MaRS (Maryland Routing Simulator) - Version 1.0 Programmer's Manual*

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

Systems Design and Analysis Group
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

UMIACS-TR-91-107
CS-TR-2723

July 1, 1991

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks]: Network Architecture and Design—*store and forward networks*; C.2.2 [Computer-Communication Networks]: Network Protocols—*protocol architecture*; C.4 [Performance of Systems]: *measurement techniques; modeling techniques; performance attributes;* I.6.3 [Simulation and Modeling]: Applications.

*This work is supported in part by RADC 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.

Contents

1 Introduction **1**

2 MaRS Directory Structure **4**

3 Components **4**
3.1 Data Structure 5
3.2 Action Routine 7

4 Parameters **8**

5 Events **10**
5.1 Command Type Events 11
5.2 Event Type Events 12
5.3 Private Type Events 14
5.4 Record Type Events 15

6 Event Manager **16**

7 Packet **17**
7.1 Packet Handling Routines 19

8 Programming New Components **19**
8.1 Interface to the Performance Monitor Component 21
8.2 Writing a New Routing Component 23
8.2.1 Data Structure 23
8.2.2 Routing Packet 25
8.2.3 Action Routine 26
8.2.4 Link Cost Handling Routines 27
8.3 Writing New Workload Components 29
8.3.1 Data Structure 29
8.3.2 Workload Packet 29
8.3.3 Action Routine 30

1 Introduction

MaRS (Maryland Routing Simulator) is a discrete-event simulator built to provide a flexible platform for the evaluation and comparison of network routing algorithms. We refer to the network to be simulated as the **target system**. The target system is characterized by discrete states and by state transitions (events) occuring at discrete time instants. Each event type has a corresponding action routine which updates the **system state** when that event occurs. The occurrence of an event corresponds to the end of the action routine. A system state is the collection of state variables necessary to describe the target system at a particular time.

MaRS views the target system as a set of components representing the physical network, the workload, and the routing algorithm, as illustrated in Figure 1.

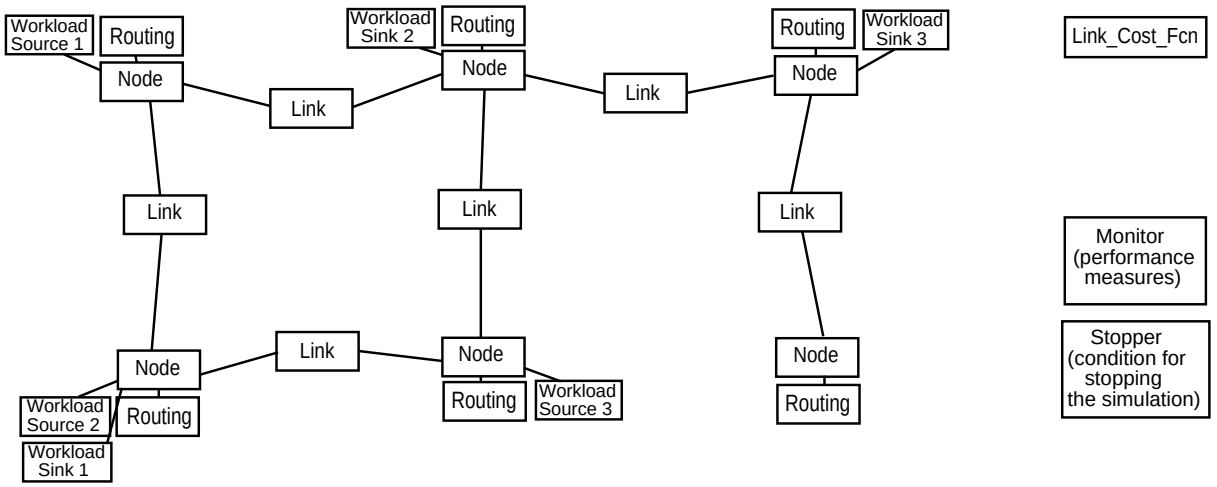

Figure 1: An example target system with performance monitor and stopper

The link and node components correspond to the **physical network**. A **link component** of type `LINK` models a bidirectional point-to-point transmission channel between two nodes. In the current version, this channel is modeled by two queues of packets, one queue for each direction. A link component is characterized by parameters such as bandwidth, propagation delay, and failure and repair distributions. A **node component** of type `NODE` models the "physical" aspects of a store-and-forward entity such as a host or switch. In the current version, it is characterized by parameters such as buffer space, processing speed, packet queueing discipline, and failure and repair distributions. Packets are transferred through node and link components from their source until they reach their destination. By connecting link components and node components, one can obtain a network of arbitrary topology.

A **routing component** must be connected to every node component, as shown in Figure 1. A routing component maintains and provides the routing information (e.g. routing table) needed by its attached node component. This enables the node component to forward packets towards their destination. To realize a particular type of routing algorithm, all routing components in the network must be of that type.

In order to compute paths, routing algorithms require a **link cost** to be associated with every link in the network. Currently, only one **link-cost component** of type `LCOSTFCN` exists that computes link costs based on one of the following criteria: hop count, utilization, delay, and hop-normalized delay. The link-cost component is not connected to any other component.

Workload is represented by **source-sink** pairs. Each source-sink pair consists of a **source component** and a **sink component** . These components are connected to node components (normally, the source's node component is different from the sink's node component).

We distinguish between the packets generated by the routing components, and by the workload components (also referred to as application/transport components) as follows. The term **routing packet** refers to packets that are generated by routing components. The term **workload packets** refers to packets generated by workload components. Workload packets can be further subdivided into **data packets**, **ack packets**, and **token packets**. Data packets carry data, ack packets carry acknowledgements to data, and token packets are used by the workload components for control purposes (e.g. roundtrip time estimate).

In addition to the above target system components, MaRS provides a performance monitor component and a stopper component.

Both the performance monitor component of type `PERF_MONITOR` and the stopper component of type `STOPPER`, like the link-cost component, are not connected to any other component. The **performance monitor component** computes network performance measures, e.g. average network throughput, average delay per packet, ..., etc. The **stopper component** allows one to define a termination condition for the simulation, based on the statistics collected by the performance monitor component during the simulation. Intuitively, the termination condition being true should indicate that the simulation has reached steady state for the performance measures of interest. In the current version, the stopper component is characterized by parameters such as a minimum number of elapsed intervals before termination is possible, length of the interval, and an error bound relating successively computed averages.

The components schedule events for one another. For example, for the transfer of a packet from a node component to a link component, the node component schedules an event that causes the link component to receive the packet and insert it in the appropriate link queue depending on which direction the packet is sent. Note that the occurrence of this event corresponds to sending the last bit of the packet on the link. Also, a component can schedule an event for itself. For example, once the link component receives a packet from one of the neighbor nodes, it schedules an event for the packet to be sent to the receiving node component (at the other end of the link). The time of this event to happen (to fire) depends on the link propagation delay, which is one of the link input parameters. Each component has an action routine which appropriately updates the system state and may schedule other events depending on the type of the event received by the component (i.e. scheduled for it).

The events that can occur in MaRS Version 1.0 are divided into four different classes: command type, event type, private type, and record type. All MaRS components have to react to the **command type events**. These events execute the basic operations of the simulator, e.g. create/delete a component. **Event type events** are defined globally in a central location. This class includes regular events such as the sending of a packet from a node component to a link component. **Private type events** are only defined internally within a component. They are only called within that component and are invisible to the other components such as the production of a packet by a workload source component. **Record type events** are meant for events that are recorded (i.e. written) to a record file[1]. The record file can be used to exactly repeat a sequence of event occurrences in different simulation runs, for example, one may use this recording option to compare different routing algorithms on different simulation runs subject to the same event occurrences, e.g. link and node failures and repairs.

This manual assumes background information about MaRS User's Manual [Alaettinoğlu *et al.*, 1991], computer networks [Tanenbaum, 1988], X Window System [Jones, 1989], and Unix [Ritchie, 1984]. It is assumed that the source code of MaRS is available for reference.

The rest of the manual is organized as follows. Section 2 shows the structure of the MaRS directory. Section 3 presents the different components of MaRS and the structure of the code controlling a component (i.e. the action routine). Section 4 describes the parameters and how they are used to carry information about components. Section 5 classifies the different event types and describe them. Section 6 presents event-related functions used by components to schedule events. Section 7 describes the structure of a packet and various packet han-

[1] using the -record option on the command line of MaRS or the record button in the control section.

dling functions provided by MaRS. All the components mentioned above are implemented in MaRS Version 1.0. They are described in more detail in [Alaettinoğlu *et al.*, 1991]. If these components are inadequate for some purpose, the user can define new components. Section 8 explains how to program a new routing or workload source/sink component.

Acknowledgment We gratefully acknowledge the use of some of the event handling and graphic interface routines of the MIT Network Simulator [Heybey, 1989]. Also, the user and programmer manuals of the MIT Network Simulator helped us in writing manuals for MaRS.

2 MaRS Directory Structure

The MaRS directory contains subdirectories for the source code, object libraries, manuals, executables, and some post-processing scripts. It is organized as follows:

`bin`	Subdirectory for the MaRS executable and post-processing scripts.
`doc`	Subdirectory for the manuals.
`proman`	Contains MaRS programmer's manual.
`usrman`	Contains MaRS user's manual.
`etc`	Subdirectory for miscellaneous files like example networks.
`include`	Subdirectory for the header files (i.e. `.h` files).
`lib`	Subdirectory for the libraries files (i.e. `.a` files).
`man`	Subdirectory for the MaRS man page.
`man1`	Contains the MaRS man page.
`src`	Subdirectory for the MaRS source code files (i.e. `.c` files).
`X`	Contains the source code of the X Window System interface.
`comps`	Contains the source code of the components.
`sim`	Contains the source code of the event manager, lists, queues, etc.

3 Components

Components are the basic building blocks of the target system. A component simulates some functions of the target system (e.g. a link component simulates a point-to-point communication channel).

At this point, it is important to distinguish between a **component instance** and a **component type**. A component is an instance of a component type. For example `LINK` is

a component type, and "Link" is a component instance of type `LINK` as shown in Figure 1. When it is not confusing, we will simply refer to both of them as "component". The component types are further grouped into **component classes**. For example, both `SPF` and `SEGAL` component types belong to `ROUTE_CLASS`.

Implementation of a component type consists of the declaration of a **data structure** and an **action routine**. The data structure defines the information needed by the component (i.e. parameters) as well as some standard information required by the simulator (e.g. pointers, flags, etc.). The action routine specifies the code to be executed when an event for this component is fired. Every component instance has its own data space, but all instances of the same type share the action routine.

The component classes and types implemented in MaRS Version 1.0 are the following (can be found in file `comptypes.h`):

Component Classes	Component Types	Comments
`LINK_CLASS`	`LINK`	
`NODE_CLASS`	`NODE`	
`ROUTE_CLASS`	`SPF`	
	`SEGAL`	
	`ExBF`	
`APTR_CLASS`	`FTP_SOURCE`	Application Transport Class
	`FTP_SINK`	
	`TELNET_SOURCE`	
	`TELNET_SINK`	
	`SMPLTRFC_SOURCE`	Simple Poisson/Uniform Traffic Source
	`SMPLTRFC_SINK`	Simple Poisson/Uniform Traffic Sink
`AUXILIARY_CLASS`	`PERF_MONITOR`	
	`STOPPER`	
	`LCOSTFCN`	Link Cost Function
`OTHER_CLASS`	None	

A detailed functional description of the components implemented is in the MaRS user's manual [Alaettinoğlu *et al.*, 1991].

From now on, we will refer to the address of the data space of a component instance as its id.

3.1 Data Structure

Each component instance have its own data space. This data space is interpreted using the data structure associated with each component type.

The data structure of a component should be the extension of the following structure defined in file `component.h`:

```
typedef struct _Component {
  /* The following is needed by the simulator, and should be common to all
     component types. */
  struct _Component
              *co_next, *co_prev; /* Links to other components in the list */
  short        co_class;          /* Class of component */
  short        co_type;           /* Type of component */
  char         co_name[40];       /* Name of component */
  PFP          co_action;         /* Action routine */
  COMP_OBJECT  co_picture;        /* Graphics object */
  list        *co_neighbors;      /* Doubly linked list of neighbors.  */
  short        co_menu_up;        /* Text window up flag */
  queue       *co_params;         /* Variable-length queue of parameters */

  /* Any other info that component needs to keep-- will vary. */
    .
    .
    .
} Component;
```

The fields are:

`co_next, co_prev`
: Used to link components into a list.

`co_class`
: The class of the component.

`co_type`
: The type of the component.

`co_name`
: The name of the component instance.

`co_action`
: Address of the action routine for this component type.

`co_picture`
: A pointer to a graphics object that displays this component on the screen.

`co_neighbors`
: Other components that are connected to this component.

`co_menu_up`
: If this field is true, then the text window associated with this component is displayed on the screen.

`co_params`
: Queue of the parameters needed by this component. This queue is used by the simulator to loop through parameters (e.g. while logging them).

3.2 Action Routine

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 is defined as the follows:

```
caddr_t action(source, this_comp, event_type, packet, argument)
Component *source;     /* Component that caused this event */
Component *this_comp;  /* Component to which this event applies */
int        event_type; /* Type of event */
Packet    *packet;     /* A packet */
caddr_t    argument;   /* Whatever */
{
       /* Usually a big switch statement on the event type */
}
```

The parameters of the action routine are:

`source`
: The component that scheduled this event. `NULL` for command type events and for events that are played from a file (refer to Section 5 for event descriptions).

`this_comp`
: Pointer to the data space of this component instance.

`event_type`
: Type of the event, the action routine is invoked for.

`packet`
: If the event requires a packet (e.g. `EV_NODE_RECEIVE`), this parameter is a pointer to that packet.

`argument`
Pointer to whatever additional information needed.

The action routine should return `NULL` if an error occurs, and should return `non-NULL` otherwise.

4 Parameters

Any information about a component that needs to be displayed on the screen, logged to disk, or saved in a network file must be stored in a **parameter**. A parameter is a data structure that (besides storing a value) stores information needed to display, save, or load the parameter. The `Param` structure is defined in `component.h`. The information stored includes pointers to functions to convert the parameter to and from a string, the name of the parameter, and flags describing how to save and/or display the parameter. The functions of the various parts of the `Param` structure are commented in `component.h`. A component may have as many parameters as needed. They are stored in a queue (which is a singly-linked list) pointed to by `co_params`. The I/O routines iterate through this queue to display the parameters as described below. The action routine may reference the parameters in any way it wants. The convention is to have a pointer for each parameter in the component's structure. As the parameter is initialized and added to the queue when the component is created, the pointer is set to point to it. Then the action routine can use a named variable to refer to the parameter rather than trying to search through the queue.

The actual value of a parameter is stored in the union at the end of its structure. Currently, the union has names for an integer, a double, or a pointer. To add another type, change the definition of the parameter structure and compile the simulator ("make mars") again. The value of the parameter is *not* used by any part of the simulator except for the action routine of the component that contains the parameter. The I/O routines read and change the value only by calling one of the functions pointed to in the parameter structure.

A parameter may serve one or more of the following purposes, depending on the flags that are set in the `p_flags` variable of the parameter structure. The constants (*Mask) are defined in the file `simx1.h`.

Initialization of parameters
The parameter values of a component affect its operation and can vary from one instance of a component to another. If the `InputMask` flag is set in the `p_flags` variable, the simulator will call the function pointed to by `p_input` with a string read either from the keyboard or from a file. This routine reads in a string, converts the string to an appropriate value and store it into the parameter.

Parameters that have the `InputMask` flag set will have their values saved (using the `p_make_short_text` routine) when the network configuration is saved.

If the `ModifyMask` is set in the `p_flags` variable, the simulator will display the default value associated with the parameter. The simulator will not ask for a value, but the user can change the value using the X window interface [Alaettinoğlu *et al.*, 1991].

Meter display

If the `CanHaveMeterMask` flag is set, a parameter can be displayed in a graphic "meter". The routine pointed to by `p_calc_val` is called to produce a value to be displayed in a meter. The returned value of the routine is expected to be a floating point number. The display of a parameter in a meter is relative to the unit of the parameter value. For example, the function for a packet queue length meter might return the length of the queue divided by ten, so that the unit in the meter display represents ten packets in the queue. Whenever the action routine changes the value of a parameter that might be displayed in a meter, it should set the `ChangeMask` flag into the `p_flags` variable of the parameter. If `ChangeMask` is set, the simulator will update the display of the parameter. The easiest way to set this flag is to call the function `log_param(c, p)`, which also takes care of logging parameter values to disk, if needed. `log_param(c, p)` updates the parameter pointed to by `p` in the component pointed to by `c`.

The default type of meter is set by the `p_display_type` variable. The constants are defined in `simx1.h`; the different meters available in MaRS Version 1.0 are `BAR_GRAPH`, `BINARY`, `LOG`, `TIME_HISTORY-A`, `TIME_HISTORY-D`, `DELTA`, `HISTOGRAM`, and `TEXT_METER`. Obviously, if the `CanHaveMeterMask` flag is not set, no value needs to be put into this element.

"Infowindow" display

If the `DisplayMask` flag is set, the parameter will be displayed in the text "infowindow" (short for information window) that appears when the user clicks on a component. The text is generated by calling the function pointed to by `p_make_text`, which returns a pointer to a string. This string is expected to contain some meaningful (e.g. with a label) human-readable representation of the value of the parameter. As with the meter above, if the value of a parameter that might be displayed in an infowindow changes, the `ChangeMask` flag must be set. Note that, in general, the `DisplayMask` flag must be set for the parameter to be of any use. Without the parameter being displayed in the infowindow, the user cannot see it or change it.

Logging to disk

If a parameter has `CanHaveLogMask` set, the user can cause its values to be written to a disk file as they change. The routine pointed to by `p_make_short_text` is used

to create a string to write. This routine should return a string that contains only the value of the parameter (no labels). For example, `p_make_text` usually points to a routine that `sprintf`'s the name of the parameter and its value into a string, while `p_make_short_text` does not include the name. To make logging happen, the action routine should call the `log_param(c, p)` routine every time the value changes (where `c` and `p` are pointers to the component and the parameter respectively). This routine writes the new value to the log if logging is turned on, and also sets the `ChangeMask` flag so that any meters will also be updated.

5 Events

MaRS is a discrete-event simulator. The events that can occur in MaRS are divided into four different classes:

Command type events `EV_CLASS_CMD`
: establish the basic operations of the simulator, e.g. create or delete a component. All components have to provide action routines for these events.

Event type events `EV_CLASS_EVENT`
: are meant for events that components send to each other. Although these events are defined globally in `eventdefs.h`, only components which receive them have to provide their corresponding action routines.

Private type events `EV_CLASS_PRIVATE`
: are specific to a component and are only known within that component. They are meant for events that components send to themselves.

Record type events `EV_CLASS_RECORD`
: are meant for events that can be recorded during a simulation. They are defined in `eventdefs.h`. During a simulation, record type events can be recorded (i.e. written) to a record file[2]. The record file can be used to exactly repeat a sequence of event occurrences on different runs of the simulator, e.g. repeat the sequence of link/node failure/repair occurrences in different simulation runs with different routing algorithms.

Next we describe the functions of the action routines associated with the different events specified in MaRS Version 1.0. Recall that the action routine of a component is defined as follows (see section 3):

[2]using the -record option on the command line of MaRS or the record button in the control section.

```
caddr_t action(source, this_comp, event_type, packet, argument)
Component *source;     /* Component that caused this event */
Component *this_comp;  /* Component to which this event applies */
int        event_type; /* Type of event */
Packet    *packet;     /* A packet */
caddr_t    argument;   /* Whatever */
{
       /* Usually a big switch statement on the event type */
}
```

Remember that `this_comp` points to the data space of the component instance to which the event applies.

5.1 Command Type Events

Command type events establish the basic operations of the simulator, e.g. create, start, reset a component. All MaRS components have to react to command type events, i.e. they have to provide routines for these events. The command type events are defined in `eventdefs.h`.

EV_CREATE

Create a new instance of a component. The `this_comp` variable must be NULL, `argument` points to the name of the new component, and the action routine returns either a pointer to a new data structure or NULL for error. The action routine must allocate the correct amount of memory for the new component's data structure, create its (empty) neighbor list, create the queue of parameters, create any packet queues, etc. This command must also initialize all the private data in the component as necessary. The only information that need not be initialized are any parameters with the `InputMask` flag set. They will be initialized by the simulator as specified in section 4.

EV_DEL

Delete an instance of a component (pointed to by `this_comp`). Detach it from any neighbors it has, free any storage associated with it, including its data structure, and perform any other necessary clean-up.

EV_RESET

Reset the state of the component. Clear out any packet queues, forget about any packets being processed, etc.

EV_START

Start stuff going for the component. For example, start a packet generator sending packets. For many components, this will be a no-op.

EV_NEIGHBOR

Attach another component as a new neighbor. `argument` points to the component to be made neighbor. A component should only allow legal neighbors. For example, a node will not allow another node to be attached as a neighbor—they must be connected by a link.

Neighbors are stored as a list of `Neighbor` structures. Each structure includes a pointer to the neighboring component, a queue in which to store packets (if needed), and a pointer to a parameter to display anything that might be associated with the neighbor. When a neighbor is added, the component must create and initialize a neighbor structure, and put it on its neighbors list. The neighbor structure is defined in `component.h`. If there is some piece of information associated with the neighbor that must be displayed, a parameter structure must be allocated, initialized appropriately, and added to the queue of parameters in the component structure.

EV_UNEIGHBOR

Remove neighbor pointed to by `argument` from list of neighbors, and free any memory used to keep track of the neighbor (such as a packet queue and the neighbor structure itself). If there is a parameter associated with this neighbor, it must be removed from the queue of parameters and freed. Return NULL if the component is not a neighbor, non-NULL otherwise.

EV_MK_PEER

Make component pointed to by `argument` a peer. Return NULL if the peer component is incompatible or not initialized yet, non-NULL otherwise.

5.2 Event Type Events

Event type events are meant for events that components send to each other. Although they are known system wide, (i.e. they are defined in `eventsdef.h` and therefore known to all components), only components that receive these events have to provide action routines for them.

EV_APTR_RECEIVE

Receive a packet from the node component pointed to by `source`. Depending on the type of the packet pointed to by `packet` different actions are taken.

For `FTP` and `TELNET`, if the packet p received is a data packet, call `EV_APTR_CONSUME` to consume p, increment the counter for received data packets, cause an ack packet to be produced and sent calling the attached node component. If p is an ack packet, increment the counter for acknowledged packets, and call the performance monitor

component to update statistics. If p is a token packet, update the round trip time estimate.

For `SMPLTRFC`, if the packet p received is a data packet, consume p, increment the counter for received data packets and ack packets (no explicit acknowledgments for this type of workload), and call the performance monitor component to update statistics.

EV_APTR_RETRANSMIT

Receive information that the workload packet pointed to by `packet` was lost due to link failure, node failure, or buffer space limitations. The action corresponding to the event depends on the type of the workload source component.

For `FTP` and `TELNET`, increment number of retransmitted workload packets. Schedule a retransmission calling the attached node component.

For `SMPLTRFC`, increment number of dropped data packets (no retransmissions for this type of workload).

EV_LINK_SEND

Receive a packet p pointed to by `packet` from a (neighbor) node component pointed to by `source`. Append p to the appropriate link queue if the link status is up. If the link status is down, do not append p to the link queue. Instead, discard p and if it is a workload packet, call `EV_APTR_RETRANSMIT` with the workload source component.

EV_NODE_RECEIVE

Receive a packet p pointed to by `packet` from an incoming link component pointed to by `source`. If the node status is up, do the following. If p is a workload packet destined for that node, call the workload sink component. If p is a workload packet not destined for that node, consult the routing table and forward the packet to the appropriate outgoing packet queue. If p is a routing packet, place it in the queue for incoming routing packets.

If the node status is down or there is no buffer space available, discard packet p. In case of a workload packet, call `EV_APTR_RETRANSMIT` with the workload source component.

EV_NODE_SEND

Take a packet p from an outgoing packet queue. `argument` points to the `Neighbor` structure of the (neighbor) link where the queue of packets that are to be forwarded through it is stored. Pass p to the link component, i.e. call `EV_LINK_SEND`.

EV_NODE_PRODUCE

Accept a packet p pointed to by `packet` from the workload source component or the attached routing component pointed to by `argument` and forward it to the appropriate outgoing packet queue. If the queue is empty, schedule `EV_NODE_SEND`.

EV_ROUTE_PROCESSING

Take a routing packet pointed to by `packet` from the queue of incoming routing packets (stored in the attached node component) and process it. Depending on the algorithms, different actions are taken.

5.3 Private Type Events

Private type events are specific to a component and are only known within that component. They only operate on the data structure of that component. They are meant for events that components send to themselves. They are defined in the component's `.h` file and not in `eventdefs.h`. Private events of a component are invisible to the other components.

EV_START_COMPUTATION

Defined for `ExBF` and `SEGAL` components. Cause the computation of costs of the outgoing links for the attached node component. Schedule next computation. Depending on the routing algorithm, other actions are taken.

EV_APTR_PRODUCE

For `FTP` and `TELNET`, increment number of produced data packets and schedule `EV_APTR_SEND` if the produce window is not full. Schedule next production if this was not the production of a response packet[3].

For `SMPLTRFC`, if the window is not full, increment number of data packets sent, produce the data packet and send it to the attached node component. (Note that this type of workload has only one window.)

EV_APTR_CONSUME

This event if not defined for `SMPLTRFC` workload. For `FTP` and `TELNET`, consume the received data packet pointed to by `packet`. For TELNET, schedule a response packet production.

EV_APTR_SEND

This event if not defined for `SMPLTRFC` workload. For `FTP` and `TELNET`, if the send window is not full, send a data packet to the attached node component and increment number of data packets sent.

EV_LINK_RECEIVE

Take a packet from the link queue pointed to by `argument` and send it to the receiving node component.

[3] A response packet is a data packet sent from the `TELNET` sink component to the `TELNET` source component.

EV_PERF_UPDATE
Cause the update of network performance measures.

EV_SPF_BROADCAST
Defined for `SPF` component. Broadcast costs of the outgoing links for the attached node component. Update routing table. Schedule next broadcast.

EV_CHECK_STOP
Defined for the `STOPPER` component. Check if termination condition is satisfied.

EV_INSTANT_RATE
Defined for `FTP`, `TELNET`, and `NODE`. Cause the update of parameters (representing instantaneous performance measures).

5.4 Record Type Events

During a simulation, record type events can be recorded (i.e. written) to a record file. The record file can be used to exactly repeat a sequence of event occurrences on different runs of the simulator, e.g. repeat the sequence of link and node failures and repairs on simulation runs with different routing algorithms.

EV_APTR_CONN_ON
Defined only for `FTP` and `TELNET`. Start a connection (i.e. train) between a given source-sink pair . `argument` points to the average number of packets per train, which is one of the component's parameters.

EV_LINK_FAILURE
Cause a failure of the link due to the failure of either the link itself or an adjacent node. The workload source components of all workload packets in the link at the time of failure are called with `EV_APTR_RETRANSMIT`. The routing components on the adjacent nodes are informed of the link failure (by sending a routing packet of type `RT_LINK_SHUTDOWN` to both).

EV_LINK_REPAIR
End the link failure. The link is considered repaired. The routing components on the adjacent nodes are informed (by sending a routing packet of type `RT_LINK_WAKEUP` to both).

EV_NODE_FAILURE
Cause a node failure. Call `EV_APTR_RETRANSMIT` for every workload packet in an outgoing packet queue. Inform the attached routing component of the node failure (by

inserting a routing packet of type `RT_NODE_SHUTDOWN` in its routing packet queue to be processed by the routing component).

EV_NODE_REPAIR

End of node failure. The node is considered repaired. Inform the attached routing component of the node repair (by inserting a routing packet of type `RT_NODE_WAKEUP` in its routing packet queue).

6 Event Manager

MaRS is a discrete-event simulator. Components send each other events in order to communicate and to send packets through the network. The event manager provides a general facility to schedule and send events. The simulator time is maintained by the event manager in units of "ticks." Currently, one tick is ten microseconds. This value may change, so to convert from ticks to microseconds, multiply ticks by the constant USECS_PER_TICK (defined in `sim.h`). (A tick will never be less than one microsecond.) The current time (in ticks) is returned by the function `ev_now()`. The time is maintained as an unsigned 32-bit value, so (at ten microseconds per tick) the simulator can run for almost 12 hours of simulated time before an overflow occurs.

There are two event-related functions that a component needs to know about: `ev_enqueue()` and `ev_call()`. `ev_enqueue()` creates a new event and places it in the event queue to be fired at the proper time. You may schedule events at the current time or at any time in the future. Scheduling events for the past is considered bad form. Events scheduled at the same time are not guaranteed to fire in any particular order. `ev_enqueue()` returns a pointer to the newly created event. The syntax of `ev_enqueue()` is as follows:

```
Event *ev_enqueue(event_type, source, dest, time, routine, packet, argument)
int       event_type;
Component *source, *dest;
unsigned  time;
PFP       routine;
Packet    *packet;
caddr_t   argument;
```

`ev_call()` creates a new event and executes the appropriate action routine *immediately*. `ev_call()` returns a pointer to the newly created event. The syntax of `ev_call()` is as follows:

```
Event *ev_call(event_type, source, dest, routine, packet, argument)
int       event_type;
Component *source, *dest;
PFP       routine;
Packet    *packet;
caddr_t   argument;
```

Notes: `PFP` is a Pointer to a Function that returns a Pointer, and is defined in `component.h`. `routine` is therefore the action routine to call when the event is fired. The arguments `packet`, and `argument` are optional—they may be replaced by NULL.

7 Packet

The target system is a message passing system. Messages are passed in packets. In MaRS, a structure is defined to model packets. This structure contains information that can be normally found in a target system's packet as well as information that is needed by the simulator.

The packet structure has been defined in `packet.h` as:

```
typedef struct _Packet {
  struct _Packet *pk_next;          /* pointer for use by queueing functions */
  Socket          pk_source_socket, /* source socket of the connection */
                  pk_dest_socket;   /* destination socket of the connection */
  Component      *pk_next_comp;     /* the component packet is going to. */
  int             pk_length;        /* length in bytes */
  unsigned int    pk_time;          /* usually departure time */
  unsigned int    pk_arrival_time;  /* arrival time to the current node */
  unsigned int    pk_sent_time;     /* the time the packet was first sent */
  int             pk_color;         /* color assigned to a packet */
  unsigned        pk_uid;           /* a unique id assigned to each packet */
  unsigned        pk_type;          /* type field : data, routing, etc. */
  char           *tail;             /* pointer to the rest of the packet */
  APTR_Packet     tr_pk;            /* transport protocol part of packet */
  RT_Packet       rt_pk;            /* routing part of packet */
} Packet;
```

The fields are:

`pk_next`

Pointer, used to queue packets together.

`pk_source_socket`
: The source socket of the connection the packet belongs to. The `Socket` type is defined as

```
typedef struct _Socket {
   Component *so_port, *so_host;
} Socket;
```

that is the ids of the node and the application/transport component instances.

`pk_dest_socket`
: The destination socket of the connection the packet belongs to.

`pk_length`
: The length of the packet. It includes the header and the imaginary data portion if any.

`pk_time`
: The time at which the packet will depart from a component.

`pk_arrival_time`
: The time at which the packet arrives to a node. This field is used to calculate average delay per packet at a node for each of its outgoing links.

`pk_sent_time`
: The time at which the data portion of this packet is first sent into the network. That is if a packet is dropped and a new one is generated, the `pk_sent_time` field of the new packet will contain the original value. When a packet is acknowledged by a sink, this field is copied to the acknowledgment packet. And when the acknowledgment packet is received by the source, this field is used to calculate average delay per packet.

`pk_color`
: Used to distinguish between packets, currently not used.

`pk_uid`
: Unique identifier assigned to each packet, currently not used.

`pk_type`
: The type of the packet. In MaRS Version 1.0, this field is either `TR_PACKET` (i.e. application/transport packet) or `ROUTE_PACKET` (i.e. routing packet).

`tail`
: A pointer that is used to implement big or varying length packets. When a packet is created, space is allocated for the entire `Packet` structure, so keeping the Packet structure small will decrease the memory requirements of the simulation. However, many of the routing algorithms require big varying length packets. When such a packet is needed, additional space can be allocated as follows:

```
Packet.tail = sim_malloc(size_of_varying_part);
```

`tr_pk`
: The application/transport part of the packet (discussed in Section 8.3).

`rt_pk`
: The routing part of the packet (discussed in Section 8.2).

If the packet structure needs to be extended for some reason, just add fields to the `Packet` structure definition. But no field should be removed from this structure.

7.1 Packet Handling Routines

Some functions are provided for the allocation and deallocation of packets. To create a new packet, use:

```
Packet *pk_alloc()
```

To delete a packet and free its space, use:

```
void pk_free(pkt)
Packet *pkt;
```

To remove all the packets and free their spaces, use:

```
void pk_free_all()
```

8 Programming New Components

This section concerns the implementation of new components. First, general steps to implement a new component are outlined. Then more detailed instructions are given to implement new routing and workload components.

The general steps to create a new component are the following:

1. Create two files, one (header file) for the data structure, `new_comp.h`, and one for the action routine, `new_comp.c`. `new_comp.h` should include the following files (all of them might not be needed):

```
#include "sim.h"          /* Contains global macros */
#include "q.h"            /* Contains queue routine and type definitions */
#include "list.h"         /* Contains list routine and type definitions */
#include "component.h"    /* Contains parameter and neighbor type
                             definitions */
#include "route.h"        /* Contains routing and local topology table type
                             definitions */
```

`new_comp.c` should include the following files (again all of them might not be needed):

```
#include <sys/types.h>
#include <stdio.h>
#include <math.h>
#include "sim.h"          /* Contains global macros */
#include "q.h"            /* Contains queue routine and type definitions */
#include "list.h"         /* Contains list routine and type definitions */
#include "component.h"    /* Contains parameter and neighbor type
                             definitions */
#include "log.h"          /* Contains loging routine definitions */
#include "comptypes.h"    /* Contains component type definitions */
#include "packet.h"       /* Contains the packet type and packet handling
                             routine definitions */
#include "eventdefs.h"    /* Contains event definitions */
#include "event.h"        /* Contains event handling routine definitions */
#include "node.h"         /* Contains node definitions */
#include "perf.h"         /* Contains performance measure definitions */
#include "new_comp.h"     /* Contains the data structure definitions
                             of this component */
```

2. Define the data structure in the header file.

3. Define private events (if any) in the header file.

4. Write the action routine. That is supply code for the command type events, and any event that is related to this component.

5. Define the new component type (and class if different from the existing ones) in the file `comptypes.h`.

6. Add an entry for the new component type in the `components_array` in the file `comptypes.c`.

7. Modify the `Makefile`, and re-compile MaRS.

8.1 Interface to the Performance Monitor Component

The performance monitor (`PERF_MONITOR`) provides performance measures to evaluate and compare different routing algorithms. To calculate these measures the performance monitor needs to obtain information from the components in the target system. The performance monitor gets this information either by polling the components, or by an interface function `pm` (called by the components in an event-driven way).

The syntax of `pm` is the following:

```
void pm(comp, comp_type, req_type, arg1, arg2, arg3)
Component *comp;
int        comp_type;
int        req_type;
int        arg1, arg2, arg3;
```

The parameters of `pm` are:

`comp`
: Id of the invoking component.

`comp_type`
: The type of the invoking component.

`req_type`
: Type of the request. In MaRS Version 1.0, the following are the valid requests:

  `NEW_COMPONENT`
  : The invoking component is just being created. If it is an interesting one, add it to poll list.

  `LINK_FAILURE`, `LINK_WAKEUP`
  : The invoking component is a link. The current number of failed links in the target system is updated.

`CONN_UP`

The invoking component is an application/transport source. The current number of active connections is incremented.

`CONN_DOWN`

The invoking component is an application/transport source. The number of active connections is decremented, and statistics about connections (e.g. average number of packets in FTP connections) are updated. In this case, `arg1` contains the life time of the connection, and `arg2` contains number of packets in the connection. In MaRS Version 1.0, the FTP and TELNET workload components use the packet/train model. A train represents a connection and is specified by its start time, number of packets, and inter-packet time distribution. It might so happen that two trains overlap (the second one starts before the first one finishes). In this case, `arg1` and `arg2` contains the total values. `arg3` contains the number of overlapping trains. (Note that if the user has chosen the parameters correctly, overlapping would not have happened.)

`PKT_RETRANSMITTED`

Number of application/transport packets dropped in the network is incremented.

`ACK_RECEIVED`

The invoking component 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 round trip delay of the packet. The statistics for throughput, and delay are updated.

`ROUTING_PKT`

The invoking component 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.

`arg1, arg2, arg3`

Depending on the request type, these arguments contain more information.

The performance monitor also collects information by polling (e.g. data load and routing load are calculated by polling the link components). The components that the performance monitor polls are determined when the `NEW_COMPONENT` request is generated by them.

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.

8.2 Writing a New Routing Component

As stated in the introduction MaRS provides a flexible platform for measuring the performance of routing algorithms. This section describes how one implements and interfaces a new routing algorithm with MaRS. Many functions are provided for convenience.

8.2.1 Data Structure

The data structure of a routing component should be an extension of the following structure which is defined in `route.h`:

```
typedef struct _Routet {
  struct _Routet *route_next,        /* Common to all components */
                 *route_prev;
  short           route_class;
  short           route_type;
  char            route_name[40];
  PFP             route_action;
  COMP_OBJECT     route_picture;
  list           *route_neighbors;
  short           route_menu_up;
  queue          *route_params;

/* the following must be present in all the routing components */
  PFI             processing_time;   /* returns an estimate of
                                        proccessing time */
  int             no_of_nodes;       /* no of nodes known in the net */
  tick_t          link_cost_time;    /* time at which link costs are
                                        last calculated*/
  Component      *node;              /* the node attached */
  Param          *routing_table;     /* routing table */
  Param          *local_top;         /* local topology table */

/* other algorithm specific parameters follows */

} Routet;
```

The fields are:

`processing_time`
: is a pointer to a function that returns an unsigned integer. This function is called

with a packet to estimate how much processing time (in microseconds) is required to process this packet. This time information is needed to schedule routing events. Such a function for each routing algorithm should be supplied.

`no_of_nodes`
: is an integer which stores the number of nodes known (i.e. from which a packet has been received) in the network. This field is mainly provided for conveniently looping in the routing table.

`link_cost_time`
: is of type `tick_t` which stores the simulated time (in ticks) at which the link costs of the attached node are last calculated.

`node`
: is a pointer to the component of type `NODE` to which this routing component is connected.

`routing_table`
: is a pointer to a structure of type `RoutingTable` (defined in `route.h`). This structure for each destination stores the cost and the next-hop. This table is updated by the routing component and used by the node component to forward packets.

```
typedef struct {
   Component   *dest;
   Component   *hop;
   unsigned int cost;
} RoutingTable[MAX_NO_OF_NODES];
```

`local_top`
: is a pointer to a structure of type `LocalTopologyTable` (defined in `route.h`). This table for each outgoing link stores the link cost and whatever necessary to calculate it. This table is created, initialized and mostly updated by the node component. However, the `l_cost` and `l_status` fields are updated by the routing component. Some important fields of this structure are the following:

```
typedef struct {
    Component    *n_nd;        /* Pointer to the neighbor node */
    Component    *n_link;      /* Pointer to the outgoing link */
    int           l_delay;     /* Propagation delay for link */
    int           l_bandwidth; /* Bandwidth for outgoing link */
    char         *l_status;    /* status of link */
    int           l_cost;      /* cost of link */
} LocalTopologyTable[MAX_LINKS_PER_NODE];
```

After defining the above standard fields required by the simulator for any routing component, other fields and parameters required by the new routing algorithm can be defined.

8.2.2 Routing Packet

The `Packet` structure contains a `pk_type` field which is set to `ROUTE_PACKET` for routing packets. The routing information is contained in the `rt_pk` field of the `Packet` structure. This field is of type `RT_Packet` defined in `packet.h` as

```
typedef struct _RT_Packet {     /* routing part of a packet */
  int           rt_type;
  union {
     /* algorithm specific packet information */
  } rt;
} RT_Packet;
```

The fields are:

`rt_type`
: is an integer that shows the routing packet sub-type. The values (defined in `packet.h`) are:

  - `RT_NODE_SHUTDOWN`: indicates that the attached node has failed
  - `RT_NODE_WAKEUP`: indicates that the attached node has become operational
  - `RT_LINK_SHUTDOWN`: indicates that the link (the id of the link is specified in the field `packet.pk_source_socket.so_host` field) has failed
  - `RT_LINK_WAKEUP`: indicates that the link (the id of the link is specified in the field `packet.pk_source_socket.so_host` field) has become operational
  - otherwise: a routing packet exchanged between routing components

`rt`
: is a union of different routing packets. It stores the information exchanged between the routing components.

To define a new routing packet, edit `packet.h` by defining a structure that represents the new packet. Add its definition to `RT_Packet.rt`. Define a new constant that will be used in `rt_type` to indicate the new packet.

If the new packet is a varying length one, place the fixed part in `RT_Packet.rt`. The space for the varying part should be allocated only when this type of packet is created (see section 7).

The routing packets are received and queued by the node component. The routing packets are passed to the routing component by the `EV_ROUTE_PROCESSING` event. This event is invoked at the time the processing of the packet is supposed to finish (the function pointed by `Routet.processing_time` is used in the calculation of this time).

8.2.3 Action Routine

The action routine should supply code for all the events that are relevant to the new routing component, including the command type events.

Besides the standard actions for command type events, the following actions should be taken:

`EV_CREATE` allocates memory for routing table

`EV_DEL` frees the memory allocated for routing table

`EV_NEIGHBOR` points the `local_top` to the local topology table in the node component's data structure (see `node.h`).

The other events that a routing component should supply an action for are the internal (i.e. private) events and the `EV_ROUTE_PROCESSING` event. The actions for internal events should be written in accordance with the specification of the routing algorithm.

The action for `EV_ROUTE_PROCESSING` event for a routing component, `comp`, should look like

```
free_routing_queue(comp);
      /* defined in route.c to free space occupied by the routing packet */

/* process the packet */
.
.
.
schedule_next_EV_ROUTE_PROCESSING(comp);
       /* defined in route.c to schedule the next EV_ROUTE_PROCESSING  */
```

The routing packets are queued at the node. The node only schedules the first `EV_ROUTE_PROCESSING` event whenever its routing queue becomes non-empty, scheduling the next `EV_ROUTE_PROCESSING` event occurrences should be done by the routing component as outlined above. The above functions `free_routing_queue` and `schedule_next_EV_ROUTE_PROCESSING` are supplied for your convenience.

To send a packet to a neighbor, schedule (or call) a `EV_NODE_PRODUCE` event for the node connected with the packet as the parameter. The last parameter of the `ev_enqueue` (or the `ev_call`) should be the link that the packet should be sent over. That is:

```
ev_call(EV_NODE_PRODUCE, my_comp, my_node_comp, my_node_comp->co_action,
packet, link_to_take);
```

8.2.4 Link Cost Handling Routines

The statistics needed for link cost calculation are collected by the node component in a structure of type `LocalTopologyTable` . But the calculation of link costs are invoked by the routing component by calling the following function:

```
compute_link_cost_metrics(routing_component);
Routet *routing_component;
```

This function, whenever called, calculates the costs of the outgoing links of the node attached to the routing component `comp` using the parameters specified by the component of type `LINK_COST_FUNC`, and resets the statistics collected by the node component (caution: calling it twice one after the other will yield a different result than just calling it once). When this function should be called depends on the specifications of the routing algorithm. With most algorithms it is called periodically.

To access the cost of a link either use the local topology table as:

```
LocalTopologyTable[link_index_in_LocalTopologyTable].l_cost
```

or the following function:

```
link_cost(routing_component, link_index_in_LocalTopologyTable)
Routet *routing_component;
int    link_index_in_LocalTopologyTable;
```

Whenever a link is failed or repaired the link cost statistics (that are collected by the node component) need to be reset by the routing component (the node component does not know that the link has failed). To reset these, in case of link failure, use:

```
link_failure_init_LTT(routing_component, link_index_in_LocalTopologyTable);
Routet *routing_component;
int    link_index_in_LocalTopologyTable;
```

and in case of link repair, use:

```
link_repair_init_LTT (routing_component, link_index_in_LocalTopologyTable);
Routet *routing_component;
int    link_index_in_LocalTopologyTable;
```

To determine the index of a link in the `LocalTopologyTable` given its id (that is address of the data structure) use

```
int index_at_LTT_by_link(routing_component, link)
Routet *routing_component;
Component *link;
```

which returns either the index or -1 (i.e. link is not found). To determine the index of a link in the `LocalTopologyTable` given the id of the node at the other end of the link, use

```
int index_at_LTT_by_node(routing_component, node)
Routet *routing_component;
Component *node;
```

which returns either the index or -1.

The `LocalTopologyTable` contains `NULL` at the `n_nd` (i.e. next_node) field after all the valid entries.

8.3 Writing New Workload Components

The data traffic is introduced to the target system by the application/transport components. The data is produced by the *source* application/transport component, and consumed by the *sink* application/transport component.

To implement a new traffic pattern (or a different protocol), a source component, and a sink component have to be implemented. In many cases, source and sink components have the same data sturcture, but different action routines even though they share most of the code.

In MaRS Version 1.0, we have integrated the application and the transport protocol into one component for efficiency reasons. Feel free to implement them seperately if you need to.

8.3.1 Data Structure

The data structure of a workload component should be the extension of the data structure described in Section 3. Since most of the fields needed by the source and the sink components are the same, one data structure can be defined for both components.

8.3.2 Workload Packet

The packet structure contains a `pk_type` field which is set to `TR_Packet` for application/transport packets. A workload packet is either a data packet, or an ack(nowledgement) packet, or a token packet. It is defined in `packet.h` as:

```
typedef struct _APTR_Packet {
  int   tr_type;
  int   response;
  int   data_size;
} APTR_Packet;
```

The fields are:

`tr_type`
: Specifies the sub-type of the packet. The different subtypes are:

  `TR_DATA`
  : Packet contains data (only logically; actually the packet contains its data size).

`TR_ACK`
: Packet is an ack packet.

`TR_TOKEN`
: Packet is a token packet which is used to estimate the round trip time of packets. The round trip time is used to set the correct timeout period.

`response`
: A flag that is sent to the sink component to notify that a response is expected for this packet. This is introduced for interactive protocols, like TELNET.

`data_size`
: The size of the imaginary data portion.

If new information is needed to be present on the workload packet, feel free to add it to this structure.

8.3.3 Action Routine

An action routine for source component and another action routine for sink component need to be written. These routines should supply code for all the events that are relevant to the new application/transport components, including the command type events.

The following event type events are called by the link and node components and therefore an action routine has to be provided for them:

`EV_APTR_RECEIVE`
: The node component schedules this event to pass a packet it has received for this application/transport component.

`EV_APTR_RETRANSMIT`
: A node or a link component lost a packet due to a failure or buffer space limitation. The packet needs to be retransmitted. This event is implemented to simplify the loss detection. The timeout period mentioned above is only used to determine when to re-send the packet.

To send a packet to the network, schedule (or call) an `EV_NODE_PRODUCE` event for the node to which the component is connected with the packet as the parameter. Make sure the last parameter of `ev_enqueue` (or `ev_call`) is `NULL`, that is:

```
ev_call(EV_NODE_PRODUCE, my_comp, my_node_comp, my_node_comp->co_action,
packet, (caddr_t)NULL);
```

Note that some calls to performance monitor need to be done using the interface function `pm`.

Index

ack packet 2, 13, 29
application/transport component 2, 29
APTR_CLASS 5
AUXILIARY_CLASS 5
CanHaveLogMask 10
CanHaveMeterMask 9
ChangeMask 9
command type event 3, 10, 11
component 4, 19
    action routine 7
    class 5
    data structure 6
    instance 5
    type 5
data packet 2, 13, 14, 29
discrete-event simulator 1, 10, 16
DisplayMask 9
EV_APTR_CONN_ON 15
EV_APTR_CONSUME 13, 14
EV_APTR_PRODUCE 14
EV_APTR_RECEIVE 12, 30
EV_APTR_RETRANSMIT 13, 30
EV_APTR_SEND 14
ev_call() 16, 27
EV_CHECK_STOP 15
EV_CLASS_CMD 10
EV_CLASS_EVENT 10
EV_CLASS_PRIVATE 10
EV_CLASS_RECORD 10
EV_CREATE 11, 26
EV_DEL 11, 26
ev_enqueue() 16
event 1, 10, 15
    class 3, 10
    manager 16
    occurrence 1
    routine 1
    type event 3, 10, 12
    type 1
EV_INSTANT_RATE 15
EV_LINK_FAILURE 15
EV_LINK_RECEIVE 15
EV_LINK_REPAIR 15
EV_LINK_SEND 13
EV_MK_PEER 12
EV_NEIGHBOR 12, 26
EV_NODE_FAILURE 16
EV_NODE_PRODUCE 14, 27, 30
EV_NODE_RECEIVE 13
EV_NODE_REPAIR 16
EV_NODE_SEND 13, 14
ev_now() 16
EV_PERF_UPDATE 15
EV_RESET 11
EV_ROUTE_PROCESSING 14, 26
EV_SPF_BROADCAST 15
EV_START 12
EV_START_COMPUTATION 14
EV_UNEIGHBOR 12
ExBF 5, 14
FTP 13, 14, 15
FTP_SINK 5
FTP_SOURCE 5
InputMask 8, 11
LCOSTFCN 2, 5
link component 1
link cost 2, 27
LINK 1, 5
LINK_CLASS 5
link-cost component 2
LINK_COST_FUNC 27
local topology table 23, 25

`LocalTopologyTable` 24, 27
Makefile 21
MaRS directory 4
MaRS Version 1.0 1, 3, 4, 9, 18, 29
message passing system 17
meter 9
MIT Network Simulator 4
ModifyMask 9
node component 1
`NODE` 1, 5, 15
`NODE_CLASS` 5
`OTHER_CLASS` 5
packet handling routines 19
packet 1, 12, 14, 17, 29
    structure 17
Param structure 8
parameter 8
`PERF_MONITOR` 2, 5, 21
performance monitor component 2, 13, 21
physical network 1
`*pk_alloc()` 19
`*pk_free()` 19
`*pk_free_all()` 19
`pm` 31
private type event 3, 10, 14
produce window 14
record file 3, 15
record type event 3, 10, 15
response packet 14
retransmission 13
roundtrip time estimate 2
`ROUTE_CLASS` 5
routing packet 2
routing algorithm 2
routing component 2, 19, 23
routing packet 15, 25
routing table 23, 24
`RoutingTable` 24
`SEGAL` 5, 14
simulator time 16
sink component 2
`SMPLTRFC` 13, 14
`SMPLTRFC_SINK` 5
`SMPLTRFC_SOURCE` 5
source component 2
source-sink pair 2, 15
`SPF` 5, 15
stopper component 2
`STOPPER` 2, 5, 15
system state 1
target system 1, 4
`TELNET` 13, 14, 15
`TELNET_SINK` 5
`TELNET_SOURCE` 5
termination condition 2
tick 16
token packet 2, 13, 29
topology 1
workload component 19, 29
workload packet 2, 13, 29
workload 2

References

[Alaettinoğlu *et al.*, 1990] C. Alaettinoğlu, K. Dussa, A. U. Shankar, and J. Bolot. Routing Testbed: Initial Design. Technical Report UMIACS-TR-90-71, CS-TR-2475, Department of Computer Science, University of Maryland, College Park, MD 20742, May 1990.

[Alaettinoğlu *et al.*, 1991] C. Alaettinoğlu, K. Dussa-Zieger, I. Matta, and A. U. Shankar. MaRS User's Manual. Technical Report UMIACS-TR-91-80, CS-TR-2687, Department of Computer Science, University of Maryland, College Park, MD 20742, June 1991.

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

[Jones, 1989] O. Jones. *Introduction to the X Window System.* Prentice-Hall, 1989.

[Ritchie, 1984] D.M. Ritchie. The Evolution of the UNIX Time-sharing System. *AT&T Bell Laboratories Technical Journal*, 63(8):1595–1605, October 1984.

[Tanenbaum, 1988] A. S. Tanenbaum. *Computer Networks.* Prentice-Hall, 1988.