Chapter 4 UMJaNetSim

This chapter provides an indepth description of the UMJaNetSim v0.5 network simulator. The first section of the chapter begins with the overview of the UMJaNetSim architecture and the UMJaNetSim Application Programming Interface (API). The aim is to provide an explanation of the UMJaNetSim architecture so that extra components can be easily integrated into the existing simulator. The second section describes UMJaNetSim features. The final section summarizes this chapter.

4.1 UMJaNetSim Architecture

UMJaNetSim [UMJANETSIM] is a flexible test bed for studying and evaluating the performance of a TCP network without the cost of building a real network. The simulator is written in JAVA Language and is developed via the object-oriented programming approach.

Figure 4.1 shows the overall architecture of the UMJaNetSim. It consists mainly of two parts: the simulation engine and the simulation topology. The simulation engine is the main controller of the entire simulation. It handles two major management tasks of the simulation, which are the event management and the GUI management. The simulation engine also handles the input/output processes (e.g. save the topology to a disk file) and provides many tools that help the simulation process. The simulation topology consists of all the simulation objects, which are also referred to as simulation components. These simulation components are the main subject of a simulation scenario, and they typically consist of a group of interconnected network components such as switches, physical links or source applications.

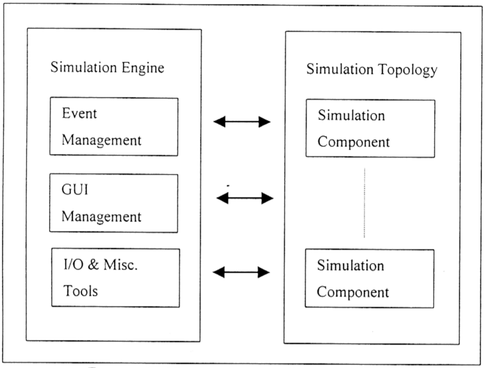


Figure 4.1 UMJaNetSim overall architecture

4.1.1 Event Management Architecture

Figure 4.2 shows the event management architecture for the UMJaNetSim. The major object of the entire application is a *JavaSim* object, which itself represents the simulation engine. The *JavaSim* object manages an event queue, an event scheduler, and a simulation clock.

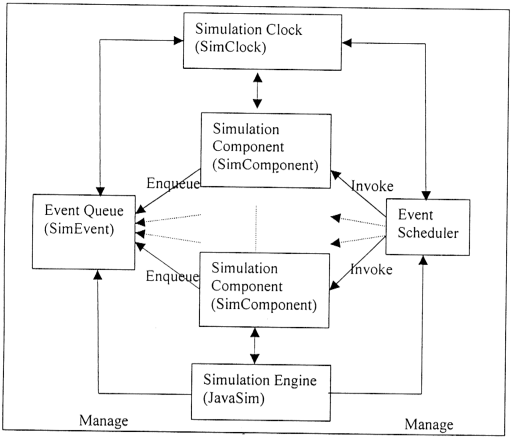


Figure 4.2 UMJaNetSim event management architecture

Typically, the simulation engine interacts with the simulation topology (consisting of all the simulation components) through two operations:

- A simulation component schedules an event for a target component (which may be the source component itself) to happen at a specific time using the *enqueue* operation.
- The simulation engine invokes the event handler of the target component when the specific time is reached. The target component will then react to the event according to its behavior.

The event queue is actually a *java.util.List* object consisting of all the scheduled events in the form of *SimEvent* objects. The events are sorted by the event-triggering time. The event scheduler always fetches and removes the first event in the event queue, and triggers the event by invoking the event handler of the target component.

4.1.1.1 The Simulation Time

In a discrete event type of simulation, the simulation time is an important concept. The UMJaNetSim uses an asynchronous approach of the discrete event model, where any event can happen at any time, up to the precision allowed by the granularity of the simulation clock. The simulation time in the UMJaNetSim is based on "ticks". The duration of a tick is configurable in the simulator. By default, a tick is equivalent to 10 nanoseconds. The *SimClock* object is the global time reference used by every component in the simulation and managed by the simulation engine. The *SimClock* object also provides helper methods for the conversion between real time and the tick.

4.1.2 GUI Management

Figure 4.3 shows the Graphical User Interface (GUI) management components of the UMJaNetSim. GUI management involves drawing the viewing area, managing various on-screen windows (or dialog boxes), and handling user inputs (e.g. menu commands). The *JavaSim* object is the overall controller for GUI management. A helper object called SimPanel handles the detailed task of drawing the topology view of the simulation components. The *SimPanel* keeps track of the latest set of simulation components and the interconnection among the components in order to present the simulation topology visually to the user. It also handles any direct component manipulation by users such as positioning of the components. Figure 4.4 shows a screenshot of the UMJaNetSim with the *SimPanel* (in the center) showing the simulation components.

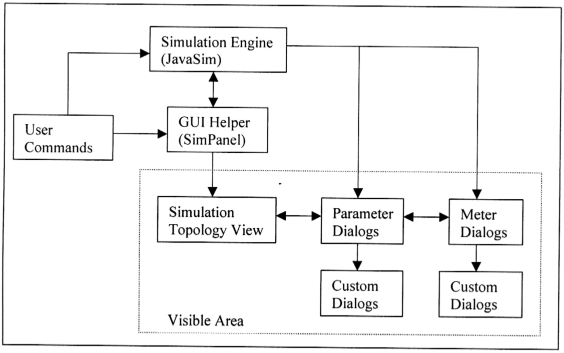


Figure 4.3 UMJaNetSim GUI management structure

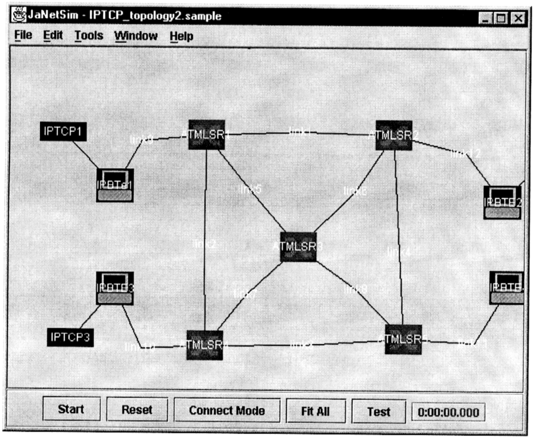
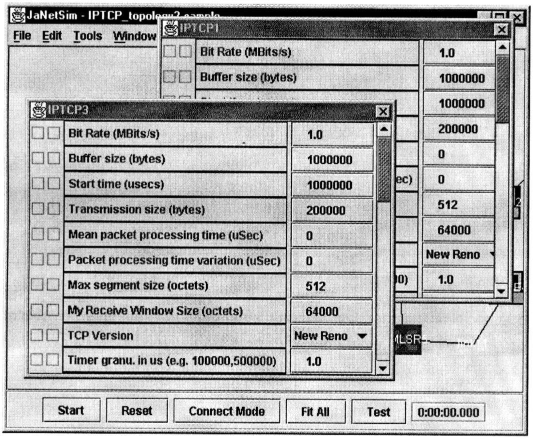


Figure 4.4 UMJaNetSim Screenshot (SimPanel)

Simulation information other than the topology view is displayed through on-screen dialog boxes (hereafter referred to as dialogs). Two primary types of dialogs are the parameter dialogs and the meter dialogs. Figure 4.5 shows the UMJaNetSim with two invoked parameter dialogs. Each simulation component is associated with a parameter dialog, which lists all its external parameters. External parameters are properties of a component that are meant to be viewable or editable by the users. For example, the transmission rate of a traffic source component may be configurable by the user, and therefore it is shown as an external parameter in the parameter dialog of this component. The UMJaNetSim allows ample flexibility in representing these external parameters, through a consistent base object called *SimParameter* (see Section 4.2.3).



IPTCP3

Bit Rate (MBits/s)	1.0
Buffer size (bytes)	1000000
Start time (usecs)	1000000
Transmission size (bytes)	200000
Mean packet processing time (uSec)	0
Packet processing time variation (uSec)	0
Max segment size (octets)	512
My Receive Window Size (octets)	64000
TCP Version	New Reno
Timer granu. in us (e.g. 100000,500000)	1.0

IPTCP1

Bit Rate (MBits/s)	1.0
Buffer size (bytes)	1000000
	1000000
	200000
	0
ec)	0
	512
	64000
	New Reno
0)	1.0

Figure 4.5 UMJaNetSim Screenshot (Parameter Dialogs)

Each external parameter may be associated with a meter dialog. A meter dialog is normally used for the graphical display of a particular output value of a component. For example, the current utilization percentage of a network link can be displayed as a graph with the x-axis as time and the y-axis as percentage value. This graph is then constantly updated throughout the execution of the simulation. The ability to present an output visually can be very helpful for instant analysis of the output. Figure 4.6 shows some meter dialogs displaying real-time graphs of simulation information.

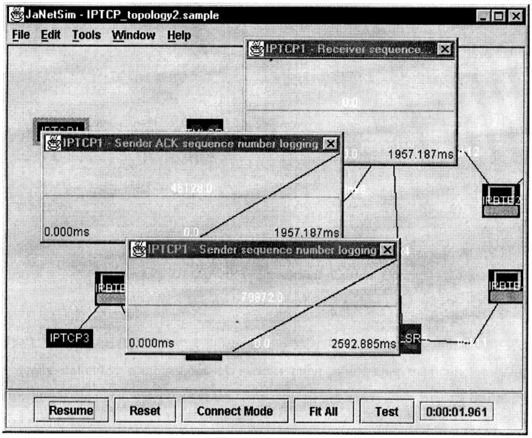
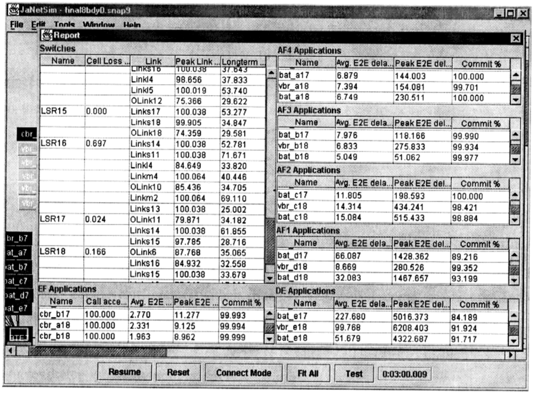

JaNetSim - IPTCP_topology2.sample
File Edit Tools Window Help
IPTCP1 - Receiver sequence...
0.0
1957.187ms
IPTCP1 - Sender ACK sequence number logging
46128.0
0.000ms
0.0
1957.187ms
IPTCP1 - Sender sequence number logging
79872.0
0.000ms
0.0
2592.885ms
IPTCP3
Resume
Reset
Connect Mode
Fit All
Test
0:00:01.961

Figure 4.6 UMJaNetSim Screenshot (Meter Dialogs)

In order to ensure full extensibility of the simulator, each parameter dialog or meter dialog can create and maintain one or more *custom* dialogs. These custom dialogs may be used to show extra information about the simulation in question. Figure 4.7 shows a custom dialog displaying custom simulation results.



Switches

Name	Cell Loss ...	Link	Peak Link ...	Longterm ...
		Links16	100.038	37.643
		Linkl4	98.656	37.833
		Linkl5	100.019	53.740
		OLink12	75.366	29.622
LSR15	0.000	Links17	100.038	53.277
		Links18	99.905	34.847
		OLink18	74.359	29.581
LSR16	0.697	Links14	100.038	52.781
		Links11	100.038	71.671
		Linkl4	84.649	33.820
		Linkm4	100.064	40.446
		OLink10	85.436	34.705
		Linkm2	100.064	69.110
		Links13	100.038	25.002
LSR17	0.024	OLink11	79.871	34.182
		Links14	100.038	61.855
		Links15	97.785	28.716
LSR18	0.166	OLink6	87.768	35.065
		Links16	84.932	32.558
		Links15	100.038	33.679

EF Applications

Name	Call acce...	Avg. E2E ...	Peak E2E ...	Commit %
cbr_b17	100.000	2.770	11.277	99.993
cbr_a18	100.000	2.331	9.125	99.994
cbr_b18	100.000	1.963	8.962	99.999

AF4 Applications

Name	Avg. E2E dela...	Peak E2E del...	Commit %
bat_a17	6.879	144.003	100.000
vbr_a18	7.394	154.081	99.701
bat_a18	6.749	230.511	100.000

AF3 Applications

Name	Avg. E2E dela...	Peak E2E del...	Commit %
bat_b17	7.976	118.166	99.990
vbr_b18	6.833	275.833	99.934
bat_b18	5.049	51.062	99.977

AF2 Applications

Name	Avg. E2E dela...	Peak E2E del...	Commit %
bat_c17	11.805	198.593	100.000
vbr_c18	14.314	434.241	98.421
bat_c18	15.084	515.433	98.884

AF1 Applications

Name	Avg. E2E dela...	Peak E2E del...	Commit %
bat_d17	66.087	1428.362	89.216
vbr_d18	8.669	280.526	99.352
bat_d18	32.083	1467.657	93.199

DE Applications

Name	Avg. E2E dela...	Peak E2E del...	Commit %
bat_e17	227.680	5016.373	84.189
vbr_e18	99.768	6208.403	91.924
bat_e18	51.679	4322.687	91.717

Figure 4.7 UMJaNetSim Screenshot (Custom Dialog)

The UMJaNetSim GUI takes advantage of the object-oriented nature of Java to allow virtually unlimited combinations of various input and output manipulation. This structure is simple in nature but at the same time allows a very high level of flexibility and extensibility.

4.1.3 Simulation Components, Parameters and Events

The primary simulation objects in the UMJaNetSim are called simulation components, each represented by a SimComponent object. From the point of view of the simulation engine, all simulation objects are *SimComponents*. This is another example of the advantages of object inheritance and polymorphism. The *SimComponent* is a well-defined base object with all the necessary interfaces that enable the interaction between the simulation engine and the component. Actual simulation components inherit the properties and methods of this base object, or in

Java terms, they are objects that "extend" this base object. In the SimComponent, the proper methods can easily modify the default interaction with the simulation engine (e.g. the component graphical image). With this, the component designer needs not be concerned with the issues of "talking" to the simulation engine, instead, the focus is on the design of the proper behaviors of the components to achieve the simulation objectives.

In order to allow configuration of component properties and display of simulation outputs, a *SimComponent* must expose a set of external parameters as stated in Section 4.2.2. Each of these parameters is an object, again, derived from a base object called *SimParameter*. The *SimParameter* object has well-defined interfaces that the simulation engine can interact with it. For example, any parameter that contains a numerical value can be displayed through a meter dialog without any programming effort from the component designer, because the appropriate mechanism to connect to a meter dialog is already built in.

The event scheduler invokes an event handler in every simulation component in order to trigger an event. In fact, this event handler is simply a well-defined method (*action()*) in the *SimComponent* that accepts a *SimEvent* object as its parameter. The *SimEvent* object has the complete description of an event including the event ID, the source component, and the optional parameters that come with the event. All components should override the *action()* method in order to react to events. All interactions between simulation components are achieved through the sending of messages in the form of a *SimEvent*. Refer Section 4.1.1 for a description about the event management architecture of the simulation.

4.2 UMJaNetSim API

In order to provide a consistent way of creating simulation components, an Application Programming Interface (API) is developed for the UMJaNetSim. The API focuses on defining well-known interfaces for simulation objects (*SimEvent*, *SimComponent* and *SimParameter*). Additionally, the API also defines essential services that the simulation engine provides. The discussion of the UMJaNetSim API will provide insights into some design issues of the simulator.

Before the discussion of individual simulation object's APIs, a look at the dynamic aspect of the simulator is in order. The UMJaNetSim utilizes the run time object-linking feature of Java through the use of the Reflection API[SUN]. This means that at the simulation compile time, the simulation engine actually does not know about the possible simulation components that may be used. A single class, the SimProvider class, provides IDs associated with these components, the names of the components and the event. This feature allows the distribution of (possibly third-party) simulation components without the need to recompile any of the code except the small *SimProvider* class that lists all the components involved.

4.2.1 Simulation Events (SimEvent)

A *SimEvent* object fully describes a particular event that is invoked on a target component. It contains an event ID, the source and destination component, the time for that event, and possibly a set of additional parameters describing the event. The UMJaNetSim takes advantage of the *Object []* array for passing of parameters to allow virtually anything to be passed through the event.

An event can be either public or private. A public event is an event passed between components of different types. A private event is an event passed between components of the same types, normally for a component itself. Public event IDs are globally significant, and need to be defined in the *SimProvider* class. Private event IDs are only locally significant, and therefore do not require global definition.

Constructor:

```
SimEvent(int aType,SimComponent src,SimComponent dest,
           long aTick,Object [] params);
```

Access methods:

```
int getType();
      //retrieve the event type

SimComponent getSource();
      //retrieve the source SimComponent

SimComponent getDest();
      //retrieve the destination SimComponent

long getTick();
      //retrieve the event-firing time

Object [] getParams();
      //retrieve the event parameters
```

4.2.2 Simulation Components (SimComponent)

The *SimComponent* object is the most important object in the simulator from a component designer's point of view. Every simulation component must inherit this base class in order to obtain the capability to interact with the simulation engine. In order to support a large variety of simulation components, every component must have a class (not to be confused with Java class) and a type. Each component class has a class ID and each component type has a type ID. These IDs are defined together with the component names in the *SimProvider* class (see Section 4.2).

Every *SimComponent* has a reference to the main object of the simulation engine, the *JavaSim* object in order to access services provided by the simulation engine. Every *SimComponent* also maintains a list of all its external parameters (in the form of *SimParameter* objects) and a list of all its *neighbor*s.

The neighbors of a component are all components that are directly connected to the component. For example, a network link component will most possibly have two neighbors, representing two components at the two ends of that link. A *SimComponent* interacts with the simulation engine by telling it about eligible neighbors (components that can be connected), while the simulation engine will signify the component when a user connects an eligible component to it.

Constructor:

```
SimComponent(String aName,int aClass,int aType,
                JavaSim aSim,Point loc);
```

Important fields:

```
protected transient JavaSim theSim;
     //a reference to the main JavaSim object

protected java.util.List neighbors;
     //a list of all (directly connected) neighbors
     //of the SimComponent

protected java.util.List params;
     //a list of all external parameters of the SimComponent
```

Neighbor Operations:

```
boolean isConnectable(SimComponent comp);
     //this is called by the simulation engine when a new
     //component is about to be connected to this component.
     //The simulation engine queries the eligibility of
     //a component to become a neighbor through this method.

void addNeighbor(SimComponent comp);
     //this is called by the simulation engine when a new
     //neighbor is connected to this component.

void removeNeighbor(SimComponent comp);
     //this is called by the simulation engine when a
     //neighbor is disconnected from this component.

void removeNeighbors(java.util.List comps);
     //this is called by the simulation engine when a
     //group of neighbors is disconnected from this
     //component.
```

4.2.2.1 Instant Information Passing

The UMJaNetSim uses a consistent way for interaction between simulation components through the passing of events. There is another way of information passing between components without the need to schedule an event. This "instant" passing of information is not used to simulate component behaviors (it is unrealistic), but instead it helps to reduce the amount of state information in a component, among other things. For example, a real network switch has a finite number of ports, each with a certain link input/output capacity (bandwidth). For simulation purposes, a virtual switch can have an infinite number of ports, where each port can be used for any types of link. The link information can be obtained during simulation run time by performing a simple query on the link component using this "instant" information passing method instead of preparing external parameters that need manual user input or rely on passing of events. This feature increases the performance and efficiency of simulations.

Component Information:

```
Object [] compInfo(int infoid,SimComponent source,
                      Object [] paramlist);
     //This method provides a way for inter-component
     //information exchange without sending run time
     //events.
```

4.2.2.2 "Copy and Paste" Support

"Copy and paste" capability is considered a standard GUI feature in modern applications. This feature is implemented in the UMJaNetSim through the use of a component-defined *copy()* method. Every component is responsible for correctly copying itself through this method. With this method properly implemented, the simulation engine will perform all other tasks involved in a copy and paste operation.

Copy Operation:

```
void copy(SimComponent comp);
     //This method is used to copy parameter values from
     //another SimComponent of the same type.
```

4.2.2.3 The Event Handler

The most important method for event handling is the *action()* method. Additionally, in order to support detailed simulation execution flows, other handler methods are needed. These include the *reset()*, *start()* and *resume()* methods. In order to emphasize the actual component behavior processing in the *action()* method, the methods involving interaction with the simulation engine are separated in the UMJaNetSim, hence these additional methods.

Event Handlers:

```
void reset();
     //Override this to perform a reset operation in order
     //to bring the status of the component back to the same
     //status as if it were just newly created.

void start();
     //Override this to perform any operations needed when
     //the simulation starts (the user clicks the "Start"
     //button)

void resume();
     //Override this to perform any operations needed when
     //the user clicks the "Resume" button after a pause.
     //One possible use is to capture any special changes
     //that have been done by the user during the pause
     //period.

void action(SimEvent e);
     //This is the event handler of this component, and will
     //be called by the simulator engine whenever a SimEvent
     //with this component as the destination triggers.
```

4.2.3 Component Parameters (SimParameter)

As stated in Sections 4.1.3 and 4.2.2, every simulation component must maintain a set of external parameters. In addition to these external parameters, a typical component normally has many properties that are internal to it such as those needed to maintain the state of the component. All parameters that need user intervention (e.g. configurable property of the component), output display and logging to a disk file should be implemented as external parameters.

The *SimParameter* is the base class for all external parameters. It provides the necessary mechanisms to interact with the simulation engine, including the support of displaying output values in meter dialogs (Section 4.1.2) and the logging of values to a simulation log file. Real time display of values through the meter dialogs provides instant feedback about the simulation outcome, while logging of values to a log file allows more detailed analyses of the simulation results.

One example of the simplest type of parameters is a parameter that simply holds one single value, such as an integer. This value can represent any properties of a component that requires an integer value, such as the buffer size of a network switch. On the other hand, a parameter can represents a complex piece of information, such as the entire routing table of a network router. Making the routing table an external parameter enables the user to view this table anytime during the simulation. This flexible use of component parameters opens up a wide range of interaction methods with the simulation components without modifying any part in the simulation engine.

Constructor:

```
SimParameter(String aName,String compName,
             long creationTick,boolean isLoggable);
     //A parameter the extends SimParameter should include
     //additional construction parameters as needed
```

Access methods:

```
String getString();
     //returns a String representation of the parameter value
     //(This is used for logging purpose)

void globalSetValue(String value);
     //This is to support setting of the same parameter values
     //for multiple components in one command.
```

Sample access methods of a parameter holding an integer value:

```
int getValue();
     //read value

void setValue(int val);
     //write value
```

As stated earlier, a parameter can be very complex, and the parameter itself can create and manage additional custom dialogs (Section 4.1.2). All the parameters are contained within the parameter dialog as described in Section 4.1.2. The parameter dialog actually contains a list of *JComponent*, a generic Swing GUI component. For a single-value type parameter (like the integer), this component can simply be a *JLabel* if it is not editable, or a *JTextField* if it is editable. A complex parameter type may just use a *JButton* that opens up new custom dialogs when invoked. The choice of components to use is dependent on the type of interaction needed by the component designer. In general, the reusability of a parameter type depends on how general the parameter is. Certain predefined parameter types (integer, double, boolean etc.) are general-purpose in nature and are useful in most cases.

GUI access method:

```
JComponent getJComponent();
```

4.3 UMJaNetSim Features

The features of the UMJaNetSim Network Simulator are summarized as follows:

- *Object-oriented.* The UMJaNetSim is fully object-oriented using the Java programming language, and therefore takes full advantage of the object-oriented paradigm.
- *Simplicity.* Based solely on the message passing mechanism between simulation components, the simulator provides a simple yet powerful way of creating various simulation environments. This use of the object-oriented approach further ensures the simplicity.
- *Portability.* Due to the platform independent nature of the Java programming language, the UMJaNetSim has high portability among various platforms. Furthermore, the simulator is readily web-enabled by using an applet version of the simulator.
- *Extensible API.* The UMJaNetSim API simplifies component development and shifts the development effort to the actual network research rather than general simulation management.
- *GUI.* The GUI of UMJaNetSim includes features that simplify simulation scenario design and manipulation. These include the ability to copy and paste simulation components, simultaneous parameter setting for multiple components, and full state saving/restoration using object serialization. Moreover, the GUI is fully extensible through the UMJaNetSim API, for example by creating custom dialogs.

4.4 Chapter Summary

The first section of this chapter discussed the architecture of UMJaNetSim v0.5, including Event Management; GUI Management; and Simulation Component, Parameters and Event.

The second section discussed UMJaNetSim Application Programming Interface (API), such as Simulation Events, Simulation Component and Component Parameter.

The third section discussed UMJaNetSim features. The next chapter describes the design, implementation and testing of the TCP simulation environment using the UMJaNetSim.

This thesis will use UMJaNetSim v0.5 as the project network simulator. A new component, NewReno, will be integrated into this network simulator. Tahoe and Reno already exists in the simulator. Performance evaluation of these three TCP versions will be done on the simulator.