Design and Implementation of MaRS: A Routing Testbed*

Cengiz Alaettinoğlu, A. Udaya Shankar†, Klaudia Dussa-Zieger, Ibrahim Matta

Department of Computer Science
University of Maryland
College Park, Maryland 20742

September 1992. Revised June 1993. Revised November 1993.

Summary

MaRS is a discrete-event simulation testbed for developing routing algorithms for wide-area computer networks. It was developed as a modified and enhanced version of an existing simulator, NetSim. MaRS allows the user to define a network configuration consisting of physical network, routing algorithm and workload. The user can control its simulation, log the values of selected parameters, and save, load and modify network configurations. MaRS provides both steady-state and instantaneous performance measures to facilitate the study of the complex dynamics that arises in routing systems (due to delayed feedback). We have used MaRS to evaluate and compare several next-hop routing algorithms.

Keywords: Computer networks, routing protocols, performance analysis, modeling, discrete-event simulation.

* This work is supported in part by Rome Labs and DARPA under contract F30602-90-C-0010 to UMIACS at the University of Maryland, and by National Science Foundation Grant No. NCR 89-04590. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, RADC, or the U.S. Government. Computer facilities were provided in part by NSF grant CCR-8811954.

†Also with Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742.

Contents

1 Introduction

In a wide-area store-and-forward computer network (e.g. the Internet), routing protocols are responsible for choosing routes for data packets, that yield good performance in terms of delay, throughput, etc. The performance of a route changes with time because it depends on the amount of traffic flowing through the links, which depends on the time-varying external load. Consequently, a routing protocol must monitor link traffic changes and accordingly adapt its routes.

Next-hop routing is a simple and robust way to do adaptive routing. Each node maintains for each destination a neighbor node id, referred to as the *next-hop*. Each data packet has its destination node id in its header. When a node receives a data packet, it forwards the packet to the next-hop for the packet's destination. The objective is that the succession of next–hops for a destination leads to the destination and is an optimal path. To achieve this objective, a next-hop routing protocol does the following functions: (a) each node maintains for each of its outgoing links a dynamic *link cost* which is updated regularly according to the traffic flowing through the link; (b) link cost information is disseminated regularly to nodes of the network; (c) based on received link cost information, each node maintains and regularly updates a next-hop for each destination.

The dynamics of such systems can be very complex due to the distributed nature of the network and the delayed feedback between route changes and link cost changes [1] (see below). A testbed is essential to understanding their performance. We have built such a testbed, called MaRS (Maryland Routing Simulator), and this paper describes its design and implementation. MaRS is developed as a modified and enhanced version of NetSim. Thus this paper also serves as a reference to some of NetSim's features.

MaRS is available publicly on the Internet at site `ftp.cs.umd.edu`, in the directory `pub/MaRS`. The first release was in June 1991. There is a mailing list, `mars-users@cs.umd.edu`[1], for discussing MaRS related issues. Currently there are around fifty subscribers on the mailing list, some of who are actively using MaRS (e.g. [2]).

The rest of the paper is organized as follows. Section 2 motivates the desired features and describes our implementation considerations. Section 3 describes the structure and the components

[1] To subscribe to this list send mail to `mars-users-request@cs.umd.edu`.

of MaRS. Section 4 describes the user interfaces of MaRS. Section 5 demonstrates how a user builds a target system in MaRS and obtains performance measures. Section 6 describes how a component is implemented in MaRS, and how it interacts with other components. Section 7 describes some results obtained using MaRS. Section 8 contains concluding remarks. The reader who is not interested in the implementation details can skip Section 6.

2 Desired Features and Implementation Considerations

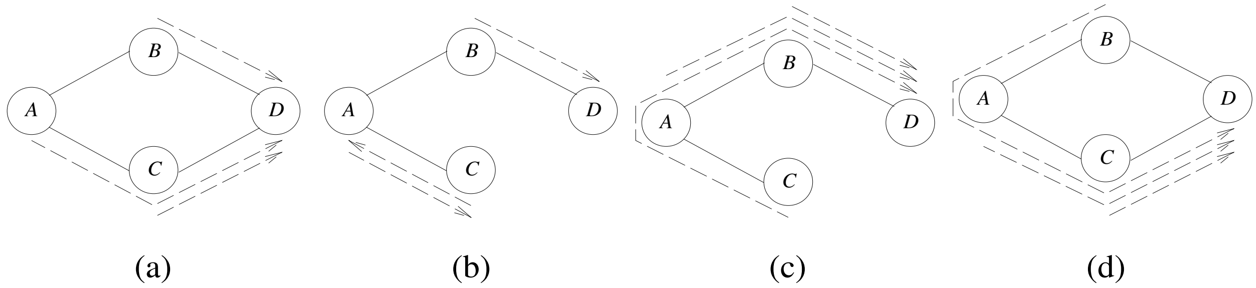

Figure 1: Complex dynamics of routing systems.

Consider the simple network topology of Figure 1(a). There are three connections: from A to D, from B to D, and from C to D. In each case, node D is the destination node. The directed dotted lines in the figure indicate the routes taken by the packets of the connections. Let us assume that the routes in Figure 1(a) are optimal routes.

Assume link $\langle C, D \rangle$ fails. Upon detecting this, C sets its next-hop to A. A loop between A and C is formed; that is, A's next-hop is C and C's next-hop is A (see Figure 1(b)). This degrades performance since all packets sent from A to C will be returned back until A receives and processes the information that the link $\langle C, D \rangle$ is down. The duration of the loop depends on the routing algorithm. In fact, some of the first routing algorithms [3] would allow C to set its next-hop to A even if there is no path to D through A, and furthermore allow the loop to stay forever.

Once A learns that link $\langle C, D \rangle$ is down, it sets its next-hop to B. Now all three connections are routed over link $\langle B, D \rangle$ (see Figure 1(c)). This increases the cost of link $\langle B, D \rangle$.

When link $\langle C, D \rangle$ is repaired, its cost is low since no traffic is flowing through it. Links $\langle B, A \rangle$

and $\langle A, C\rangle$ also have low cost since no traffic is flowing in this direction. After the repair, due to the high cost of $\langle B, D\rangle$, all three connections will be routed over link $\langle C, D\rangle$ (see Figure 1(d)), instead of the optimal routes (i.e. Figure 1(a)).

When the link costs are next calculated (assuming simultaneous update of all link costs), the cost of link $\langle C, D\rangle$ will be high and the cost of link $\langle B, D\rangle$ will be low. Hence all three connections will be routed over $\langle B, D\rangle$. In the next round, all three connections will be routed over $\langle C, D\rangle$, and so on. That is, instead of optimal routes, the routes taken by the three connections will oscillate between two non-optimal sets (in practice, to avoid this, link costs are subjected to exponential averaging, which causes link costs to change more slowly).

The dynamics are complex even for a simple four-node topology. With more realistic topologies, the system's behavior can be even more complex, for example: loops can involve multiple hops; breaking one loop may cause another loop. In general, there are many kinds of *topology changes* (failures and repairs of nodes and links, start and termination of connections) that affect the optimality of routes. The effects of the topology changes can interfere with each other over time, resulting in very complex behaviors.

Consider for example the loops which are established while trying to adapt to topology changes. There are routing algorithms which (1) allow long-lived loops, (2) allow only short-lived loops (as in the example), and (3) do not allow any loops. It is well known that allowing long-lived loops severely degrades performance [3]. However, it is not clear whether avoiding loops completely is superior to allowing only short-lived loops. This is because routing algorithms that avoid loops completely usually require nodes to use old next-hops or have no next-hops during the period when new (loop-free) routes are calculated.

A testbed is essential to understanding the complex dynamics of routing systems. Ideally, we would like the testbed to accurately model features that are important to routing performance, and for efficiency ignore or appropriately abstract away features which are not important. For example, if the link queueing discipline is not important, then instead of storing the sequence of packets awaiting transmission, we could just store the number of packets in the queue. However, it is not clear which features (e.g. flow control, link scheduling, buffer management) are important for routing performance. We opted to take a conservative approach by having easily extendible modules to represent these features. The testbed should have the following capabilities:

- Adding new routing algorithms to the testbed should be easy, as also should adding new kinds of workload, nodes and links (e.g. new link scheduling disciplines). Such additions require programming. Hence the testbed should be modular.

- The user should be able to easily build arbitrary network topologies and define different workload distributions. This should not involve any programming.

- The user should be able to specify various kinds of topology changes with various temporal characteristics, both aperiodic and recurrent, deterministic and stochastic. For example, the user should be able to fail or repair a link at a specified time and observe if and how the system converges to a new steady state.

- It should provide standard performance measures such as average delay, average throughput, average load, as well as more specialized measures such as the current next-hops to a destination, the number of routing protocol messages used, the number of failed links. It should provide both average and instantaneous versions of these measures. Instantaneous measures (e.g. throughput during the last 500msecs) are needed to observe the system adapting to radical changes (e.g. link failure).

We considered two approaches to a software testbed: *process emulation* and *discrete-event simulation*. In process emulation, every component of the computer network that is to be studied, henceforth called *target system*, is represented by a separate process. Each process executes code corresponding to the code of its component. An advantage of process emulation is that the actual code of the target system can be tested. However, in order to faithfully capture the dynamics of the target system, it is necessary to periodically synchronize all the processes. The overhead for this process synchronization is a serious disadvantage.

In discrete-event simulation, the target system is abstracted by a set of state variables and a set of events. The behavior of the system is represented by a sequence of timed event occurrences (explained in Section 3). Each event occurrence updates the values of the state variables and the "simulated time". A major advantage of discrete-event simulation is that it is very flexible: more state variables and events can be introduced to make the simulation more realistic at the expense of increasing computational cost.

We chose discrete-event simulation over process emulation because it offers the greatest flexibility in modeling the target system. In order to reduce programming work, we examined three discrete-event simulators that were available to us: NetSim from MIT [4]; COMNET II.5 from CACI Products Company [5]; and DeNet (Discrete Event Network) from University of Wisconsin-Madison [6]. NetSim is designed for simulating a network of message-passing components. It is written in C and runs under UNIX and X windows. COMNET II.5 is a commercial package similar to NetSim. DeNet is a general purpose simulator that simulates a set of communicating objects. It is written in Modula2.

We found NetSim most suitable for our purposes for several reasons. First, the source code was readily available, and its style of coding and documentation is good. Thus modifications and expansions seemed easy to do. Second, it is written in a very portable form of C. Third, it does not enforce modular composition rules, unlike DeNet. Thus, modularity can be second to computational efficiency.

NetSim is structured into three parts, a simulation engine, a user interface, and a set of components that simulate physical network (i.e. links and switches, etc.), workload, and transport protocols. The set of components that came with NetSim is designed to study flow control, rather than routing. Most flow control studies consider a main connection over a (usually short) fixed route, with other fixed-route cross connections interfering with the main connection. Typically, the performance of the main connection (throughput, delay, etc.) is observed while the cross connections start, send packets, and terminate. NetSim supports only fixed routes which have to be hand-specified by the user. The routes are stored in a shared data structure accessible by all components.

The user cannot specify failures and repairs of links and nodes. Nodes (switches) do not maintain statistics essential for link cost calculation. The user can only specify a fixed number of connections. Each connection can be started at a specified time. But once it ends it cannot be restarted, in contrast to a workload model where connections start and terminate continuously. NetSim cannot model FTP and Telnet traffic directly.

NetSim does not provide system wide performance measures (e.g. overall throughput). It does provide per connection and per component performance measures (e.g. throughput of a connection, utilization of a link), but these are not sufficient for our purposes.

NetSim also has features which are not important for our purposes. Some features (e.g. a restricted flow control) could interact with routing in uncontrolled ways. Some features could slow down our simulations (e.g. probabilistically setting an error bit for each packet transmission to model channel errors, which involves random number generation).

In building MaRS, we replaced all the components that came with NetSim with our own components. We used the simulation engine of NetSim, but with extensive modifications. We built a much nicer new graphical user interface using Motif [7]. We kept the old X interface of NetSim for users of MaRS that do not have Motif, but did extensive modifications to the interface code. In implementing our components, we used the same programming structure as in NetSim. In terms of number of lines, MaRS extends NetSim from 18K lines to 40K lines. In our descriptions in this paper, we point out our use of NetSim's code and concepts, and our modifications.

3 Structure of MaRS

In discrete-event simulation, the target system is modeled by a set of **variables** and a set of **events**. Associated with each event is a **routine** that can update the variables and cause other event routine executions. A simulation of the target system corresponds to a succession of event occurrences over time, i.e. a sequence of the form (e_1, t_1), (e_2, t_2), ..., (e_n, t_n), where $(e_i,\ t_i)$ denotes the execution of event e_i's routine at time t_i.

The simulator executes a loop, with one event occurrence simulated in each iteration. It maintains the following two variables: **simulated time**, indicating the time of the last event simulated; and **event list**, consisting of a set of (e, t) pairs, where e is an event and t is a time. An (e, t) pair in the event list means that e is to be executed at time t. In each iteration, the simulator removes an (e, t) pair with the smallest t from the event list, updates the simulated time to t, and executes

e's routine. We refer to this as "firing event e at time t". The execution of e's routine can schedule new events, i.e., add new (e', t') pairs into the event list.

MaRS has the same structure as NetSim: a **simulation engine**; a **user interface**; and a **set of components** which accommodates a variety of target systems and performance measures.

The simulation engine manages the event list and the user interface (including X window, command line options, and files). When MaRS is started, the simulation engine initializes variables of the user interface, processes command line options (including reading any files), and then goes into the basic simulator loop. In each iteration, it processes inputs from the keyboard and mouse, executes the next event from the event list, and updates the graphical user display.

The components are for modeling the target system and certain simulation functions. For modeling the physical nodes and links of the target system, MaRS has **link components** and **node components**. For modeling routing protocols, MaRS has a **link-cost function component** and three **routing components**, SPF, ExBF, and SEGAL, corresponding to three different routing algorithms [8, 9, 10, 11]. For modeling application and transport protocols (e.g. user traffic and flow control), MaRS has three types of **workload components**, FTP, TELNET, and Simple_Traffic, corresponding to three different types of user traffic. Each type of workload component has two versions, a **source** and a **sink**. MaRS has one component concerned with simulation functions, the **performance monitor**, which collects statistics.

A component in MaRS (and in NetSim) consists of a data structure and a set of events, each with an associated routine. A component can be **instantiated** by appropriately initializing its data structure. Multiple instances of a component are needed to model a target system. For example, each node (link) of the target system is represented by an instance of a node (link) component. By interconnecting multiple instances of various types of components, a target system of arbitrary topology, routing protocol, and workload can be obtained. For readability, we will often refer to a component instance as a component, when it causes no confusion.

Figure 2 shows a target system of five node component instances and six link component instances. Each link component is connected to exactly two node components. A node component models the "physical" aspects of a store-and-forward entity such as a host or a switch. Each node component is connected to an SPF routing component and zero or more workload components. Each workload component is either a source or a sink of some type (i.e. FTP, Telnet, or Simple_Traffic). For each source component instance, there is a corresponding "peer" sink component instance. There is a single instance of a link-cost function component; it is not connected to any component.

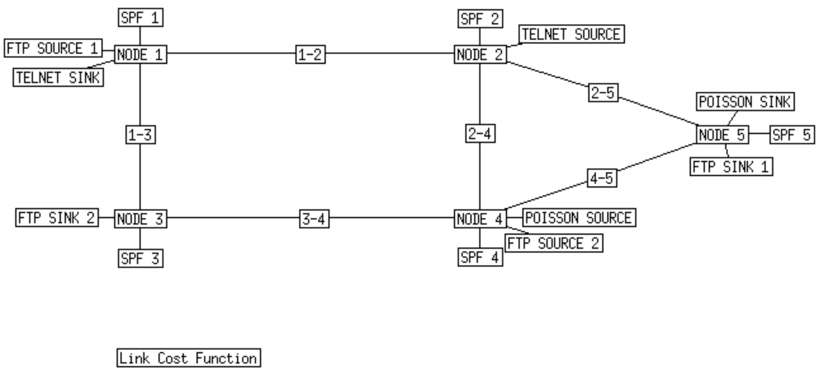

Figure 2: Target system.

Component instances can interact with each other by calling events, by scheduling events, or in some special cases by sharing variables. That is, if e and f are events of different component instances, e's routine can (1) call f's routine, or (2) schedule f with some time value t, or (3) access the variables of f's routine. (Details in Section 6. Option (1) is not possible in NetSim.) Component instances interact with the simulation engine as follows. A component instance can make file system calls to store performance measures that are to be logged. The simulation engine can access component data structures to determine the displays to be updated. The simulation engine can call events of a component in response to user interface commands (either from command line, input file, or mouse/keyboard).

In the rest of this section, we briefly describe each type of component. Full details can be found in [12].

Link

A link component models a bidirectional transmission channel between two nodes. It contains two queues of packets, each representing the packets in transit in one direction. A link can be subject to failure and repair, with up-times and down-times described by (probability) distributions. When a link fails, all packets that are in transit are lost.

Input parameters for a link component include propagation delay, bandwidth, distribution of up/down times, etc. Output parameters include instantaneous utilizations of (workload and routing) packets for each direction.

Link-cost function

The link-cost function component maintains and periodically updates a cost for each link in the network. At each update, the link-cost value is calculated by applying a transformation to a statistic referred to as "raw cost" which is observed on the link during the previous update period. The raw cost usually reflects the load on the link. MaRS currently supports the following raw link-cost functions: hop-count, utilization, (queueing and propagation) delay, and hop-normalized delay [13]. MaRS currently provides a transformation that includes exponential averaging, scaling with a slope and an offset, minimum and maximum bounds, and movement limit. Parameters include the link-cost update period and parameters of the transformation.

Routing

The routing component instances execute a routing algorithm. At each node, the attached routing component builds and maintains a **routing table**, which specifies a next-hop for each destination. The next-hop can take the value *nil* (e.g. because the destination is not reachable from the node). The routing component also maintains additional information that is specific to the algorithm (e.g. cost of the route via each neighbor, costs of remote links). The routing components regularly exchange information in the form of **routing packets**; more precisely, a routing component passes a routing packet to its local node, which then sends it to a neighboring node (via the link between them), which then passes it to its local routing component. A routing component has input parameters such as the topology broadcast period for SPF (see Section 6), and output parameters such as the routing table.

Workload

The workload components model the user traffic in the target system. In each workload source-sink pair, the source component produces **workload packets** and passes them to its node component. These packets are then forwarded by the node components (using their next-hops) until they reach their destination node, where they are consumed by the sink component. In FTP and TELNET workloads, each source-sink pair goes through a succession of connections. Each connection incorporates a send-window flow control mechanism and an acknowledgment-with-retransmission mechanism[2] using roundtrip time estimates. The Simple_Traffic workload does not involve connections, flow control, or acknowledgment-and-retransmission mechanism. Here, the source produces packets with interpacket times distributed either exponentially or uniformly.

Workload input parameters include average number of packets per connection, average delay between connections, packet size, delay between packets, window sizes, etc.; and output parameters include instantaneous throughput and delay.

Node

A node component has a separate queue of packets for each outgoing link, and a single common queue for all routing packets received on its incoming links. The node passes queued routing packets to the local routing component. For each incoming workload packet that is not destined for a local sink, the node consults the local routing table and appends the packet (after a processing delay) to an appropriate outgoing queue. A packet generated by a local source is treated similarly. In the outgoing links, routing packets have priority over workload packets. A received packet is dropped if there is no available buffer space (and this depends on the packet priority) or if it is a workload packet whose destination's next-hop is *nil*.

[2] Retransmissions are needed because packets can be lost due to node or link failures, buffer space limitation, or *nil* next-hop.

A node can be subject to failure and repair, with up-times and down-times described by distributions. When a node fails, all packets that are stored in it are lost.

Input parameters of a node component include total buffer space, distribution of up/down times, etc., and its output parameters include node status, buffer utilization, queue lengths, etc.

Performance monitor

The performance monitor component collects and evaluates statistics about the network. The performance measures provided by MaRS are measured in one of two ways: periodically-updated and event-updated. A periodically-updated performance measure is updated periodically at time instants separated by a fixed length interval called "update period"[3]. An event-updated performance measure is updated whenever a related event occurs. For example, the total network throughput is updated periodically, while the current number of failed links in the network is updated whenever the status of a link changes from *up* to *down* or vice versa.

A performance measure can be either an average measure or an instantaneous measure. An "average measure" is computed based on statistics collected after a specified "startup interval" from the beginning of the simulation (to reduce transient effects due to the network being initially empty). An "instantaneous measure" is computed based only on current state or statistics collected during the last update period. For many of these measures, time variance is also calculated.

Periodically-updated performance measures include (average or instantaneous) network throughput, delay per data packet, data load, routing load, number of dropped workload packets, etc. Event-updated performance measures include current number of active connections, average number of data packets in an FTP/TELNET connection, average life-time of an FTP/TELNET connection, maximum data packet delay, etc.

[3] Note that this period is different from the link-cost update period.

4 Interface Design and Implementation

4.1 User Interfaces

Currently MaRS provides a **command line interface**. In addition, MaRS can be compiled with one of two optional graphical interfaces: a **simple X interface** (the user interface that came with NetSim) or an **X-Motif interface**.

The command line options include the specification of the length of the simulation run, the seed for the random number generator, the name of the network file, the name of the record file, the length of the startup interval, and the performance monitor update period (network file and record file are explained below).

The simple X interface [12] is almost entirely mouse-driven. The display is divided into three parts: **control section**, **network display**, and **text window**. The control section is comprised of a clock and a set of control buttons which allow the user to load and save network files (described in Section 4.2), start and stop the simulation, change the execution mode of the simulator (continuous time, event step, time step), record events, etc. The user works in the network display to create, delete, move and connect component instances. Every component instance is created with a set of default parameters. To display the parameters, the user clicks on the component instance and a component information window opens up. Further clicking on a specific parameter allows the user to modify its value (if it is an input parameter) by entering a new value through the text window. Parameter values of a component can be logged into a file or displayed with a meter. The text window is used to enter parameter values and to display error messages and warnings.

The X-Motif interface [14] provides some additional functionality over the simple X interface. Components can be clustered and displayed as groups. The grouping option provides the user with the possibility to display a more structured network, e.g. a hierarchical network. The size of the network display is dynamic and the network display can be scrolled. The meters are displayed in a separate window (the MaRSMeterWindow). The X-Motif interface is easier to use and is more customizable.

With both of the graphical interfaces, output parameters can be displayed with a meter by clicking on the meter field of the parameter in the component information window. Currently there are eight meter types available in MaRS: binary meter, bar graph, log, time history A, time history D, delta meter, histogram and text meter. The most commonly used meter types are time history A, time history D and text meter. Time history A displays the parameter value periodically versus time, whereas time history D displays the value (versus time) only when the parameter value changes. The text meter is used to display various tables such as routing tables. A detailed description of all meters can be found in [12]. The meter type and meter parameters can be changed by clicking on the meter and entering the new values. The meter window can be moved, resized, raised and lowered by clicking the appropriate mouse buttons.

A graphical interface is very helpful in building a network, debugging a routing algorithm, etc. However, it slows down the simulation considerably. For this reason, only the command line interface should be used when speed is important and the network display is not needed. Even when MaRS is compiled with the simple X interface or the X-Motif interface, it can still be executed with the command line interface and without the graphical interface.

4.2 Files

MaRS works with four ASCII files; referred to as the **network file**, the **snap file**, the **record file** and the **log file**. We describe these next.

A network file includes a description of the network to be simulated. It consists of three distinct parts. The first part is a list of all component instances with their input parameter values. The second part indicates the interconnection of the instances. The third part lists the peer components (e.g. the source-sink workload pairs). The network file can be created either manually, or automatically by clicking on the save button when operating MaRS graphically. When running MaRS, the network file can be specified on the command line, or by using the load option of the graphical interfaces. The network file includes all information needed to carry out a simulation. Because the X-Motif interface provides additional functionality, some additional information is stored in a separate network file (named `net.motif` if the network file is named `net`). By storing this information in a separate file, the network file (`net`) is compatible with all MaRS interfaces.

A snap file is basically a network file, with the following additions: (1) the values of current output parameters; (2) the seed for the random number generator; (3) the time of the snapshot; and (4) flags to indicate the parameters that have meters or are logged. A snap file is created automatically after any simulation for which a stop time is specified, or by clicking on the snap button when operating MaRS interactively.

We often need to simulate different routing algorithms subject to the same sequence of event occurrence times for a special set of events. For example, to compare how different routing algorithms respond to failures and repairs (of nodes and links), we want simulations of the different algorithms where the sequence and occurrence times of failure and repair events is the same in all simulations. To achieve this, we cannot generate failure and repair events in the usual way (using random numbers). We classify a subset of the events, including link/node failures and repairs, connection coming up, etc., as **record-class events**. MaRS provides **record** and **play** options which can be specified from the command line or from a graphical interface. With the record option, the occurrences of record-class events are written to a record file (the record file can also be created manually). With the play option, the record-class events are read and scheduled from the record file. Each entry in a record file specifies an event, its occurrence time, the name of the concerned component, and other relevant parameters. Using this feature, one creates a record file with the first run of a series of simulations, and in all other runs the record-class events are played from this file. The record/play-back capability does not exist in NetSim.

MaRS can write the values of selected parameters to a log file. The header of the log file contains a list of the parameters logged. The parameters are assigned unique numbers. A parameter value entry consists of the unique parameter number, the timestamp at which the parameter value was logged and the actual parameter value. The data written in the log file can be used for further post-processing and analysis of the simulation.

The file interface is the same as in NetSim, except for the parts that deal with record/play-back capabilities and X-Motif interface's parameters.

5 An Example Network Configuration

To illustrate the use of MaRS we present a small example. To start MaRS, a user types “mars” on the command line and hits return. A graphical interface pops up, in this example, the X-Motif interface. Clicking on the create option in the edit menu opens up a window displaying all available components. Dragging any component out of this window creates an instance of that particular component with default parameters.

The components that define a physical network are the node component and the link component. Any arbitrary physical network can be built by creating node and link component instances and connecting them (by clicking on the components). When a link is connected to a node, the link represents an outgoing and incoming channel of the node. Figure 3 illustrates a physical network with five nodes and six links (e.g. link between nodes 3 and 4 is identified as 3-4).

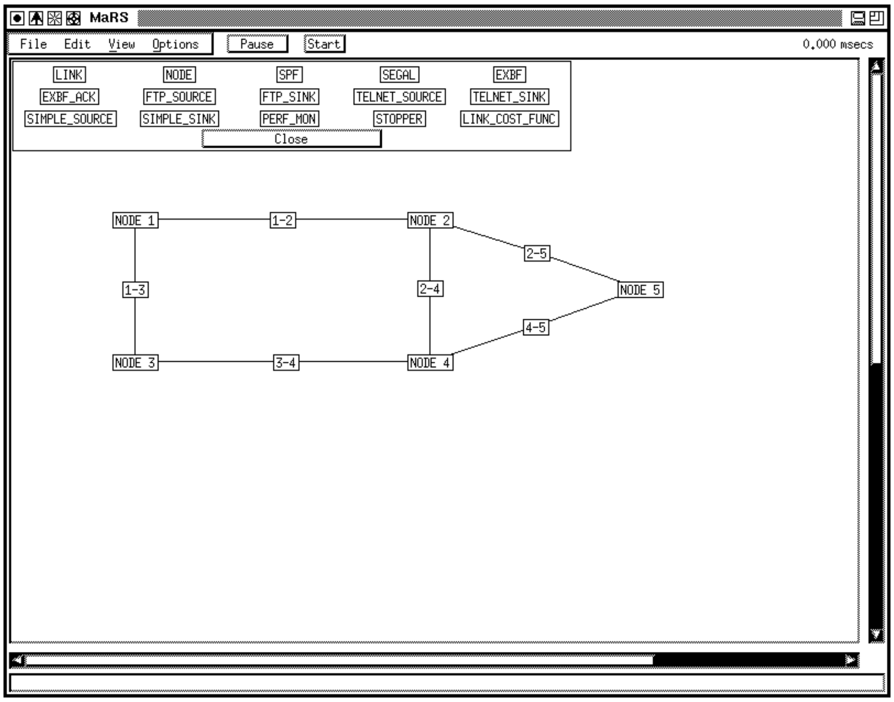

Figure 3: Physical network.

To add a routing protocol to the physical network, a routing component instance has to be created for each node component instance and connected to it. The routing components that are currently available in MaRS are SPF, ExBF and SEGAL. All routing component instances in the physical network have to be of the same type. In addition a link–cost function component instance has to be created. It is not connected to any other components; it provides information about the link–cost function to all routing components.

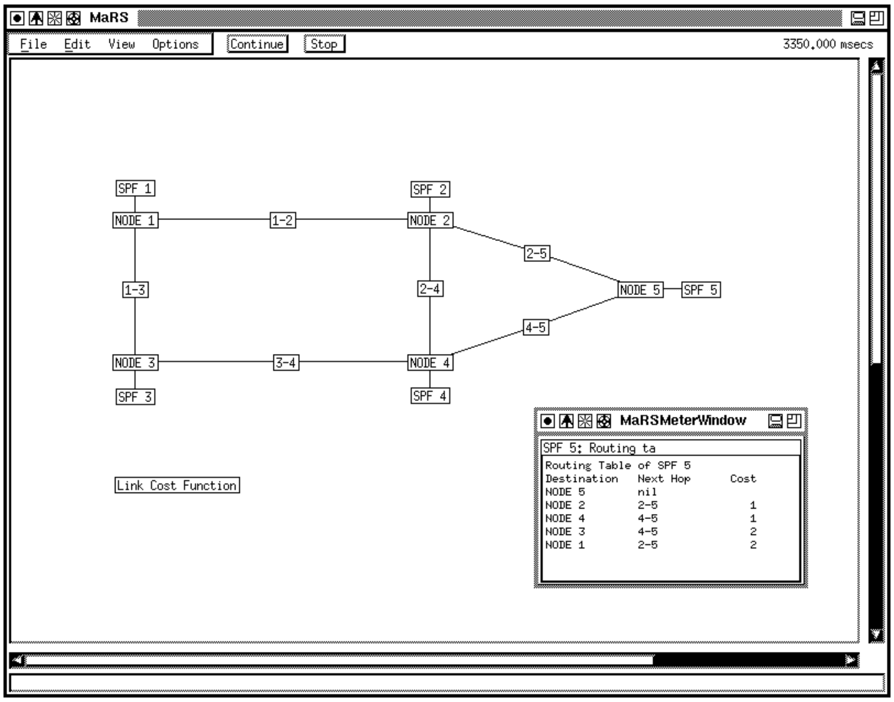

Figure 4: Physical network with routing (after 3.350 seconds).

In Figure 4, SPF is chosen as routing algorithm. The link–cost component is displayed in the left lower portion of the figure. By clicking on the start button, the simulation is started. Figure 4 shows the simulation after 3.350 seconds (time is shown in the upper right corner of the display border). In the right lower portion, the routing table of *SPF 5* displays entries for all nodes in the physical network.

The workload components that are currently available are FTP_SOURCE, FTP_SINK, TELNET_SOURCE, TELNET_SINK, SIMPLE_SOURCE and SIMPLE_SINK. Workload source and sink components are connected to node components; for each workload source there has to be a workload sink. To inform a workload source about its workload sink (and vice versa), the two components have to be made peers by appropriately clicking on them. Figure 5 shows four workload source component instances and four workload sink component instances.

To obtain performance measures for the network simulated, an instance of the performance monitor has to be created. The performance monitor is not connected to any other component.

Figure 6 shows the performance monitor together with some of its output parameters, and meters for two performance measures. The parameters of any component can be displayed by clicking on the component. Further clicking on the Meter option (M) of the parameter creates a meter for the parameter. Clicking on the Log option (L) causes the parameter values to be written to a log file. The parameters shown in the meters are the instantaneous network throughput and the instantaneous average delay per packet. At this point the simulation has progressed for 4310.760 milliseconds.

6 Implementing a Component

In this section, we describe how to implement a new component in MaRS, and illustrate with the SPF component. This description in most parts applies to NetSim as well. We explicitly state additional requirements and features of MaRS that do not exist in NetSim. We also describe how components interact with each other and with the performance monitor. For brevity, we describe the highlights only. In many places, we replace C code by English sentences, omit variables, and use different but more descriptive names for variables. A more complete (and accurate) description of a routing component implementation can be found in [15].

We first describe the SPF routing protocol. Each node maintains a view of the network topology with a cost for each link. To keep these views up-to-date, each node periodically (or whenever a failure/repair occurs) broadcasts the link costs of its outgoing links to all other nodes using flooding (i.e. it sends link cost information to its neighbors, they forward this information to their neighbors and so on). As a node receives this information, it updates its view of the network topology and applies Dijkstra's shortest path algorithm[16] to choose its next-hop for each destination. Note that because of the store-and-forward nature of the network, a node may receive old link cost information after it has received new information. To avoid using old information, SPF uses sequence numbers.

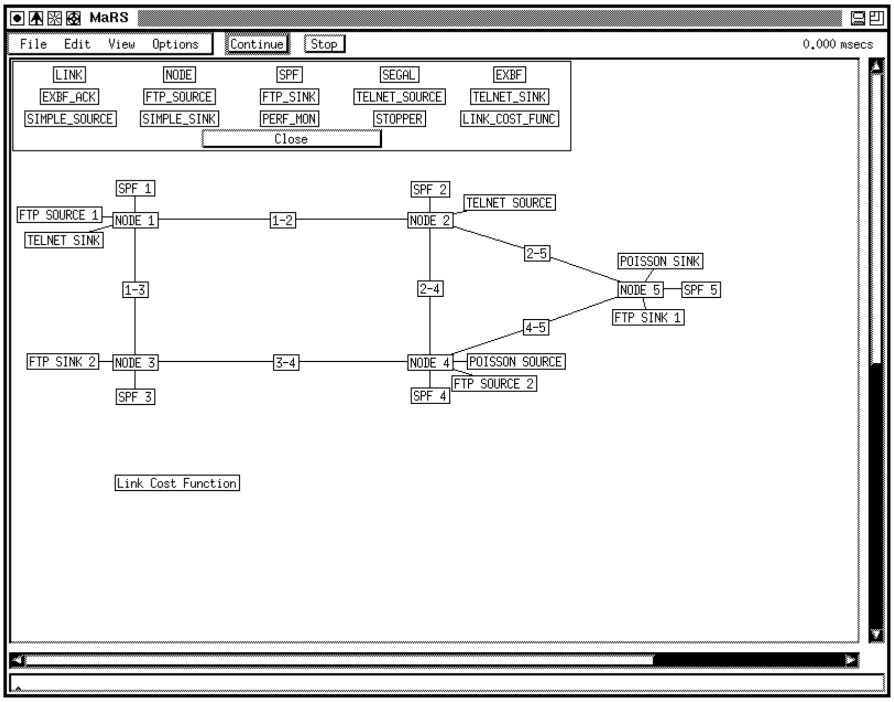

Figure 5: Physical network with routing and workload.

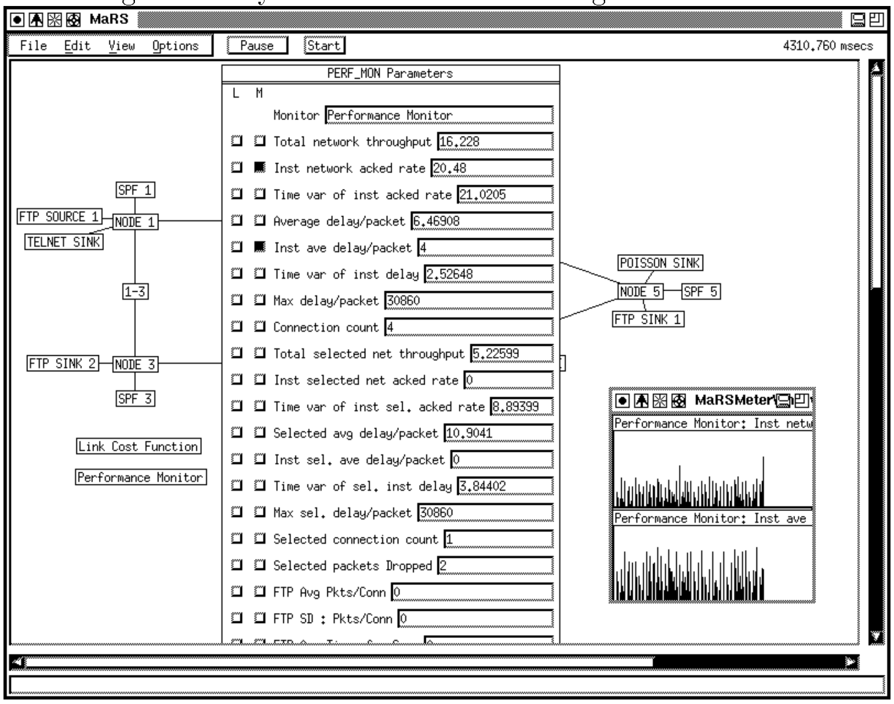

Figure 6: Physical network with routing, workload and performance monitor.

A component implementation consists of a **data structure** declaration and an **action routine**.

```
typedef struct _SPF_t {

/* Fields common to all components */
  struct _SPF_t   *next, *prev; /* To make link list of components */
  short           class;        /* Class of component (``Routing'' for SPF) */
  short           type;         /* Type of component (``SPF'' for SPF) */
  char            name[40];     /* Name of component instance */
  PFP             action;       /* Pointer to action routine */
  GRPH_OBJECT     picture;      /* Graphics object displaying component instance*/
  list           *neighbors;    /* List of neighbor component instances */
  short           menu_up;      /* Text window up flag */
  queue          *params;       /* Queue of parameters */

/* Fields common to all routing components */
  PFI             processing_time;    /* function estimating proccesing time of routing
                                         events; used to schedule routing events */
  int             no_of_nodes;        /* number of nodes in the network known
                                         to this SPF instance */
  unsigned int    link_cost_time;     /* time local link costs are last calculated */
  Component       *node;              /* pointer to local node */
  Param           *routing_table;     /* routing table: stores the next-hop
                                         and the cost for each destination, */
  Param           *local_topology;    /* local topology table: stores link costs and
                                         necessary statistics for each outgoing link */

/* SPF specific fields */
  Param           *global_topology;   /* view of the network topology */
  Param           *seq_no;            /* message sequence no. last used
                                         by this SPF instance */
  Param           *seq_no_table;      /* last sequence no. received
                                         from other SPF instances */
  Param           *brdcast_period;    /* average broadcast period length */
  Param           *brdcast_period_sd; /* broadcast period standard deviation */

} SPF_t;
```

Figure 7: SPF data structure.

The data structure contains fields needed for the operation of the component and for the operation of the simulation engine. The action routine specifies the code to be executed when an event for this component occurs. The events either model the component's behavior (e.g. send, receive, etc.) or are needed by the simulation engine (e.g. instance creation, connecting to a component, etc.).

Data Structure

The data structure of SPF is shown in Figure 7. It contains three sets of fields. The first set of fields is common to all components. These fields are needed to interface the component with the simulation engine and the user interfaces. The second set of fields is common to all routing components (hence these fields are not required by NetSim). These fields are shared with other components; e.g. the routing table is shared with the local node. This set is empty for non-routing components. The last set of fields is specific to the SPF component.

Each parameter of a component must be declared to be of type `Param` structure, and added to the `params` queue. The `Param` structure contains the following: fields indicating the parameter's value, either integer, double or pointer; flags indicating the parameter's type (i.e. input/output) and capabilities (e.g. can it have a meter or be logged); status flags (e.g. dirty bit indicating whether the parameter is being logged); and fields pointing to appropriate functions to input, log, display, etc. the value of the parameter. The parameters of a component can be accessed in two ways: by traversing the `params` queue (this is how the simulation engine accesses the parameters); and by defining other fields (e.g. `routing_table` in Figure 7) in the data structure of the component that point to the entries in the `params` queue.

Interaction with other Components

The simulator time is maintained in units of "ticks" in an unsigned 32-bit integer variable. Currently, one tick is ten microseconds, so the simulator can run for almost 12 hours of simulated time before an overflow occurs. The components can access the current time by using the `ev_now()` function.

Interaction between component instances is usually either by *calling* events or by *scheduling* events. The function `ev_enqueue()` schedules an event e by enqueueing the pair (e, t) in the event list where t is the event occurrence time. When the simulated time equals t, the action routine of the appropriate component is invoked. The function `ev_call()` executes the appropriate action routine *immediately*, after which control returns to the calling routine. Note that scheduling f with the current value of simulated time corresponds to "asynchronously" calling the action routine of f. The headers of `ev_enqueue()` and `ev_call()` are given in Figure 8. The `ev_call()` function

```
int ev_call(event, source, dest, action, packet, argument)

Event *ev_enqueue(event, source, dest, time, action, packet, argument)

   int          event;          /* Type of event to be fired */
   Component   *source,         /* The component instance causing the event */
               *dest;           /* The component instance the event is for */
   unsigned     time;           /* Time of event firing */
   PFP          action;         /* Address of the action routine of dest */
   Packet      *packet;         /* optional, packet to pass */
   caddr_t      argument;       /* optional, extra arguments */
```

Figure 8: Interface.

does not exist in NetSim. We have observed that for many events, an occurence at time t schedules new events whose occurence time is also t. With `ev_enqueue()` function, the simulation engine inserts these event-time pairs into the event list, sorts the event list, and then removes and fires these events. By introducing the `ev_call()` function, this scheduling overhead is reduced to just firing events.

Components can also interact by sharing data structures; typically, only one of the components has write access and all other components have read-only access. Sharing data structures is very limited, and done only when it is significantly more efficient than calling or scheduling events.

Packets

If any additional message types are introduced by the new component they should be defined in the `packet.h` file. The SPF component uses link state packets (of type `LINK_STATE_PKT`), which contains the link costs of the outgoing links of the sender. In addition, the failure/repair of the local node and its links are reported to the routing component by the packets of the following special types: `NODE_SHUTDOWN_PKT`, `NODE_WAKEUP_PKT`, `LINK_SHUTDOWN_PKT`, and `LINK_WAKEUP_PKT`.

```
caddr_t spf_action(source, spf, event, packet, argument)
     Component *source;         /* Component that scheduled or called this event */
     register SPF_t *this_spf;  /* Component to which this event applies */
     int event;                 /* Type of event */
     Packet *packet;            /* A packet (if required) */
     caddr_t argument;          /* Other parameters (if required) */
{
  switch (event)  {  /* A big switch statement on the event type */

    /* Events generated by the simulation engine and user interfaces */
    case EV_CREATE_INSTANCE: <event handler>
    case EV_DELETE_INSTANCE: <event handler>
    case EV_CONNECT: <event handler>
    case EV_DISCONNECT: <event handler>
    case EV_RESET: <event handler>
    case EV_STOP: <event handler>
    case EV_START:
      spf_start(spf); break;
    case EV_STOP: <event handler>

    /* Events generated by components */
    case EV_SPF_BROADCAST:
      spf_broadcast(spf); break;
    case EV_RECEIVE:
      process the received packet; break;
    }

  return non-null if success;
}
```

Figure 9: SPF action routine.

Action Routine

The action routine is invoked whenever an event for this component is fired. The action routine specifies the code to be executed for each event related to the component. The action routine must return `NULL` if an error occurs, and `non-NULL` otherwise.

The action routine for SPF is specified in Figure 9. The arguments are standard for other types of components. The code consists of a big switch statement with one case for each event. The events are divided into two sets. Events from the first set (i.e. `EV_CREATE_INSTANCE` to `EV_STOP`) are generated by the simulation engine and the user interfaces. The event handlers for these events are standard for the most part. For example, on `EV_CREATE_INSTANCE`, the memory for the data structure of the instance and the memory for the parameters is allocated and initialized. On EV_RESET, parameters are reset to their initial values. On EV_START, error conditions are checked, if a periodic event exists for the component then its first occurrence is scheduled, etc. Events from the second set (i.e. EV_SPF_BROADCAST and EV_RECEIVE) are generated by component instances. The event handlers for these events capture the behavior of the target system simulated by this component.

We expand the event handlers of EV_START and EV_SPF_BROADCAST in Figure 10. In spf_start (the event handler of EV_START), note that the EV_SPF_BROADCAST event is scheduled rather than called even though its time (given by ev_now()) is 0. This ensures that when the broadcast is actually done all components have been started.

The event handler of EV_SPF_BROADCAST, spf_broadcast(), first schedules the next EV_SPF_BROADCAST event (remember that the broadcasts are periodic). It checks the status of the local node, and cancels the broadcast if the node is down. If the node is up, the link costs of the outgoing links are computed using the statistics collected by the local node and the link cost function as specified by the link cost component. Then, a broadcast is done by creating a link state packet for each outgoing link, and passing it to the node for transmission (by *calling* EV_NODE_PRODUCE event). Finally, new next–hops (and costs) are calculated by applying the Dijkstra's shortest path algorithm. In the EV_NODE_PRODUCE call in ls_broadcast(), note that the outgoing link is also specified as an argument to the EV_NODE_PRODUCE event. On EV_NODE_PRODUCE event with the non-NULL argument, the local node skips the routing table lookup and uses the argument as the link to take. This is needed because the shortest path for the neighbor might not be through that link.

6.1 Interface to the Performance Monitor

The performance monitor component (PERF_MONITOR) provides performance measures to evaluate and compare different routing algorithms (this component does not exist in NetSim). To calculate these measures, the performance monitor needs to obtain information from the components of the target system. The performance monitor gets this information either by read accessing the data structure of the components (e.g. by polling), or by an interface function pm (see Figure 11), which

```
static caddr_t spf_start(spf)
register SPF_t *spf;
{
   check for error conditions (e.g. no local node is connected);
   /* schedule the first EV_SPF_BROADCAST event */
   ev_enqueue(EV_SPF_BROADCAST, spf, spf, ev_now(), spf->action, NULL, NULL);
   return success;
}

static caddr_t spf_broadcast(spf)
register SPF_t *spf;
{
   next_time = calculate next broadcast time using spf->brdcast_period
                  and spf->brdcast_period_sd;
   ev_enqueue(EV_SPF_BROADCAST, spf, spf, next_time, spf->action, NULL, NULL);

   if (the node connected is down)
      return success;

   /* node is UP */
   compute_link_costs(spf);  /* computes the new costs of the outgoing links */
   ls_broadcast(spf);        /* broadcast the new link costs */
   Dijkstra_spf(spf);        /* update next-hops and costs using Dijkstra's
                                shortest path first algorithm on spf->global_topology */
   return success;
}

static caddr_t ls_broadcast(spf)
register SPF_t *spf;
{
   for (link in spf->local_topology && link is UP) {
      create a packet of type LINK_STATE_PACKET;
      fill the packet with the link costs from spf->local_topology;
      /* pass the packet to the node; it will do the sending */
      ev_call(EV_NODE_PRODUCE, spf, spf->node, spf->node->action, packet, link);
   }
}
```

Figure 10: `spf_broadcast()` and `ls_broadcast()` functions.

is called by the components in an event-driven way. Some of the requests and their effects are the following:

`ACK_RECEIVED` The invoker is an application/transport source which received an acknowledgment packet. In this case, `arg1` contains the number of data bytes that are acknowledged, and `arg2` contains the roundtrip delay of the packet. The statistics for throughput, and delay are

```
void pm(comp, comp_type, req_type, arg1, arg2, arg3)
   Component *comp;              /* id of the invoking component */
   int        comp_type;         /* type of the invoking component */
   int        req_type;          /* request type */
   int        arg1, arg2, arg3;  /* additional information if required */
```

Figure 11: The syntax of pm.

updated.

LINK_FAILURE (LINK_WAKEUP) The invoker is a link that has failed (become repaired). The current number of failed links in the target system is updated.

ROUTING_PKT The invoker is a routing component which created some routing packets. In this case, arg1 contains the number of routing packets created. The total number of routing packets used is updated.

NEW_COMPONENT The invoker is just being created. This request is needed for efficient polling.

It is the responsibility of the programmer to put appropriate calls to pm into the code. Otherwise, the performance measures calculated by performance monitor may not be correct.

The performance monitor also collects information by polling (e.g. data load and routing load are calculated by polling the link components). Performance monitor maintains *poll lists* in order to avoid polling and checking all components while only some components needs to be polled. When a component invokes pm with a NEW_COMPONENT request, it is added to the appropriate poll lists depending on its type.

7 Some Results using MaRS

We have found MaRS to be very useful in understanding the complex dynamics of routing algorithms. As described earlier, a routing algorithm may form loops in its routes while adapting to topology changes, which degrades performance. In [1, 17], we compared the performance of two new distance-vector routing algorithms, SEGAL [10, 11] and ExBF [9], and a link-state algorithm, SPF [8]. SPF and ExBF allow short-lived loops, whereas MS avoids loops entirely. (In response to a workload or topology change, to avoid loops MS starts a diffusion computation during which nodes keep their old routes until they obtain new loop-free routes. In case of a failure, affected nodes reset their next-hops to *nil* during the diffusion computation.)

In [1], we considered the NSFNET-T1-backbone network with varying uniform and skewed workloads and failure distributions. Our overall conclusion was that ExBF is as good as SPF for good link cost functions, whereas SEGAL is worse. In [17], we examined the relationship between "workload-independent" measures (such as number of messages needed to adapt routes after a link failure) and "workload-dependent" measures (such as delay and throughput). We examined both transient characteristics and steady-state characteristics. Our overall observation was that workload-independent measures are not good indicators of workload-dependent measures. In the rest of this section, we present some of our results from [17].

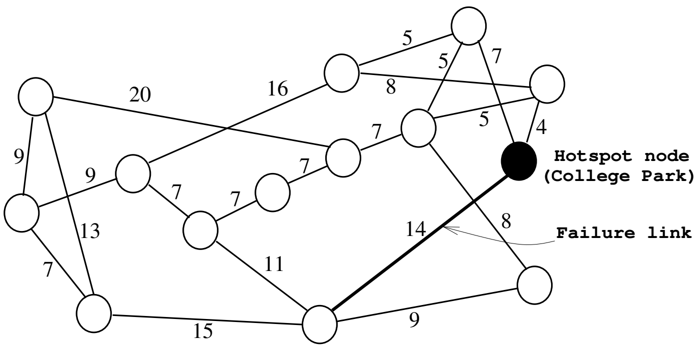

Figure 12: NSFNET-T1-Backbone (14 nodes, 21 bidirectional links, average degree 3).

The physical network models the NSFNET-T1-Backbone illustrated in Figure 12. Link propagation delays in milliseconds are indicated. All links have the same bandwidth, 1.5Mbit/sec. All nodes have adequate buffer space for buffering packets awaiting processing and forwarding. Workload packets can be processed in parallel. The processing time for workload packets equals 1 msec.

We compare SPF, MS and ExBF by examining the behavior after a link failure (at time 100 seconds) and repair (at time 200 seconds). The failure link is indicated in Figure 12. The link cost function used in this paper is the hop-normalized-delay function [13].[4]

[4] It monitors the *average packet delay* (queueing and transmission) and *average packet transmission time* for the link during the last update period. Assuming an $M/M/1$ model, it calculates the link utilization. The link utilization is first exponentially averaged with the previous value of the average link utilization. The result is scaled with a

Routing packets received at a node are processed sequentially. The processing time of an SPF packet is 6msecs, the processing time of a MS packet is either 2msecs or 3msecs depending on the packet type, the processing time of an ExBF packet is 4.5msecs.

We consider a skewed workload referred to as *hotspot-with-background workload* with parameters H and B, where H is the average number of FTP connections from every node to the hotspot node (hotspot node is indicated in Figure 12), and B is the average number of FTP connections between every pair of nodes excluding the hotspot node. Each connection has data packet size equal to 512 bytes, inter-packet generation time equal to 150 msec, and window size for flow control equal to 8 packets. All connections start when the simulation begins and are unending.

We observe transient performance measures such as overshoots, undershoots, and recovery times of various instantaneous measures after the link failure and repair.[5] We define *recovery time* of an instantaneous measure to be the time needed for the measure to reach its steady-state value after a link failure, or a link repair. The overshoot (undershoot) of an instantaneous measure is its maximum (minimum) value above (below) the steady-state value during the recovery time.

Figure 13 shows the instantaneous delay, throughput, and routing load versus time, for $B = 1$ and $H = 4$. Figure 14 shows the same for $B = 1$ and $H = 7$.

We begin our observations by examining the behavior of instantaneous delay. Before the failure (at time 80 seconds), it is around a steady-state value (e.g. 75 msecs for $H = 4$). As a result of the link failure at time 100 seconds, it increases to a new steady-state value (e.g. 95 msecs for $H = 4$). The recovery time of the instantaneous delay after the failure is small (less than 1 second for $H = 4$; 7 seconds for $H = 7$). There are no undershoots and overshoots for $H = 4$. For $H = 7$, there is an undershoot followed by an overshoot. When the link repairs at time 200 seconds, the instantaneous delay shoots up and decreases slowly to the same steady-state value before the link failure. Similar observations can be drawn for instantaneous throughput.

Our explanation for the high overshoots and large recovery times after the link repair is that

slope and an offset and then subjected to minimum and maximum bounds. Then its value is adjusted so that the difference from the previous value of the link cost is less than a movement limit. The resulting number is the new cost of the link. The parameters of hop-normalized-delay function used are as follows: offset equals 0, slope equals 10, minimum limit equals 1, maximum limit equals 10, and movement limit equals 1. The link-cost update period is uniformly distributed with mean 10 seconds and standard deviation 1 second.

[5] An *instantaneous measure* is a function of time updated periodically every 500 msec.

the repaired link becomes very attractive to many source-destination pairs since it reports a low cost.

Regarding the behavior of instantaneous routing load, after the link failure, it shoots up and falls down to the same steady-state value as before. The same happens after the link repair as well. The recovery time in instantaneous routing load for ExBF can be as high as 50 seconds whereas it is less than 5 seconds for SPF and MS. We observe no relation between the behavior of instantaneous routing load and the behaviors for instantaneous delay and throughput.

In spite of allowing short-lived loops, SPF and ExBF perform better than MS in all the simulations (also with other H and B values) we have performed. The diffusion computation mechanism used by MS to avoid short-lived loops degrades its performance (e.g. causes MS to drop packets). This implies that short-lived loops are preferable.

8 Concluding Remarks

MaRS provides a flexible platform for the evaluation and comparison of routing algorithms. Several people are extending MaRS in various ways. It has been extended [18] to study type-of-service routing with two traffic classes: minimum delay and maximum bandwidth. For each outgoing link, there are two queues, one for each traffic class; a round-robin scheduling discipline is used to share the link between the two queues. The routing table is modified to maintain different routes for each traffic class. MaRS is being extended [19] to study multicasting. There is also recent work in building a commercial routing testbed based on MaRS [20].

The memory required by MaRS to simulate a network is linear in the number of nodes and links, the number of connections, and the number of packets in transit; the latter is limited by the number of connections and the window size. Thus the memory space is linear in the number of nodes, links, and the product of connections and window size.

The CPU requirement of MaRS is best indicated by the number of events processed per second of simulated time. This too is linearly bounded by the number of nodes, links, and connections-window size product. One reason that the number of links and nodes comes in to the bound is the periodic calculation of some statistics. The connections-window size product comes in to the bound because each packet transmission or reception is an event.

The performance realised in practice also depends on the machine architecture. We present numbers for two architectures: a Sun Sparc 10/30 with 32M bytes of memory, and a DECStation 5000 with 16M bytes of memory. A 500-second simulation of NSFNET with 182 connections (one connection per every pair of nodes) required 2.2M bytes of memory; it took 2287 seconds of user time on the Sparc and 1926 seconds of user time on the DECStation. A 500-second simulation of NSFNET with 1092 connections (six connection per every pair of nodes) required 10M bytes of memory; it took 4453 seconds of user time on the Sparc and 3847 seconds of user time on the DECStation. There were no page faults and swap outs, except for the experiment with 1092 connections on the DECStation, where there were 35000 page faults and 14 swap outs (other programs were running on these machines). Note that even though the workload increased by a factor of 6 (from 182 connections to 1092 connections), the CPU requirement increased by a factor of 2 only.

Through usage, we have become aware of some of the limitations and deficiencies of MaRS. The error reporting is inadequate; for example, if all routing component instances are not of the same type, MaRS crashes without reporting an error. There is no mechanism to dump the core at the end of a simulation run, such that a new run of MaRS can be initialized to resume the simulation from the original run's last state.

We are currently investigating several extensions to MaRS, including a distributed simulation engine (to simulate large networks more efficiently), hierarchical routing algorithms with type-of-service and policy requirements, new traffic characterizations to model video and audio connections, nodes with packet scheduling capabilities (to guarantee real-time delivery), and links modeling ethernets and token-ring networks.

9 Acknowledgments

We gratefully acknowledge the use of the structuring concepts, its simulation engine code, and the simple X interface code of the MIT NetSim. We thank Eric Bull for implementing the X-Motif interface. We thank Jean Bolot, Ólafur Gudmundsson, and Ashok Agrawala for various discussions

during the design and implementation of MaRS.

References

[1] A.U. Shankar, C. Alaettinoğlu, I. Matta, and K. Dussa-Zieger. Performance Comparison of Routing Protocols using MaRS: Distance-Vector versus Link-State. In Proc. *ACM SIGMETRICS/PERFORMANCE*, volume 20, pages 181–192, Newport, Rhode Island, June 1992.

[2] Lee Breslau. Electronic mail, October 1991. Department of Computer Science, University of Southern California.

[3] J. M. McQuillan, G. Falk, and I. Richer. A Review of the Development and Performance of the ARPANET Routing Algorithm. *IEEE Transactions on Communications*, COM-26:1802–1811, Dec 1978.

[4] A. Heybey. The Network Simulator. Laboratory of Computer Science, Massachusetts Institute of Technology, October 1989.

[5] CACI Products Company. COMNET II.5 Overview. Available from CACI Products Company, March 1990.

[6] M. Livny. DeNet Overview. Technical note, University of Madison-Wisconsin, November 1989.

[7] W. A. Parrette. *Motif Programming in the X Window System Environment.* McGraw-Hill, 1993.

[8] J. M. McQuillan, I. Richer, and E. C. Rosen. The New Routing Algorithm for the ARPANET. *IEEE Transactions on Communications*, COM-28(5):711–719, May 1980.

[9] C. Cheng, R. Riley, S. P. R. Kumar, and J. J. Garcia-Luna-Aceves. A Loop-free Bellman-Ford Routing Protocol Without Bouncing Effect. In Proc. *ACM SIGCOMM '89*, pages 224–237, September 1989.

[10] P. M. Merlin and A. Segall. A Failsafe Distributed Routing Protocol. *IEEE Transactions on Communications*, COM-27(9):1280–1287, September 1979.

[11] A. Segall. Advances in Verifiable Fail-Safe Routing Procedures. *IEEE Transactions on Communications*, COM-29(4):491–497, April 1981.

[12] C. Alaettinoğlu, K. Dussa-Zieger, I. Matta, and A. U. Shankar. MaRS (Maryland Routing Simulator) – Version 1.0 User's Manual. Technical Report UMIACS-TR-91-80, CS-TR-2687, Department of Computer Science, University of Maryland, College Park, June 1991.

[13] A. Khanna and J. Zinky. A Revised ARPANET Routing Metric. In Proc. *ACM SIGCOMM '89*, pages 45–56, September 1989.

[14] Eric Bull. MaRS (Maryland Routing Simulator) – Version 2.0 X-Motif Interface Manual. Available by anonymous ftp at `ftp.cs.umd.edu:pub/MaRS`, August 1992.

[15] C. Alaettinoğlu, K. Dussa-Zieger, I. Matta, Ó. Gudmundsson, and A. U. Shankar. MaRS (Maryland Routing Simulator) – Version 1.0 Programmer's Manual. Technical Report UMIACS-TR-91-107, CS-TR-2723, Department of Computer Science, University of Maryland, College Park, July 1991.

[16] E. Dijkstra. A Note on Two Problems in Connection with Graphs. *Numer. Math.*, 1:269–271, 1959.

[17] C. Alaettinoğlu, A. U. Shankar, K. Dussa-Zieger, and I. Matta. Responsiveness of Routing Protocols to Link Failures and Repairs. Technical Report UMIACS-TR-92–76, CS-TR-2928, Department of Computer Science, University of Maryland, College Park, September 1992.

[18] I. Matta and A.U. Shankar. Type-Of-Service in Adaptive Next-Hop Routing. Technical Report CS-TR-2963, Department of Computer Science, University of Maryland, College Park, September 1992.

[19] Danny J. Mitzel. Electronic mail, October 1992. Department of Computer Science, University of Southern California.

[20] Sam Steppel. Private Communication. Computer Sciences Corporation.

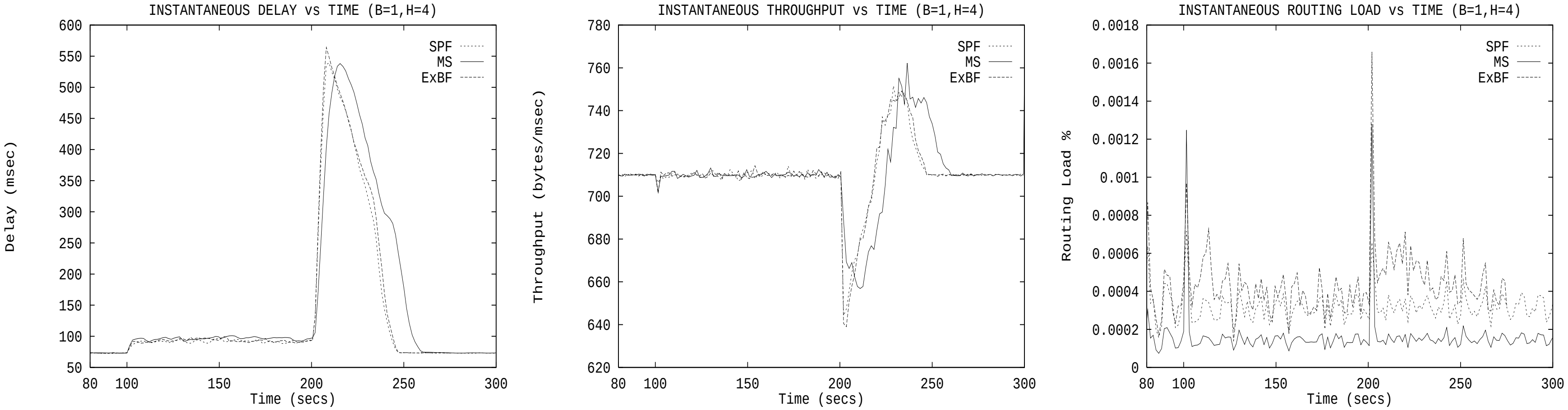

Figure 13: Responsiveness to a link failure and repair for $B = 1$ and $H = 4$. Each curve is average of 15 experiemnts.

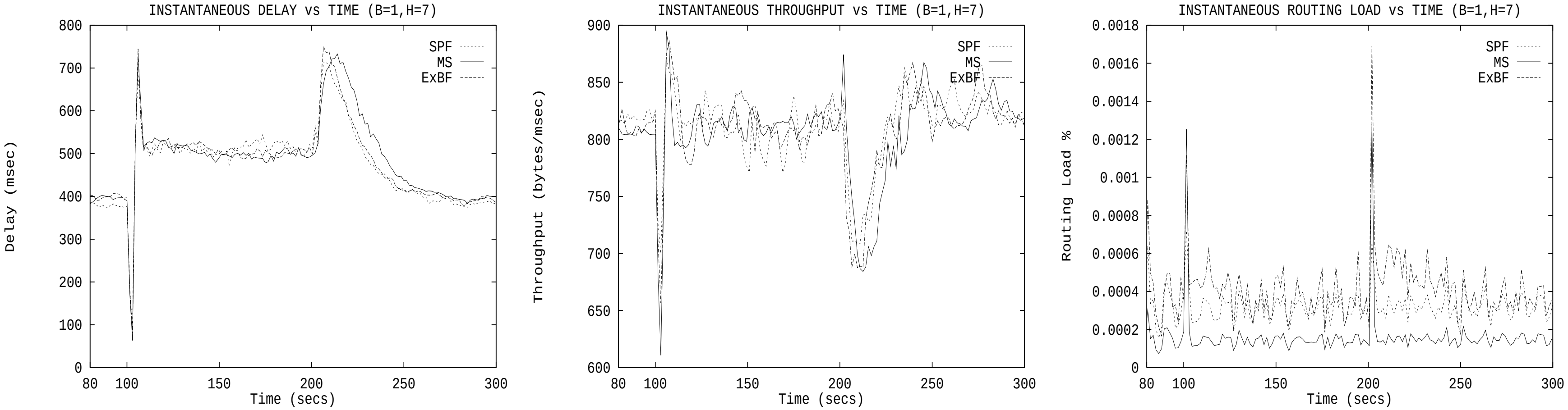

Figure 14: Responsiveness to a link failure and repair for $B = 1$ and $H = 7$.