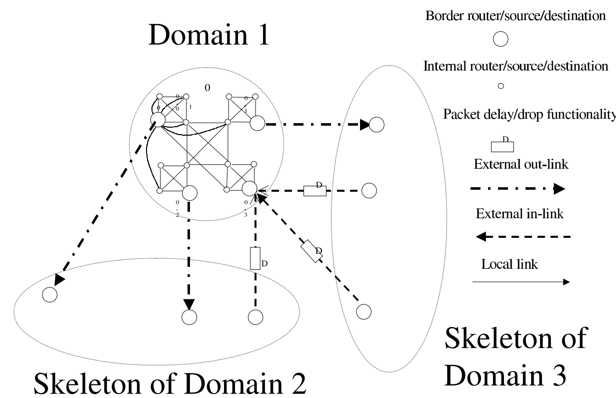


Figure 2: Active Domain with Connections to Other Domains

- The ability to suspend the simulation to enable exchange of data on path delays using message passing between processors simulating individual domains. During the simulation freeze, each individual simulation domain exchanges information on packets generated and dropped along links leaving the domain (cf. Figure 2).

  The network in Figure 2 is split into three individual domains, named 1, 2 and 3. Each of the domain simulations runs concurrently with the others and they exchange information about the path delays incurred by packets leaving the domain. The interval for exchange of this information is user configurable (in the Tcl script). For example, each domain may run its individual simulations for one second from n-th to $n+1$-st second of the simulation time, and pause thereafter, Then, information about delays of packets leaving the domain during this time interval is passed onto the target domain to which these packets are directed. If these delays differ significantly from what was assumed in the target domain, the simulation of the time interval $(n, n+1)$ is repeated. Otherwise, the simulation progresses to the time interval $(n+1, n+2)$. The threshold value of the difference between the current delays and the previous ones under which the simulation is allowed to progress in time it is set by the user. This threshold impacts the speed of the simulation progress and defines the precision of the simulation results.

  New event for the ns scheduler, *Freeze* is defined generically. It pauses the simulation at intervals defined by the user. During the event execution, it executes functions provided by the user in Freeze definition. On return, Freeze

reactivates the simulation.

- The ability to record information about the delays and drop rate experienced by the packets leaving the domain. Each delay measures the time expired from the instance a packet leaves its source to the time it reaches the domain boundary. Packet drop rates are computed for each flow separately. Also recorded is information about each packet source and its intended destination. Having this information enables us to replicate the source from the original domain to the boundary of the target domain (sources in skeletons of domains 2 and 3 in Figure 2) and postpone an arrival of each packet produced by the replicated source at the domain boundary by the delay measured in the source (and transient, if necessary) domains. Also, with probability defined by packet drop rates, packets are randomly dropped during the passage to the boundary of the destination domain (D boxes in Figure 2).

- The ability to define domain members and identify individual sources within the domain that generate packets intended for nodes external to the domain. This feature enables us to directly connect a source to the destination domain to which it sends packets. We refer to such replicated source as a *fake source* and to the link that connects it to the domain internal nodes as a *fake link*, as explained below. The domain is defined by the user using a Tcl level command which takes as its parameters the nodes that the user marks as belonging to the domain. Then, the simulation of this domain is created by deactivating all domains external to the selected domain.

2.2 Details of modifications to ns

2.2.1 Domain definition: Domain is a Tcl-level scripting command that is used to define the nodes which are part of the domain for the current simulation. In the first iteration of the simulation the traffic sources outside the domain are inactive. The traffic generated within the domain is recorded and the statistics calculated. In the following iterations, the sources active within other domains with a link to the domain in question are activated.

When a domain declaration is made in the Tcl script, the nodes defined as a parameter to this command are stored in the form of a list. Each time a new domain is defined, the new node list is added to a domain list (a list of lists). The user selected domain is made active. Any link with one end connected to a node in this domain and the other end connected to a node in another domain is defined as a

cut-link. All packets sent on these links are collected for their delay and drop rate computation.

Source generators connected to sources outside the active domain are deactivated. This is done by a new Tcl script statement that attaches an inactive status to nodes outside the active domain (cf. 2.2.3. Traffic Generator description below).

2.2.2 Connector: The connector performs the function of receiving, processing and then delivering the packets to the neighboring node or dropping the packets. A modification has been made to this connector class which now has the added functionality of filtering out packets destined for the nodes outside the domain and storing them for statistical data calculation.

A connector object is generally associated with a link. When a link is set up, the simulator checks if this link connects nodes in different domains. If this is the case, this link is classified as a cross-link and the connector associated with this link is modified to record packets flowing across it. Each such packet is either forwarded to the neighboring node or is marked as leaving the domain based on its destination.

2.2.3 Traffic Generator: *TrafficGenerator Class* is used to generate traffic flows according to a timer. This class is modified, so that for the domain simulation, the traffic sources can be activated or deactivated. Initially, at the start of the simulation, the traffic generator suppresses nodes outside the domain from generating any traffic.

2.2.4 Fake Link: Fake links are used to connect the fake sources to a particular cross-link on the border of the destination domain. When a fake traffic source is connected to a domain by a fake link, the packets generated by this source are sent into the domain via the fake link and not the regular links which are set up by the user network configuration file. The fake link adds a delay and, with certain probability, drops the packet to simulate packet's behavior during passage through the regular route. With the fake traffic sources and fake links, the statistical data from the simulation of another domain are collected, and the traffic to the destination domain is regenerated.

When a fake link is built, the source connector and the destination connector must be specified. A fake link shortens the route between the two connector objects. Each connector is identified by the nodes on both ends of it. Link connectors are managed in the border object as a link list. The flow id to build up a fake link is specified, one fake link is used for one flow.

Fake link is used to simulate a particular flow, so when the features (packet delay and drop rate) of this flow change, the fake link object needs to be updated. After updating the parameters of the fake link object, the performance of the corresponding fake link changes immediately. Fake links themselves are managed in the border object as a link list.

2.2.5 Connectors with Fake Targets: In the original version of ns, connectors are defined as *an NsObject with only a single neighbor*. But our new ns simulation required this definition to be changed to build fake links to shortcut the routes for different packet flows. These fake links are set up according to the network traffic flows and each flow from the fake sources will need a separate fake link. The flows that go through one source connector may reach different cross-link connectors at the destination border, so there will be fake links connecting this connector to some different connectors. Different flows going into one connector are sent to different fake links, which are defined as fake targets here. Thus, the connector could now be defined as *an NsObject with one neighbor and a list of fake targets*. When the fake connection is enabled in a connector, this connector would have a list of fake links (fake targets), and would classify the incoming packets by flow id and send them to the correct destinations.

The connector class will maintain a list of fake targets. Once a new fake link is set up from this connector, it will be added to this connector's fake target list (this is done by the shortcut method defined in the Border class).

2.2.6 Border: Border is a new class added to the ns. It is the most important class in the domain simulation. A border object represents the active domain in the current simulation. The main functionality of the border class includes:

- Initializing the current domain: setting up the current domain id, assigning nodes to different domains, setting up the date exchange etc.
- Collecting and maintaining information about the simulation objects, such as a list of traffic source objects, a list of the connector objects and a list of the fake link objects maintained by the border object.
- Implementing and controlling the fake traffic sources: setting up and updating fake links, etc.

The border object is set up first, and its reference its made available to all objects in the simulation. A lot of other ns classes need to refer to the variables and methods

in the border object. The border class has an array which for each simulation object stores the domain name to which this object belong. This information is collected from domain description files that are created by the domain object implementation. The names are created for the files assigned to each domain to store some persistent data needed for inter-domain data exchange and restoration of the state from the checkpoint.

All traffic source objects created in the simulation are stored. These traffic sources can be deactivated or activated using the flow id. All the connector objects created in the simulation are stored. These connectors are identified by the two nodes to which they are connected. The connector information is used to create fake links.

The traffic sources outside the current active domain are deactivated while setting up the network and domains. When a fake link is set up for a flow, the traffic source of this flow will be reactivated. The border class searches the traffic source list to find the object, and calls the reactivate() method of the matching source object to reactivate this flow.

When the border receives flow information from other domains, it will set up a fake link for this flow, and initialize the parameter of the fake link using the received statistical data. When setting up a fake link, it goes through the connector list to find the source and the destination of the connector objects, and then shortcuts the route between them by adding a fake target into the source connector. All the created fake link objects are stored in the border as a linked list ready for further update.

2.2.7 Checkpointing: This feature has been included in ns to enable the simulation to easily rerun over the same simulation time interval. We use diskless checkpointing, in which each client process creates a child when it leaves a freeze point. The child is suspended, but preserves a state of the parent at the freeze time. The parent proceed to the next freeze point. Once there, the parent decides whether to return to the previous state, in which case it unfreezes the child and then kills itself, or to continue the simulation to the next time interval, in which case the suspended child is killed. This method is efficient because the process memory is not duplicated initially; later only pages that become different for the parent and child are duplicated during execution of the parent. The only significant cost is the execution of fork statement creating a child, which however is several orders of magnitude smaller than saving state to disk. More details have been provided in a separate section later.

2.2.8 Synchronizing Individual Domain Simulations: Individual domain simu-

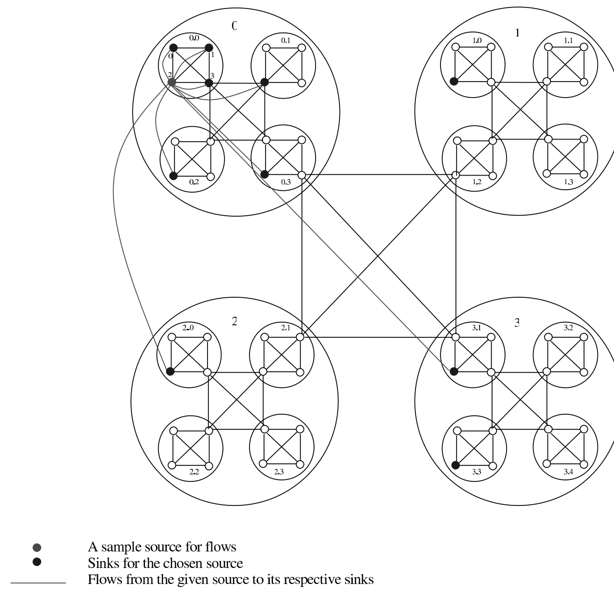


Figure 3: 64-node configuration showing flows from a sample node to all other nodes in a network

lations are distributed across multiple processors using a client-server architecture. Multiple clients connect to a single server that handles the message passing between them. The server is defined as a single process to avoid the overhead of dealing with multiple threads and/or processes. The server uses two maps (data structures). One map keeps track of the number of clients that have already supplied the delay data for the destination domain. The other map is toggled by clients that need to perform checkpointing. All messages to the server are preceded by *Message Identification Parameters* which identify the state of the client. A decision whether to checkpoint the current state or to restore the saved state is made by the client based on the comparison of packet delays and drop rates in two subsequent iterations.

A client indicates to the server whether it requires checkpointing in the contents of the message itself. A client which has to checkpoint causes all other clients to block until it has resent the data to the server and the server has delivered it to the destination domain (in other words a domain on another machine). This is achieved by exchanging the maps at the end of each iteration during the simulation freeze.

The steps of collaboration of simulators and the server are shown in Figure 1.

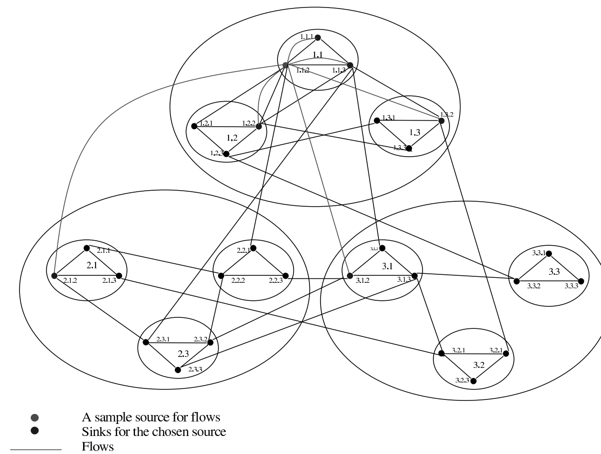


Figure 4: 27-node configuration and the flows from the sample node

3 Performance

We use two sample network configurations, one with 64 and the other with 27 nodes to test the performance of our simulation method. Both of these networks are divided into classes of domains. The rate at which sources generate traffic are varied to generate temporal congestion in the network, especially at the nodes at the borders of the domains. All sources produce Constant Bit Rate (CBR) traffic with constant packet size of 64 bytes.

The 64-node network is designed with a great deal of symmetry. The smallest domain size is four nodes; there is full connectivity between these nodes. Four such domains together are considered as a larger domain in which there is full connectivity between the four sub-domains. Finally, four large domains are fully connected and form the entire network configuration (cf. Figure 3).

The 27-node network is a PINNI network [10] with a hierarchical structure. Its smallest domain is composed of three nodes. Three such domains form a larger domain and three large domains form the entire network (cf. Figure 4).

3.1 64-node network

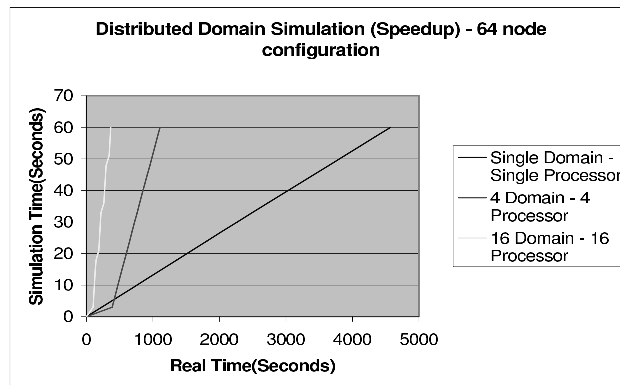


Figure 5: Simulation times for the 64-node network decomposed into domains of different sizes

Each node in the network is identified by three digits $x.y.z$, where $0 \leq x, y, z \leq 3$, that identify domain, subdomain and node rank within the subdomain to which the node belongs.

Each node has nine flows originating from it. In addition, each node also acts as a sink to nine flows. The flows from a node $x.y.z$ go to nodes:

$x.y.(z+1)\%4$ $x.y.(z+2)\%4$ $x.y.(z+3)\%4$
$x.(y+1)\%4.z$ $x.(y+2)\%4.z$ $x.(y+3)\%4.z$
$(x+1)\%4.y.z$ $(x+2)\%4.y.z$ $(x+3)\%4.y.z$

Thus, this configuration forms a hierarchical and symmetrical structure on which the simulation is tested for scalability and speedup.

In a set of measurements, the sources at the borders of domains produce packets at the rate of 20000 packets/sec for half of the simulation time. The bandwidth of the link is 1.5Mbps. Thus, certain links are definitely congested and congestion may spread to some other links as well. For the other half of the simulation time, these sources produce 1000 packets per second. Since such flows require less bandwidth than provided by the links connected to each source, congestion is not an issue. All other sources produce packets at the rate of 100 packets/sec for the entire simulation. For these measurements we defined sources that produced only CBR traffic and the speedup was measured by comparing simulation times of domains to the simulation time of the entire network (excluding synchronization time).

We measured speed up for this configuration over simulation of 60 seconds of traffic. The simulation interval was set at 14.9999 seconds, resulting in five freezes. The simulation speedup with 16 domains (each with size of four nodes) was approximately 18, as shown in Table 1 and Figure 5. The decomposed simulation required at most two iterations to converge to the solution in each simulation time interval. Despite repetitive simulations over some of the intervals, the decomposed simulations achieved superlinear speedup. The differences in the total number of packets in each flow, the number of dropped packets and the sizes of the queues at the routers were well below 1% for all three different domain sizes.

3.2 27-node configuration

The network configuration shown in Figure 4, the PINNI network adopted from [10], consists of 27 nodes arranged into 3 different levels of domains containing three, nine and 27 nodes, respectively. Each node has six flows to other nodes in the configuration and is receiving six flows from other nodes. The flows from a node $x.y.z$ can be expressed as:

$x.y.(z+1)\%3$ $x.y.(z+2)\%3$
$x.(y+1)\%3.z$ $x.(y+2)\%3.z$
$(x+1)\%3.y.z$ $(x+2)\%3.y.z$

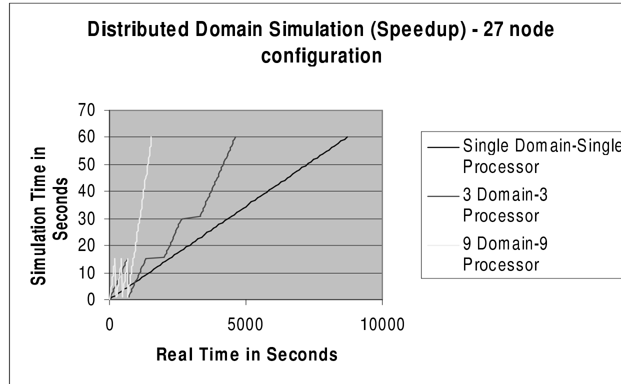


Figure 6: Simulation times for 27-node network decomposed into domains of different sizes

size of domain	27-nodes	64-nodes
large = 1 proc/domain	3885.5	1714.5
medium = 3(4) procs/domains	729.5	414.7
small = 9(16) procs/domains	341.9	95.1
speed up for small domain	11.4	18.0

Table 1: Measurements results on IBM Netfi nities (times are in seconds)

In these set of measurements, as above, the sources at the borders of domains produce packets at the rate of 20000 packets/sec for half of the simulation time. The bandwidth of the link is 1.5Mbps. Thus, congestion is defi nitely produced on certain links shown above and congestion may be produced on certain other links. For the other half of the simulation, these sources produce 1000 packets which is less than the total bandwidth of the links connected to each of them. All other sources produce packets at the rate of 100 packets/sec for the entire simulation. We measured the speed up for this confi guration over 60 seconds of simulated traffi c. The simulation interval was set at 14.9999 seconds, resulting in fi ve freezes.

The speedup of simulation with 9 domains was approximately 11 compared with a single network (sequential) run. The graphs of the results are shown in Figure 6 and the numerical results are presented in Table 1. This confi guration is less regular then the 64-node confi guration and as result, the number of iterations needed for convergences varied from two to four. Despite that, the decomposed simulations showed superlinear speedup. The differences in the total number of packets in each flow, the number of dropped packets and the sizes of the queues at the routers were

well below 1% for all three different domain sizes.

4 Checkpointing - What and Why?

Checkpointing is a new feature which has been added to mitigate the effect of large variance in *Average Delay* calculations for each of the *Fake Sources* pumping data into each domain.

This checkpointing routine is called when on comparing the current statistical data of average delays more than 10 percent variance is found. This means that the data (delays and drop probabilities) used in the previous iteration was in error and this data is updated by the delays obtained in the current iteration.

We wanted to introduce this feature with the least amount of code restructuring as possible hence we are using the Unix Signal features which allows us to stop(suspend) and restart a process as explained in the following pages.

4.1 Some History

The original idea was to use third part libraries one of them being the *Dynamite Checkpointing* library since it was a transparent checkpointing library for Unix.

This library works by writing the address space of the process and its mapped shared libraries to a file. This is done when a process receives a signal *USR1* which is a user signal. This is also called preserving the memory image of the process.

This checkpointing file is an ELF executable which first loads the text and data segment into the memory followed by the heap and stack segments respectively.

Once the loading has occurred *siglongjmp* is made to jump back to the checkpoint signal handler to restore the signal mask.

The state of the signal handlers are restored using the *sigaction* and the previously open files are restored. When the signal handler returns, the operating system restores all CPU registers and the application resumes execution.

4.2 Current Checkpointing Implementation

Some drawbacks that were found in using the third party library were:

- The implementation depended on the image of the process address and data space being copied to an ELF executable which involves IO operations and utilizes a signifi cant amount of disk space.
- Also this implementation of checkpointing depends on certain third party libraries which are currently not widely supported by many operating systems.

Hence an alternative algorithm is listed below:

At the instant of Freeze:

- The main process forks a child process. This child process is suspended until the main process signals it to awaken.(This is currently implemented using SIGSTOP and SIGCONT signals).
- The main process proceeds to the end of execution. At the point of the next freeze if the process needs to go back (ie. iterate) it sends a wake up signal(SIGCONT) to the child process which originally in suspend (SIGSTOP) mode. Now this child process becomes the main process.
- If instead the process realizes that it can go ahead, it awakens the child process and kills it(SIGKILL).

Thus it can be observed that at any one point in execution time there is only one process. These steps are repeated as per the number of freeze events required.

4.3 Advantages

The advantage of this technique is that we just keep a copy of the process in memory in the child process and there is no need to resort to any IO.

5 BGP Extensions to SSFNet for Genesis support

5.1 Introduction

The Border Gateway Protocol (BGP)is the de facto standard inter-domain routing protocol in today's global Internet.BGP builds and maintains forwarding tables to be used by a router when forwarding data packets around an inter-network, across the Autonomous Systems (AS). The forwarding table are built up in a distributed fashion: all routers running BGP in the entire inter-network share reachability information with each other. When faced with multiple routes to the same destination, a selection is made based on several factors, many of which can be configured by the administrator. Most commonly, shortest autonomous system (AS) path length is the primary factor.

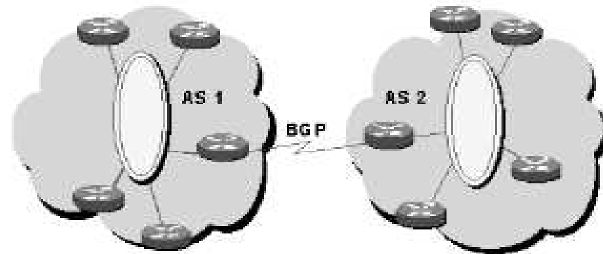


Figure 7: BGP between AS'

5.2 Goals

BGP has been increasingly used for some forms of traffic engineering. Our goal is to provide a novel outbound load-balancing technique using BGP LOCAL_PREF settings and aided by online simulation. Outbound means each AS has several links and we want to distributing traffic to theses links such that a complex set of objectives are met. These objective functions will represent more realistic needs of large networks like those of ISPs and defence networks than what is done today.

To incorporate Genesis into the BGP implementation on SSFNet, one of the first

goals was to Domainize the flow of BGP messages ie. allow individual BGP routers on different processors to behave as if they are communicating directly with one another. What Genesis aims at achieving is to completely bypass direct communication between the routers. The final goal being to harness the power of each individual processor while reducing the processing load per processor by disallowing all domains to activate. largeThis selective activation is what enables each domain simulation to converge faster.

BGP under Genesis To support simulation in which BGP changes impact background traffic while preserving speed and scalability we use Genesis approach. Our goal was to port BGP to Genesis using SSFNET, and to measure the scalability and speed of BGP simulation under Genesis.

Information Exchange Unlike TCP and UDP, the information exchanged between BGP Speakers (Peers) on the border routers are Route Withdrawals or Additions. So it is important to pass between Genesis domains information carried by these messages. This required a modification to the Farmer-Worker architecture to support transfer of route information and not only the traffic characterization needed for TCP and UDP flows.

Synchronization BGP Speakers involved in route exchanges have to be in the mutually complimentary states of Send and Receive during the course of updating their routing tables with new route information. Thus there is a need to maintain the state of the simulation by recognizing when a BGP Speaker issues an update and which destination has to be in the Receive state at the same time. This is an issue in Genesis where each domain is simulated independently of the others and the only means of maintaining the global state of any object is through the Farmer-Worker architecture.

5.3 Modification to the SSFNet BGP implementation - Mini Freeze

The basic modification to the existing BGP implementation is to support synchronized peer updates on route changes. By that it means that if a peer in domain 1 wishes to inform the peer in domain 2 it does so by creating a socket connection with the peer in domain 2. The problem with this is that in order to synchronize the transfer of route updates it is required that both the domain simulations are in the same state. For example if at Simulation time 5seconds Router 1 in domain 1 sends an update to Router 2 in domain 2. It is necessary for domain 2 to receive this update at time 5 of its simulation clock. For this we propose the following

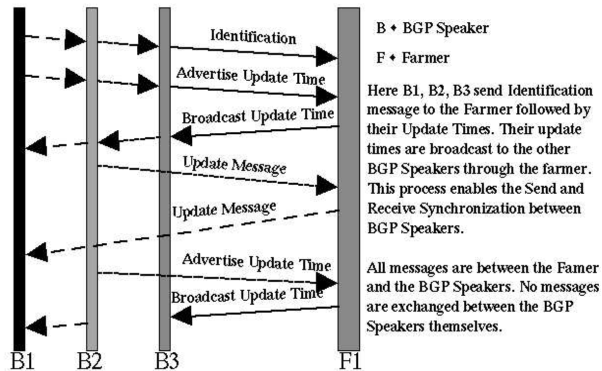


Figure 8: Message Synchronization between BGP Peers

additions to the existing Genesis/BGP implementation:

Mini Freeze

Socket communication framework for transferring route update information across routers in different domains to support the Mini Freeze event.

The Mini Freeze will be scheduled by the destination domain which contains the router for which the update is the target. This is valid because parsing of the DML fi le containing the network model gives information as to the scheduled route update times.

So the source domain need not schedule a Mini Freeze since all it has to do is setup the socket connection when router update is scheduled with an external peer entry. This is done when the peer entry calls its SendMsg() function in the BGPSession class for the peer entry.

The technique to implement the Mini Freeze would be to schedule the regular Mini Freeze event ahead of time based on the parameters returned from parsing the DML fi le. *One of the issues to be discussed is that if the peer entry that schedules this Mini Freeze event does not receive a route update during the Freeze.*

5.4 Reusability of existing Genesis classes to support BGP peer decomposition - Confirming simulation correctness

Once the update has been executed the thing that has to be taken care of is the delays in packets that has been affected by this route change. Consider the following scenario:

At time t_1 = 5sec in domain 1 Router 1 advertises a route withdrawal in domain 1. Assume that the source in Domain 1 now uses route 2. As indicated in the figure. In Genesis we allow this source operation in domain 2 by means of a source proxy. We can get this route update to Domain 2 at time 5sec using the Mini Freeze event. But for Genesis to maintain correct simulation state we will have to have the source proxy indicate this delay for packets generated after 5 + t seconds.

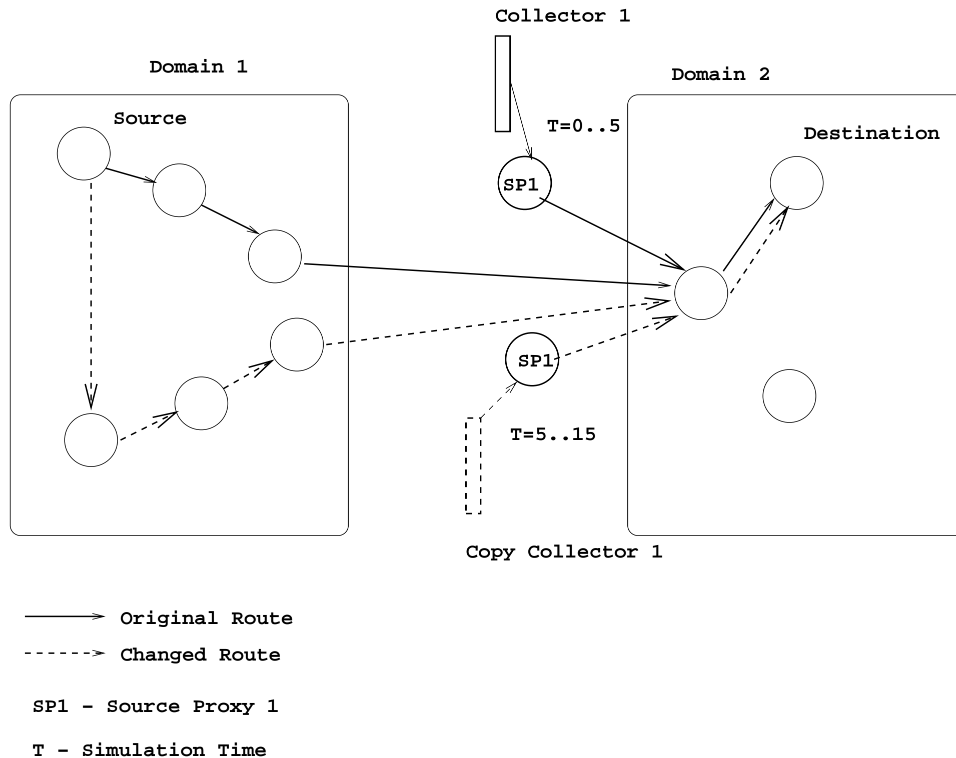


Figure 9: Design Model

To do this one novel way without much modification to the Genesis infrastructure would be to create a Collector copy for each flow. This collector copy would be

used by the source proxy when the route goes down. This collector contains delays for packets following the modified route.

5.5 State Saving

State saving is a new feature added as an extension to support Genesis-based BGP. Currently we support only route updates as of writing this report. But other BGP messages have also been incorporated which is outside the scope of this document.

State saving as an extension to BGP involves writing the update messages to file. Recording the following parameters:

- Freeze Time
- Message Contents (Update/Withdraw etc)
- AS Router ID

5.6 SSFNet Additions

BGPFreeze: This event allows for BGP synchronization points to be inserted into the simulation. It is configurable through the DML file.
BGPFreezeTimer: This is a derivative of the inbuilt Timer class of SSFNet. It counts down time to different BGP synchronization points.
ConnectServer: This handles all communications between each BGP Speaker and the Farmer.
BGPServer: This is a multithreaded Farmer that schedules all BGP Update events and synchronizes between various BGP Speakers.
BGPSession: Modified This class has been modified from the original SSFNet version to account for the messaging subsystem that allows for the independent operation of each domain.
Net: Modified The Net class has been modified to allow for the selective activation and deactivation of the domains.

6 Model verification through Self-Similarity

There has been some interest in experimentally observed self-similar, or long rang dependent, behavior in for instance local area network (LAN) traffic [1] and VBR video traffic.

The cause of self-similarity in traffic statistics in real networks is an open question at present. It is the belief that many traffic flow control protocols are intrinsically capable of generating chaotic [2] behavior which are believed to be self-similar [**?**].

We in the last year have developed a distributed network simulation model GeNeSIS (General Network Simulation Integration System) which allows for scalable multi-domain real-time modeling of the network. This model addresses scalability at experiment design, network decomposition and network simulation levels [6]. One major goal after proving that this network did indeed achieve almost super-linear speedup, was to validate this model.

Now we approached the problem of validation from two angles. One was experimentation - using ns [3] trace files to measure delays and drop rates of selected flows, arrival rates of acknowledgment packets, arrival times and distribution of packets. The other was analytical - check simulated system behavior against known system input/output behavior: for example confirm that a self similar nature of the aggregation of the TCP flow is preserved.

7 Models to verify self-similarity

The generally accepted argument for the "Poisson-like" nature of aggregated traffic, namely, that aggregate traffic becomes smoother (less bursty) as the number of of traffic sources increses, has very little to do with reality. None of the commonly used traffic models is able to capture the *fractal*-like behavior of self-similar traffic [1].

The general most accepted way to prove self-similarity is to measure the burstiness of traffic. This is identified by the 'Hurst Parameter' [7]. In order to check for the possible self-similarity of the traffic data, the following graphical tools can be applied viz. variance time plots, pox plots of R/S, and periodogram plots. The *Hurst Parameter* H, when calculated for the average packet arrival time or jitter is close to unity. This in turn implies the maximum possible degree of self-similarity.

The rescaled adjusted range statistic - pox plots of R/S:
The rescale adjusted range statistic, *R(n, k)/S(n, k)*, is calculated for a selection of subsets of the arrival times for the packets T_i, starting at *n* and of size *k + 1* [8]. The adjusted range *R(n, k)* has the following physical interpretation. Suppose that the times series T_i represents the amount of water per time unit flowing into a reservoir. Furthermore, water flows out of the reservoir at a constant rate, this rate being such that the reservoir contains the same amount of the water at the *n + k*-th time unit as the *n*-th time unit. Then *R(n, k)* is the maximum capacity of the reservoir such that it will not overflow in the period *n* to *n + k* inclusive.

The calculation of *R(n, k)* proceeds as follows. Given *n* and *k*, the mean

$\mu(n,k) = \frac{1}{k+1}\Sigma_{i=n}^{n+k}T_i$ (1)

and standard deviation

$S(n,k) = \sqrt{\frac{1}{k+1}\Sigma_{i=n}^{n+k}(T_i - \mu(n,k))^2}$ (2)

are calculated. Then,

$R(n,k) = max_{0\leq l\leq k}(\Sigma_{j=n}^{n+l}T_j - (l+1)\mu(n,k)) - min_{0\leq l\leq k}(\Sigma_{j=n}^{n+l}T_j - (l+1)\mu(n,k))$ (3)

Plot $log_{10}(R(n,k)/S(n,k))$ against $log_{10}(k)$. Vary n, k....starting value of n is chosen randomly in the range 1 to N - k where N could be the number of packets recieved. Use linear regression to fi t a straight line through the R/S plot. Slope of this line being an estimate of H - the Hurst parameter.

Periodogram Analysis:
One of the characteristics of a self-similar time is that the logarithm of its spectral density

$log_{10}f(w) \sim (1-2H)log_{10}|w| + C$ (4)

as $w \rightarrow 0$, with C a constant. This is used as the basis for another method of estimating H [4]. First the periodogram

$I(x_j) = \frac{1}{2\pi N_p}.|\sum_{m=1}^{N_p} A_m e^{imx_j}|^2$ with $x_j = \frac{2\pi j}{N_p}$ (5)

is calculated, with $j = 1...\lfloor (N_p - 1)/2 \rfloor$ and N_p typically 10^5. A multiple linear regression [9] is then performed to find the coefficients $a_0....a_3$ in

$$log_{10} f(x) = a_0 log_{10} |1 - e^{ix}| + a_1 + a_2 x + a_3 x^2 \quad (6)$$

which best fit $f(x_j)$ to $I(x_j)$ for $j = \lfloor (N_p - 1)/2 \rfloor$. In 6, the logarithm term represents long range dependence and the remaining terms represent the short range dependence in the time series. It is clear from 6 that

$$\lim_{x \to 0} log_{10} f(x) = a_0 log_{10} |x| + a_1 \quad (7)$$

Standard deviation of aggregates:
A direct approach to calculate *H*, which is also described in [1] and [5]. First for a give *k*, the *k*-aggregated series

$$A_n(k) = \frac{1}{k} \sum_{m=1}^{k} A_{(n-1)k+m} \text{ with } n = 1...\lfloor N/k \rfloor \quad (8)$$

is calculated. The standard deviation of this series is then found from

$$A_n(k) = [\frac{1}{\lfloor N/k \rfloor} \sum_{n=1}^{\lfloor N/k \rfloor} (A_n(k) - \mu)^2]^{\frac{1}{2}} \quad (9)$$

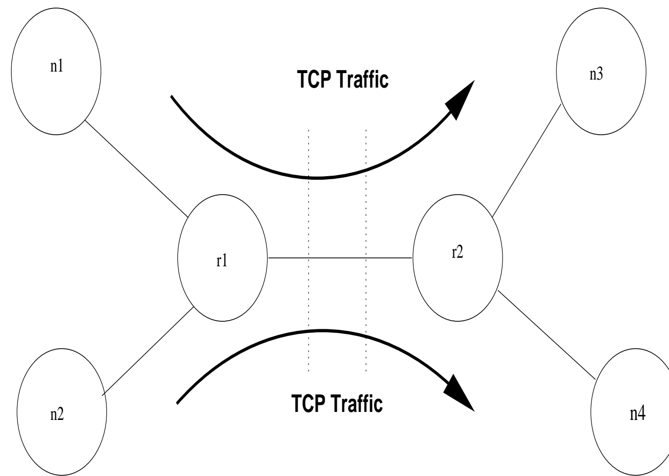
where $\mu = \mu(1, N)$ is the mean of the entire series $A_n, n = 1...N$. As stated in [1] and [5], the asymptotic slope of a plot of $log_{10} S(k)$ against $log_{10} k$ is equal to *H - 1*. The aggregation *k* is varied in exactly the same way as it was in the R/S calculation.

Autocorrelation Function:
The autocorrelation function of the time series

$$C(k) = \frac{1}{N-k} \sum_{j=1}^{N-k} (A_j - \mu)(A_{j+k} - \mu) \quad (10)$$

has also been calculated, with $\mu = \mu(N, k)$ and with *k* ranging from 100 to N/10. Although the calculation does not lead to an estimate for H, the autocorrelation function displays behavior characteristic of self-similar processes, for large *k*.

8 Test Network Configuration



All links are 1.544 Mbit T–1, queues are droptail.

Figure 10: Sample network configuration used to validate self-similarity

As shown in Figure reffigure1 the network currently under consideration for measurement of self-similarity consists of 2 routers r1 and r2. There are 2 nodes connected to each of the 2 routers viz. n1 and n2 to r1, n3 and n4 to r2. The domains can be identified by the 2 dotted lines separating them . The domain on the left is called Domain 1 and the domain on the right is called Domain 2. Domain 1 is the source of all packets.

This configuration is implemented in ns. The configuration script is written in Tcl. The system is then implemented on GeNeSIS to segreagated the domains onto 2 processors to verify similarity across the distributed model. The data collected is written to trace files. Here the difference in arrival times of the packets are recorded. Also recored with each arrival time is the difference in sequence numbers to allow for dropped packets.

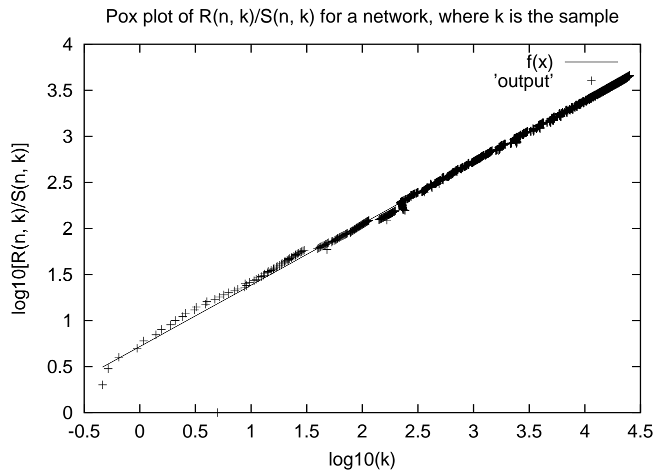


Figure 11: Timing Graphs

9 Analysis

For the analysis we chose the rescaled adjusted range statistic - pox plots of R/S.
The data collected from the test setup is then fed to a program to calculate $log_{10}(R(n,k)/S(n,k))$ and $log_{10}(k)$.

A plot of $log_{10}(R(n,k)/S(n,k))$ and $log_{10}(k)$ is made. A curve fi tting algorithm like the least sqaure method is applied and a line is fi t through the pox plot. The slope of this line gives an approximate value of H.

The plot in Figure10 shows the pox-plot for the test network described in the previous section. The slope of the line is approximately 0.699. A good indication of the self-similarity evident in TCP networks.

10 Complex Networks

In addition to the analysis on the simple network shown in the previous section, Genesis was validated against the network that inspired the seminal paper on self-similarity [1]. This network consists of 2 internal networks which are then connected to the external Internet through 2 routers. The traffi c that has been defi ned to be mostly TCP interactive traffi c and some FTP sources.

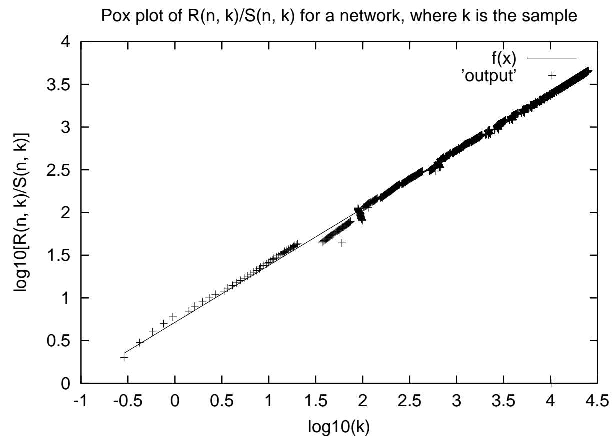


Figure 12: Single Domain Run on unmodifi ed ns Slope of straight line gives H

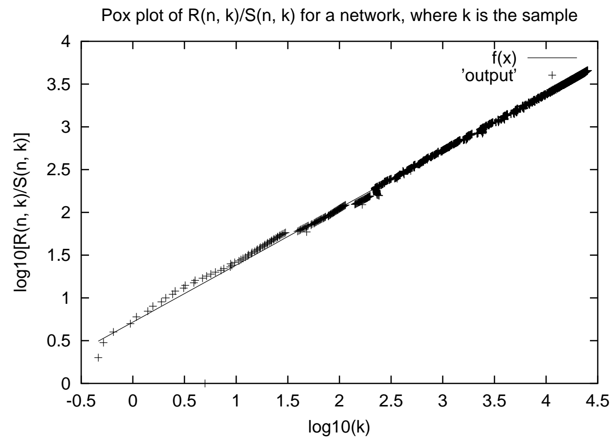


Figure 13: GeNeSIS run Slope of straight line gives H

The network confi guration is given in the fi gure below:

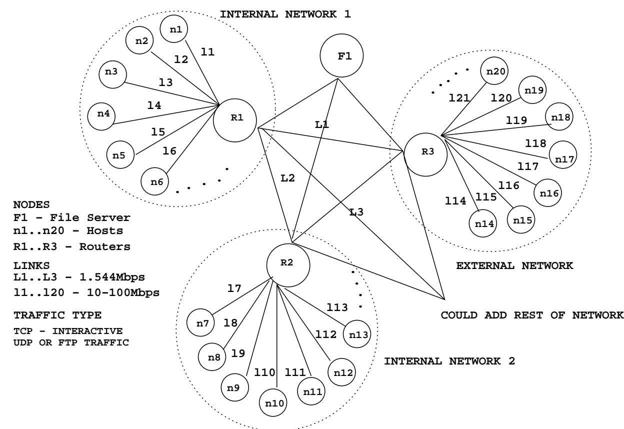


Figure 14: Bell-Core Network Sample Network Confi guration

This network confi guration was run through the regular unmodifi ed ns and was also simulated using Genesis. To validate our model the pox plots for rescaled adjusted range statistic is compared. Both plots are shown below.

It can be observed from the plots that the Hurst parameter given by the slope of the line through the pox plots are approximately of the same value which is 0.667. A clear indication of self-similarity [1].

11 Conclusion

Genesis has been validated against many well known and sample networks. All the results indicate that the selfsimilar nature of the traffi c is maintained across the model. The Hurst parameter values is apporximately 0.667 which is good indicator of self-similarity.

References

[1] W.E. Leland, M.S. Taqqu, W. Willinger & D.V. Wilson, "On the self-similar nature of Ethernet Traffi c [extended version]", *IEEE Transactions on Networking*, **2**, 1, 1994, pp. 1-15

[2] E. Ott, “Chaos in dynamical systems”, Cambridge University Press, Cambridge, 1993, ISBN 0-521-43799-7

[3] ns(*network simulator*). http://www-mash.cs.berkeley.edu/ns.

[4] J. Beran, “Fitting long-memory models by generalised linear regression”, *Biometrika*, **80**, 4, 1993, pp. 817-822

[5] M.W. Garrett & W. Willinger, “Analysis, modelling and generation of self-similar VBR video traffi c”, *Proceedings ACM SIGCOMM ’94*, London, 1994, pp. 269-280

[6] Szymanski, B., Y. Liu, A. Sastry, & K. Madnani, “A system for Large-scale Parallel Network Simulation”, CS TR 01-14, Submitted to the PADS 2002 conference.

[7] H.E. Hurst, “Long-term storage capacity of reservoirs”, *Transactions of the American Society of Civil Engineers*, **116**, 1951, pp. 770-799 itWater Resources Research,]

[8] B.B. Mandelbrot & J.R. Wallis, “Some long-run properties of geophysical records”, **5**, 2, 1969, pp/ 321-340

[9] W.H. Press, S.A. Teukolsky, W.T. Vetterling & B.P. Flannery, “Numerical Recipes in C” (2nd edition, 1992), Cambridge University Press, ISBN 0-521-43108-5

[10] Bhatt, S., R. Fujimoto, A. Ogielski, and K. Perumalla, “Parallel Simulation Techniques for Large-Scale Networks” *IEEE Communications Magazine*, 1998.

[11] Fujimoto, R.M., “Parallel Discrete Event Simulation,” *Communications of the ACM*, vol. 33, pp. 31-53, Oct. 1990.

[12] Law, L. A., and M. G. McComas, “Simulation Software for Communication Networks: the State of the Art,” *IEEE Communication Magazine*, vol. 32, pp. 44-50, 1994.

[13] Ye, T., D. Harrison, B. Mo, S. Kalyanaraman, B. Szymanski,K. Vastola, B. Sikdar, and H. Kaur, “Traffi c Management and Network Control Using Collaborative On-line Simulation,” *Proc. International Conference on Communication, ICC2001*, 2001.

[14] L.R. Klein, “Quantitative Studies of International Economic Relations,” *Chapter The LINK Model of World Trade with Application to 1972-72*, North Holland, Amsterdam, 1975.

[15] Y. Shi, N. Prywes, B. Szymanski and A. Pnueli, “Very high level concurrent programming,” *IEEE Trans. Software Engineering*, SE-13:1038-1046, September 1989.

[16] B. Szymanski, Y. Shi and N. Prywes, “Synchronized distributed termination,” *IEEE Trans. Software Engineering*, SE-11:1136-1140, September 1987.

[17] M. Yuksel, B. Sikdar, K.S. Vastola and B. Szymanski, “Workload generation for ns simulations of wide area networks and the internet,” *Proc. Communications Networks and Distributed Systems Modeling and Simulation Conference*, Pages 93-98, SCS Press, 2000.