March 29, 2002
RT0454
Network 31 pages

# Research Report

## Preliminary Prototype of End-To-End Performance Simulator

Tsuyoshi Id¥'e and Toshiyuki Hama

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

**Research Division**
**Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich**

**Limited Distribution Notice**
This report has been submitted for publication outside of IBM and will be probably copyrighted if accepted. It has been issued as a Research Report for early dissemination of its contents. In view of the expected transfer of copyright to an outside publisher, its distribution outside IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or copies of the article legally obtained (for example, by payment of royalities).

# Preliminary Prototype of End-To-End Performance Simulator

Tsuyoshi Idé Toshiyuki Hama

March 29, 2002

**Abstract**

This paper describes a preliminary prototype of end-to-end performance simulator, which forms a part of a 2002/1Q milestone of Server System Institute Project #34 titled "*Autonomic e-Business Server Management and Analysis*". As understood from the title, we intend to sketch some of essential aspects of the whole model, rather than realistic data analysis. What we focus on is correlation effects between events. With the aid of a simple mathematical solution, we demonstrate that the distribution of response time is considerably affected by the correlation. Explicit source codes are listed in Appendix.

## 1 Introduction

Today, we observe that growing complexity of the information technology (IT) infrastructure threatens to undermine the very benefits IT aims to provide. Obviously, there are acute needs to manage the complexity in an autonomic manner.

To study the performance of IT infrastructure including the Internet, intensive efforts have been devoted. Studies on TCP algorithms [1], routing algorithms [2], and statistical physical studies on Internet traffic [3] are such examples. Although they successfully describe individual phenomena to some extent, they are still far from providing the basis of the autonomic computing environment. Hence, it is very significant to integrate these fragments of theories, and to implement a simulator that calculates the end-to-end performance, encapsulating the complexity of the whole IT infrastructure.

Our prototype is the first step of the ambitious project, 2002 Server System Institute Project #34 titled "*Autonomic e-Business Server Management and Analysis*". There are many metrics to evaluate the end-to-end performance. To be concrete, we mainly consider response

time of requests, which is defined as the difference between when a request happens at an end-user client and when a response to the request reaches the end-user.

In this paper, we introduce a discrete-event simulator with a minimal construction under a simple linear topology, and sketch some of essential aspects of the whole model, rather than realistic data analysis. In what follows, we first explain principles of the model we adopt. Then, we explain results of calculation. What we focus on is the correlation effect between events. With the aid of a simple mathematical solution, we will demonstrate that the distribution of response time is considerably affected by the correlation. In Appendix, explicit source codes are listed.

## 2 Discription of Model

We describe the overview of our prototype in Fig. 1. As shown, the model contains three ingredients: an event source, network components, server components. The network components consist of models of a "last mile" and a backbone, and the server components consist of a HTTP server, a Web application server (WAS), and a database sever (DB). We describe them in what follows.

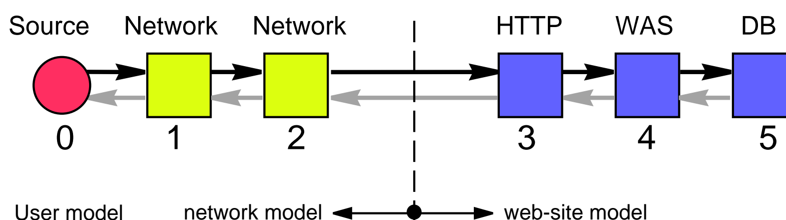


Figure 1: The overview of the model

### 2.1 Source

The source provides a model of access patterns of end-users. We assume that there are three kinds of requests, namely, (1) acquisition of only HTML data, (2) acquisition of HTML data and execution of some process at the WAS, and (3) acquisition of HTML data and execution of some process at both the WAS and the DB. These request types are chosen randomly in accordance with a predetermined probability when each request is generated at each time point. We summarize the model in Fig. 2, where the above three requests are denoted by H, W,

and D, respectively. All of requests are treated as 0.5 kB files at the source, thereafter the size is magnified by a factor of 25 when the files reach the HTTP server, so that all of requests have the same data size of 12.5 kB when they return and die at the source.

Since this is a request-based calculation, we disregarded detailed behavior of Internet protocols such as TCP/IP. Obviously, this leads to an oversimplification since retransmission mechanisms and branching of paths are essential to reproduce the observed self-similar behavior of the Internet traffic [4]. This subject will be discussed extensively in the next phase of our prototype.

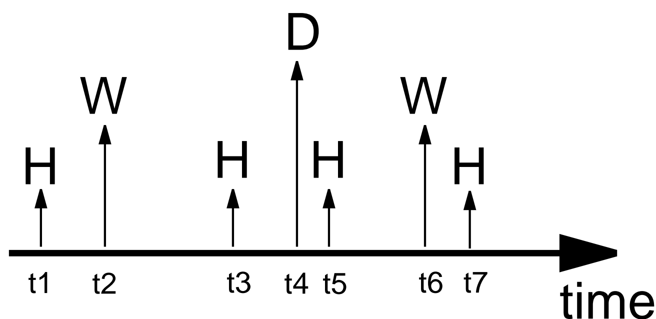


Figure 2: Explanation of the source. The time points , $t_1$, $t_2$... are chosen according to the Poisson distribution. Three destinations (H, W, D) are probabilistically assigned for each points (see the text).

## 2.2 Network model

The network component marked with 1 in Fig. 1 is a simple first-in-first-out (FIFO) queue with the infinite size. As a simple model of the "last-mile", we give each request a delay time simply determined by $D/B$, as schematically shown in Fig. 3, where $D$ is the size of data of a request, and $B$ is a bandwidth. We set $B$ to be 0.1 kB/ms in the results shown below. By definition, the delay time of inbound paths is different from outbound paths, corresponding to the difference in file sizes.

The network component marked with 2 is also a simple FIFO queue with the infinite size. As a simple model of backbones, a fluctuation of the bandwidth due to the background traffic is taken into account. Because of the fluctuation, delay time at this component is not deterministic, but of some distribution. Detailed analysis of this simple model will be described in the next section.

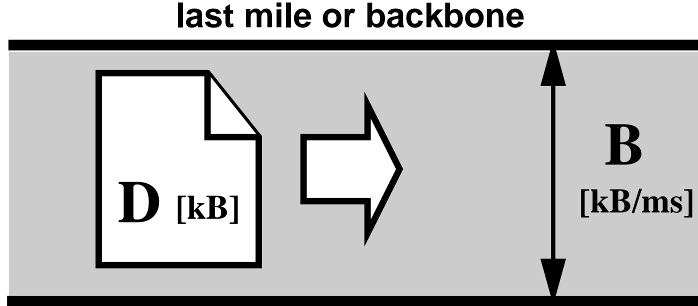


Figure 3: Schematic description of network components. Delay time is simply calculated as $D/B$. To simulate the fluctuation due to the Internet, the bandwidth is randomly sampled under a probability distribution (see Sec. 3.1).

### 2.3 Server model

In the network components, each request is treated independently. In the server components, however, there is interference between events. For example, processing time of an event will get longer if other events come into the same server because of reduction of resources allocated to that event. In general, one has to consider several kinds of resources such as CPU, memory, I/O, etc., and also complex transactions of server side programs. In this prototype, however, we considered only CPU-like resource, and assumed that the resource is allocated equally for each event.

Figure 4 explains the procedure for calculating time for events to be spent at server components. At a time point of $t_1$, we have $N$ events in the component ($N = 3$ is shown in the figure), and times at which the component finishes jobs to process the events are given as $T_1$, $T_2$, etc. When another event enters the server component at a time point of $t_2$, the times are renewed as $T_1'$, $T_2'$..., where

$$T_1' = t_2 + \frac{(N+1)D_1'}{C},$$

etc. Here, $D_1'$ is unprocessed amount of job of the event 1 at the time point of $t_2$, and $C$ is the total CPU-like resource of the component. Similar renewal procedure is needed also when an event completes its process and goes out of server components.

It is important to understand that a variety of time orders of events can occur in this renewal process. For example, if a newly joined event has very small amount of job, the sever component will complete the job of this newly joined event the earliest. Thus, we must recalculate the all times whenever events enter.

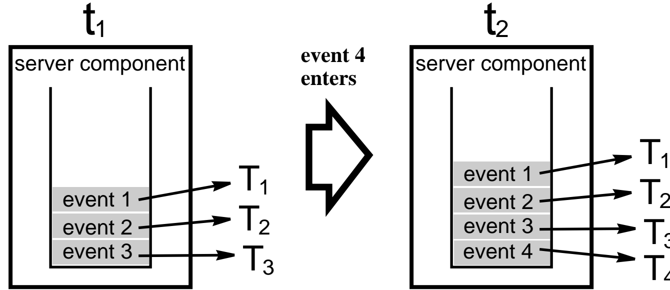


Figure 4: Explanation of the interference between events in a server component. Finishing times of events are renewed when another event enters the component. Note that $T_4$ may be smaller than the other, depending on its job amount.

### 2.4 Procedure of calculation

The key of this discrete event simulation is to manage effectively the time-renewal process in the server components. For this purpose, we furnished each of nodes (source, network components, and servers) with a priority queue with the infinite size, called eventList, as shown in Fig. 4. In addition, we defined another priority queue called node-Manager, where all nodes are ordered according to times at which each node finishes a job of an event the earliest.

With these queues, the whole procedure of calculation is essentially a problem of sorting: Take the top node (say A) in the nodeManager, and identify an event that makes a transition to another node (say B). Renew the eventList of both A and B, and then renew the nodeManager.

## 3 Results

### 3.1 Analytic function of the distribution of response time

To validate the calculations of response time, which is the most important metric to assure service-level agreements, we derive an analytical function of the distribution before showing results of calculation. The delay time, $Y$, through the backbone is written as

$$Y = \frac{D}{B},$$

where $D$ and $B$ are the size of the request and the bandwidth, respectively. For $B$, we have assumed the following function:

$$B = B_0 - 2B_0 rx,$$

where $0 \leq D < 1$ is a dimensionless constant and $-1 \leq x < 0.5$ is a stochastic variable subject to a constant distribution. $B_0$ is the average of $B$.

The above two equations define a transformation from a stochastic variable $x$ to $Y$, i.e. $Y$ is a function of $x$. A standard Jacobian manipulation shows the probability distribution of $Y$,

$$p(Y) = \frac{dx}{dY} p(x) = \frac{D}{2B_0 r Y^2}$$

for

$$\frac{D}{(1+r)B_0} \leq Y < \frac{D}{(1-r)B_0}.$$

$p(Y)$ is zero outside the above domain. According to this function, the contribution of the backbone in response time has inverse-square dependence on time over a finite range.

### 3.2 Results of calculation

Figure 5 shows the distribution of response time with varying average time interval ($\beta$) of the Poisson distribution at the source, and the total CPU-like resource in the HTTP server. As shown in (a), bandwidths of the network components are set to be 0.1 kB/ms (the last-mile) and 0.15-0.45 kB/ms (the backbone). The breakdown of user requests is fixed to be (HTTP, WAS, DB)=(0.8, 0.1, 0.1). The total resources of WAS and DB server components are 0.1 and 0.5 kB/ms, respectively.

The curve at the top of (a) is the distribution for $\beta = 2000$. In this parameter, each request is processed almost independently since this value is much larger than average response time itself. We can see that the curve has a finite range and sharp peak at about 140ms, being consistent with the result in Sec. 3.1. For smaller $\beta$, we observe spectral weight of the peak at the lower bound shift to longer response time. This trend is clearer with decreasing $C$. The reason of this effect is attributed to the interference effect inside the HTTP server. Roughly speaking, we can understand the line shapes as the inverse-square function with some modification by correlation between events in server components.

The interference or correlation effect is the most clearly observed for $C = 0.5$, where the distribution suddenly expands far larger scale (see (b)). Note that the drastic transition of this kind cannot be described by theories in which the correlation between events is not treated explicitly.

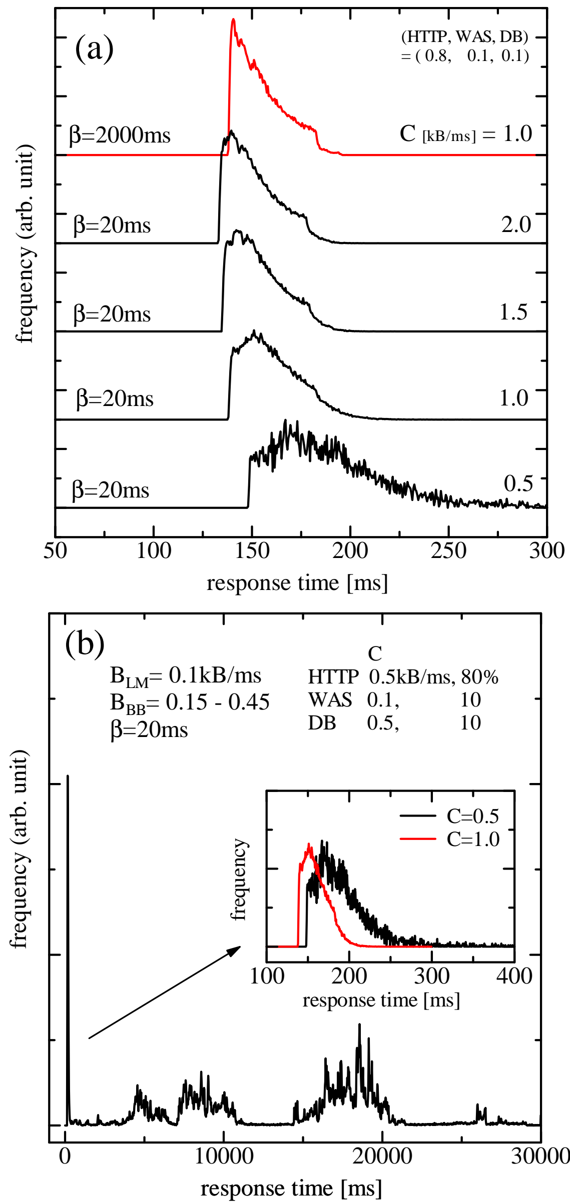


Figure 5: Distribution of response time. (a) The dependence on the total CPU-like resource of the HTTP server. (b) Wider-range view of the bottom curve in (a), which is also shown in the inset.

# 4 Concluding Remarks

We implemented a toy model of the end-to-end performance simulator. Although the model omits several matters such as topologies of the network, the contribution of background traffic in the Internet, etc., it clearly demonstrates the essential role of the correlation effects between events, which is not reproducible by using conventional queuing theories.

Naturally, further sophistication is needed to analyze data in the real world. Since the burst nature of Internet traffic is crucial in capacity planning of Web sites, refinement of the traffic model is of particular importance. It is well known that the Poisson distribution is not consistent with the actual access pattern of Web sites [5]. Recently, Koide proposed an extended Poisson distribution to reproduce the actual traffic pattern [6, 7, 8]. As aforementioned, Fukuda, Takayasu, and Takayasu [4] demonstrated that both a function of TCP algorithm and branching of network are indispensable to reproduce the observed self-similarity even with the Poisson distribution at the end-user. Apart from the traffic pattern, there are many subject to study to simulate actual data faithfully. These subjects will be discussed the next terms of the project.

# Acknowledgements

The authors thank Drs. S. Tezuka, A. Koide, and J. Shimizu for stimulating discussion.

# References

[1] David M. Nicol, "Discrete Event Fluid Modeling of TCP", Proceedings of the 2001 Winter Simulation Conference, 1291.

[2] K. Oida, M. Sekido, "An agent-based routing system for QoS guarantees", Proceedings of IEEE International Conference on Systems, Man, and Cybernetics, 833-838, 1999.

[3] A. Yu. Tretyakov, H. Takayasu and M. Takayasu, "Phase transition in a computer network model", Physica A: Statistical Mechanics and its Applications, **253**, 1998.

[4] K. Fukuda, H. Takayasu, and M. Takayasu, "Self-Similar Traffic Originating in the Transport Layer", Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS'01), p.55-60, SCS, Phoenix, USA, Jan. 2001.

[5] R. Gusella, "A Measurement Study of Diskless Workstation Traffic on an Ethernet", IEEE Trans. on Communication, **38**,1557-1568, 1990.

[6] A. Koide, "Study of Internet Traffic -1, Extended Poisson Distribution", IBM Research Report, RT0428, 2001.

[7] A. Koide, "Study of Internet Traffic -2, Micro Model of Traffic", IBM Research Report, RT0429, 2001.

[8] A. Koide, "Study of Internet Traffic -3, Analysis of Real Data", IBM Research Report, RT0430, 2001.

# A Appendix: Source Codes

## A.1 Structure of classes

We sketch the structure of classes of the simulator in Fig. 6. As shown, all of the components in Fig. 1 extends an abstract class of Node. There are two kinds of priority queue. One manages all of nodes, and the other is instantiated in each node object to manage events. Network utilizes FifoQ class to calculate delay time of events (see Sec. 2.4).

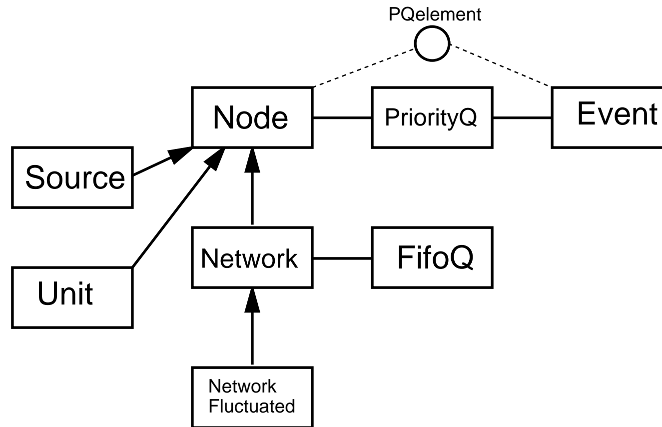


Figure 6: Relationship between classes.

## A.2 Event

Event describes fundamental attributes of events. Each event has its own clock to record when it is generated, when the present process started, etc. It also includes information of job to be processed. Each event is regarded as an element of priority queues inside the component.

```
package com.ibm.fmds.simulator;
import java.util.*;
class Event implements PQelement
{
    private double tGen;    // when event generated
    private double tStart;  // when job started
    private double tFinish; // when job will finish
    private double restWork; // residual amount of job
    private double workAmount; // amount of the total job
    private double age;     // age of event
    private int idReq;      // ID of request
    private Node idSource;
    private Node presentNode;   // reference to the present node
    private boolean returnback = false;
```

```
    ////////////////////////////////////////////////////////
    public double getFinishTime(){return  tFinish;}
    public double getGenTime(){return tGen;}
    public Node getPresentNode(){return presentNode;}
    public int getReqID(){return  idReq;}
    public double getRestWork(){return restWork;}
    public Node getSource(){return idSource;}
    public double getStartTime(){return  tStart;}
    public double getValue(){return tFinish;}
    public boolean isThisReturnPath() {return this.returnback;}
    public void resetRestWork(){this.restWork=this.workAmount;}
    public void resetWorkAmount(double mag)
        {this.workAmount = mag*this.workAmount;}
    public void setFinishTime(double x){tFinish=x;}
    public void setGenTime(double x){tGen=x;}
    public void setPresentNode(Node id){presentNode=id;}
    public void setReqID(int id){idReq=id;}
    public void setRestWork(double x){restWork=x;}
    public void setSource(Node id){idSource=id;}
    public void setStartTime(double x){tStart=x;}
    public void setWorkAmount(double x){workAmount=x;}
    public void thisIsReturnPath() {this.returnback = true;}
}
```

## A.3 Node

Node is an abstract class of all the components of source, unit, and network. The topology of system is represented in getNextNode method, which tells events what is the next node. Tentatively, the linear topology is assumed there.

```
package com.ibm.fmds.simulator;
import java.util.*;
abstract class Node implements PQelement
{
    protected int id = -1;
    protected Node[] nodeList;
    protected EasyAnalyzer easyAnalyzer=null;
    ////////////////////////////////////////////////////////
    abstract public void enter(Event event);
    abstract public int howManyEvents();
    //------------------------------------------
    public Node getNextNode(Event event)
    {
        /* Return the reference of the next node
        judging from information of event and node.
        The following topology is tentatively assumed.
            S     N     NF    U     U     U
            @-----*-----*-----*-----*-----*
            0     1     2     3     4     5
        */

        Node nextNode= null;
        int reqID = event.getReqID();
        boolean comeback=event.isThisReturnPath();
```

```
if(this.id == 0)
{
    nextNode = nodeList[1];
}
else if(this.id == 1)
{
    if(comeback==false)
        nextNode = nodeList[2];
    else
        nextNode = nodeList[0];
}
else if(this.id == 2)
{
    if(comeback==false)
        nextNode = nodeList[3];
    else
        nextNode = nodeList[1];
}
else if(this.id == 3)
{
    if(reqID==0){
        nextNode = nodeList[2];
    }
    else{
        if(comeback==false)
            nextNode = nodeList[4];
        else
            nextNode = nodeList[2];
    }
}
else if(this.id == 4)
{
    if(reqID==0){
        System.out.println("Id=0 is not reachable to WAS !!");
    }
    else {
        if(comeback==false)
            nextNode = nodeList[5];
        else
            nextNode = nodeList[3];
    }
}
else if(this.id == 5)
{
    if(reqID==2){
        nextNode = nodeList[4];
    }
    else{
        System.out.println("Id=0&1 is not reachable to DB!!");
    }
}
else
{
    System.out.println("?? There is no fork.");
}
```

```
        return nextNode;
    }
    //-------------------------------------------
    public void setEasyAnalyzer(EasyAnalyzer ea) {this.easyAnalyzer = ea;}
    public void setNodeID(int ide) {this.id=ide;}
    protected void setNodeList(Node[] nl) { nodeList = nl;}
}
```

## A.4 Network

Network is a class that implements the model of the last-mile component. It simply gives delay time according to file sizes of events.

```
package com.ibm.fmds.simulator;
import java.util.*;
class Network extends Node
{
    protected int maxSize;
    protected double totalResource; // bandwidth [kB/ms]
    protected FifoQ eventList;
    ////////////////////////////////////////////////////////
    public Network(int maxsize, double totalresource)
    {
        this.totalResource = totalresource;
        this.maxSize = maxsize;
        eventList = new FifoQ(this.maxSize);
    }
    //-------------------------------------------
    public void enter(Event addedEvent)
    {
        // present time = time of the transition
        double timeNow = addedEvent.getFinishTime();
        // job amount is reset
        addedEvent.resetRestWork();
        double restWork = addedEvent.getRestWork();
        // the present job starts now
        addedEvent.setStartTime(timeNow);
        // what is the time spent
        double timeSpent= this.getTimeSpent(addedEvent);
        addedEvent.setFinishTime(timeNow + timeSpent);
        // insert an event that makes a transition to this node
        eventList.insert(addedEvent);
    }
    //-------------------------------------------
    private double getTimeSpent(Event event) {
        // the time spent is proportional to amount of job
        double result=0;
        if(event.isThisReturnPath()==false){ // outbound
            result = event.getRestWork() / this.totalResource;
        }
        else{// inbound
            result = event.getRestWork() / this.totalResource;
        }
        return result;
    }
```

```
    //-----------------------------------------implementation of PQelement
    public double getValue()
    {
        double value;
        if( eventList.noOfElements() == 0)
        {
            value = 1.e10;
        }
        else
        {
            Event topEvent = (Event) eventList.peekFront();
            value = topEvent.getFinishTime();
        }
        return value;
    }
    //-----------------------------------------
    public int howManyEvents()
    {
        return eventList.noOfElements();
    }
    //-----------------------------------------
    public double peekTopTime()
    {
        Event event = (Event)eventList.peekFront();
        return event.getFinishTime();
    }
    //-----------------------------------------
    public Event removeTopEvent()
    {
        // remove the top event from this node
        Event transitionEvent= (Event) ( (this.eventList).remove() );
        // return the event removed
        return transitionEvent;
    }
}
```

## A.5 NetworkFluctuated

NetworkFluctuated is a class that implements the backbone component, extending Network. For detailed explanation, see Sec.3.1.

```
package com.ibm.fmds.simulator;
import java.util.*;
class NetworkFluctuated extends Network
{
    double ratio;
    Random random;
    ////////////////////////////////////////////////////////
    public NetworkFluctuated
        (int maxsize, double totalresource, double rat, long seed){
        // making a FifoQ object
        super(maxsize, totalresource);

        this.ratio=rat;
        random = new Random(seed);
    }
```

```
    //-----------------------------------------
    // insert an event that makes a transition to this node
    public void enter(Event addedEvent)
    {
        // the present time = the time to make a transition
        double timeNow = addedEvent.getFinishTime();
        // reset amount of job
        addedEvent.resetRestWork();
        double restWork = addedEvent.getRestWork();
        // Now the process starts
        addedEvent.setStartTime(timeNow);
        // What is the time spent
        double timeSpent= this.getTimeSpent(addedEvent, this.ratio);
        addedEvent.setFinishTime(timeNow + timeSpent);
        // insert an event that makes a transition to this node
        eventList.insert(addedEvent);
    }
    //-----------------------------------------
    private double getTimeSpent(Event event, double ratio) {
        // bandwidth fluctuates over the range of ‘‘deviation"
        // -ratio*totalResource < deviation < ratio*totalResource
        double U= random.nextDouble();
        double deviation = 2.*(U - 0.5 )*ratio*this.totalResource;
        return  event.getRestWork() / ( this.totalResource + deviation);
    }
}
```

## A.6 Source

Source generates events based on the exponential distribution. In its constructor, each event is given one of three request IDs. As shown in `enter()`, the source is also a sink of events.

```
package com.ibm.fmds.simulator;
import java.util.*;
class Source extends Node
{
    Event event;
    double initialWorkValue;
        // amount of job of events
    private java.util.Random generator;
        // random number for time interval
    private java.util.Random idgenerator;
        // random number for request IDs
    private double [] idProbability;
        // probability of request IDs
    public static double mean;
        // mean time interval
    public boolean randomInit=false;
        // a flag to indicate initialization of Random
    ///////////////////////////////////////////////////////
    public Source(double initialWorkValue_, double[] idprobability
        , double m, long seed_interval, long seed_id)
    {
        event = new Event();
```

```
    // work amount
    this.initialWorkValue=initialWorkValue_;
    event.setWorkAmount(this.initialWorkValue);
    event.resetRestWork();
    randomInit(seed_interval, seed_id);
    this.mean=m;
    double timeGen = 0.5*interval();
    event.setGenTime(timeGen);
    event.setFinishTime(timeGen);

    this.idProbability=idprobability;
    int id = this.idSampler(idProbability);
    event.setReqID(id);
    event.setSource(this);
    event.setPresentNode(this);
}
//------------------------------------------
public void enter(Event event)
{
    // Source is also a sink of events.
    // send returned events to easyAnalyzer
    easyAnalyzer.countTheEvent(event);
}
//------------------------------------------
public void enter(Event event, EasyAnalyzer ea)
{
    ea.countTheEvent(event);
}
//------------------------------------------
public double getValue()
{
    return event.getFinishTime();
}
//------------------------------------------
public int howManyEvents()
{
    return 1;
}
//------------------------------------------
public int idSampler(double[] idProb)
{   // idProb[]: probability for each IDs
    int generagedID=-1;

    // cumulative distribution function cumulativeP[]
    int numberOfID=idProb.length;
    double[] cumulativeP = new double[numberOfID];  // zeroset
    for (int ii =0; ii < numberOfID ; ii ++) cumulativeP[ii]=0.;
    cumulativeP[0]=idProb[0];
    for (int ii =1; ii < numberOfID ; ii ++){
        cumulativeP[ii]=cumulativeP[ii-1]+idProb[ii];
    }

    // sampling an ID number in [0,1]
    double U=this.idgenerator.nextDouble();
    if( U < cumulativeP[0]){
```

```
        geneagedID = 0;
    }
    else{
        for(int ii=1; ii <numberOfID; ii ++ ){
            if( (cumulativeP[ii-1] <= U) && (U < cumulativeP[ii])){
                generagedID = ii;
                break;
            }
        }
    }
}
//------------------------------------------
private double interval()
{
    double x;
    double U;
    if(this.randomInit==false)
    {
        System.out.println("Error. Initialize Random.");
        x=0.;
    }
    else
    {
        U=generator.nextDouble(); // U is uniform random number in [0,1)
        x=-1.*mean*Math.log(U);   // exponential distribution with mean
    }
    return x;
}
//------------------------------------------
public double peekTopTime()
{
    return event.getFinishTime();
}
//------------------------------------------
public void randomInit(long seed_interval, long seed_id)
{
    this.generator = new Random(seed_interval);
    this.idgenerator = new Random(seed_id);
    this.randomInit=true;
}
//------------------------------------------
public Event removeTopEvent()
{
    // Remove nothing but generates a new event
    Event transitionEvent= null;
    transitionEvent = event;
    event = new Event();
    event.setFinishTime(this.interval()+transitionEvent.getFinishTime());
    event.setGenTime(event.getFinishTime());
    event.setStartTime(transitionEvent.getFinishTime());
    event.setWorkAmount(this.initialWorkValue);
    event.resetRestWork();
    event.setReqID( this.idSampler(this.idProbability) );
    event.setSource(this);
    event.setPresentNode(this);
    return transitionEvent;
```

```
    }
}
```

## A.7 Unit

Unit is one of the most important class, where the correlation effects are modeled. A brief explanation of this class has been shown in Sec. 2.3.

```
package com.ibm.fmds.simulator;
import java.util.*;
class Unit extends Node
{
    int maxSize;
    double totalResorce;
    double magnification;
    PriorityQ eventList;
    ////////////////////////////////////////////////////////
    public Unit(int maxsize, double totalresorce)
    {
        this.totalResorce = totalresorce;
        this.maxSize = maxsize;
        eventList = new PriorityQ(this.maxSize);
    }
    //-------------------------------------------
    public Unit(int maxsize, double totalresorce, double mag)
    {
        this.totalResorce = totalresorce;
        this.maxSize = maxsize;
        this.magnification = mag;
        eventList = new PriorityQ(this.maxSize);
    }
    //-------------------------------------------
    public void enter(Event addedEvent)
    {
        // Information of an event that makes a transition here
        double timeNow = addedEvent.getFinishTime();
            // the present time = when the transition occurs
        if(this.isThisTurn(addedEvent)==true){
            addedEvent.thisIsReturnPath();
        }
        if(addedEvent.isThisReturnPath()&&(this==super.nodeList[3])){
            // If this is the HTTP server along an inbound path,
            // magnify the data size.
            addedEvent.resetWorkAmount(this.magnification);
        }
        addedEvent.resetRestWork();
        double restWork = addedEvent.getRestWork();

        // now job starts
        addedEvent.setStartTime(timeNow);

        // insert an event that makes a transition
        int oldNumberOfEvent = (this.eventList).noOfElements();
        int newNumberOfEvent = oldNumberOfEvent +1;

        // If this unit is empty....................
```

```
    if(oldNumberOfEvent==0)
    {
        // the event occupies all of resource
        addedEvent.setFinishTime(timeNow+restWork/this.totalResorce);
        // simply add the event
        eventList.insert(addedEvent);
    }
    // If this unit is not empty................
    else
    {
        // resource amount for each event
        double newResorcePerEvent
            = this.totalResorce/(double)newNumberOfEvent;
        double oldResorcePerEvent
            = this.totalResorce/(double)oldNumberOfEvent;
        // update information of the newcomer
        addedEvent.setFinishTime(timeNow+restWork/newResorcePerEvent);
        // calculate all of due times
        // assume the number of events change at timeNow
        // from oldNumberOfEvent to newNumberOfEvent
        this.enterUpdateTime(timeNow, oldNumberOfEvent, newNumberOfEvent);

        // all information is updated; add it to eventList
        eventList.insert(addedEvent);
    }
}
//--------------------------------------------
// method to update due times when an event comes
private void enterUpdateTime(double timeNow,
    int oldNumberOfEvent, int newNumberOfEvent)
{
    double oldResorcePerEvent
        = this.totalResorce/(double)oldNumberOfEvent;
    double newResorcePerEvent
        = this.totalResorce/(double)newNumberOfEvent;
    double restTime;
    double restWork, timeFinish, timeStart;
    double restWorkRevised, timeFinishRevised, timeStartRevised;

    if(oldNumberOfEvent==0)
    {
        // when empty
    }
    else
    {
        for(int ii=0; ii < oldNumberOfEvent; ii++)
        {
            // residual job amount when the present round starts
            restWork = ((Event)eventList.elementAt(ii)).getRestWork();
            // start time of the present round
            timeStart=((Event)eventList.elementAt(ii)).getStartTime();

            // residual job amount when the present round ends
            restWorkRevised =
                restWork - (timeNow-timeStart)*oldResorcePerEvent;
            // when the present round ends
```

```
            timeFinishRevised =
                timeNow + restWorkRevised / newResorcePerEvent;

            // update internal information events
            ((Event)eventList.elementAt(ii)).setStartTime(timeNow);
            ((Event)eventList.elementAt(ii)).setFinishTime(timeFinishRevised);
            ((Event)eventList.elementAt(ii)).setRestWork(restWorkRevised);
        }
    }
}
//-------------------------------------------
// for abstract method of ‘PQelement’
public double getValue()
{
    double value;
    if( eventList.noOfElements() == 0)
    {
        // when empty
        value = 1.e10;
    }
    else
    {
        Event topEvent = (Event) eventList.peekMin();
        value = topEvent.getFinishTime();
    }
    return value;
}
//-------------------------------------------
public int howManyEvents()
{
    return eventList.noOfElements();
}
//-------------------------------------------
public boolean isThisTurn(Event event) {
    boolean answer=false;
    /* The if-branching rule is based on
       the following tentative topology.
            S     N     NF    U     U     U
            @-----*-----*-----*-----*-----*
            0     1     2     3     4     5
    */
    if(event.getReqID()==0){
        if(this == nodeList[3]) answer=true;
    }
    else if(event.getReqID()==1){
        if(this == nodeList[4]) answer=true;
    }
    else if(event.getReqID()==2){
        if(this == nodeList[5]) answer=true;
    }
    return answer;
}
//-------------------------------------------
public double peekTopTime()
{
    return  eventList.peekMinValue();
```

```
}
//-----------------------------------------
public Event removeTopEvent()
{
    // 1.remove the event from the eventList
    int oldNumberOfEvent = this.eventList.noOfElements();
    Event transitionEvent= (Event) ( (this.eventList).remove() );
    // 2.if that place is the turning point, raise a flag
    //if(this.isThisTurn(transitionEvent)==true){
    //  transitionEvent.thisIsReturnPath();
    //}
    // 3.update due times in the new eventList
    if( oldNumberOfEvent ==1)
    {
        // When empty
    }
    else
    {
        double timeNow =transitionEvent.getFinishTime();
        int newNumberOfEvent = oldNumberOfEvent -1 ;
        this.removeUpdateTime(timeNow, oldNumberOfEvent, newNumberOfEvent);
    }
    // 4.return the removed event as a return value
    return transitionEvent;
}
//-----------------------------------------
// update due times when an event finishes and leave from the eventList
private void removeUpdateTime(double timeNow,
    int oldNumberOfEvent, int newNumberOfEvent)
{
    double oldResorcePerEvent
        = this.totalResorce/(double)oldNumberOfEvent;
    double newResorcePerEvent
        = this.totalResorce/(double)newNumberOfEvent;
    double restTime;
    double restWork, timeFinish, timeStart;
    double restWorkRevised, timeFinishRevised, timeStartRevised;

    if(newNumberOfEvent==0)
    {
        // when empty
    }
    else
    {
        for(int ii=0; ii < newNumberOfEvent; ii++)
        {
            // residual job amount when the present round starts
            restWork = ((Event)eventList.elementAt(ii)).getRestWork();
            // start time of the present round
            timeStart=((Event)eventList.elementAt(ii)).getStartTime();
            // residual job amount when the present round ends
            restWorkRevised =
                restWork - (timeNow-timeStart)*oldResorcePerEvent;
            // when the present round ends
            timeFinishRevised =
                timeNow + restWorkRevised / newResorcePerEvent;
```

```
                // update internal information of events
                ((Event)eventList.elementAt(ii)).setStartTime(timeNow);
                ((Event)eventList.elementAt(ii)).setFinishTime(timeFinishRevised);
                ((Event)eventList.elementAt(ii)).setRestWork(restWorkRevised);
            }
        }
    }
}
```

## A.8 PQelement and PriorityQ

PriorityQ is a class of a priority queue object, whose elements are ordered by a value that is given with `getValue()`.

```
package com.ibm.fmds.simulator;
import java.util.*;
interface PQelement
{
    abstract double getValue();
}

package com.ibm.fmds.simulator;
import java.util.*;
class PriorityQ
{
    protected PQelement [] queArray;
    protected int nItems;
    protected int maxSize;
    /////////////////////////////////////////////////////////
    public PriorityQ(int s)          // constructor
    {
        maxSize =s;
        queArray = new PQelement[maxSize];
        nItems = 0;
    }
    //------------------------------------------
    public void insert(PQelement item)  // insert item
    {
        int j=-1;
        if(nItems==0)                        // if no items,
            queArray[nItems++] = item;        // insert at 0
        else                                // if items,
        {
            try
            {
                for(j=nItems-1; j>=0; j--)         // start at end,
                {
                    if( queArray[j].getValue() < item.getValue() )
                    {     // if new item larger,
                        queArray[j+1] = queArray[j]; // shift upward
                    }
                    else                          // if smaller,
                        break;                     // done shifting
                }  // end for
```

```
            queArray[j+1] = item;             // insert it
            nItems++;
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("\n **Error! "+" nItems="+nItems+
            " j+1="+(j+1) + " item.getValue()="+ item.getValue()+
            " maxSize=" + maxSize+ "\n");

        }
    }  // end else (nItems > 0)
}  // end insert()
//--------------------------------------------
public void removeItemOf(PQelement item)  // remove item
{
    int theIndex=-1;
    for(int ii =0; ii < nItems; ii++)
    {
        if(queArray[ii] == item)
        {
            theIndex = ii;
            break;
        }
    }
    if(theIndex != -1)
    {
        // remove a item of queArray[theIndex]
        // and shift items from queArray[theIndex+1]
        // to queArray[nItems-1] by one
        for(int ii =theIndex+1; ii < nItems; ii++)
        {
            queArray[ii-1] = queArray[ii];
        }
        nItems--;
    }
    else
    {
        // do nothing when no item
    }
}
//--------------------------------------------
public PQelement remove()           // remove minimum item
    { return queArray[--nItems]; }
//--------------------------------------------
public PQelement peekMin()          // peek at minimum item
    { return queArray[nItems-1]; }
//--------------------------------------------
public double peekMinValue()         // peek at minimum VALUE
    { return queArray[nItems-1].getValue(); }
//--------------------------------------------
public boolean isEmpty()         // true if queue is empty
    { return (nItems==0); }
//--------------------------------------------
public boolean isFull()          // true if queue is full
    { return (nItems == maxSize); }
//--------------------------------------------
```

```
    public int noOfElements()
        { return nItems; }
    //-------------------------------------------
    public PQelement elementAt(int index)
        { return queArray[index]; }
    //-------------------------------------------
    public void showAllElements()
    {
        PQelement x;
        for (int ii=this.nItems-1; ii>=0; ii--)
        {
            x=(PQelement)queArray[ii];
            System.out.println(x.getValue());
            x=null;
        }
    }
}
```

## A.9 FifoQ

FifoQ describes the FIFO queues in the last-mile and the backbone models.

```
package com.ibm.fmds.simulator;
public class FifoQ {
    private int maxSize;
    private PQelement[] queArray;
    private int front;
    private int rear;
    private int nItems;
    ////////////////////////////////////////////////////////
    public FifoQ(int s) // constructor
    {
        maxSize = s + 1;
        queArray = new PQelement[maxSize];
        front = 0;
        rear = -1;
        nItems =0;
    }
    //-------------------------------------------
    public void insert(PQelement j) // put item at rear of queue
    {
        if (rear == maxSize - 1) // deal with wraparound
            rear = -1;
        queArray[++rear] = j; // increment rear and insert
        nItems++;
    }
    //-------------------------------------------
    public boolean isEmpty() // true if queue is empty
    {
        return (nItems==0);
    }
    //-------------------------------------------
    public boolean isFull() // true if queue is full
    {
        return (nItems==maxSize);
```

```
    }
    //------------------------------------------
    public int noOfElements() // number of items in queue
    {
        return nItems;
    }
    //------------------------------------------
    public PQelement peekFront() // peek at front of queue
    {
        return queArray[front];
    }
    //------------------------------------------
    public PQelement remove() // take item from front of queue
    {
        PQelement temp = queArray[front++]; // get value and incr front
        if (front == maxSize) // deal with wraparound
            front = 0;
           nItems--;
        return temp;
    }
}
```

## A.10 SimulationHandler and EasyAnalyzer

We prepared also two tentative classes, SimulationHandler and EasyAnalyzer, in order to run the simulator and to obtain data. In this preliminary stage, we set most of numerical parameters by putting down directly in `runSimulator` method of SimulationHandler class, although they should be input though the GUI.

```
package com.ibm.fmds.simulator;
import java.util.*;
import java.io.*;
public class SimulationHandler
{
    // for connection with GUI (see Simulator.run())
    protected com.ibm.fmds.util.FMDSSimulatorTrace trace = null;
    /////////////////////////////////////////////////////
    public static void main(String[] args) {
        SimulationHandler test=new SimulationHandler();
        test.runSimulator("AA009.dat", "AA009c.dat");
    }
    //------------------------------------------
    public void runSimulator(String freqFilename, String cFreqFilename) {
        if(trace != null )trace.append("Simulation starts\n");
        System.out.println("Simulation starts");

        int Ne = 100000; // the total number of events
        int numberOfNodes = 6; // the total number of Node components
        int numberOfSources = 1; // the number of Source components
        int numberOfNetworks = 2; // the number of Network components
        int numberOfUnits = numberOfNodes - numberOfSources- numberOfNetworks;
        int numberOfID=3; // the number of kinds of request
        int maxNumberOfSeats = 100;
```

```
    // capacity of eventList in Unit and Network components
double bandwidth_LM = 5.0e-3; // bandwidth of the last-mile component
double bandwidth_BB = 0.15; // bandwidth of the backbone component
double ratio=0.8; // a parameter representing fluctuation in bandwidth_BB
long seed_bandwidth=932874L;

double mean = 25.; // average time interval of event generation
long seed=2234123748L;
double throughput_HTTP = 0.7;
double throughput_WAS = 0.1;
double throughput_DB = 0.5;
double initialWorkValue = 0.5;
double magnify = 25.;
    // magnification parameter in HTTP server component
String filename=freqFilename;
String filename_c=cFreqFilename;

// probability for each request IDs
double [] idProbability = new double[numberOfID];
idProbability[0]=0.8; // static HTML page request
idProbability[1]=0.1; // servlet execution
idProbability[2]=1.-idProbability[0]-idProbability[1]; // DB access

Node[] node = new Node[numberOfNodes];
Node topNode, nextNode;
Event topEvent;
PriorityQ nodeManager = new PriorityQ(numberOfNodes);
int loopFlag = 0;
int prensentNodeID = -1;
int nextNodeID = -1;

node[0] = new Source(initialWorkValue,idProbability,
                mean,seed,seed*8312097L);
node[1] = new Network(maxNumberOfSeats, bandwidth_LM);
node[2] = new NetworkFluctuated(maxNumberOfSeats,
                bandwidth_BB,ratio,seed_bandwidth);
node[3] = new Unit(maxNumberOfSeats, throughput_HTTP, magnify);
node[4] = new Unit(maxNumberOfSeats, throughput_WAS, magnify);
node[5] = new Unit(maxNumberOfSeats, throughput_DB, magnify);

for (int ii = 0; ii < numberOfNodes; ii++) {
    node[ii].setNodeID(ii);
    node[ii].setNodeList(node);
}

// charge an event into Sources
for (int ii = 0; ii < numberOfSources; ii++) {
    nodeManager.insert(node[ii]);
}

EasyAnalyzer results = new EasyAnalyzer();
results.clearResults();
results.setTrace(trace); // give 'trace' (see Simulator.run())
node[0].setEasyAnalyzer(results);
//......................................... main loop begins
while(results.counter < Ne){
```

```
            topNode = (Node) nodeManager.remove();
            topEvent = topNode.removeTopEvent();

            nextNode = topNode.getNextNode(topEvent);
            if (nextNode != topNode)nodeManager.removeItemOf(nextNode);

            nextNode.enter(topEvent);
            if (!(topNode.howManyEvents() == 0)) nodeManager.insert(topNode);
            if (nextNode != topNode) nodeManager.insert(nextNode);

            topNode = null;
            topEvent = null;
            nextNode = null;
        }
        //......................................... main loop ends
        if(trace != null )trace.append("\narranging the data...\n");
        results.convolution(Ne);
        results.dataStatistics();
        results.spewData(filename, filename_c);

        if(trace != null )trace.append("\nSimulation finished.\n\n");
        System.out.println("\nSimulation finished.");
    }
    //----------------------------------------
    public void setTrace(com.ibm.fmds.util.FMDSSimulatorTrace newTrace) {
        trace = newTrace;
    }
}
```

EasyAnalyzer analyzes results of the simulator in a simple manner. The distribution of response time is given as two types of graphs: One is a simple histogram, and the other is a continuous curve, which is calculated by convoluting the histogram with Gaussian. The main reason of the convolution is to eliminate the dependency of line shape on division of the time axis. Namely, if the axis is divided into infinitesimally small intervals (i.e. `increment`$\rightarrow 0$), we will observe only zero or one frequency at any time point, resulting in a monotonous, almost constant line shape. Under the condition of conservation of spectral weight, which is equal to the total number of events generated in a round of simulation, the convoluted curve provides an appropriate representation of the distribution.

```
package com.ibm.fmds.simulator;
import java.io.*;
public class EasyAnalyzer {
    // For connection with the GUI
    protected com.ibm.fmds.util.FMDSSimulatorTrace trace=null;

    public static int counter=0;
        // counter for arrived events
    public static double increment=2.;
        // minimal interval of response time [ms]
    public static double increment_convoluted=4.0;
```

```
    // minimal interval of convoluted response time [ms]
public static double lowerBoundOfTime=2000.;
public static double upperBoundOfTime=4000.;
public static double sigma=increment_convoluted;
    // sigma of Gaussian for convolution
public static int[] responseTime;
    // distribution of response time
public static double[] responseTime_convoluted;
    // convoluted distribution of response time

public static int arraySize;
public static int arraySizeC;

private double average=0.;
private double variance=0.;
private double minResponseTime=0.;
private double maxResponseTime=0.;

static{
    EasyAnalyzer.arraySize
        =(int)((EasyAnalyzer.upperBoundOfTime
                - EasyAnalyzer.lowerBoundOfTime)
                /EasyAnalyzer.increment);
    EasyAnalyzer.arraySizeC
        =(int)((EasyAnalyzer.upperBoundOfTime
                - EasyAnalyzer.lowerBoundOfTime)
                /EasyAnalyzer.increment_convoluted);
    if((EasyAnalyzer.arraySize <0)||(EasyAnalyzer.arraySizeC<0))
        System.out.println(
        "invalid definition of range of time in EasyAnalyzer");
    responseTime = new int[arraySize];
    responseTime_convoluted = new double[arraySizeC];
}
/////////////////////////////////////////////////////////
public void clearResults() {
    for(int ii=0; ii < EasyAnalyzer.arraySize; ii++){
        EasyAnalyzer.responseTime[ii]=0;
    }
    for(int kk=0; kk < EasyAnalyzer.arraySizeC; kk++){
        EasyAnalyzer.responseTime_convoluted[kk]=0.;
    }
    this.counter=0;
}
//------------------------------------------
public void convolution(int Ne)
{
    double y, time;
    double cy, ctime;
    double argument;

    for(int ii=0; ii < EasyAnalyzer.arraySize; ii++){
        y = (double)EasyAnalyzer.responseTime[ii]/(double)Ne;
        time = (double)ii * EasyAnalyzer.increment
            + EasyAnalyzer.lowerBoundOfTime;
        for(int kk=0; kk < EasyAnalyzer.arraySizeC; kk++){
            // convolute the distribution with Gaussian
```

```
            ctime = (double)kk * EasyAnalyzer.increment_convoluted
                + EasyAnalyzer.lowerBoundOfTime;
            argument = -1.*(time-ctime)*(time-ctime)
                /(2.*EasyAnalyzer.sigma*EasyAnalyzer.sigma);
            cy= y*Math.exp(argument)
                / ( Math.sqrt(2.*Math.PI)*EasyAnalyzer.sigma );
            EasyAnalyzer.responseTime_convoluted[kk]=
                EasyAnalyzer.responseTime_convoluted[kk] + cy;
        }
    }
}
//--------------------------------------------
public void countTheEvent(Event event)
{   // catch events and calculate their age
    if(Math.IEEEremainder(EasyAnalyzer.counter, 10000) == 0. ){
        System.out.print(EasyAnalyzer.counter/1000 + "%...");
        if(trace != null )
            this.trace.append(EasyAnalyzer.counter/1000 + "%...");
    }
    EasyAnalyzer.counter++;
    double roundTripTime;
    roundTripTime = event.getFinishTime()-event.getGenTime();

    double numberOfInterval
        = (roundTripTime-EasyAnalyzer.lowerBoundOfTime)
            /(double)EasyAnalyzer.increment;

    int index= (int)numberOfInterval;
    if(index<0){
        index=0;
        //EasyAnalyzer.responseTime[index]
        //  = EasyAnalyzer.responseTime[index] + 1;
    }
    else if(index >= EasyAnalyzer.arraySize){
        index=EasyAnalyzer.arraySize-1;
        //EasyAnalyzer.responseTime[index]
        //  = EasyAnalyzer.responseTime[index] + 1;
    }
    else{
        EasyAnalyzer.responseTime[index]
            = EasyAnalyzer.responseTime[index] + 1;
    }

}
//--------------------------------------------
public void dataStatistics() {
    double time;
    double totalWeight=0;
    int y;
    double yy;

    // calculate the total weight
    this.minResponseTime = EasyAnalyzer.upperBoundOfTime;
    this.maxResponseTime = EasyAnalyzer.lowerBoundOfTime;
    for(int ii=0; ii < EasyAnalyzer.arraySize; ii++){
        y= EasyAnalyzer.responseTime[ii];
```

```
            totalWeight = totalWeight + (double)y;
        }

        // calculate average of distribution of response time
        for(int ii=0; ii < EasyAnalyzer.arraySize; ii++){
            time = (double)ii * EasyAnalyzer.increment
                + EasyAnalyzer.lowerBoundOfTime;
            y= EasyAnalyzer.responseTime[ii];
            average=average + time*(double)y/totalWeight;
            if(y >= 1){
                if(this.minResponseTime>time)
                    this.minResponseTime = time;
                if(this.maxResponseTime<time)
                    this.maxResponseTime = time;
            }
        }
        // calculate variance of distribution of response time
        double squareAverage=0;
        for(int ii=0; ii < EasyAnalyzer.arraySize; ii++){
            time = (double)ii * EasyAnalyzer.increment
                + EasyAnalyzer.lowerBoundOfTime;
            y= EasyAnalyzer.responseTime[ii];
            squareAverage = squareAverage + time*time*(double)y/totalWeight;
        }
        this.variance = squareAverage - average*average;

        // Output
        System.out.println("\n");
        System.out.println("totalWeight=" + (int)totalWeight);
        System.out.println("average=" + this.average);
        System.out.println("min=" + this.minResponseTime);
        System.out.println("max=" + this.maxResponseTime);
        System.out.println("variance=" + this.variance);
        System.out.println("standard deviation=" + Math.sqrt(this.variance));

        if(trace != null ) this.trace.append("\n");
        if(trace != null )
            this.trace.append("totalWeight=" + (int)totalWeight+"\n");
        if(trace != null )
            this.trace.append("average=" + this.average+"\n");
        if(trace != null )
            this.trace.append("min=" + this.minResponseTime+"\n");
        if(trace != null )
            this.trace.append("max=" + this.maxResponseTime+"\n");
        if(trace != null )
            this.trace.append("variance=" + this.variance+"\n");
        if(trace != null )
            this.trace.append("standard deviation="
                + Math.sqrt(this.variance)+"\n");
    }
    //-------------------------------------------
    public double getAverage() {
        return this.average;
    }
    //-------------------------------------------
    public double getMaxResponseTime() {
```

```
        return this.maxResponseTime;
    }
    //-------------------------------------------
    public double getMinResponseTime() {
        return this.minResponseTime;
    }
    //-------------------------------------------
    public double getVariance() {
        return this.variance;
    }
    //-------------------------------------------
    public void setTrace(com.ibm.fmds.util.FMDSSimulatorTrace newTrace) {
        this.trace = newTrace;
    }
    //-------------------------------------------
    public void spewData(String frequency, String cfrequency)
    {
        double x, y;
        try{
            FileWriter fw = new FileWriter(frequency);
            //BufferedWriter bw = new BufferedWriter(fw);
            PrintWriter pw = new PrintWriter(fw);
            for(int ii=0; ii< EasyAnalyzer.arraySize; ii++){
                x= EasyAnalyzer.lowerBoundOfTime
                    + (double)ii* EasyAnalyzer.increment;
                y= EasyAnalyzer.responseTime[ii];
                pw.println(x + ", " +y);
            }
            fw.close();
            pw.close();
        }
        catch(IOException e){
            System.out.println("No path: " + frequency);
        }

        try{
            FileWriter cfw = new FileWriter(cfrequency);
            //BufferedWriter cbw = new BufferedWriter(cfw);
            PrintWriter cpw = new PrintWriter(cfw);

            for(int ii=0; ii< EasyAnalyzer.arraySizeC; ii++){
                x=EasyAnalyzer.lowerBoundOfTime
                    + (double)ii* EasyAnalyzer.increment_convoluted;
                y=EasyAnalyzer.responseTime_convoluted[ii];
                cpw.println(x + ", " +y);
            }
            cfw.close();
            cpw.close();
        }
        catch(IOException e){
            System.out.println("No path: " + frequency);
        }
    }
}
```