# A Performance Model for a Network of Prototype Software Routers

By

Pengfei Wu, B.E.

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements of the degree of

Master of Applied Science

Ottawa-Carleton Institute for Electrical and Computer Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, K1S 5B6
Canada

July 20th, 2003

The undersigned recommend to the Faculty of Graduate Studies
and Research the acceptance of the thesis

# A Performance Model for a Network of Prototype Software Routers

Submitted by Pengfei Wu, B.E in partial fulfillment of the
requirements of the degree of Master of Applied Science

---

Dr. Rafik Goubran, Chair

Department of Systems and Computer Engineering

---

Dr. C. Murray Woodside, Thesis Co Supervisor

---

Dr. Chung-Horng, Lung Thesis Co Supervisor

# Abstract

It is important to understand performance aspects of a computer system from the software architecture and its configurations. For a system that has various possible configurations, the performance analysis will become difficult. This thesis describes a compositional model-building approach which is useful when there are many possible configurations. This approach is based on assembling the sub models and is demonstrated in a network software router, CGNet.

In addition, the derivation of parameters must be well addressed for performance modeling. The measurements and parameter estimation techniques can be deployed in completing the performance model. We apply these approaches and techniques to discussing the real problems we encountered in parameterizing the performance model.

The converter tool, which automates the compositional model-building approach, has been developed for CGNet. This tool is a solution to bridging the gap between the network configurations and the performance analysis. It is a configuration-based model-building tool for CGNet. With the tool, we can generate the performance model from configurations.

# Acknowledgements

I would like to thank my co-supervisor, Dr. C. Murray Woodside, for his guidance and advice throughout my study in Master program, especially during the research for this thesis. I would also like to thank my co-supervisor, Dr. Chung-Horng, Lung, for his guidance and suggestions throughout this research. Their seasoned guidance, consistent support, wise advice, and numerous encouragements have been integral to the success of this research. Here I would like to say, " thank you very much" to express my deepest gratitude.

I would like to thank all my friends for everything. You have provided assistance for me from study and life when I need it. I would also like to thank the members of RADS Lab for creating a wonderful working environment and for their support. The financial assistance from Carleton University is greatly appreciated.

Finally, special appreciation goes to my family. Their support, understanding and love are never stopped since I was born. I am proud of you and thank you.

# Table of Contents

# List of Figures

# List of Tables

# List of Symbols

| **Symbols** | **Description** |
|---|---|
| $P_{\mathrm{M}}$ | The process whose CPU time is calibrated |
| $P_{\mathrm{I}}$ | The CPU-intensive process |
| $L_{\mathrm{I}}$ | A given number of loops for $P_{\mathrm{I}}$ |
| $T_{\mathrm{I}}$ | The execution time of $P_{\mathrm{I}}$ |
| $T_{\mathrm{M}}$ | The CPU time taken by the process $P_{\mathrm{M}}$ |
| **Node** | All routers in an operational network |
| **OutLinks(XX)** | All bidirectional links connected to node XX |
| **Neighbor(XX)** | All nodes connected to node XX through links in **OutLinks(XX)** |
| **G(XX)** | All elements outside **Node** that can send the data traffic to node XX |
| gXX | The merged generator connected to node XX |
| $r_{XX}$ | The speed of the generator gXX |
| **Destination(XX)** | All the destination nodes that the generator gXX sends data traffic to |
| $P_{XX,ZZ}$ | The proportion of data traffic for destination ZZ from generator gXX |
| $r_{XX,ZZ}$ | The speed of data traffic for destination ZZ from generator gXX |
| **S(XX)** | All elements outside **Node** that can receive the data traffic from node XX |
| sXX | The merged traffic sink connected to node XX |

| | |
|---|---|
| XX-YY | The link between node XX and node YY |
| Speed(XX-YY) | The speed of link XX-YY |
| Cost(XX-YY) | The cost of link XX-YY |
| $T_{XX}$ | The execution time of node XX executable |
| $n_{RG}$ | Number of packets received from generator during $T_{XX}$ |
| $n_{RN}$ | Number of packets received from **Neighbor(XX)** during $T_{XX}$ |
| $n_R$ | Number of packets received and switched during $T_{XX}$ |
| $n_{SD}$ | Number of packets sent to **Neighbor(XX)** during $T_{XX}$ |
| $n_{SK}$ | Number of packets sent to the traffic sink during $T_{XX}$ |
| $n_S$ | Number of packets sent to the traffic sink and **Neighbor(XX)** during $T_{XX}$ |
| $n_L$ | Number of packets lost in XX during $T_{XX}$ |
| $n_{GS}$ | the number of the packets the local generator has sent |
| (XX, ZZ) | Packet class for packets that are originated from router XX and destined to router ZZ |
| **Path(XX,ZZ)** | The sequence of routers along path for packet class (XX, ZZ) |
| **ForwardR(XX, ZZ)** | all intermediate routers between XX and ZZ along the path for packet class(XX,ZZ) |

| | |
|---|---|
| *source* router | The router which packet class (XX, ZZ) is originated from |
| *destination* router | The router which packet class (XX, ZZ) is destined to |
| *forwarding* router | The intermediate router between *source* router and *destination* router along path for packet class (XX,ZZ) |
| *upstream* router | The router which sends packet class (XX,ZZ) to the immediate next-hop router |
| *downstream* router | The router which receives packet class (XX,ZZ) from *upstream* router |

<link_name, packetclass_name [,packetclass_name]>

| | |
|---|---|
| | The label of the interface in the node sub model for link |
| <gAA, AAUser> | The label of the interface in the node AA sub model for generator |

| | |
|---|---|
| $d_{XX-YY}$ | The network delay for packet with size 85 bytes through link XX-YY |
| $T_C$ | The execution time of the *compulmt* process |
| TP | The total CPU time for node process |
| $a_R$ | CPU time per packet received |
| $a_{SW}$ | CPU time per packet switched to outgoing queue |
| $a_{SD}$ | CPU time per packet sent to outgoing socket for next hop |
| $a_{SK}$ | CPU time per packet sent to outgoing socket for traffic sink |
| $a_{RSW}$ | CPU time per packet received and switched to outgoing queue |
| $a_P$ | CPU time per packet received, switched and sent to outgoing socket |

| | |
|---|---|
| *multiplier* | A multiplier to vary the rates of all generators in CGNet |
| **N** | Nodes in the converter tool |
| **G** | Generator in the converter tool |
| **L** | Links in the converter tool |
| **S** | Traffic sinks in the converter tool |
| **P** | Host processors in the converter tool |
| **RT** | Routing tables for all nodes in the converter tool |
| **Usr** | User task of LQN model in the converter tool |
| **Rcv** | Receiving task of LQN model in the converter tool |
| **Swi** | Switching task of LQN model in the converter tool |
| **Snd** | Send task of LQN model in the converter tool |
| **Snk** | Sink task of LQN model in the converter tool |
| **Net** | Network delay task of LQN model in the converter tool |
| **RE** | Routing table entry in the converter tool |

# Chapter 1 Introduction

This chapter briefly describes the thesis research. Section 1.1 introduces the purpose of performance modeling. Section 1.2 discusses the challenges of performance modeling. Section 1.3 presents the objectives of the thesis. Section 1.4 shows the overview of the thesis. Section 1.5 summarizes the contributions of thesis. Section 1.6 lists the outline of thesis.

## 1.1 The Purpose of Performance Modeling

Designing a computer system, or configuring it after implementation to meet certain performance criteria is a significant problem. In most cases developers and designers tend to ignore performance issues in order to meet tight deadlines and aggressive schedules. This may lead to catastrophic results after the application is used in a performance critical environment. It may cost a substantial amount of time and money to identify and correct the performance problems.

Depending on measurement data alone is not fully effective in ensuring that the computer system will meet performance expectations. It only offers a snap shot of the computer system, and expresses the performance factors exactly for that specific situation.

Performance modeling is a good alternative to ensure that the software architecture will meet the performance objective. The performance model is essential to identify serious performance problems at the architectural and early design stages of the life cycle in software engineering. Performance modeling in the early stage of software design life cycle can not only reduce the risk of performance-related failures by giving early warnings to potential performance problems, but also avoid the snowball effect of performance problems.

Performance models can provide performance predictions under varying environmental conditions or design alternatives, and these conditions can be used to help detect problems. Performance analysis addresses the sensitivity of the performance in utilization, repetition and synchronization, which allows us to rapidly explore alternatives to correct the problems before they could arise in the system. Performance analysis offers the feedback on performance aspect, which gives us more insight into the system we are building.

## 1.2 Challenges of Performance Modeling

Innovations in software not only push software engineering to a higher level, but also call for careful attentions to the performance. An object-oriented approach as a new technology in software engineering presents a special problem for software performance engineering. Performing a given function in object-oriented methods is likely to require the collaborations among many different objects from several classes. The interactions can be numerous and complex and often obscured by polymorphism, making the interactions difficult to trace. Distributed systems challenge the performance intuition. Constructing distributed systems involves a complex combination of choices about processing and data location, platform size, network configuration, middleware implementation, etc. How to construct a performance model for an object-oriented distributed software system has been important in model-oriented software performance engineering area since the mainstream use of Web applications, Common Object Request Broker Architecture (CORBA), and Enterprise JavaBeans.

Once we convert the behaviors and constraints of the computer system into an appropriate performance model format such as a simulation model, a queueing network model, or a Petri-net model, the actual demand parameters in terms of execution times and frequency of their execution should be estimated, measured and inserted. The hardest part of software performance engineering is getting the data you need for the performance model. Final performance prediction results are sensitive to budgeted parameters so much that the answer to how to get the parameters for the model is the first factor to make the model-oriented approach trustworthy.

Many tools based on queueing networks or stochastic Petri nets and extensions make it easy to derive the performance prediction with a parameterized performance model. We can obtain the quantitative characteristics of a computer system from the performance prediction as well as tell where the problems came from. The real objective of the performance model is to offer suggestions and practical solutions that software designers and user are concerned the most.

## 1.3 Objectives of This Thesis

This thesis discusses how to create a quantitative model for an object-oriented and distributed application in the Internet, and describes how to integrate a performance model seamlessly into Internet traffic engineering. The research focuses on the three problems mentioned in the section 1.2: 1) the compositional approach to building performance model for the object-oriented distributed application system; 2) the measurement for the parameters and validation of models for the communication system; 3) the suggestion and solution for the application system. Instead of designing a case study for the explanation of pure academic theory, we have chosen one practical system to propose the solutions of scalability from the performance perspective of the system.

To mitigate the complexity to build the performance model for a complex system, a compositional approach is proposed for constructing the model in the thesis. The approach is based on assembling sub-models for different operations in the sub systems. We begin to study the system by use cases and scenarios at any phase of the development process. We can review software descriptions such as requirement specifications, design documents, and source code implementation to understand the scenarios. We build the structure of the sub-models corresponding to scenarios or scenario fragments, which can be assembled into a large complex model.

To obtain the parameters characteristics of the performance model, we use the "Displacement" technique for robust measurement. The measurement can also be used to verify and validate the models, and monitor the computer system. The computer system can be so complicated that it requires the estimation techniques to achieve parameters for the performance model.

We can quantify the performance of the software system's architecture and design by solving the SPE models, and identify whether any performance problems may exist. If any problems exist, the system with proposed suggestions and design alternatives could be re-modeled, re-parameterized and re-solved until the system meets the performance objectives.

## 1.4 Overview of Thesis Work

The thesis involves two main different research areas: Software Performance Engineering and Traffic Engineering for network communication. We can derive traffic characteristics of the operational network system through measurement and prediction within the realm of performance evaluation.

Internet routers, the most active elements of the whole network, perform packet-processing tasks at high rates by dividing the work into hardware and software. To increase the performance of both large and small routers, it is important to understand performance aspects of the protocols, the hardware, and also of the software designs. Performance of an operational network is the outcome of the behaviour of each router, of the interactions between routers, and the behaviour of the network. CGNet is a prototype software router of modest scale that was developed to study the interaction of all of these factors. CGNet is an emulation of an entire network that can run in the lab on one PC or on a network. A typical network can be modeled by CGNet and all elements in the network can be presented as components in CGNet.

In the research we investigate CGNet and predict the performance of the overall network in terms of throughput, packet loss, and utilization of the router. This objective is achieved by using the Layered Queueing Network (LQN) model. LQN is an extended queueing network model that specifies the calls between entries so that the layered requests for service and the layered contention delays in the path are represented in a simple canonical way. In a layered queueing network, a software process (thread) is represented as an entry in the task can act as both clients and servers to the other processes. The layered queueing network model has the same parameters as the queueing network model such as the average number of visits, the average service time at the device, and their scheduling disciplines. Analytic modeling techniques based on approximate mean value analysis are used to provide performance estimates for the behavior of the system being studied.

In this research, a performance model of CGNet was created and evaluated, to see if the model can be used to supplement emulation and to see if the model gives the same predictions. A layered queueing network was used to capture the effect of contention for software thread resources and threading levels. The model can estimate the performance characteristics such as throughput, utilizations of threads, buffers, and packet losses. However, some difficulties in the measurements leave some questions unanswered about models for arbitrary network configurations.

A traffic-engineering framework is proposed in this thesis, which involves three main components: 1) CGNet model, 2) LQN model, 3) Performance predicted, as illustrated in Figure 1. The framework consists of three main steps. First, we choose an operational network we are concerned about and model the overall network by CGNet. Second, we construct an LQN model of CGNet by hand or automatically generate it by a tool I developed. Third, we can use the solver of the LQN model, *lqns*, *parasrvn*, and *spex* to solve the model and obtain the predicted performance of the overall network. After that, we can propose the solutions of an operational network for reconfiguration and scalability according to the performance characteristics of a communication system predicted by the model.

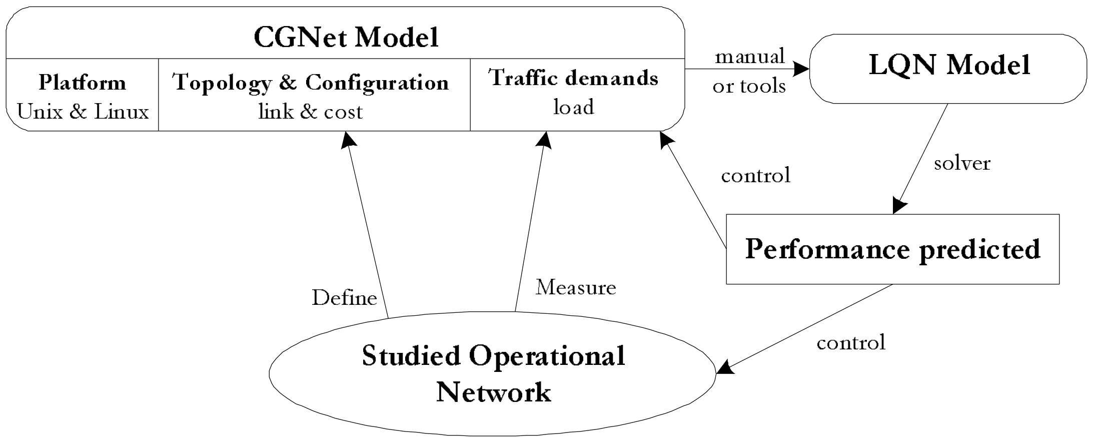


Figure 1.1 Main components of the traffic-engineering framework

## 1.5 Thesis Contributions

There are several contributions of this research, which are as follows:

- The compositional model building approach based on assembling the sub-models for different network operations at each node is proposed in the thesis. The sub-model corresponds to standard scenario fragments, which can be assembled to a large complex model. (See chapter 4)
- Performance parameters for the CGNet prototype were determined by measurements and estimations in which we make use of the "Displacement" technique and the least square estimation technique. Some discoveries on thread switching overhead and limitations of LQN solvers were derived from the observations in the research. (See chapter 5 and chapter 6)
- The converter tool has been developed in the thesis that can automate the compositional approach and bridge between the configuration and the LQN performance model. (See chapter 7)

## 1.6 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 presents the background of topic in the thesis such as traffic engineering, software performance engineering, Use Case Maps (UCM), "Displacement" technique. Chapter 3 introduces CGNet and its components, which emulate the elements in a network, as well as the configuration and the execution of CGNet. Chapter 4 describes the compositional model building approach based on assembling the sub models. The sub models corresponds to scenarios or scenario fragments. Chapter 5 shows the measurement and parameter estimation for the parameter purpose. It also discusses the observations in the derivation of parameter. Chapter 6 validates the model in the three aspects: throughput, packet loss, and utilization of host processor. Chapter 7 offers the converter tool to translate from the configuration to the LQN model and presented the algorithm of the converter tool. Finally Chapter 8 draws the conclusion and contains suggestions for the performance purpose.

# Chapter 2 Background

This chapter provides the general background of the research presented in this thesis. Traffic engineering and software performance engineering (SPE) are the basic background of research. Section 2.1 briefly reviews traffic engineering. Section 2.2 introduces (SPE). Section 2.3 presents Use Case Maps (UCMs). Section 2.4 shows the Layered Queueing Network (LQN) model. Thus we can understand the basic notations of UCMs and the LQN model. At last section 2.5 describes the "Displacement" technique for measurement.

## 2.1 Traffic Engineering

"The aspect of Internet network engineering which deals with the issue of performance evaluation and performance optimization of operational IP networks" is referred to as Internet traffic engineering in the IETF [Awduche02]. An important objective of Internet Traffic Engineering is to enhance the performance of an operational network at both traffic and resource levels. It addresses the problems of allocating resource efficiently and reliably in the network so that user constraints are met and operator benefit is maximized.

The major challenge of Internet traffic engineering is network performance evaluation, which is important for assessing the effectiveness of traffic engineering methods, and for monitoring and verifying compliance with network performance goals. Results from performance evaluation can be used to identify existing problems, guide network re-optimization, and aid in the prediction of potential future problems.

Performance evaluation of a network can be derived in many different ways. The most typical techniques are analytical methods, simulation, and empirical methods based on measurements. When we use analytical methods or simulation, we should capture the operational characteristics of the network elements in the constructed models. These characteristics include topology of the network, bandwidth of links, buffer size in nodes, flow rate from outside.

It is quite complicated to evaluate the performance of the operational network contexts. A number of techniques such as abstraction, decomposition, and approximation can be used to simplify the analysis. Queueing models and approximation schemes based on asymptotic and decomposition techniques can be widely used in the network analysis. For example network calculus [Cruz91] is a mathematical approach to model network behavior. It may simplify network analysis relative to classical stochastic techniques. When we use analytical techniques, we should make sure that the models faithfully reflect the relevant operational characteristics of the modeled network elements for precision purpose. Simulation can be used to evaluate network performance with computation or to verify analytical approximations. But simulation can be computationally costly compared to analytical methods. Empirical methods are also used in the performance evaluation. For instance, the probe packets are used in MATE [Elwalid01] so that the node can compute the statistics.

An appropriate approach to a given network performance evaluation problem may involve a hybrid combination of analytical techniques, simulation, and empirical methods.

## 2.2 Software Performance Engineering

Software Performance Engineering (SPE), an engineering approach to performance, is defined in the book [Smith02] as "a systematic, quantitative approach to constructing software systems that meet performance objectives." SPE is a software-oriented approach and it focuses on architecture, design, and implementation alternatives of the proposed software. We can evaluate the performance characteristics of architecture and design alternatives and select them for the objectives of software performance in responsiveness and scalability. SPE includes the model-based approach to performance prediction as well as techniques for data collection, management of uncertainties, model validation, and performance solution.

Model prediction is a good choice and modeling methodology is the core to SPE. We can build and analyze the models of proposed software. We can then explore its performance characteristics to determine if it will meet its requirements. Model prediction is widely accepted and we hail that model prediction makes it possible to prevent performance problems from surfacing late in the life cycle of the software development process. But as systems grow more complex, parameters and related measurements for the performance model present more challenges in SPE than before. Once we have the parameters and solve the model, we can quantify the performance of the software's architecture and design. We can determine whether it is to meet performance objectives or not. If it does not meet the objective, the performance characteristics may indicate where problems are and why they could happen. Performance improvement and solution is the final objective of SPE. We can deploy performance solutions by applying performance principles, patterns, anti patterns, and tuning [Smith02]. Therefore, software performance engineering is a combination of three main steps:

1. Performance modeling
2. Performance measurement
3. Performance solution

In the following subsections, the SPE model procedure is outlined first, and performance modeling and data collection are described.

### 2.2.1 SPE Model Procedure

Connie Smith proposed the perform methodology in the book [Smith90] which can be used in analyzing the performance of a system. It is outlined as follows:

1. Capture performance requirements and understand the system functions.
2. Understand the architecture of the system and develop a performance model.
3. Capture the processing steps and define software execution characteristics.
4. Estimate resource usage and insert them as model parameters.
5. Solve the model and analyze the results and make design suggestions.

The methodology is widely used as SPE approach in the computer system and you can make some revisions for your specific case study. It can be deployed in any stage of software development processing if only can you construct models which can indicate the characteristics of a computer system, and obtain the reasonable parameters of the model. You repeat the procedures and derive feedback from performance analysis so that you can adjust your design for performance purposes. After you have modeled a number of design alternatives, you can tell some of them own good performance characteristics and others do not. Then you can determine which one is the final software plan.

### 2.2.2 Performance Modeling

The goal of SPE and model-based approach is to reduce the risk of performance failures. People [Smith99] notice the benefit of "the earlier, the better" principle in the development stages. But we cannot deny the fact that our knowledge of the software design processing and implementation details are sketchy in the early phases of software development. The insufficiency of knowledge for the proposed software drives us to learn more from the concept of system, requirement specification and the design descriptions if possible.

Software execution models such as execution graphs can not only help us understand the processing steps of the system, but also identify serious performance problems at the architectural and early design phases. Class diagrams, deployment diagrams, sequence and collaboration diagrams in UML [Gomaa01] support execution graphs and software execution models well. As we know more completely about the software's design and implementation details, we can refine the software execution model in the critical parts. The results of the software execution model is needed as input parameters for the system execution model, which capture the software performance characteristics of system with contention for system resource among workloads and multiple users.

The system execution model is a dynamic model and it characterizes the software's performance accounting for the contention effects. The system execution model is represented as a network of queues and servers, where a queue represents jobs waiting for service, and a server represents a component providing service. Performance evaluation tools [Bolch98] have been invented and they have contributed significantly to the design of complex computer systems and networks. They can be categorized into two sets: one is based on queueing networks such as LQN [Woodside95A], QNAP2 [Veran85]; and the other is based on stochastic Petri nets and extensions, for instance, SPNP [Ciardo89]. In most cases they all provide analytic/numerical methods as well as a simulation-base solution.

### 2.2.3 Performance Data Collection

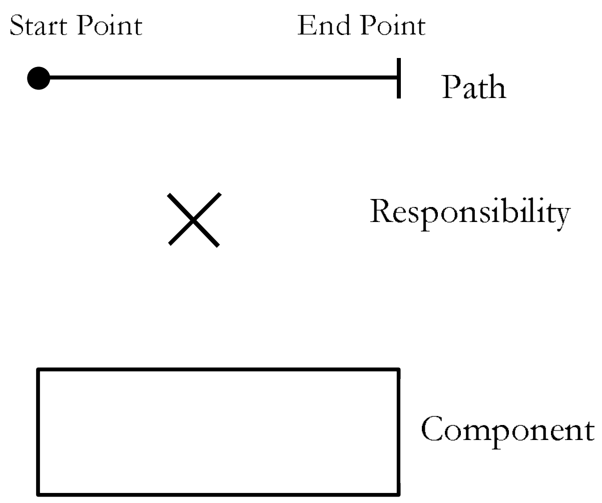
When we construct the performance model, we need the necessary data to solve the software performance model. We can conduct performance walkthroughs [Smith02] by bringing together the people who can help us understand the workload intensity, the execution environment, and their interaction with the software. Expert judgement and experience [Smith99] play important roles in the performance walkthroughs. The precision of model results depends on the fidelity of the estimates. The verification and validation of the estimates is important for the precise prediction. Resource requirements are difficult to estimate and gather in our initial SPE studies. In some cases we employ the mathematical formula or model to calculate the value we expect to obtain if the specification is inadequate.

Performance measurements [Jain91] also provide the input data for SPE model if the prototypes or implementations are available. Lots of profiling and kernel instrumentation tools are available with programming language specifications such as *gprof* [Graham82], or *paradyn* tool [Miller95]. Although these tools are well defined and widespread, they have difficulties in the complicated cases with many communication operations. Sometimes you are required to write your own specific measurement program for your application. It is very difficult to plan research that will provide the general solution and overcome the difficulties in the performance measurements.

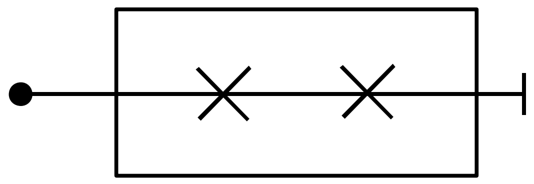
## 2.3 Use Case Map

The Use Case Map (UCM) model is a high-level design model to help a user express and reason about a system's large-grained behaviour, which was contributed by R. J. A. Buhr. UCMs [Buhr96] are defined as causal scenarios, architectural entities or behavior patterns. UCMs are a visual notation for use cases [Jacobson92] with extension of a high level of abstraction. UCMs express behaviour along the paths by sets of responsibilities. The trace of a set of responsibilities along a path can be referred to a scenario. The scenarios enable a user to grasp the behaviour of the system without getting lost in execution details. Therefore Use Case Maps can be the basis of a performance model because both of them can capture the specification of systems. There is a difference between them: the former unfolds the scenarios to easy understandings of the system but the latter captures the behaviour with quantity characteristics. This will be considered in chapter 4.

A UCM map is a collection of elements that describe one or more scenarios unfolding through a system. The basic elements [Buhr96] in UCM are paths, responsibilities and components. UCMs are composed of paths with responsibility points that may traverse components as well as scenarios vs. system.



(a) basic elements in UCM



(b) a simple example path with responsibilities overlaid on a component

Figure 2.1 Basic notation and a simple example of Use Case Map

A path can be interpreted in the behavioural terms as a scenario and its visual representation is a line with a start point and an end point. A filled circle represents a start point, which indicates the stimulus and a set of preconditions to start the path. A bar ends a path and shows the results of the path (see Figure 2.1 (a)). The paths are routes along which chains of causes and effects propagate through the system.

A rectangular black box shown in Figure 2.1 (a) is a component. The components represent entities and objects that are encountered during the execution of a scenario. They can represent both hardware and software resources in a system such as objects or modules, processes and threads, or physical devices.

There may be named responsibility points along any path so that a path shows more scenario detail. Responsibilities shown in Figure 2.1 (a) are denoted with crosses along the path. Responsibilities represent the localized actions and functions that a system must perform at the specified point.

There is a very simple example in Figure 2.1 (b) that represents a scenario. The scenario is that a path with responsibilities is overlaid on one component. Although it is simple, it well indicates the relationship among path, responsibility and component.

Complicated scenarios make UCM construction express the patterns that are not purely point-to-point. If explicit concurrency appears in the same scenario, it should be expressed as parallel path segments split by the AND forks or gather by the AND joins. Figure 2.2 shows an example of a Use Case Map for the session of data upgrading with joining and forking. The AND fork is a single path in the scenario splitting into two or more parallel paths. The AND fork indicates the beginning of the concurrency. As the parallel paths after the AND forks are progressing, the AND join is used to end the concurrency. No scenario can continue executing until all the parallel paths have joined. For example in figure 2.2 only after *upgrading data* and *progressing display* have been done, can *done message display* be performed. The OR fork and the OR join represent the alternatives paths. The OR fork expresses that a single path splits into two or more alternative paths. Only one of the possible branches may be traversed after the OR fork. The OR join shows the alternative paths to merge in a single path. The OR join indicates that at least one of the possible paths leading to the OR join needs to be traversed before proceeding further. In figure 2.2 there are two alternatives after the system verifies the user: valid user and invalid user. We use an OR fork after *verifying user* and an OR join indicates that the transaction for update data is finished.

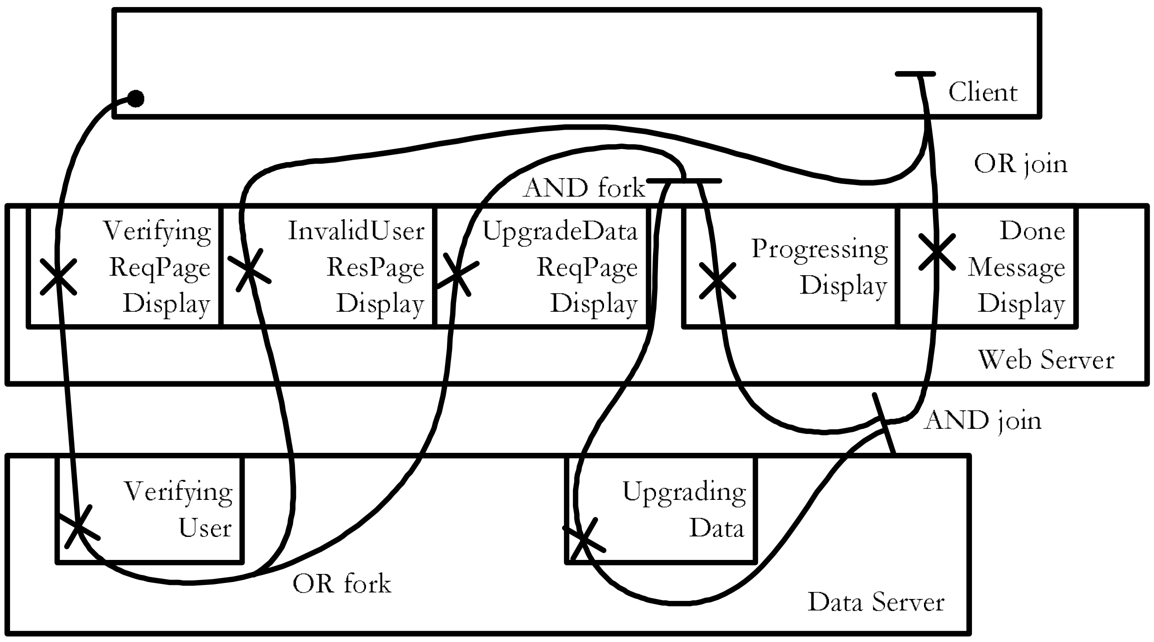


Figure 2.2 An example of UCM with forking and joining for the upgrading transaction

## 2.4 Layered Queueing Network Model (LQN)

Layered Queueing Networks (LQN) [Rolia95] [Woodside95A] was developed as an extension of the Queueing Networks for performance modeling of the complex and distributed systems. An LQN model describes the architecture of system by the sets of resources and expresses the interaction between the resources. It is capable of modeling most of the features such as multi-threaded processors, devices, locks, communication and so on [Franks00]. LQN models can be solved to capture the contentions and identify the performance bottlenecks [Neilson95]. LQN models also provide the comprehensive descriptions for the performance characteristics of a computer system. An LQN model is the target performance model for this research.

### 2.4.1 LQN Notation

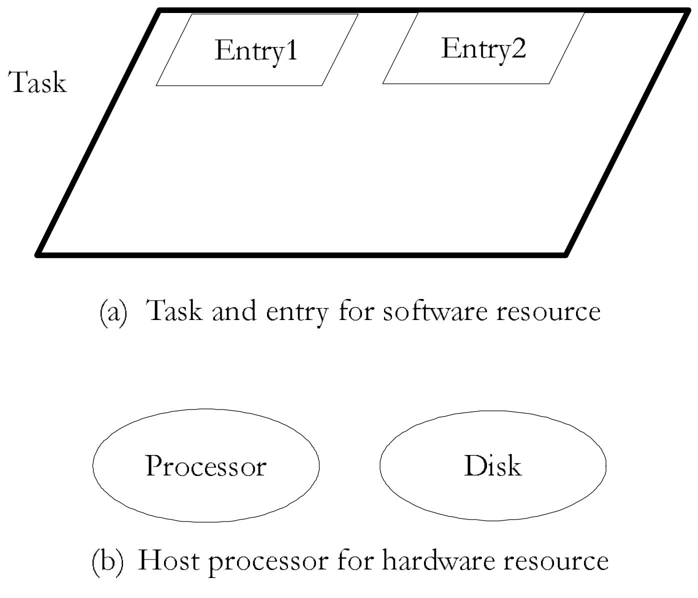

Task
Entry1
Entry2
(a) Task and entry for software resource
Processor
Disk
(b) Host processor for hardware resource

Figure 2.3 The LQN notation in terms of task, entry and host processor

The LQN model describes a system by the sets of resources such as software and hardware. The software resources are processes, threads, operations, semaphores and so on. The hardware resources are CPUs, disks, and interface controller. These resources can be modeled in LQN in terms of tasks, host processor, and entry. A task is a software object, which carries operations, and it has the properties of resources, including a queue, a discipline, and a multiplicity. A task has one or more entries, representing different operations it may perform. A task has a host processor, which represents the physical entity that carries out the operations. Hence, the physical execution can be delegated by the logic of operation expressed by the entry in the task. The processor has a queue and a discipline for executing its tasks. The visual notation of the elements in the LQN model such as task, entry, and host processor is shown in Figure 2.3.

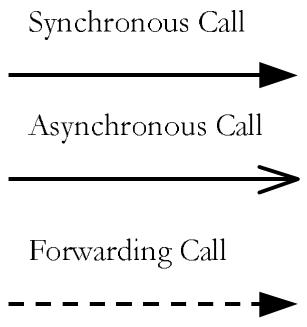


Figure 2.4 Calls for service request in LQN models

The interactions between software and hardware can be expressed as service requests, named as calls in LQN models. Calls are shown in LQN by messaging arrows in Figure 2.4 Calls for service request. The tasks may send and receive the service requests and play the client/server role respectively. If tasks do not receive any requests, they are called reference tasks and they represent the load generators or users of the system.

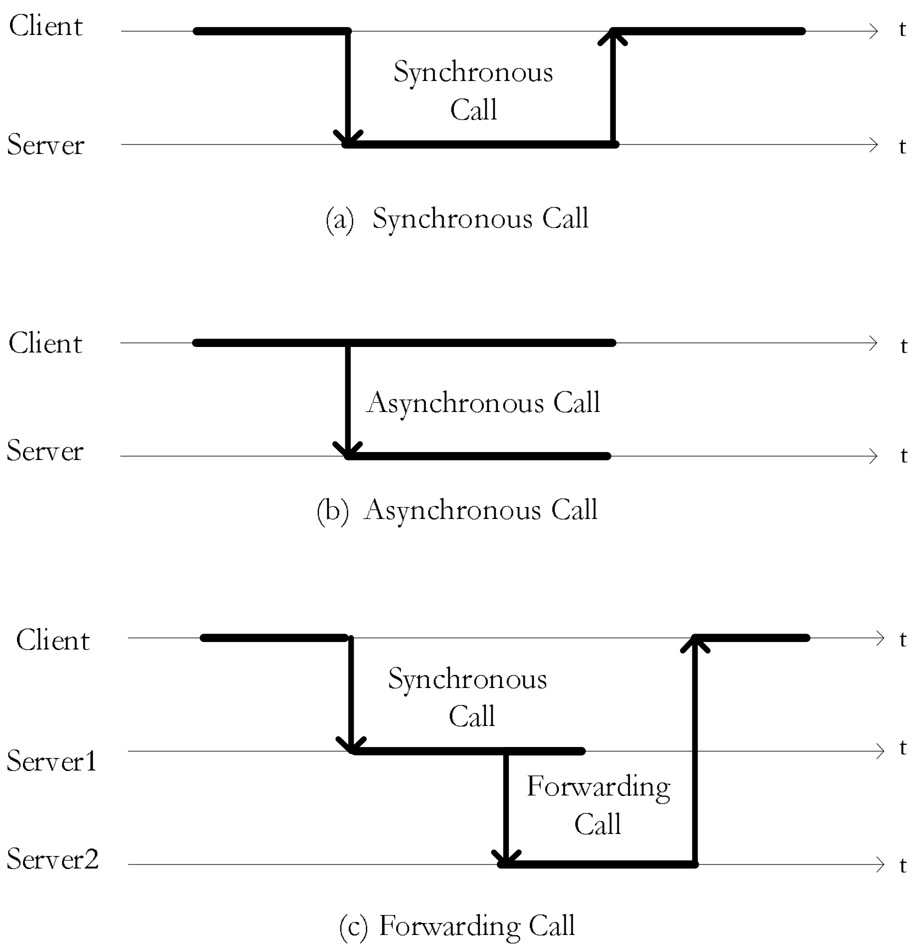


Figure 2.5 Time lines of LQN models synchronous, asynchronous, and forwarding calls

There are three types of calls or messages between tasks and they are synchronous, asynchronous, and forwarding calls. The tasks receive the service request at the designated interface point as entry. If the server task receives a synchronous call from the client task (see Figure 2.5 (a) Time lines of LQN models synchronous call), the server task is responsible for returning a reply after the request has been completed. The client task is blocked until it receives the reply from the server task. If the server task receives an asynchronous call from the client task (see Figure 2.5 (b) Time lines of LQN models asynchronous call), the client task continues executing during the execution of the server task and does not need to wait for the response from the server task. In the forwarding call, the client task is blocked until it receives a reply as the synchronous call. The intermediate server task partially processes the service request and then forwards the request to another server task. This server task is responsible for sending a reply to the client task and unblocks the client task. The intermediate server task can continue operation after forwarding the call. Figure 2.5 (c) shows the time lines of LQN models forwarding call.

LQN models also define the demands, which include the total average amounts of host processing and average number of calls for service operations required to complete an entry.

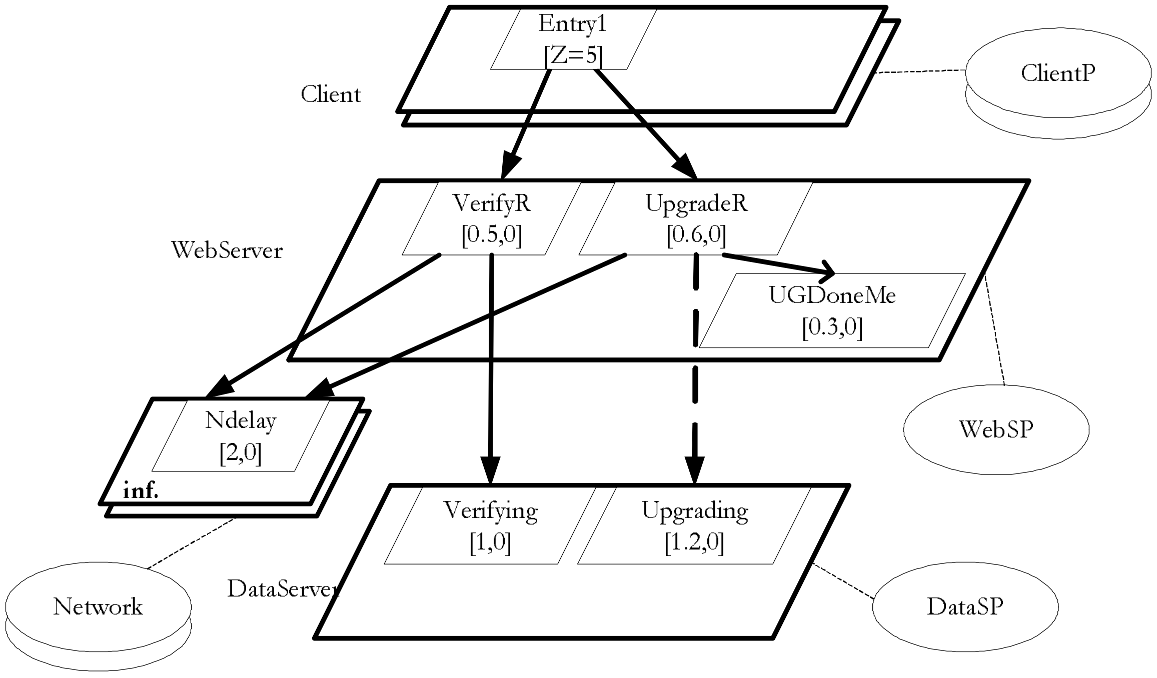


Figure 2.6 A typical example of an LQN model for the database upgrading transaction

Figure 2.6 illustrates a typical LQN model with the basic notation in terms of tasks, entries, host processors, calls and demands. It has modeled the behaviour of the web database system for the upgrading transaction. Users send requests to upgrade the database. Two main steps should be performed: user validation and database upgrade. The information for user and data upgrading should be input on the web page and these information have been sent to the database server to validate and update the database server. There is a network delay between web server and database server. Once the upgrading has been done, the webpage with finished information should be shown on web server.

### 2.4.2 LQN Tools

This subsection introduces LQN tools available to support the LQN notation. There are two solvers for LQN models that share the same input file format. One is named as Layered Queueing Network Solver (LQNS) and the other is ParaSRVN, a simulator.

LQNS is an analytical solver [Franks00] that breaks the LQN model into separate queueing network sub-models. The individual queueing network can be solved by using mean-value analysis (MVA). The MVA results of each sub-model can be used as the MVA parameters to other sub-models. ParaSRVN simulates the LQN model by creating tokens for each call and following those tokens through the system.

The experiment controller SPEX uses an expanded modeling syntax and supports repetitive sequences of model to run. It deploys parameter controls and extracts specified results of performance characteristics.

For the case study in this thesis, we use the simulator ParaSRVN and experiment controller SPEX to predict the performance of a system with the confidence intervals.

## 2.5 Displacement Technique for Measurement

There have been significant advances in modeling formalism and model solution techniques so that it makes performance modeling relatively easy. We need the necessary data to parameterize the performance models and solve them. Data collection for solving models, validating performance models, and monitoring the system can be the most difficult task in SPE processing. The trustable parameters are also the most challenging in the trustable performance prediction from performance models. "Displacement" technique [Woodside97] is chosen as a solution for the measurements in the research.

### 2.5.1 Background

Traditional implementations of the UNIX operating system provide coarse grained, statistical measurements of CPU utilization. In the standard CPU demand tool the running process is charged with CPU time including two parts: 1) user time if the processor is in user mode, and 2) system time if the processor is in system mode. In fact there could be three relevant CPU states: user mode, system mode, and interrupt mode. In some implementations a significant amount of the execution in interrupt mode is conducted, but the CPU demand tools in the usual kernel instrumentation can not capture it. McCanne and Torek have documented that the standard CPU-demand tools cannot capture Interrupt Service Routine (ISR) execution in UNIX in the paper [McCanne93].

In fact it is difficult to directly measure the CPU demand of communications protocols and middleware such as sockets and RPCs. For some applications there is a significant amount of the protocol execution done by interrupt service routines (ISRs). If there is no process switching, the clock ticks that occur during the ISR execution can be allocated to the interrupted process. But if there is process switching, it is impossible to know which process is the destination of a message at the moment of the interruption and which process should be charged. That is because the information of the interrupt has not yet been processed.

The "Displacement" technique does not focus on the process under measurement any more and introduces the measurable CPU-intensive process. It records the displacement of a CPU-intensive process $P_I$ by the process under measurement $P_M$ and the prerequisite is the process $P_I$ that can be measured accurately. Then we can compute the CPU effort taken by the process $P_M$.

### 2.5.2 Displacement Technique Implementation

The original paper presents the displacement procedure and assumes that the process under measurement $P_M$ is or can be configured with a repetition of the operation to be measured. It requires that there is no interaction for $P_M$ in the calibration experiment and the behaviour of the operation is repeatable. Thus, the CPU demand per operation can be calculated. Here we modify this procedure to remove the use of a repeated execution in $P_M$.

We are going to calibrate the normal process $P_M$. We choose the CPU-intensive process $P_I$ to execute a fairly short loop over some arithmetic operations. The duration of its loop affects the resolution of the measurements to be obtained. $P_I$ is configured to run for a given number $L_I$ of loops. The wall-clock time is obtained and printed out at the beginning and the end of $P_I$. The procedure of "Displacement" techniques is shown in Figure 2.7.

**Step 1**: The CPU time per loop of $P_I$ is calibrated by running it alone on a workstation for some number $L_I(1)$ as illustrated in Figure 2.7 (a); let the wall-clock time interval be $T_I(1)$ seconds. The estimated CPU time for one loop traversal of $P_I$ is

$$\tau_I = T_I(1) / L_I(1) \ \textit{sec/loop} \tag{2-1}$$

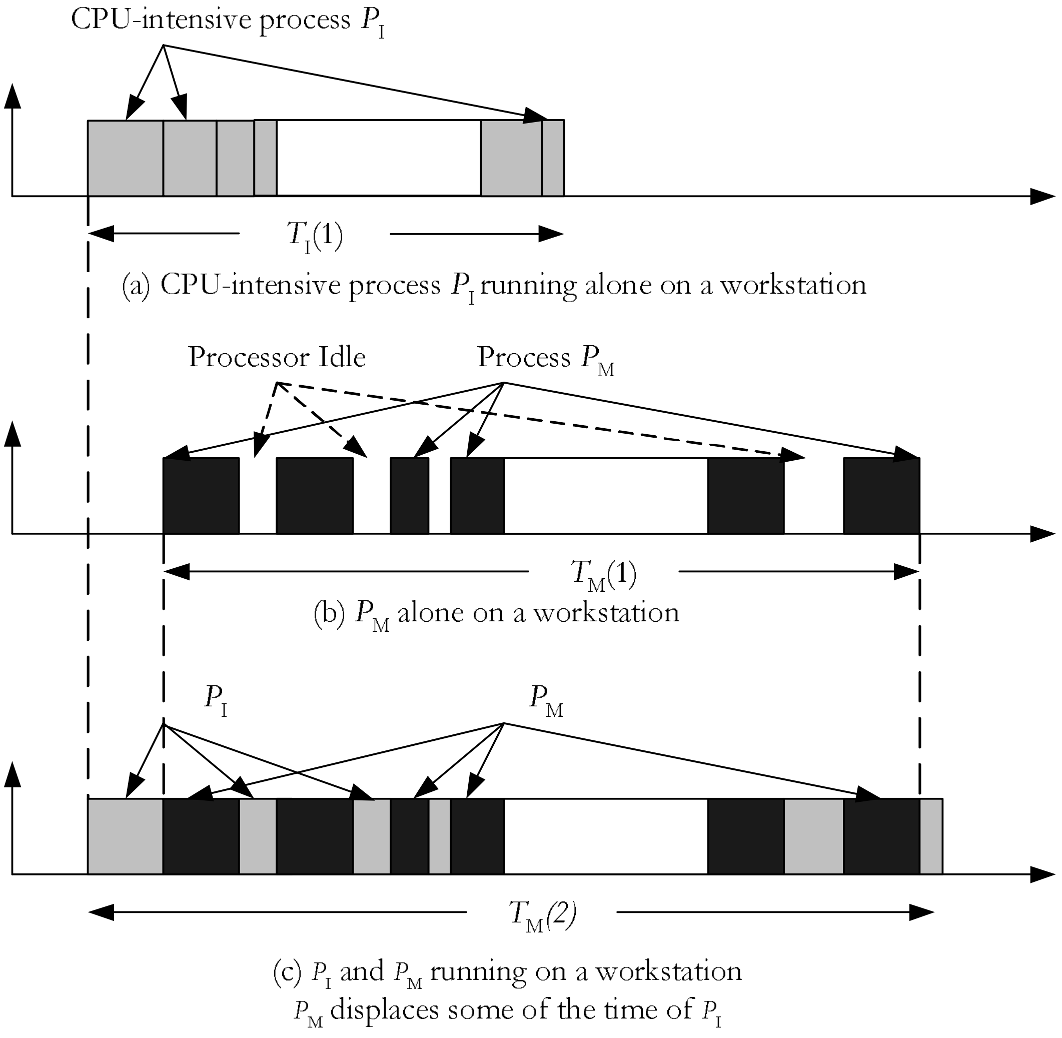

CPU-intensive process $P_I$
$T_I(1)$
(a) CPU-intensive process $P_I$ running alone on a workstation
Processor Idle
Process $P_M$
$T_M(1)$
(b) $P_M$ alone on a workstation
$P_I$
$P_M$
$T_M(2)$
(c) $P_I$ and $P_M$ running on a workstation
$P_M$ displaces some of the time of $P_I$

Figure 2.7 The procedure of "Displacement" technique implementation

**Step 2**: The two processes $P_M$ and $P_I$ are then run together in such a way that $P_I$ starts a little before and ends a little after the other, as illustrated in Figure 2.7 (c). The loop counter is set to the values $L_I(2)$ for $P_I$; it may require some experiments to adjust $L_I(2)$ to be long enough. The wall-clock time interval for $P_I$ is recorded as $T_I(2)$ seconds. Then the CPU time taken by the $P_M$ during Step 2 is

$$T_M(2) = T_I(2) - L_I(2) \quad \tau_I \qquad (2\text{-}2)$$

Figure 2.7 (b) only shows that $P_M$ is executed alone on one machine. It is obvious that $P_M$ has higher priority than $P_I$ when both of them are running together. Figure 2.7 (c) indicates $P_M$ replaces the execution time of process $P_I$ and $P_I$ can fill the gap of CPU idle during the execution of $P_M$.

### 2.5.3 Discussions

This displacement procedure in subsection 2.5.2 assumes that the two processes are running on a quiet workstation and there are only $P_M$ and $P_I$ during the two tests. We also assume the overhead for loop count in $P_I$ can be ignored. Thus we can obtain the CPU demand for the process $P_M$ with the interaction. The trade off is that if $P_M$ consists of different operations, we cannot obtain the CPU demand for each operation.

The solution of the overhead for loop count in $P_I$ is that we rerun $P_I$ alone with the loop account set to $L_I(2)$ on one quiet machine to obtain the wall-clock time $T_I(3)$ as step 1. The difference between $T_I(2)$ and $T_I(3)$ is the total CPU time of process $P_M$ we expected.

# Chapter 3 Description of CGNet

This chapter briefly provides a description of CGNet, including main components, and network description files to model the typical network. Section 3.1 presents an overview of CGNet including the component descriptions. It offers the general information for CGNet. Section 3.2 describes the configuration of CGNet. Section 3.3 shows a high-level architecture of CGNet and the related scheduling policy. Section 3.4 describes how to define, run, and collect information from CGNet for an operational network.

## 3.1 Overview of CGNet

CGNet [Hobbs01] is a network emulation tool. CGNet was designed as prototype at Nortel Networks to emulate a network in the laboratory. The tool was developed to test the feasibility of the traffic engineering technology, and it can be used to support general tests of many different kinds of networks.

The CGNet model includes nodes, sources (generator) and destinations (sinks) for the traffic, a connection topology with stated link capacities, and controller which can send control commands to the nodes. The capacity of generator and the proportion traffic which should be sent to each destination can be adjusted. We can also configure the capacity of links according to the variability of a real network. Periodic statistic reports are generated from each node once a network is running. They are sent to a statistics sink as normal traffic. The operator can send control commands through the manual controller to control the network. Figure 3.1 [Hobbs01] shows a typical network modeled by CGNet. All routers, links and traffic in a real network are abstracted into the components of CGNet and in sub section 3.4.1, there are more detailed discussions on how to model a real network into components and connections by CGNet.

CGNet is an object-oriented application. CGNet software was written in C++, but many portions were in C style with about 30,000 lines of code. It contains complicated data structures and algorithms. It also includes the complex logical interactions between functionalities and expert knowledge in network communication and traffic controlling.

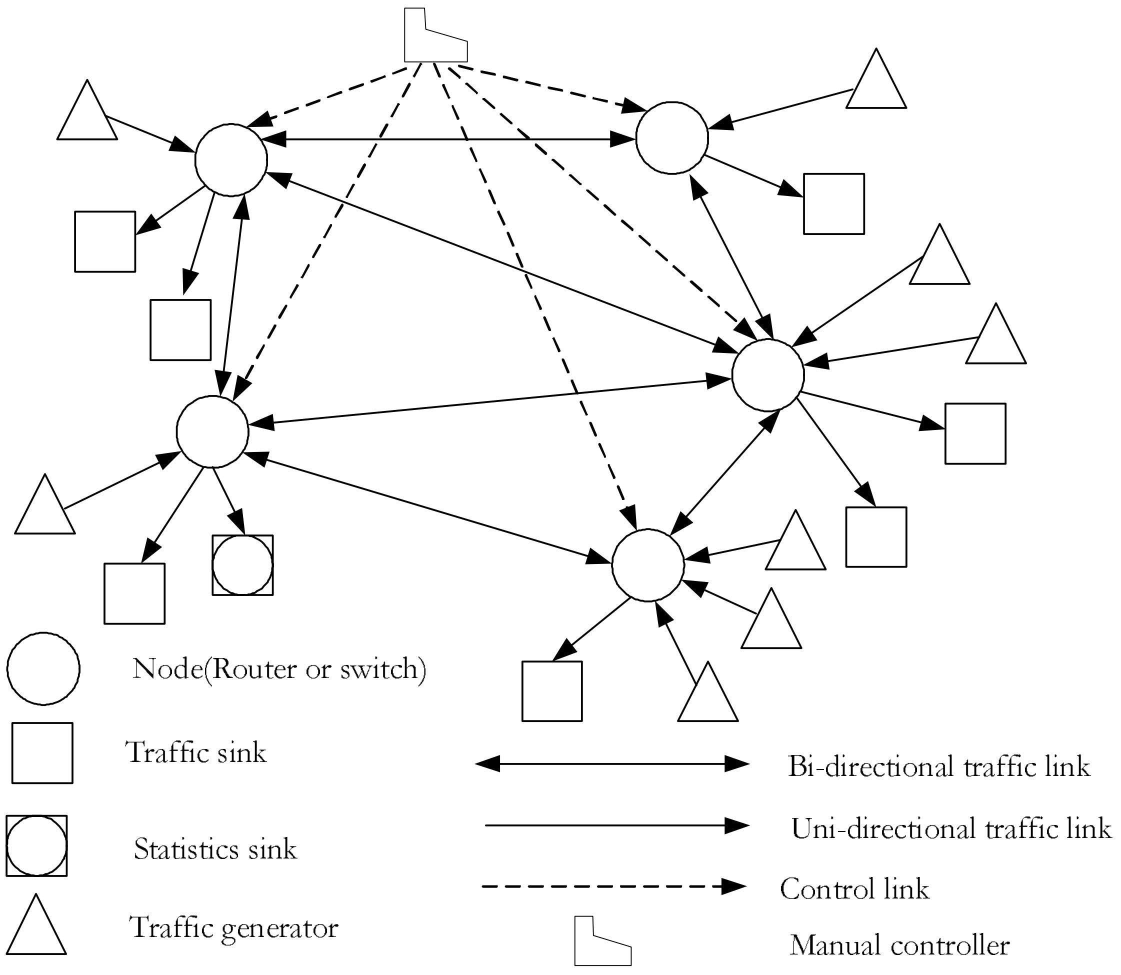


Figure. 3.1 A typical network that can be modeled by CGNet

We can configure the components of CGNet through network description files and run CGNet executables on one or more workstations. Table 3.1 shows the components of

CGNet with illustration as well as their descriptions [Hobbs01].

| CGNet Components | |
|---|---|
| ○ | **Node**: a node in CGNet represents a router in the real network. But for special use and purposes, a node in CGNet could be imaged as a switch. |
| △ | **Generator**: A generator stands for a payload traffic source, which can generate traffic at a pre-configured speed. The proportion of traffic is sent to each destination in the network. |
| □ | **Traffic sink:** A traffic sink is a destination for generated traffic. |
| ⧇ | **Statistics sink**: Each node in the network generates periodic statistic reports while CGNet is running on the machines. The statistic reports are sent as normal traffic to a statistics sink. |
| ⏢ | **Manual controlle**r: This device is used to send control commands to a node to control the network manually. In CGNet, each node's control port can be accessed from multiple controllers. |
| ⟷ | **Bi-directional traffic link**: This represents the transport media between nodes in the network. It can be thought as two unidirectional links with the same speed and cost. The traffic on these links can be conventional payload traffic as well as administration traffic, such as routing updates and statistics reports. |
| ⟶ | **Uni-directional traffic links:** These links carry traffic from a node to a traffic sink or from a traffic generator to a node. In CGNet the implementation of a uni-directional link is identical to that of a bi-directional link but in general only one direction is used. |
| ⇢ | **Control link**: a manual or intelligent controller sends commands to a node through it. |

Table 3.1 CGNet components and their attributes

## 3.2 CGNet Configuration

CGNet has defined network description files [Hobbs01]. Once CGNet executables read them, the connections can be established and the network can be setup according to network description files. Table 3.2 shows the network description files and the description of each file.

| File Name | Descriptions |
|---|---|
| **globalinfo** | This file contains a variety of global information that applies to the entire networks. |
| **nodeinfo** | It defines the characteristics of all the nodes within the network and might contain statistics interval and host name of execution platform for the nodes. |
| **generatorinfo** | It shows the characteristics of the network's generators so that generator knows which node it connects with, its total capacity, the proportion of traffic that should be sent to each destination in the network, and host name of execution platform for the generators. |
| **sinkinfo** | It describes the characteristics of each traffic sink including connected node name, statistics filename, and host name of execution platform for the traffic sink. |
| **linkinfo** | It offers the characteristics of the network inter-node links, which are bi-directional and symmetric in speed and cost. The link information includes the bandwidths and cost of each link between nodes in the network. |

Table 3.2 Configuration files for establishing CGNet

## 3.3 Description of the Software Architecture

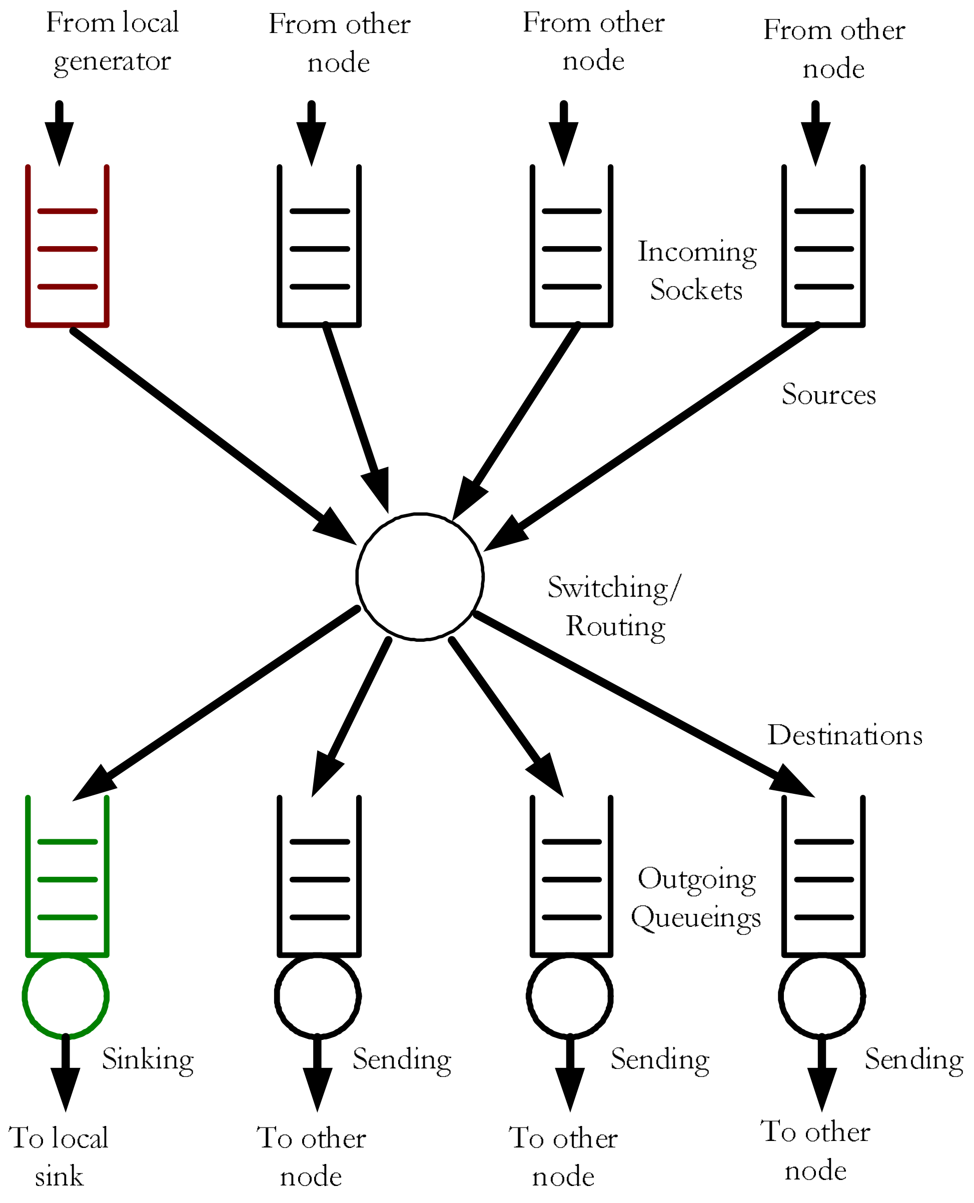


Figure 3.2 High-level architecture of the node

This section focuses on the software architecture and the behaviour such as scheduling policy, and packet handling during the execution of CGNet. Each component such as a node, generator, traffic sink, statistics sink and controller requires one executable to run on the machines and other link components can be established after CGNet starts up on the machines.

The generator sends packets with the pre-configured speed to the node to which the generator is connected. The destination for each packet has been randomly defined once it has been generated. The packets traverse the node along the path specified in the routing table and arrive at the destination, traffic sink. The traffic sink just consumes the packets when the sink receives it.

The node process has the same operations: the main thread receives packets from the incoming sockets, parses packets and switches them to the outgoing queue; the sending or sinking thread sends packets the outgoing sockets; the sending thread also emulates the network delay for the link. Once a packet comes into the node from its local generator or some other node, it waits in the buffer space of the incoming socket. The packet is processed when the switching/routing server selects that socket and reads the packet from it. The switching/routing server polls all the incoming sockets within one node to process the packets with the round-robin cyclic service discipline [Takagi90]. The packet is read and parsed, then switched to the corresponding outgoing queue according to the routing table. There is one thread associated with each outgoing queue. The thread can be either a sinking thread or a sending thread. For the sink thread, it just dequeues the packet and writes it to the outgoing socket. But for the sending thread, it also dequeues the packet and writes it to the outgoing destination socket, and then emulates the network delay. The network delay is emulated by the UNIX select function with a timeout value which does not block the CPU processor. High level architecture for the node has been shown in Figure 3.2.

## 3.4 A CGNet Experiment

In this section we focus on how to run CGNet and what kind of information we can obtain from it. Subsection 3.4.1 describes the definition of CGNet for the operational network and how to configure it. Subsection 3.4.2 shows how to run and terminate it. Subsection 3.4.3 presents the output of CGNet

### 3.4.1 Definition of the Operational Network by CGNet

We choose an operational network that we study and focus on all routers within the network. We define the routers as nodes in CGNet. The set of nodes within the network is denoted **Node** and let **Node**= {AA, BB, CC, ……}. For each node in set **Node**, we identify all connections to the node. If the other end of the connection is also the element of **Node**, we define the connection as a link. For each XX ∈ **Node,** all links connected to node XX can be defined as a link set **OutLinks(XX)** = { XX-AA, XX-BB, XX-CC, ……}, and the set of neighbor nodes **Neighbor(XX)** = {AA, BB, CC, ……}. Obviously XX ∉ **Neighbor(XX)** and for each YY ∈ **Neighbor(XX)** we have XX-YY ∈ **OutLinks(XX)**. All elements outside **Node** that can send the data traffic to node XX are defined as generators **G(XX)**= {gXX1, gXX2, ……}. All elements outside **Node** that can receive the data traffic from node XX are defined as traffic sinks **S(XX)**= {sXX1, sXX2, ……}. We repeat to define links, the generators and traffic sinks for each node. Thus we define the elements in the operational network as the components of CGNet. For simplicity, we merge all generators for one node into one generator and combine all traffic sinks for one node in one traffic sink. For each node XX ∈ **Node**, we can denote the merged generator as gXX for **G(XX)**, and the combined traffic sink sXX for **S(XX)** respectively. The corresponding data rates are also combined together.

We define the attributions and quantify the quantity characteristics for each element in the network. We choose the host name of execution platform such as nodes, generators and sinks because each of them requires one running instance of the appropriate executable. In addition for each link we should collect link information, the speed Speed(XX-YY) and the cost Cost(XX-YY). All bi-directional links are assumed to be symmetric in speed and cost. For a generator and a traffic sink we should define the node name, for instance the node of the generator gXX and the traffic sink sXX is XX. The generator supports a Constant Bit Rate (CBR). We measure the traffic speed of generators and proportion of traffic sent to the destination. For example, the data traffic generated by the generator gXX is sent to traffic sinks sAA, sBB, sCC, …… respectively. We let **Destination(XX)** = { AA, BB, CC,……} Here XX, AA, BB, CC, …… ∈ **Node** and XX ∉ **Destination(XX)**. We can define packet class as (XX, YY) that has a unique source-destination pair, and here YY ∈ **Destination(XX)**. We use $r_{XX}$ to denote the speed (capacity) of generator gXX. Then we investigate the destinations of packets generated by gXX for a period long enough. For each destination ZZ ∈ **Destination(XX)** there are $n_{g(XX,ZZ)})$ packets sent to sZZ from generator gXX. We can then define $r_{XX}$ and pair (ZZ, $n_{g(XX,ZZ)}$) for each destination. It is easy to obtain the proportion of data traffic for destination ZZ from generator gXX is $P_{XX,ZZ} = \frac{n_{g(XX,ZZ)}}{\sum_{YY \in Destination(XX)} n_{g(XX,YY)}}$. Then the speed of data traffic for destination ZZ from generator gXX is $r_{XX,ZZ} = r_{XX} * P_{XX,ZZ} = r_{XX} * \frac{n_{g(XX,ZZ)}}{\sum_{YY \in Destination(XX)} n_{g(XX,YY)}}$.

### 3.4.2 Execution of CGNet

From the subsection 3.4.1 we define CGNet for an operational network and store the configuration information into network description files. This subsection focuses on the execution of CGNet.

Starting up a network requires many commands to be executed. We create scripts to start the various network components such as nodes, generators, traffic sinks and statistics sinks. We choose the machines on which we start up the network and the host name of the machine should be consistent with the definition of the components in the network description files. We can choose one or more machines to run CGNet for the purpose of experiments. We put the CGNet executables somewhere on the UNIX/LINUX path, and start the executables from the directory where network configuration files are stored. The programs are tolerant to different start-up sequences, each waiting patiently for neighbors to initialize if they are started first. The generator will not send packets until the connected node starts up. The CGNet executables read the configuration files, network description files during initialization so that they can configure themselves and connect to each other. Once all the components and connections are setup, the communication is established.

The original version of CGNet uses an infinite loop to keep reading the packets in the incoming sockets and processes them. If you want to terminate an experiment, you can kill one node process. It causes the broken pipe error if other processes try to write the sockets of the node you have killed. The broken pipe signal can be used to terminate the executables, nodes, generators and traffic sinks so that CGNet is terminated. In this research, each node terminates after a set number of received packets. The termination is propagated by broken pipe after the first node terminates. Thus we can adjust the running time and measured packet number arbitrarily for our purpose. We use $T_{XX}$ to express the execution time of node XX, XX $\in$ **Node**. Thus there are $r_{XX} \times T_{XX}$ packets sent from generator gXX to node XX.

### 3.4.3 CGNet output

Once a network is running, special network statistics are captured by statistics executable we call it as statistics sink

The node process in the network updates statistics information for every data packet. These statistics are gathered for a statistics interval as defined in the *nodeinfo* file. The node sends the statistics report to its selected statistics sink at the end of each statistics interval and the statistics database is reset. Each statistic represents only the events of the previous statistics interval.

A tool to collect all statistics information for each node has been developed in the research. We can obtain the statistics information for each node XX, XX $\in$ **Node** as follows.

Number of packets received from generator: $n_{RG}$

Number of packets received from **Neighbor(XX)**: $n_{RN}$

Number of packets sent to **Neighbor(XX)** : $n_{SD}$

Number of packets sent to the traffic sink: $n_{SK}$

Number of packets lost in XX because the outing queues are full: $n_L$

We define $n_R$ = $n_{RG} + n_{RN}$ as number of received and switched packets and $n_S$ = $n_{SD} + n_{SK}$ as the number of sent packets. In fact, each node performs the statistics for every packet it receives, which means all the packets received should be processed. There is $n_R = n_S + n_L$ and it covers the case where there is no packet loss, $n_L$ =0.

# Chapter 4 Constructing a Performance Model for CGNet

This chapter describes how to build the performance model for CGNet system with one sample configuration. The network and its topology are introduced in section 4.1. We trace the scenarios in the system so that we can understand the behaviour of the system. We build the template model for each scenario and merge them into a sub model for each node. All the sub models for nodes are assembled and composed into a complex performance model for the network. The procedures of constructing the performance model for CGNet are broken down into the following sections:

4.2 Determining scenarios for data traffic across the network

4.3 Determining detailed scenarios in a node

4.4 Mapping scenarios to the node-path sub models

4.5 Assembling the node-path sub models into a node sub model

4.6 Composing the node sub models into the system model

The compositional strategy has been generalized in the section 4.7, and a tool for building the model will be described in chapter 7.

## 4.1 Network Description

To present the compositional modeling approach, we choose one sample network and focus on the performance characteristics of the stable state of the network. In this section we briefly introduce the network for which we build the performance model on its topology and components.

The topology of the sample network modeled by CGNet is shown in Figure 4.1. There are five nodes in the network and each node has a generator, a traffic sink and a statistics sink. The network includes bi-directional traffic links between the nodes so that they can communicate with one another. The definition of nodes, generators, sinks and links are configured in the network description files in CGNet as described in chapter 3. The generators, traffic sinks and statistics sinks for all nodes are running on one machine but each node is executing on its other machine separately.

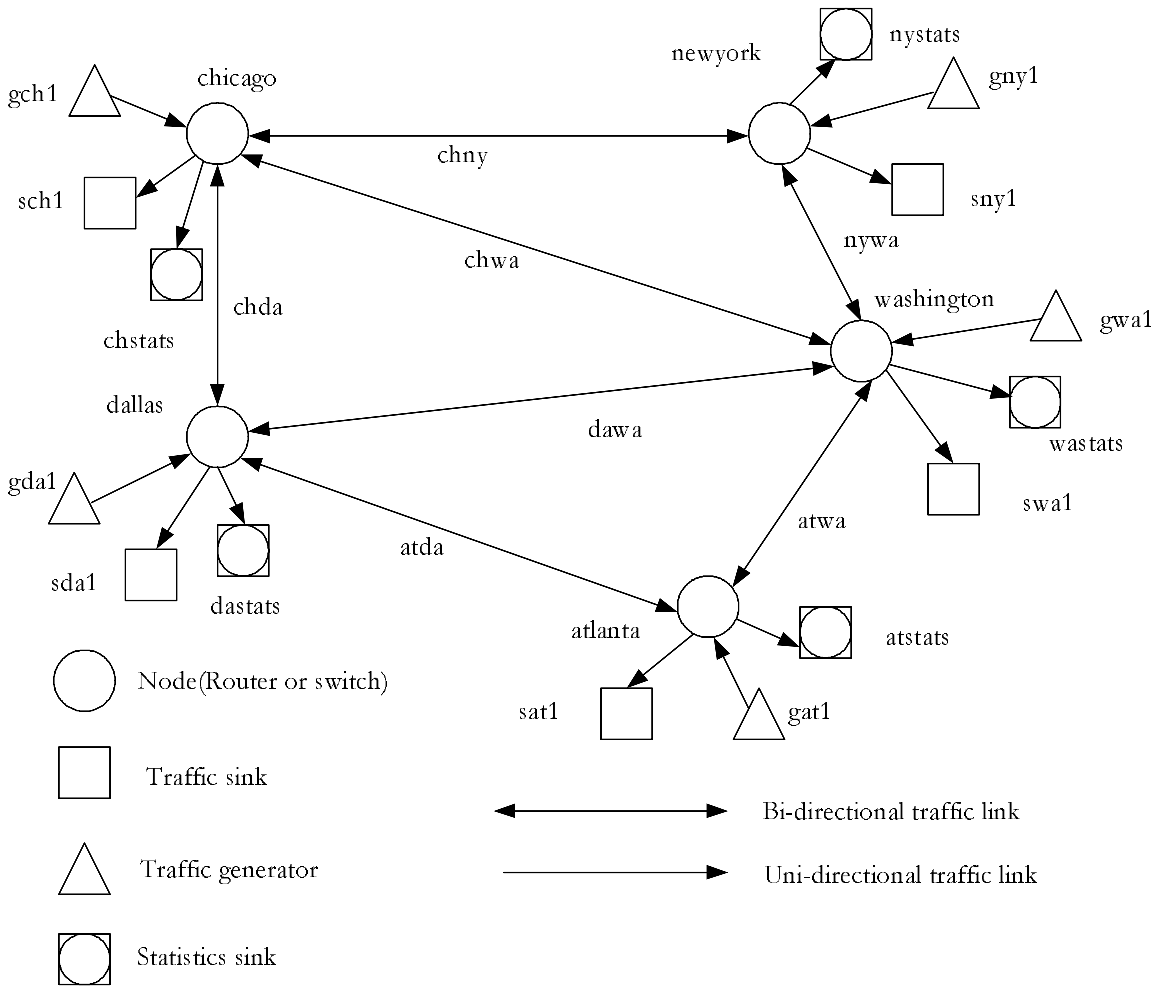


Figure 4.1 Topology of an example network model by CGNet

One may notice that the controller components of CGNet are absent in the network. It is reasonable to leave them out because here we pursue the performance analysis of the stable state of the network.

## 4.2 Determining Scenarios for Data Traffic across the Network

This section presents the operations of the routers for a typical packet traversing across the network to help us understand the behaviour of the network. We focus on the main components, and understand the interactions between them so that we can derive a panorama of the system especially in the performance characteristics.

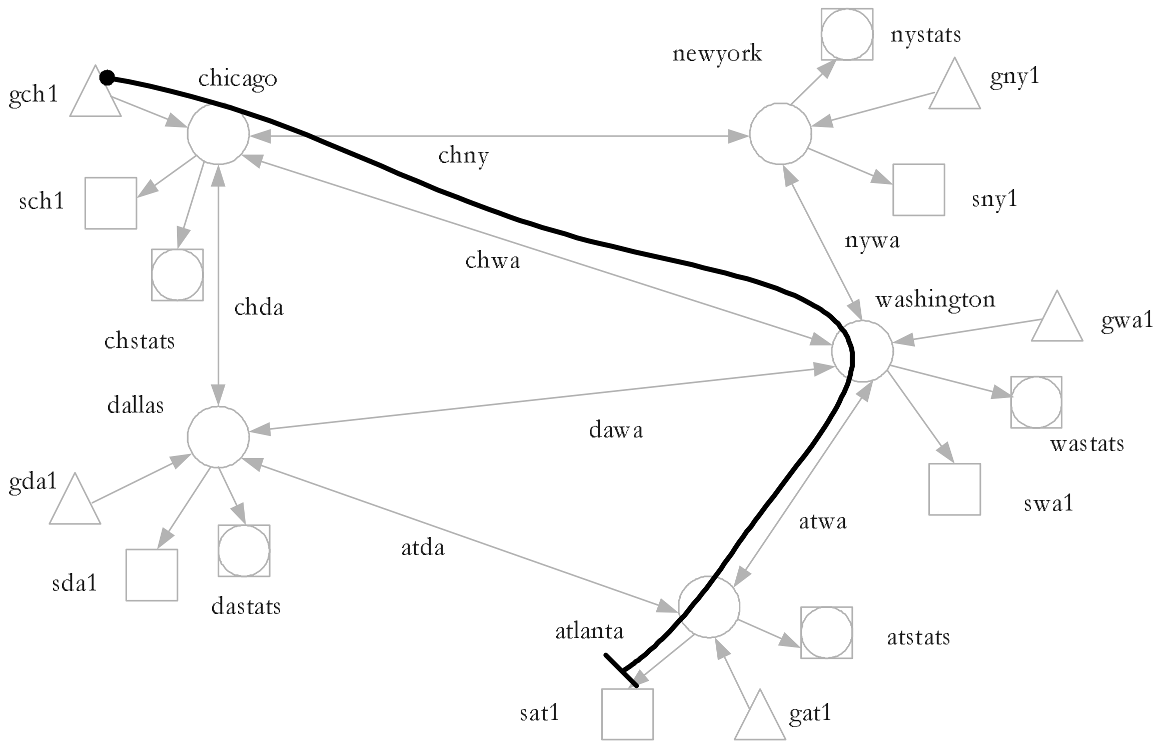


Figure 4.2 Path view of a packet going from a generator to a traffic sink

The path of a data packet follows the same pattern for all routers through the network. Each packet starts from a generator and ends at a sink. It traverses the routers along the path defined in the routing table. We can define the packet class for the packets that share the same generator and the traffic sink. For each packet class (XX, ZZ) defined in section 3.4.1, node XX is the *source* router and node ZZ is the *destination* router. Certainly there is some packet class that is related to routers other than the *source* router and the *destination* router. The *forwarding* router is the intermediate router to forward this packet class. We define **ForwardR(XX, ZZ)** = {AA, BB, CC, ……}. For each packet class, the number of elements in **ForwardR(XX, ZZ)** could be zero or more and XX, ZZ $\notin$ **ForwardR(XX, ZZ)** . We also can define the path for packet class (XX, ZZ) as **Path(XX,ZZ)** = <XX, AA, BB, CC, …, ZZ>. Among **Path(XX,ZZ)** , AA, BB, CC, … stand for the elements in the set **ForwardR(XX, ZZ)**. In the sequence of **Path(XX,ZZ)** , any neighbor nodes EE, FF have been further defined: EE is the *upstream* router of FF for packet class (XX,ZZ) and FF is the *downstream* router of EE for packet class (XX,ZZ). We can have a node XX that has no *upstream* router for packet class (XX,ZZ) and node ZZ that has no *downstream* router for packet class (XX,ZZ). The path of the packet class has been defined when the network was established.

The path view [Woodside95B] for a packet traversing from generator to traffic sink through the network is described by the Use Case Maps in figure 4.2. As the path illustrated in figure 4.2 we define the packet class (CH, AT) for the path and the path has been overlaid on the router in the network. The *source* router is node Chicago (CH); the *destination* router is node Atlanta (AT) and the *forwarding* router is node Washington (WA). We have **ForwardR(CH, AT)** ={WA}, and **Path(CH, AT)** = <CH, WA, AT>.

There may be other sources of stimuli in the original CGNet that cause work in network, like administration and statistic packets. We can trace out the scenarios other than the one described in the previous paragraph. Fortunately the frequency of them handled by the node is so low that we can ignore it during the performance analysis for the network.

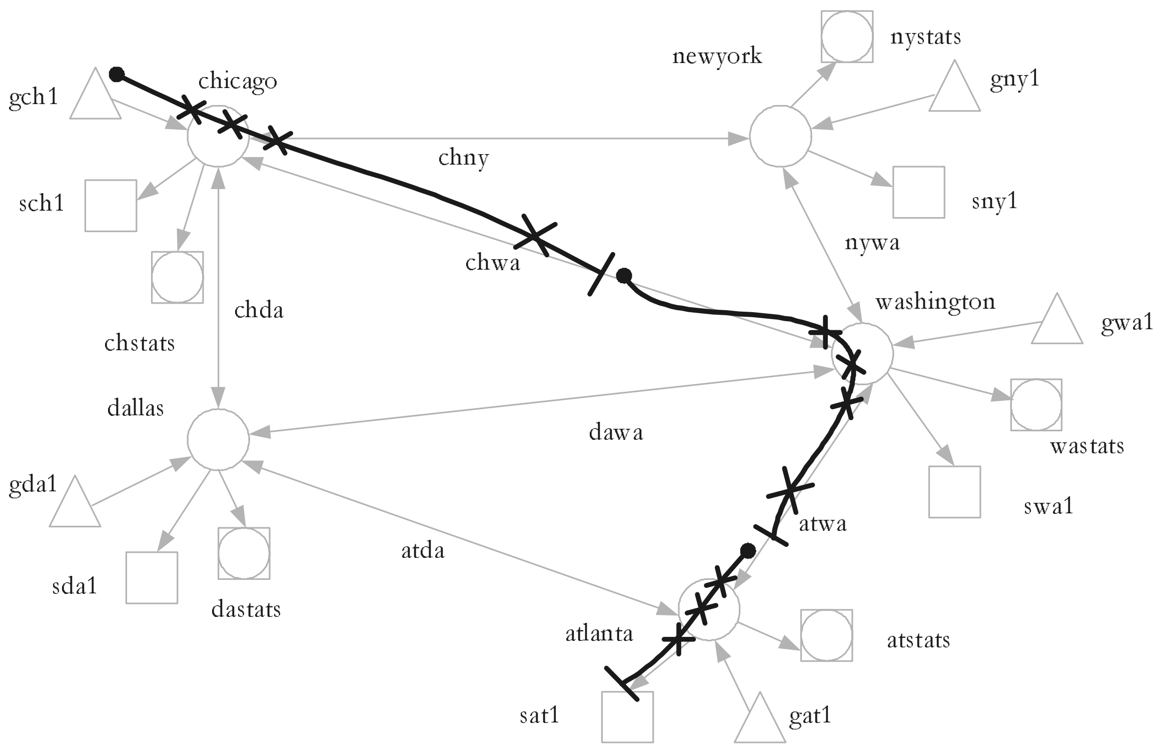


Figure 4.3 High level of UCMs for packet class (CH, AT) through Network

A packet of packet class (XX, ZZ) goes from a generator to a sink along the path **Path(XX,ZZ)**. Each node along **Path(XX,ZZ)** performs the same operations. These operations are briefly described as follows: the main thread receives the packet from the incoming socket, switches the packet according to the routing table to the corresponding outgoing queue; and the sending thread sends the packet to the next hop, or the sinking thread sends it to the traffic sink; In addition to that, the sending thread emulates the network delay. These operations can be defined as responsibilities in UCMs, which can be overlaid on the path illustrated as Figure 4.2. The path can be broken into path fragments for each node and the similarity of responsibilities for each node is more obvious. The scenario fragment can be traced out through each path fragment. Figure 4.3 shows the high level of UCMs for the packet class (CH, AT), which traverses the network from the *source* router to the *destination* router.

## 4.3 Determining Detailed Scenarios in a Node

A Router is an intelligent and complex device. It processes all the packets from the incoming sockets. The diversity of the packet classes the router processes can drive the router to play different roles for each packet. The role could be the *source* router, the *forwarding* router, or the *destination* router. Whichever role it is, it performs the packet processing for each packet class according to the routing table. We choose a node XX with four incoming sockets and four outgoing sockets as an example shown in Figure 4.4, to describe the possible UCMs for each packet class.

Three typical packet classes can traverse through the node XX: packet class (XX, BB), packet class (AA, XX), or packet class (AA, BB). For packet class (XX, BB), node XX is the *source* router. The scenario starts from gXX and the packet class is sent to the *downstream* router of node XX for packet class (XX, BB). The scenario for packet class (XX, BB) in node XX is shown by the UCM in the right of Figure 4.4. For packet class (AA, XX), node XX is the *destination* router. The scenario ends at XX's traffic sink and packet class is received from the *upstream* router of node XX for packet class (AA, XX). The scenario packet class (AA, XX) in node XX is shown by UCM in the left of Figure 4.4. For packet class (AA, BB), node XX is the *forwarding* router. The packet class is received from the *upstream* router of node XX and sent to the *downstream* router of node XX for packet class (AA, BB). The scenario packet class (AA, BB) in node XX is shown by UCM in the middle of Figure 4.4.

Therefore we can identify all packet classes which traverse through the node. The scenario fragment of each packet class can be traced and responsibilities for each scenario fragment are well defined. We can derive all the scenario fragments within each node.

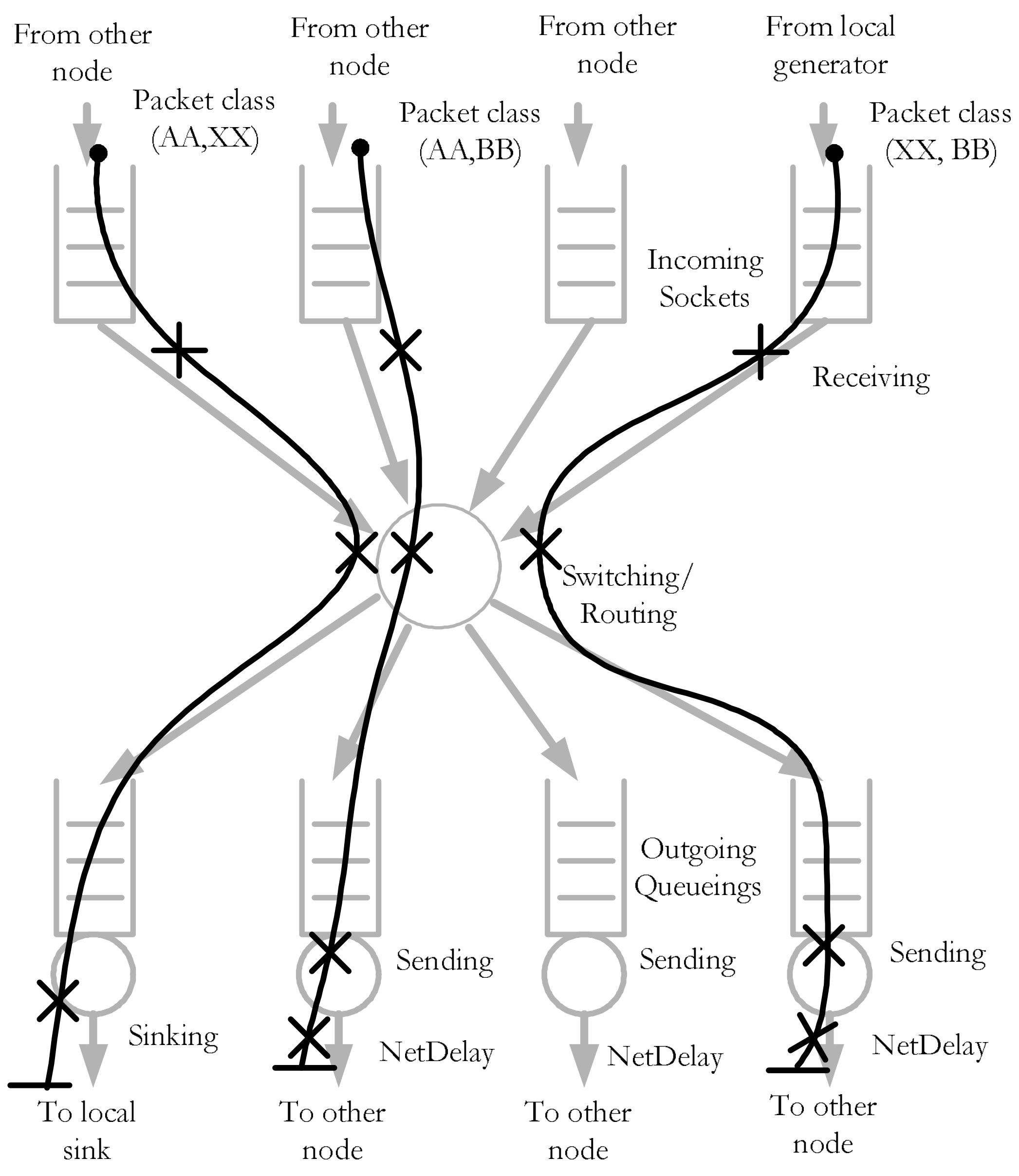


Figure 4.4 Use Case Maps of packets going through one node in CGNet

## 4.4 Mapping Scenarios to Node-Path Sub Models

This section focuses on mapping the scenario for a single path through one node to a sub model, called a node-path sub model. We choose the node Atlanta in the network shown in section 4.1 and build all node-path sub models for all scenarios involving node Atlanta. The work has been broken into two steps: mapping possible scenario to the template node-path sub model which will be shown in subsection 4.4.1, and identify all scenarios involving node Atlanta and substitute for the notation name of the template node-path sub model for each scenario shown in subsection 4.4.2.

### 4.4.1 Mapping Scenarios to Template Node-Path Sub Models

Before we perform mapping a scenario to the template node-path sub model, we should define the tasks for our sub model. From previous discussions on the responsibilities of each node it is clear that we can define the receiving, switching, sending, sinking and network delay tasks for the sub model. Separate receiving tasks are defined in order to provide a separate buffer for each task. Thus we can model buffer overflows separately for each link. The execution demand of each entry in the receiving task is set to 0, and the call to the entry in the switching task is a synchronous call. For the sending task, we know the sending thread sends a packet and emulates the network delay with the UNIX select function with a timeout value. The network delay does not block the processor but can block the sending thread. The network delay tasks are introduced and they have their own host processor as an infinite server. Network delay is a pure delay as calculated in the LQN model and will not consume CPU time on the host processor of receiving, sending, switching and sinking task.

In fact we can associate the tasks with the hardware resources. In one node each receiving task is associated with an incoming socket. The sending task, together with its network delay task, is corresponded to an outgoing socket. There is only one switching task that is in charge of the switching/routing server. The sinking task is connected to the traffic sink that is connected to the node.

To name the notation of the LQN model in terms of the entry, task, and the host processor we conveniently define the rules for naming in the LQN model construction. The rule for the name of a node in LQN model uses the first two capital letters of the node name defined in the CGNet model. All the names of the receiving, switching, sending, sinking and network delay task have RCV, SW, SEND, SINK, DELAY respectively as key words.

We choose a typical packet class (XX, ZZ) and explain how to map a scenario fragment to a template node path sub model. The packets of this packet class are generated by the generator gXX and the destination of packets is the traffic sink sZZ. Node XX receives the packet class from gXX, parses and switches it and sends it to YY through link XX-YY according to the routing table in node XX. YY gets the packet class from XX-YY, and does the same work to ZZ via link YY-ZZ. ZZ receives it from YY-ZZ, finds that the next hop is the local traffic sink from the routing table of ZZ, and sinks it to the local traffic sink. The number of intermediate YY can be zero or more. XX, YY, ZZ represent the *source* router, a *forwarding* router and the *destination* router respectively for the packet class (XX, ZZ) and **Path(XX, ZZ)** is <XX, YY, ZZ>. We build a node-path sub model along the path node by node for each scenario fragment. The node-path sub models cover all types of scenario fragments in a node discussed in section 4.3. Each of them can be used as a template node-path sub model.

At first we define general variable node names AA, BB, CC, DD: AA is the node we built the model for; BB is the *upstream* router; CC is the *downstream* router, and DD is the *destination* router. Thus we can express the role changing for node XX, YY, ZZ and define the task and entry names conveniently.

The receiving task of node AA is in charge of receiving packets from the incoming socket. Each entry in the receiving tasks should express the operation on the packets with different destinations. We separate the packets here because they can lead to different path. The packets come from the local generator or from other nodes as *upstream* routers. For the former we can define the receiving task named as "AA_RCV". For the latter we define it as "BBAA_RCV". The receiving task in Figure 4.5 node XX is the "AA_RCV" type and the receiving tasks in Figure 4.6 node ZZ and in Figure 4.7 node YY are the "BBAA_RCV" type. In task "AA_RCV" the entry name is "RCV_AA_DD" (see Figure 4.5 node XX). In task "BBAA_RCV" the entry name is "BBAA_AA_DD" (see Figure 4.7 node YY) if DD $\neq$ AA; but if DD = AA, the entry name is "BBAA_AA_SE" (see Figure 4.6 node ZZ).

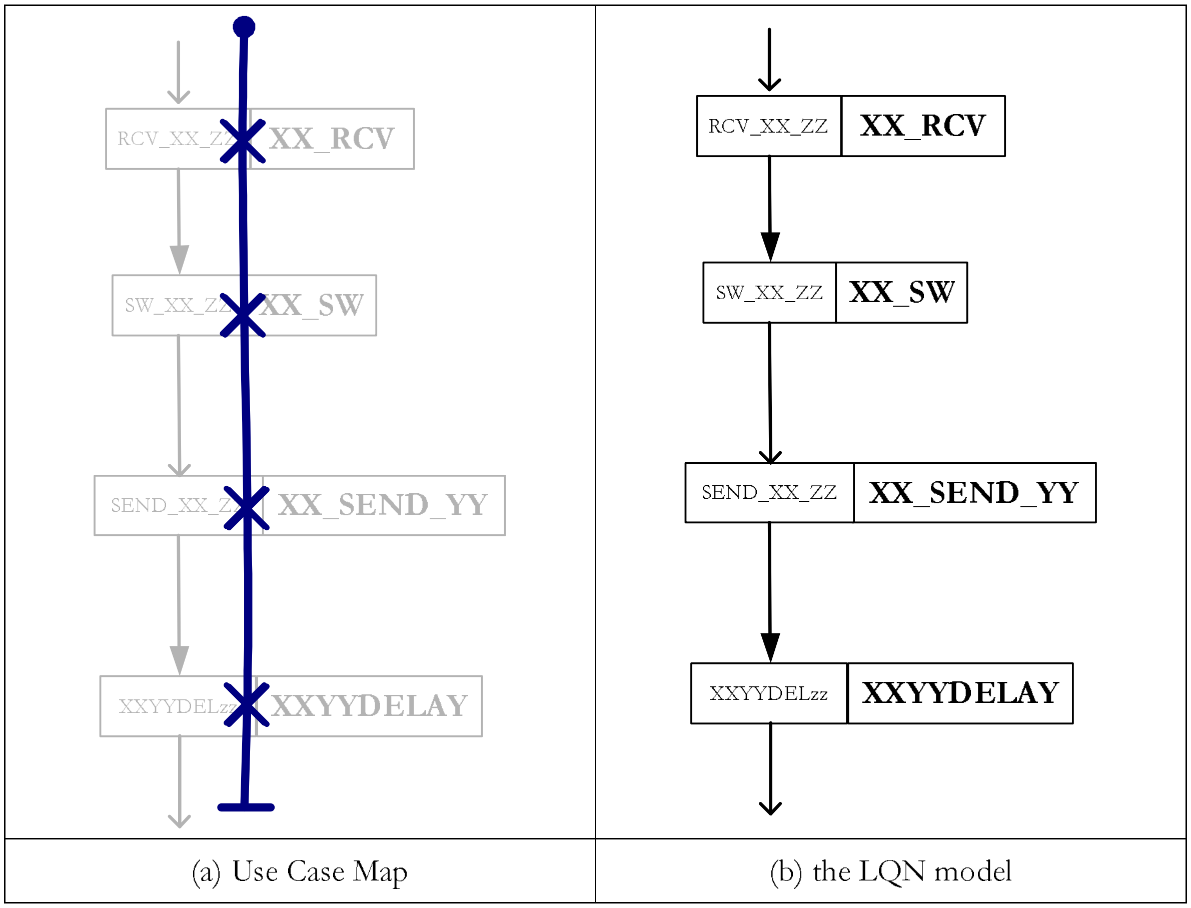
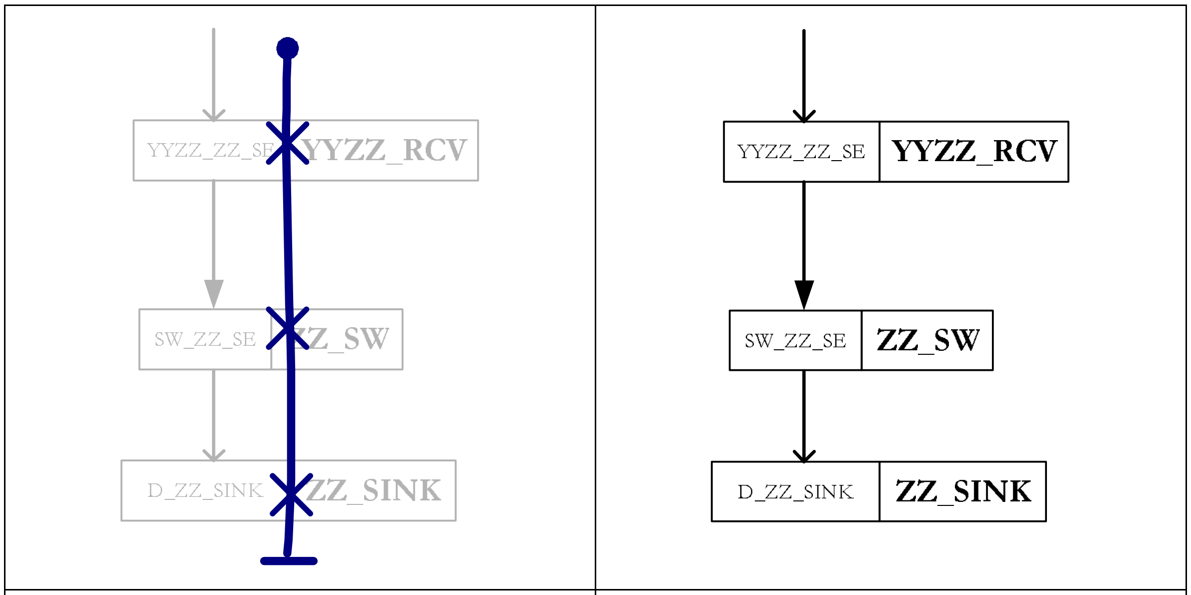


Figure 4.5 Handling of Packet Class (XX, ZZ) in node XX with the *downstream* router YY

The switching task is handling parsing and switching packets to the outgoing queues. Each entry in the switching task is for the packets with different destination. The switching task name is “AA_SW”. Node XX, YY, and ZZ in Figure 4.5, Figure 4.6, and Figure 4.7 respectively share the same style in the name of the switching task. The entry name is named as “SW_AA_DD”, and Figure 4.5 Node XX, and Figure 4.7 Node YY follow it for the case DD $\neq$ AA. If DD = AA, the entry name is “SW_AA_SE” shown in Figure 4.6 Node ZZ.

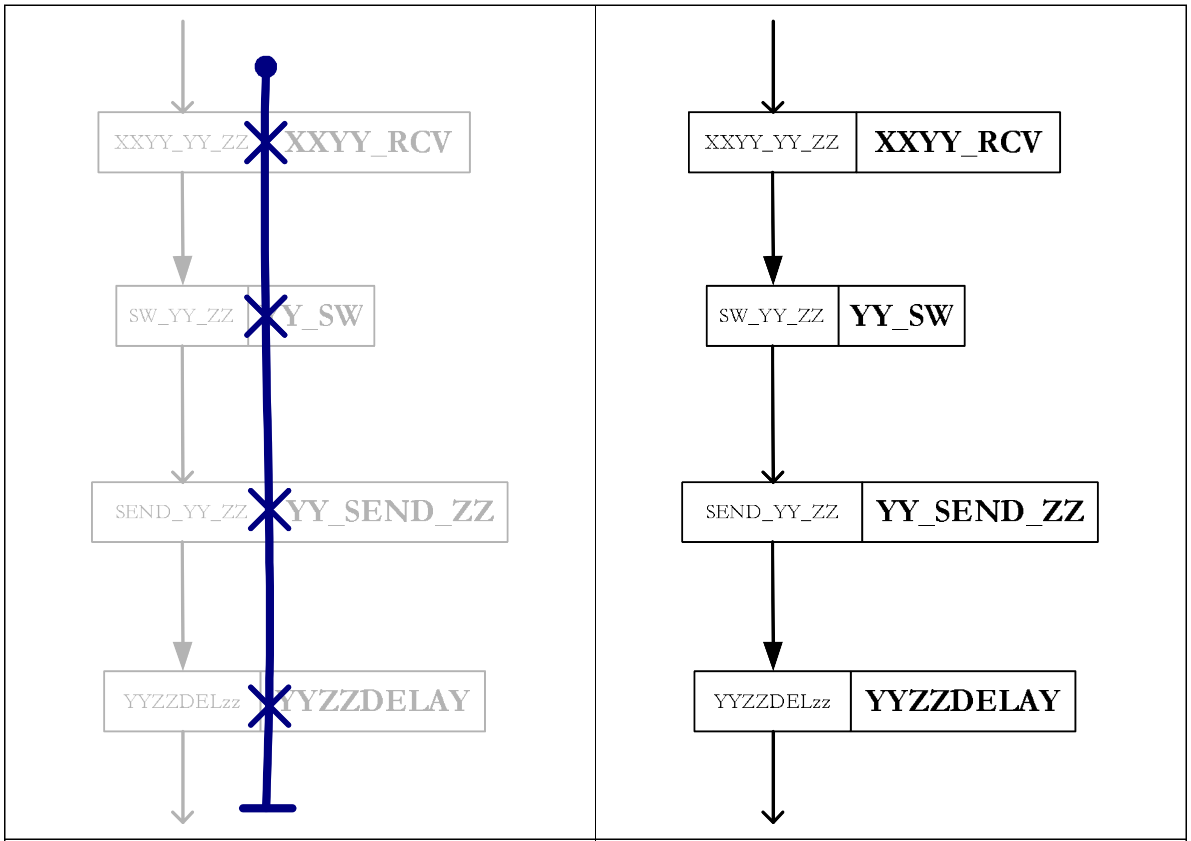


(a) Use Case Map (b) the LQN model

Figure 4.6 Handling of Packet Class (XX, ZZ) in node ZZ with the *upstream* router YY

The sending task represents the sending thread, which dequeues the packet and sends it to outgoing sockets. It is related to the network delay task. The network delay task emulates the network delay. Entries in these two tasks are still for different destinations. The sending task name is “AA_SEND_CC” and the entry for sending task is “SEND_AA_DD”. The network delay task name is “AACCDELAY” and the entry for the network delay task is “AACCDELdd” (dd is the small letters of the name of the *destination* router). There is no sending task and network delay for packet class (XX, ZZ) in node ZZ. Figure 4.5 Node XX and Figure 4.7 Node ZZ show the sending tasks and network delay tasks following the naming rules.

The sinking task performs the same as the sending thread in sending the packet to an outgoing socket but there is no network delay. There is only one entry because it is only for the packets at their *destination* router. The sinking task name is "AA_SINK" and entry name is "D_AA_SINK". Only node ZZ has the sink task and Figure 4.6 defines the sinking task.



(a) Use Case Map | (b) the LQN model

Figure 4.7 Handling of Packet Class (XX, ZZ) in node YY with the *upstream* router XX and the *downstream* router ZZ

When we perform the substitutions of the node-path sub models in the following subsection, we only substitute for AA, BB, CC, DD and dd. The complete naming rules for entry, task and host processor in the LQN model are shown in appendix A [Marcotty86].

In the previous definition of naming conventions for entry and task, we have described mapping the responsibilities to the entries in tasks in the LQN model together. Then we investigate the interaction between entries and mapping them to the type of calls between entries in different tasks.

The receiving tasks read all the packets from the socket and these packets come from the generator or other nodes. The call to the receiving task should be an asynchronous call. Once the packets are received from the sockets to the workspace, the switching task performs switching/routing for the packets according to the routing table. Thus the call from receiving task to the switching task is synchronous.

After the routers make the decision about which link the packet should be switched to, the packet will be put in the outgoing queue for sending or sinking. The entry in the switching task does not expect any reply and continues to perform switching/routing again. Hence the entry in the switching task makes the asynchronous call to the entry of the sending task or sinking task.

CGNet uses one thread for each outgoing queue, which is in charge of sending/sinking packets. The sending thread emulates the network delay in the sending case. It is blocked during the emulated network delay. The synchronous call is used from the sending task to the network delay task. The thread writes the packets to the outgoing socket and does not wait for a reply from the processing of the next hop. So from the network delay task to the receiving task of the next hop, the call is asynchronous. It is consistent with expressing a call to receiving task in the previous paragraph.

Packet class (XX, ZZ) is generated by gXX and the *downstream* router of node XX for packet class (XX, ZZ) is YY. The UCM for packet class (XX, ZZ) in node XX is shown in Figure 4.5 (a). The mapped node-path sub model is shown Figure 4.5 (b).

Packet class (XX, ZZ) arrives at the *destination* router ZZ and the responsibilities along the path in node ZZ are shown as the UCM in Figure 4.6 (a). The corresponding node-path sub model is derived from the UCM and is shown in Figure 4.6 (b). YY is the *upstream* router of node ZZ.

Node YY receives packet class (XX, ZZ) from node XX and sends it to node ZZ. Figure 4.7 (a) shows the UCM with responsibilities. The *forwarding* router YY has the *upstream* router XX and the *downstream* router ZZ for this packet class. We can obtain the node-path sub model of node YY in Figure 4.7 (b)

### 4.4.2 Substituting Template Node-Path Sub Models for Atlanta, An illustration

Before we substitute the node-path sub model for node Atlanta for the template LQN sub model, we must first go through the system and collect information to define the roles of node Atlanta for each packet class. From topology in Figure 4.1, there are three sources that can send packets directly to node Atlanta: node Dallas, node Washington and local generator. We have **Neighbor(AT)** ={DA, WA}and **OutLinks(AT)** = {AT-DA, AT-WA}. From the routing table we can derive the *destination* router for the packet class and the next hop. If one packet class comes from node XX ∈ **Neighbor(AT),** node XX is the *upstream* router for node AT and there must be at least an entry in routing table of node XX with next hop AT. If one packet class goes to node YY ∈ **Neighbor(AT),** node YY is the *downstream* router for node AT and the routing table of node XX with an entry for next hop YY. We retrieve the routing tables from the relevant nodes, node Atlanta, node Dallas and node Washington and list them in Table 4.1. The entries in the routing table involving next hop

Atlanta are made bold (see table 4.1(b) (c)).

| Sink Name | *Destination* Router | Through Link | Next hop |
|---|---|---|---|
| sat1 | ATLANTA | null | sat1 |
| sch1 | CHICAGO | ATWA | WASHINGTON |
| sda1 | DALLAS | ATDA | DALLAS |
| sny1 | NEW YORK | ATWA | WASHINGTON |
| swa1 | WASHINGTON | ATWA | WASHINGTON |

(a) Routing table for node Atlanta

| Sink Name | *Destination* Router | Through Link | Next hop |
|---|---|---|---|
| sat1 | **ATLANTA** | **ATDA** | **ATLANTA** |
| sch1 | CHICAGO | CHDA | CHICAGO |
| sda1 | DALLAS | null | sda1 |
| sny1 | NEW YORK | DAWA | WASHINGTON |
| swa1 | WASHINGTON | DAWA | WASHINGTON |

(b) Routing table for node Dallas

| Sink Name | *Destination* Router | Through Link | Next hop |
|---|---|---|---|
| sat1 | **ATLANTA** | **ATWA** | **ATLANTA** |
| sch1 | CHICAGO | CHWA | CHICAGO |
| sda1 | DALLAS | DAWA | DALLAS |
| sny1 | NEW YORK | NYWA | NEW YORK |
| swa1 | WASHINGTON | null | swa1 |

(c) Routing table for node Washington

Table 4.1 Routing tables for part of nodes in the network.

The substitution rules for template node-path sub model are outlined as follows:

1. If node Atlanta receives a packet class from any BB $\in$ **Neighbor(AT)** and sends it to node CC $\in$ **Neighbor(AT)** , we infer the *destination* router DD of this packet class from the routing table. In the template node-path sub model of Figure 4.7 we substitute AT for "YY", and BB for "XX". We substitute CC for "ZZ" in the task name and substitute DD for "ZZ" in the entries of the receiving, switching and sending task. For the entry in the network delay task, we substitute CC for "ZZ" and dd for "zz" ('dd' is small letters of the *destination* router DD). Node AT is the *forwarding* router.
2. If node Atlanta receives a packet class from any BB $\in$ **Neighbor(AT)** and sinks it in the local traffic sink, We substitute AT for "ZZ", and BB for "YY" in the template node-path sub model of Figure 4.6. Node AT is the *destination* router.
3. If node Atlanta receives a packet classes from local generator and sends it to any CC $\in$ **Neighbor(AT)**, we infer the *destination* router DD of this packet class from generatorinfo. In the template node-path sub model of Figure 4.5, we substitute AT for "XX", CC for "YY", DD for "ZZ" and dd for "zz" ('dd' is small letters of the *destination* router DD). Node AT is the *source* router.

For each node in **Neighbor(AT)** we identify the possible traffic to node Atlanta. The routing table in Table 4.1 (b) for Dallas shows there is an entry with next hop ATLANTA and the sink name is sat1. We check the routing table of node Atlanta table 4.1 (a) and know packets are sent to the local traffic sink. It is possible to have packet classes (XX, AT) with **Path(XX,AT)** = < XX, …, DA, AT>. These packet classes are sent from node DA through DA-AT to node AT. Node AT is the *destination* router and Rule 2 should be used. We can derive a node-path sub model for packet class (XX, AT) at node AT with the *upstream* router DA in Figure 4.8. Here XX is an unknown variable and it could be DA or something else.

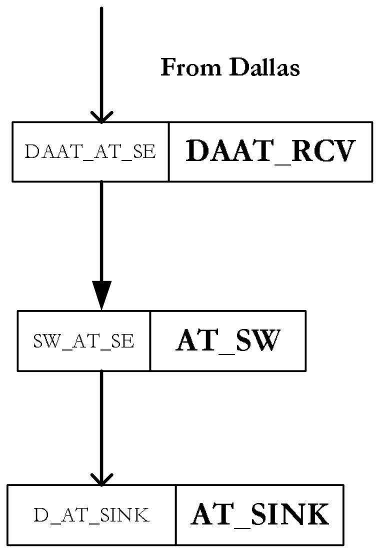


Figure 4.8 The node-path sub model for packet class (XX, AT) at node AT with the *upstream* router DA

For node Washington, there is only one entry with next hop ATLANTA in the routing table for node Washington in Table 4.1 (c) and the sink name is sat1 too. We can infer the packet classes (XX,AT) with **Path(XX,AT)** = < XX, …, WA, AT> as we did for node Dallas. These packet classes are sent from node WA through WA-AT to node AT. Rule 2 is used again. We can also derive the node-path sub model for packet class (XX, AT) at node AT with the *upstream* router WA in Figure 4.9. XX is an unknown variable and it could be WA or something else.

For the local generator, we investigate data traffic from the file *generatorinfo* and identify the packet classes. There are four packet classes: (AT, CH), (AT, DA), (AT, NY), and (AT, WA). The next hop from the routing table of node AT in Table 4.1 (a) provides a hint of the *downstream* router for each packet class. Node WA is the *downstream* router for packet classes (AT, CH), (AT, NY), and (AT, WA) and node DA is the *downstream* router for packet class (AT, DA). For each packet class node AT is the *source* router and rule 3 should be performed.

We can obtain a bunch of node-path sub models for packet class (AT, XX) at node AT with a *downstream* router YY ∈ **Neighbor(AT)** in Figure 4.10. XX is CH, DA, NY and WA respectively, and YY depends on the value of XX. We obtain node-path sub models: packet class (AT, CH) in Figure 4.10 (a), packet class (AT, DA) in Figure 4.10 (b) , packet class (AT, NY) in Figure 4.10 (c) , packet class (AT, WA) in Figure 4.10 (d).

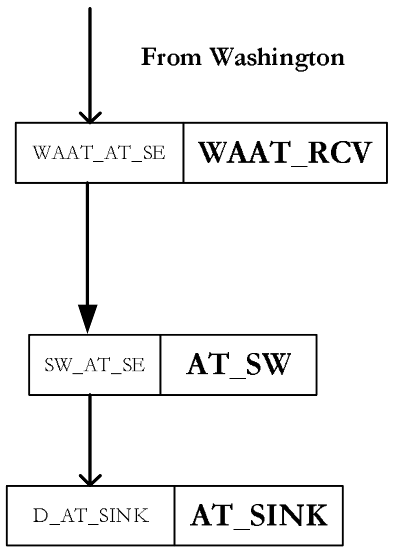
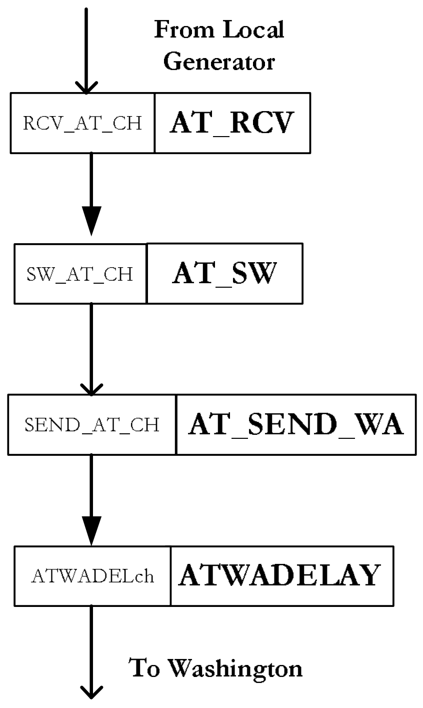
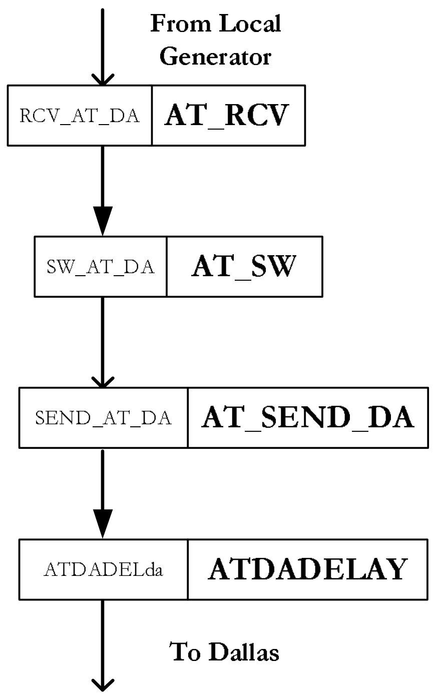


Figure 4.9 The node-path sub model for packet class (XX, AT) at node AT with the *upstream* router WA

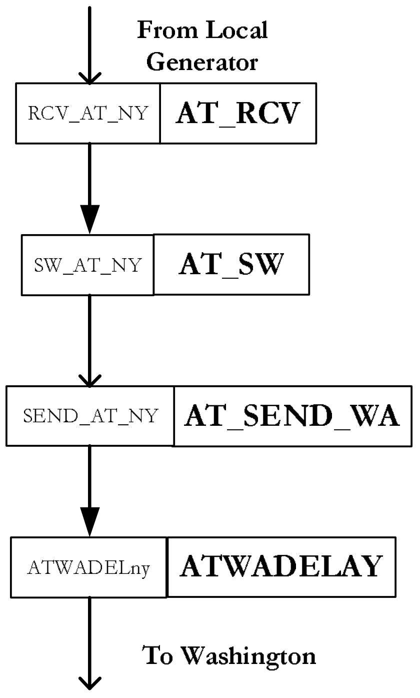
According to the current configuration, there is no packet class which defines node Atlanta as a *forwarding* router. If there was a packet class (XX, ZZ), there could be AT ∈ **ForwardR(XX, ZZ)** and **Path(XX,ZZ)** = < XX, …, WA, AT, DA, …, ZZ> or < XX, …, DA, AT, WA, …, ZZ>. For this packet class, the routing table in the *upstream* router had an entry with next hop AT for the *destination* router ZZ. The routing table in node AT has an entry with a next hop *downstream* router for *destination* router ZZ. We have the packet class and its *upstream* router and *downstream* router, thus rule 1 could be used for the corresponding node-path sub model. So far we have obtained all node-path sub models for node AT.



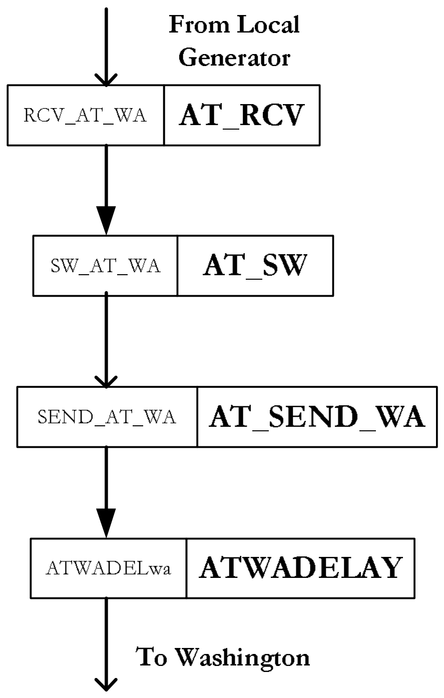
(a) The packet class (AT, CH) with the *downstream* router WA



(b) The packet class (AT, DA) with the *downstream* router DA



(c) The packet class (AT, NY) with the *downstream* router WA



(d) The packet class (AT, WA) with the *downstream* router WA

Figure 4.10 The node-path sub models for packet Classes from local generator in Atlanta

## 4.5 Assembling Node-Path Sub Models for a Node

Now we move to merge the node-path sub models so that we can acquire the node sub model. We still choose a node delegated by node Atlanta.

In section 3.4.1, we know the traffic generated by generator is sent to the destination with different proportions as well as the capacity of the generator defined in the configuration file. One user pseudo task with name "XXUserT" is introduced into the node sub model and its host process is "XXUserProc". The entry of User task with name "XXUser" can receive the call as the external arrival rate, which is equal to the capacity of generator $r_{XX}$ . It can send the request to the receiving task of the generator with different proportions and the proportion is $P_{XX,ZZ}$ for the packet class with destination ZZ. For node Atlanta, the arrival rate is $r_{AT}$ and the proportions are $P_{AT,CH}$, $P_{AT,DA}$, $P_{AT,NY}$, $P_{AT,NY}$.

We have associated the task with the hardware resources when we defined the task in section 4.4. Different resources separate the tasks and all the tasks for the different resources can not be merged any more. The type of call between entries in the task will not change in spite of the assembling and merging. Each entry in one scenario defined in subsection 4.3.1 reflects the responsibility of an operation for one packet class in a node. If the request is an asynchronous call to the entry, all requests should line in the queue. If the asynchronous requests are in the same queue, the entries that handle these requests should put all in one task.

We can define the compositional rules for assembling the node-path sub model as follow:

1. Location principle: If the entries share the same resource, all the entries for these tasks can be assembled onto one task. Note the number of entries will not change.

2. Similarity principle: If entries handle the packets with the same destination in the tasks, the entries can be merged in one entry of one task. Rule 1 location principle is a prerequisite. The currently processing packet in one node is the destination-oriented policy.
3. Inheritance principle: We notice the sequence of responsibilities in UCM is consistent with the entry sequence as the LQN model for each packet class. The entry sequence in the LQN model is defined as from receiving task to switching task, from switching task to sending/sink task, and from sending task to network delay task, and from the network delay task to the receiving task in next hop node. If the similarity principle is applied in the previous entry, the following entries can inherit the similarity principle from previous entry.

In node Atlanta, all requests (packet classes) from the generator are waiting in a queue located in the same incoming socket. The entries in the receiving task in Figure 4.10 (a) (b) (c) (d) share the hardware resource. Rule 1 location principle is deployed and we merge all entries of the receiving task for the generator in Figure 4.10 (a) (b) (c) (d) in one receiving task for the incoming socket for the local generator.

All packet classes pass through the switching/routing server. Whichever incoming socket packet classes come from, all the entries of switching tasks in Figure 4.8, 4.9, and 4.10 (a) (b) (c) (d) are combined together. Rule 1 location principle is employed here and we can derive the switching task for the node. In Figure 4.8 and 4.9 the switching tasks handle the packet class with the same destination. Rule 2 similarity principle is used here. Both entries are merged into one entry in the switching task.

For the sending and network delay tasks, the routing table indicates that packet classes with the destinations Chicago, New York, and Washington should go through the link ATWA and all these packet classes should wait in the same outgoing queue for the outgoing socket. By rule 1 location principle, we assemble the entries of the sending task and the network delay tasks in Figure 4.10 (a) (c) (d) into one sending and one network delay for the corresponding outgoing socket. The sending task and network delay task in Figure 4.10 (b) handle the packet class with the destination Dallas and can exist as an independent task in the node sub model for node Atlanta.

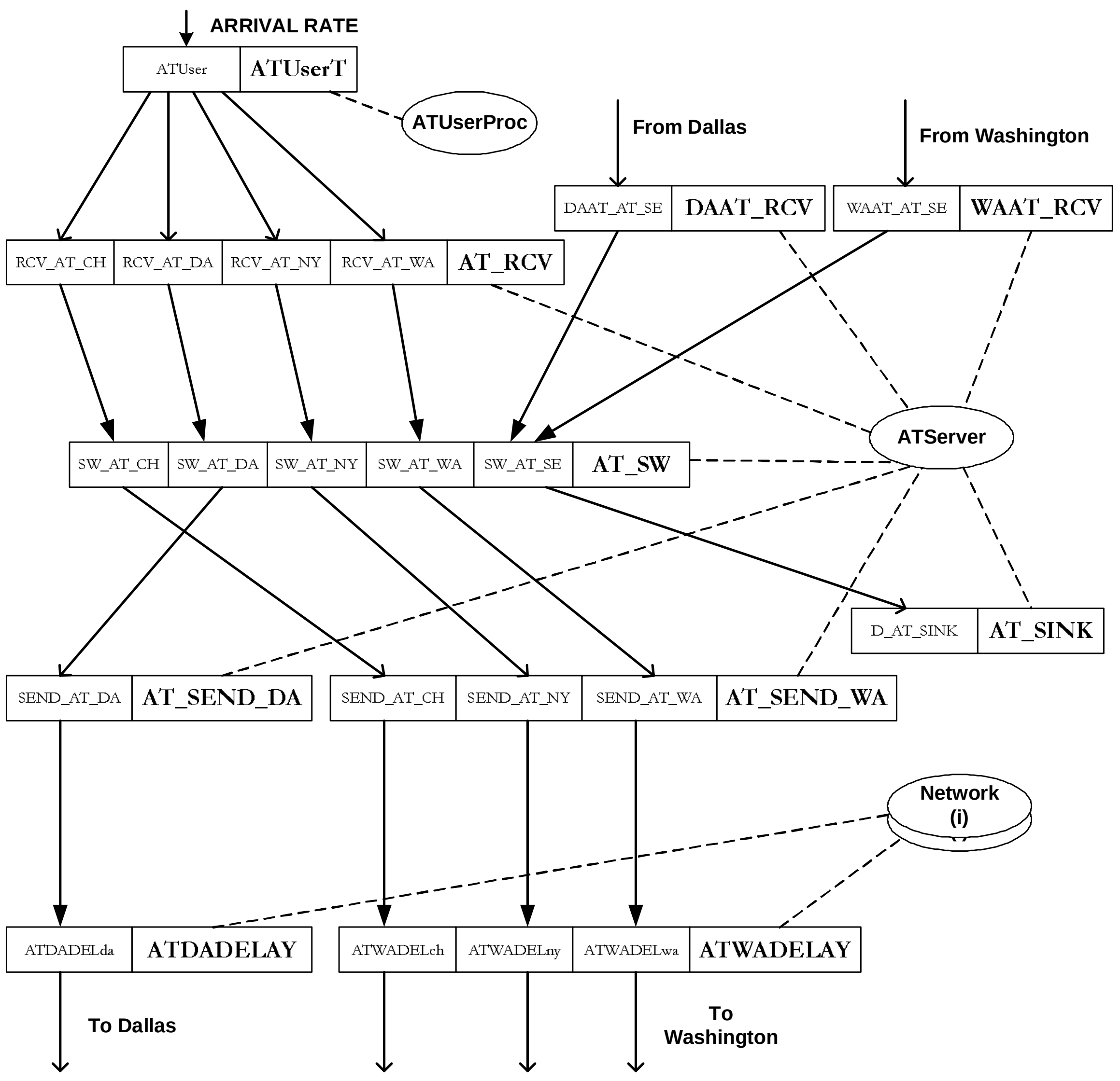


Figure 4.11 The structure of the node sub model for Atlanta

The sinking tasks in Figure 4.8 and Figure 4.9 deal with the operations that sink the packet classes in node Atlanta and should be combined in one task with one entry. Rule 3 Inheritance principle can be used here and the sinking task inherit the similarity in the switching task.

After the combining and merging of the node-path sub models for the scenarios in node Atlanta, we derive the node sub model for node AT, which is shown in detail in Figure 4.11. The host processor for each task depends on the physical entity that carries out the operations.

## 4.6 Composing node sub models into System Model

The behaviour of the network includes interactions between routers. In fact the scenarios of section 4.5 are the scenario fragments of the packet classes within one node. The assembling of node-path sub models for scenario fragments can be used to capture the interaction among the components within one node. The complete scenarios can describe all the behaviour of the packet class in the network to reflect performance characteristics for the network. This section focuses on the composition of the node sub model derived from scenarios within a node into a system model.

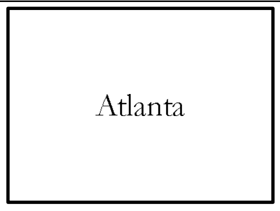
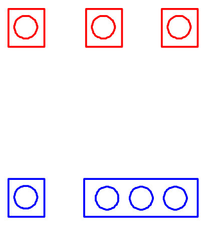
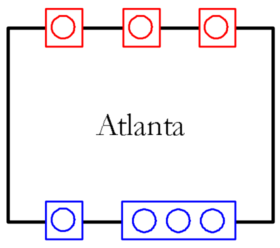
| | | |
|---|---|---|
|  |  |  |
| (a) Sub Model | (b) Interface | (c) Complete sub model |

Figure 4.12 The components for the high-level sub model

Building the sub model for node Atlanta based on the scenario fragment has been described in section 4.5 and we can repeat the procedures for the other four nodes: Chicago, Dallas, New York, and Washington. In Figure 4.12 (a), we use a box to express the sub model and the LQN notation for the sub model is put in the box.

Each sub model has defined some tasks that interact with other models. The communication is the request from one entry in the network delay task to the other entry in the other node. Each sub model also receives the arrival flow from the local generator. A circle in Figure 4.12(b) represents a packet class that can be grouped by link to stand for the interface. A rectangle in Figure 4.12(b) is the link's graphic expression in the components of the high-level sub model. But if the link is from local generator to node, we use a circle to represent entry in the User task. If this node receives the packet class, or packets from the local generator, we overlay the interface in the upper line of the sub model box. If this node sends the packet class to its neighbor node, we overlay the interface in the lower line of sub model box. Figure 4.12 (c) is an example of the complete expression of the high-level sub model of node Atlanta.

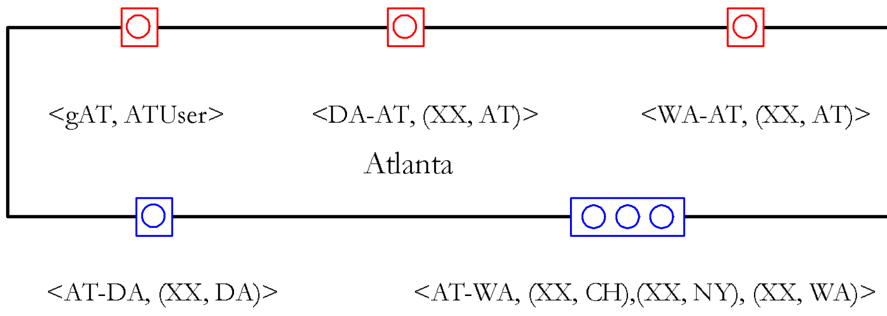


Figure 4.13 The labeled high-level sub model for node Atlanta

We use node AA, its *upstream* router BB, its *downstream* router CC and a packet class with its *destination* router DD to make further explanations for labeling the interface. We use XX to stand for the *source* router of the packet class. For node AA, the interface on the upper line of sub model box can be labeled as <BB-AA, (XX, DD)>, and the number of (XX, DD) can be one or more. We can label the interface on the lower line of sub model box as <AA-CC, (XX, DD)>, and the number of (XX, DD) can be more than one. The interface on the upper line of sub model box can be labeled as <gAA, AAUser>. Figure 4.13 shows the labeled high-level sub model for node Atlanta.

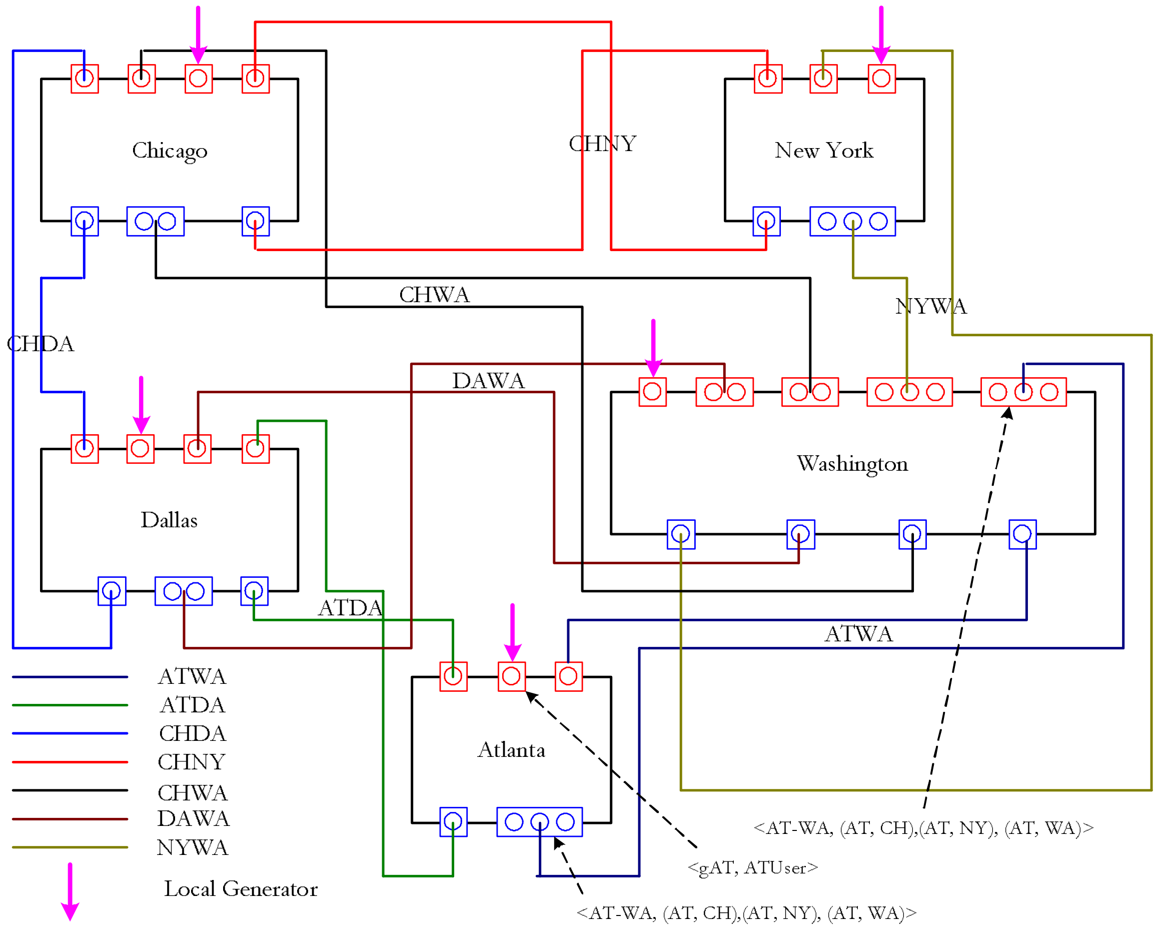


Figure 4.14 A high-level performance model for the network

We associate the calls defined in the LQN node sub model with the interface of the node component. For the incoming interface, we connect the call which is to the entry

"BBAA_AA_DD" in the receiving task BBAA_RCV with the interface <BB-AA, (XX, DD)>. We also connect open arrive calls for the generator which is to the entry AAUser in the user task "XXUsr" to the interface <gAA, AAUser>. For the outgoing interface, we can connect the call, which is from the entry "AACCDELdd" in the network delay task "AACCDELAY", to the interface <AA-CC, (XX, DD)>.

For each link, we connect the outgoing interface on the lower line of the sending node to the incoming interface on the upper line of the receiving node by packet class. Thus <AA-CC, (XX, DD)> in the lower line of node AA can be joined to <CC-AA, (XX, DD)> in the upper line of node CC. For instance, Figure 14 shows the connection from the outgoing interface <AT-WA, (AT, CH), (AT, NY), (AT, WA)> in node Atlanta to the incoming interface <AT-WA, (AT, CH), (AT, NY), (AT, WA)> in node Washington. The label of the open arrival call to the node Atlanta is also indicated in Figure 14. We obtain a high-level performance model for the network in Figure 14.

## 4.7 Generality of Compositional Strategy for Building Models

This section describes the generic compositional strategy derived from the model building practice in section 4.2, 4.3, 4.4, 4.5 and 4.6.

In general, in any computer system we begin by identifying the requests to traverse this system. We trace the request from its origin to its destination. The request may consume software and hardware resources. We treat the resources as servers. For simplicity, we can decompose the request into pieces and trace the path for each piece. If we can identify the path for each request (or request piece), we can then apply the compositional strategy for building the performance model. The compositional strategy is outlined as follows:

1. Define the scenario and trace the path for each request;
2. Decompose the scenarios into fragments within the component. Here a component is a generic object, for example a node in CGNet;
3. Map the scenario fragments into the component-path sub models in terms of entry, task, and call used in the LQN model. If the system is an Object-Oriented design, we can identify the alternatives of the scenario fragment of each class, then map all possible alternatives of the scenario fragment to the template sub model;
4. Assemble the component-path sub models to a component sub model for each component. Here we assemble the entries of the task for the resource. The assembling rules illustrated in section 4.5 are summarized as follows:
    I. Location principle: If the entries share the same resource, all the entries for these tasks can be assembled to one task.
    II. Similarity principle: If entries handle the requests that have the same responsibilities including the execution demand along the path after this point in the tasks, the entries can be merged into one entry to one task. The Location principle is a prerequisite.
    III. Inheritance principle: The responsibility in the scenario is corresponding to the entry in the model. The sequence of entries in the LQN model is the same as that of responsibilities in UCM. If the Similarity principle is applied to the entry in the previous task, the entries for the following responsibilities can inherit the Similarity principle.
5. Compose the component sub models into the system model. Following the path of the request piece traversing the different components, we can join the call of the incoming interface in one component to that of the outgoing interface in another. This gives the performance model for the system.
6. The assembling and composition in step 5 can, in principle, be applied recursively at multiple levels of component decomposition.

# Chapter 5 Data Collection and Measurement

In chapter 4 we have developed the structure of the LQN model. This chapter focuses on the techniques for data collection for various scenarios, measurement for the execution information of the system, and parameter estimation for the LQN model. We begin by presenting the experiment setup for measurement purposes. Section 5.2 shows the collection of necessary parameters for a complete performance model and how the LQN model is reproduced with those parameters. Section 5.3 provides the procedure for measurement with the "Displacement" technique. Section 5.4 discusses parameter estimation with the least square estimation and regression models are also presented. Section 5.5 describes the derivation of parameters for the simple four-node configuration. Section 5.6 discusses the parameters we derive from the five-node configuration with complicated topology.

## 5.1 Experiment Setup for Measurement

As stated above, we assumed in chapter 4 that we had the necessary data such as CPU demand of an entry and the call's intensity to create the performance models. To increase the accuracy of measurement, we ran the node components on separate machines.

For CGNet with its topology shown in Figure 4.1, we choose six SUN SPARCstation 2 machines as an experimental environment. The operating system on these machines is Sun Solaris 5.7. The NIC and Hub speeds are 10Mbps and all machines are in an isolated local area network. We run each node on one machine. All the generators and traffic sinks run on the other one because they only perform the simple task. Experiments indicate that all generators and traffic sinks can perform the job well as we expect from the configuration file.

## 5.2 Data Collection

In this section we present the necessary data collection so that we can complete the performance model. We can collect data from execution information and derive it from configuration.

From the current CGNet implementation, the average packet size is 85 bytes in the network. The maximum length of the outgoing queue is 78000 bytes. Thus the buffer size of the outgoing queue is 78000 /85 = 917 packets.

| Name | Speed (bps) | Node | Destination1 | | Destination2 | | Destination3 | | Destination4 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Dest. | Weight | Dest. | Weight | Dest. | Weight | Dest. | Weight |
| *gat1* | *67000* | *AT* | *NY* | *204* | *WA* | *178* | *CH* | *185* | *DA* | *103* |

Table 5.1 Definition of generator gAT in CGNet

From the configuration of CGNet we can obtain the information of generators and links, which affect the parameters such as external arrival rates and execution demand for entries of the network delay task in the performance model. Here we use the sample node Atlanta as before. Table 5.1 shows the information of the generator gAT. The information of links connected to Atlanta is shown in table 5.2.

| Name | Node1 | Node2 | Speed (bps) |
|---|---|---|---|
| *atda* | *AT* | *DA* | *75000* |
| *atwa* | *AT* | *DA* | *105000* |

Table 5.2 Definition of links AT-DA and AT-WA

Thus based on the previous definition in section 3.4.1 and table 5.1, we have

$r_{AT}$ = 67000bps =67000/(85*8)= 98.53 packets/sec.

$P_{AT,CH}$ = 185/(204+178+185+103) = 0.276

$P_{AT,DA}$ = 103/(204+178+185+103) = 0.154

$P_{AT,NY}$ = 204/(204+178+185+103) = 0.304

$P_{AT,WA}$ = 178/(204+178+185+103) = 0.266

We define network delay $d_{XX-YY}$ for a packet through link XX-YY. We have $d_{XX-YY}$ = packet size/Speed(XX-YY), so based on table 5.2, we obtain.

$d_{AT-DA}$ = 85 bytes /75000bps = (85*8)/75000 sec

$d_{AT-WA}$ = 85 bytes /105000bps = (85*8)/75000 sec

We define the CPU time for the actions, receiving, switching, sending and sinking a packet as follow:

$a_R$ = CPU time per packet received (from local generator or from other nodes)

$a_{SW}$ = CPU time per packet switched to outgoing queue

$a_{SD}$ = CPU time per packet sent to outgoing socket for next hop

$a_{SK}$ = CPU time per packet sent to outgoing socket for traffic sink

These are the entry parameters in the LQN model as well as the network delay $d_{XX-YY}$. In section 4.4.1, we have discussed the receiving task and know the receiving task is used in the LQN model only for the purpose of providing a separate buffer for an incoming socket. We set $a_R$ = 0 and migrate the amount of $a_R$ to $a_{SW}$ when we define the parameters in the

LQN model. We retain $a_R$ for clarity through the rest chapter but keep this fact in mind.

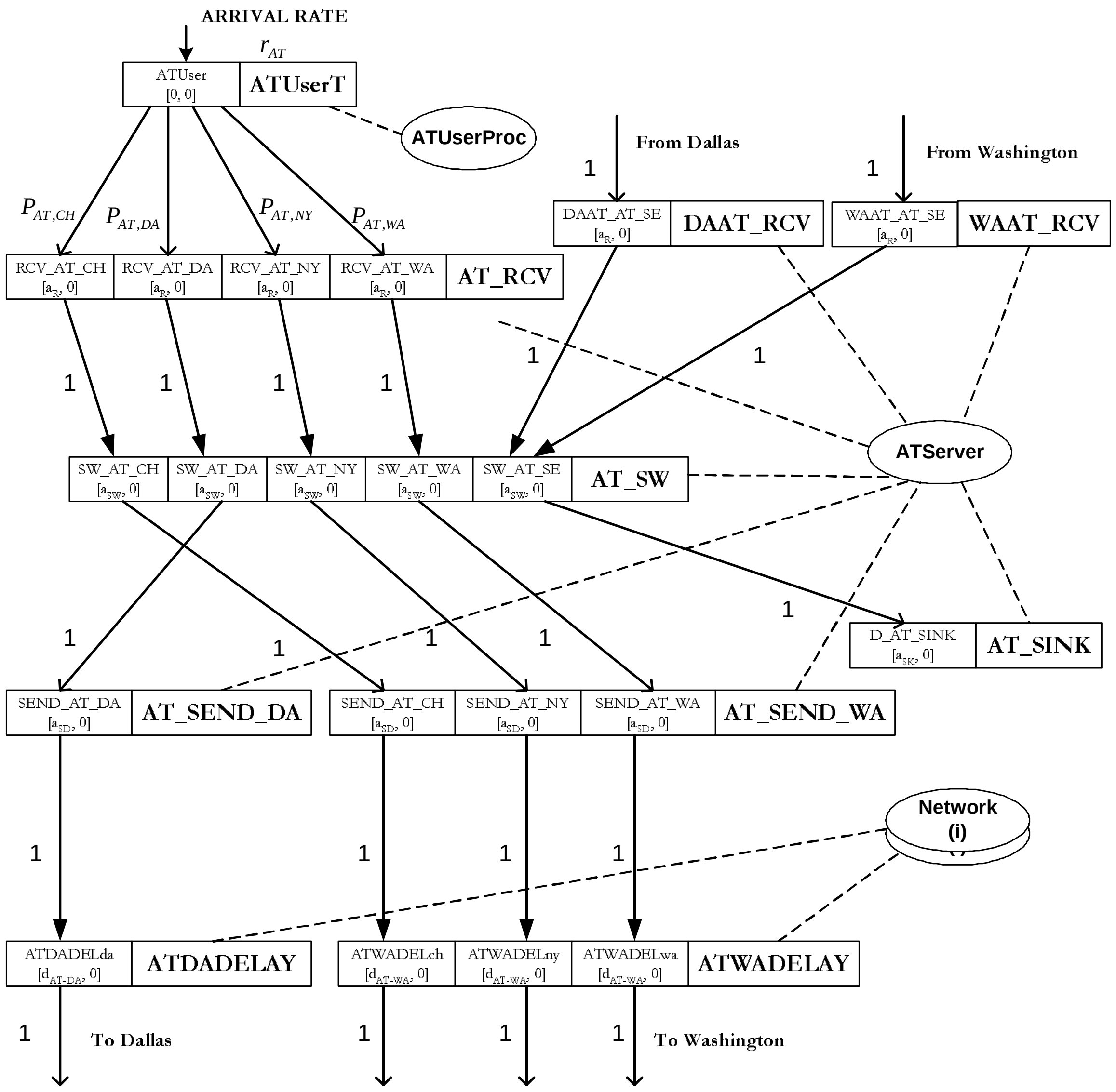


Figure 5.1 Node sub model for Atlanta with parameters

We also can infer from the CGNet implementation, that the number of each call from the entry in receiving tasks to the entry in switching task, from switching task to sending tasks and sink tasks, from sending task to network delay task, from network delay task to receiving task in the next hop is 1.

We collect and define the information for each node and insert them into the LQN model. The parameterized LQN sub model for node AT is shown in Figure 5.1.

## 5.3 Measurement

In this section we describe our efforts to acquire the parameters for the LQN model. Measurement with "Displacement" technique should be performed so that we can obtain the total CPU time of a process.

### 5.3.1 Motivation

We defined the $a_R$, $a_{SW}$, $a_{SD}$ and $a_{SK}$ in section 5.2 but did not offer the value or solution for them. Measurement is the only way to acquire them. All the parameters indicate the demands of the operations. There are two potential solutions to obtain them: 1) measure the operations directly, 2) repeat them to obtain the total time then divide for each. Normally if the CPU spends much time on one operation, both solutions are feasible. But for CGNet we can imagine how little the time the system spends on processing one packet. In addition within CGNet the receiving/switching is the main process, and the sending/sinking is performed by one thread for each outgoing queue. Also the overhead for operation switching is ignored in the first approach. The overhead for the environment in the second approach is also missed. All these overheads cannot be neglected compared to the time the system spends on one operation. Missing them could lead to an imprecise prediction of the performance model.

The solution in this thesis is to measure those parameters together. We let CGNet run for a period of 20 minutes to obtain the CPU time for each node. We can obtain the total CPU time, then split it to each $a_R$, $a_{SW}$, $a_{SD}$ and $a_{SK}$. Each of those parameters includes a portion of the overhead. How to split the total CPU time for each parameter will be discussed in section 5.4.

| Node name | *gprof* (sec) | *time* command (sec) | | |
|---|---|---|---|---|
| | CPU time | User CPU time | System CPU time | Total CPU time |
| Atlanta | 744.51 | 542.77 | 325.67 | 868.44 |
| Chicago | 1700.17 | 1257.97 | 744.25 | 2002.22 |
| Dallas | 1349.26 | 1001.76 | 575.04 | 1576.8 |
| New York | 1576.24 | 1209.22 | 632.75 | 1841.97 |
| Washington | 1819.16 | 1358.04 | 835.00 | 2193.04 |

Table 5.3 The comparison of CPU time from two profilers for one case

Profiler *gprof* is chosen to measure the C++ program of CGNet following profiling steps outlined in the manual [Fenlason97]. The time spent in each function or its subroutines can be obtained from the output. Another profiler, *time* command [Fink02], can show the CPU time as well as the elapsed time between invocation of the utility and its termination. We expect the consistent outputs of these profilers for the same process. Table 5.3 shows the comparison of results of the two profilers for one case. The gap between column 2 and column 5 is too large to give confidence in the values.

We can make the hypothesis that the time for interrupt service routine (ISR) execution is not correctly accounted for by both profilers. In fact each node performs reading packets from sockets and writing packets to sockets. Most parts of the tasks are executed by the ISR. The Paper [McCanne93] has well documented the fact an arbitrary amount of the CPU time has not been charged in the profiler. The paper [Woodside97] for "Displacement" technique deduces that it is difficult, even impossible, to know which process should be charged when the ISR is invoked for communication protocol applications. That drives us to adopt the "Displacement" technique in measurement.

## 5.3.2 Measurement with Displacement Technique

In this subsection we focus the measurement with "Displacement technique" for CGNet. The objective is to obtain the CPU time for each node when we run CGNet for each case. Although the *time* command does not perform well in accurate measurement, we can use it to roughly determine how long the node is executed and how much CPU time is used for the node.

The "Displacement" technique introduces a dummy process *compulmt*. The *compulmt* process is a CPU intensive process. It is introduced to fill the CPU idle time during the execution of CGNet, so that the CGNet node process can displace the execution time of the *compulmt* process. The *compulmt* process executes a fairly short loop over some arithmetic operations and the number of loops is programmable. In the beginning and end of the *compulmt* process, the "wall-clock" measurement is performed and the difference is the execution time of the *compulmt* process. The code of the *compulmt* process is shown in appendix B and we can change `ITERATIONS` to vary the duration of *compulmt* process execution.

To make sure the *compulmt* process does not affect the execution of CGNet, priority policy is applied when we run CGNet and the compulmt process one the same machine. The *nice* command [Frank98] is used to set priority for the process.

The procedure of measurement for the CPU cost for each node executable can be outlined as follows:

1. Run the CGNet executables with *time* command ahead. Record and estimate the CPU idle time for each node executable from results of the *time* command;
2. Choose the suitable *compulmt* process for each node executable and make sure the execution time of *compulmt* process can cover the CPU idle time. The execution time of *compulmt* process on quiet machine is $T_C(1)$.
3. Run the *compulmt* process first with lower priority followed immediately by the CGNet node executable with higher priority on the same machine. The two processes can keep the CPU busy. CGNet node executable can displace the time of the *compulmt* process by its high priority. The CGNet node executable yields to the *compulmt* process when it is idle. That priority policy of the scripts can be implemented as in figure 5.2.

```
nice +19 time ~pfwu/binmu5/compulmt &
cd ./nodeat
time ~pfwu/binmu5/node atlanta . $1 0 &
cd ..
```

Figure 5.2 An example script of executing the *compulmt* process and node executable

   When two processes have finished, we obtain the execution time of the *compulmt* process $T_C(2)$ .
4. Calculate the CPU time for the node process from the difference between two executions of the *compulmt* process as $T_C(2) - T_C(1)$.

We can obtain the CPU time of CGNet node process through this approach, instead of some instrumentation techniques that handle the communications operations poorly.

## 5.4 Parameter Estimation

Section 5.3 has provided the total CPU time for the node executable but not the host processing demand for each entry or functionality. We try to break the CPU time into piece and estimate the $a_R$, $a_{SW}$, $a_{SD}$ and $a_{SK}$. The Least Square Estimation techniques (see [Follenweider93] or any text on regression in statistics) can be used for this purpose.

We can make an acceptable assumption that $a_R$, $a_{SW}$, $a_{SD}$ and $a_{SK}$ are consistent and repeatable in the test while the environment where the node executables run is not changed. We use TP to denote the total CPU time for a run, then

$$\text{TP} = n_R * ( a_R + a_{SW} ) + n_{SD} * a_{SD} + n_{SK} * a_{SK} \tag{5-1}$$

The sending thread and sinking thread share the same code in CGNet. The difference is the sending thread emulates the network delay with the select statement with timeout but the sink thread does not. We can assume that the difference is small, so $a_{SD} = a_{SK}$. The equation (5-1) yields

$$\begin{aligned} \text{TP} &= n_R *( a_R + a_{SW} ) + (n_{SD} + n_{SK} ) * a_{SD} \\ &= n_R *( a_R + a_{SW} ) + n_S * a_{SD} \end{aligned} \tag{5-2}$$

In the case where there is no packet loss in the outgoing queues, we have $n_R = n_S$. We obtain from the equation (5-2):

$$\text{TP} = n_S *( a_R + a_{SW} + a_{SD} ) \tag{5-3}$$

If there is packet loss with losses $n_L$, we have $n_R = n_S + n_L$. It is assumed that lost packets are only processed through the receiving and switching steps. We obtain from the equation

(5-2):

$$TP = n_S * (a_R + a_{SW} + a_{SD}) + n_L * (a_R + a_{SW}) \quad (5\text{-}4)$$

For simplicity we define $a_P = a_R + a_{SW} + a_{SD}$ and $a_{RSW} = a_R + a_{SW}$. Substituting them yields

For case without packet loss: $TP = n_S * a_P$ (5-5)

For case with packet loss: $TP = n_S * a_P + n_L * a_{RSW}$ (5-6)

From equation (5-5) and (5-6) we can hypothesize a simple regression model in (5-7) and a multiple regression model in (5-8) to model the relationships between the total CPU time and the number of packets handled by CGNet,

$$Y = \beta_1 x_1 \quad (5\text{-}7)$$

$$Y = \beta_1 x_1 + \beta_2 x_2 \quad (5\text{-}8)$$

One method, the least square approach, chooses the estimators $\hat{\beta}_1$ and $\hat{\beta}_2$ that minimize the sum of squared errors (SSE). Here we have Y is TP, $x_1$ is $n_S$ and $x_2$ is $n_L$. Further more the equation (5-8) can cover the case without packet loss if $x_2$ is set to 0.

The original CGNet has provided the nominal rates for the generators. We can introduce a rate *multiplier* so as to change rates, to vary the workload of CGNet for each case. We have Rate = *multiplier* * nominal rate

We collect the sample data TP, $n_S$ and $n_L$ by changing the *multiplier*. Then we use them to estimate the unknown parameters in the regression model equations (5-7) and (5-8). The regression analysis procedures have been outlined in [Scheaffer86] and the statistical tool

SAS [Cary99] is used in the research.

From estimates $\hat{\beta}_1$ and $\hat{\beta}_2$ we obtain $a_P$ = $\hat{\beta}_1$ = $a_R$ + $a_{SW}$ + $a_{SD}$ and $a_{RSW}$ = $\hat{\beta}_2$ = $a_R$ + $a_{SW}$, so $a_{SD}$ = $\hat{\beta}_1$ - $\hat{\beta}_2$, $a_R$ + $a_{SW}$ = $\hat{\beta}_2$. In this section we assumed $a_{SD}$ = $a_{SK}$ and in section 5.2 we have $a_R$ = 0. In summary, we have $a_R$ =0, $a_{SW} = \hat{\beta}_2$, $a_{SD}$ = $a_{SK}$ $= \hat{\beta}_1$ - $\hat{\beta}_2$.

## 5.5 CPU Cost for a Simple Four-Node Configuration

We have an LQN model with two parameters, $a_{SW}$ and $a_{SD}$, and two regression equations (5-7) and (5-8) for $a_P$ and $a_{SW}$. A great number of measurements were taken and the results were difficult to interpret. A simple configuration with four nodes and one packet class was examined to clarify the issues.

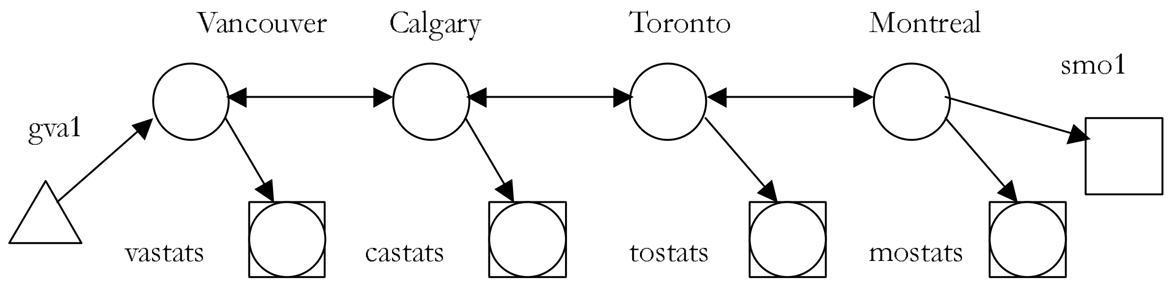


Figure 5.3 Simple four-node configuration of CGNet

In the four-node configuration, there are four nodes: Vancouver, Calgary, Toronto, and Montreal. Only one packet class generated in gVA traverses through the four nodes and sinks in sMO. The topology of the network is shown in Figure 5.3. The speed of the generator gVA is 100000bps and link speed is 105000bps. All the data traffic is one-way traffic in this linear configuration. The configuration avoids the switching overhead between the sending/sink threads. We run each node on a machine and the generator, traffic sink and statistics sink on the fifth machine.

| **TP (secs)** | $n_S$ **(packets)** | $n_L$ **(packets)** |
|---|---|---|
| 91.66745114 | 34424 | 0 |
| 98.62636304 | 34582 | 0 |
| 96.60449505 | 34582 | 0 |
| 76.53196609 | 34554 | 0 |
| 83.91038609 | 34559 | 0 |
| 80.26346898 | 34560 | 0 |
| 132.8928231 | 63577 | 0 |
| 146.5299519 | 63578 | 0 |
| 133.910804 | 63887 | 0 |
| 157.6920601 | 79478 | 0 |
| 166.6995101 | 77641 | 0 |
| 164.395846 | 77638 | 0 |
| 207.2725751 | 97907 | 0 |
| 203.719165 | 97914 | 0 |
| 204.2064589 | 98358 | 0 |
| 210.2447071 | 108119 | 0 |
| 210.771836 | 108128 | 0 |
| 213.553525 | 108486 | 0 |
| 243.4811021 | 126957 | 0 |
| 242.248595 | 126966 | 0 |
| 239.770828 | 127558 | 0 |
| 269.7792711 | 146738 | 0 |
| 267.153322 | 146747 | 0 |
| 268.1924909 | 147434 | 0 |
| 280.4792662 | 158891 | 0 |
| 275.229635 | 158900 | 0 |
| 280.879622 | 159345 | 0 |

(a) the light workload without packet loss

| TP (secs) | $n_S$ (packets) | $n_L$ (packets) |
|---|---|---|
| 276.945847 | 166060 | 7425 |
| 277.0247301 | 161368 | 22607 |
| 281.7895311 | 157021 | 36966 |
| 274.7519361 | 147570 | 48670 |
| 299.7562981 | 152192 | 64696 |
| 309.827373 | 152387 | 79051 |
| 342.3031641 | 161439 | 99396 |
| 346.7120831 | 161504 | 114808 |
| 362.4453422 | 161780 | 130810 |
| 388.2538232 | 166096 | 149881 |

(b) the heavy workload with packet loss

Table 5.4 Collection of sample data for four-node configuration

With the speed of generator increasing, we perform measurements for the CPU time for each node as in section 5.3 and collect the sample data to estimate parameters as in section 5.4. We tabulate the sample data in table 5.4 and separate them into 5.4 (a) for the light-load without packet loss cases and 5.4 (b) for the heavy load with packet loss.

| Variable | Parameter Estimate | Standard Error | 95% Confidence Limits | |
|---|---|---|---|---|
| $\hat{\beta}_1$ | 0.00197 | 0.00003573 | 0.00190 | 0.00205 |
| $\hat{\beta}_2$ | 0.00022815 | 0.00014582 | -0.00006276 | 0.00051905 |

**Root MSE:** 37.49628

Table 5.5 output for all cases for four-node configuration

We use a multiple regression model in equation (5-8) for all data in table 5.4. Part of the results on estimated value and root mean squared error (MSE) from the SAS multiple regression routine for the CPU time is reproduced in Table 5.5. From $\hat{\beta}_1$ and $\hat{\beta}_2$ in table 5.5, we know that lost packets do not incur much cost and most efforts of CPU in handling a packet is in sending, not receiving a packet (derived $a_{SW}$ << derived $a_{SD}$).

| Variable | Parameter Estimate | Standard Error | 95% Confidence Limits | |
|---|---|---|---|---|
| $\hat{\beta}_1$ | 0.00200 | 0.00003297 | 0.00192 | 0.00207 |

**Root MSE:** 38.34787

Table 5.6 output for light workload for four-node configuration

To see if there is any difference, we consider the light-load cases without loss in table 5.4 (a) and the heavy-load cases with losses in table 5.4 (b) separately. Table 5.6 shows part of the output from the SAS simple regression routine for the light load for four-node configuration. Part of the output from the SAS multiple regression routine for the heavy load for four-node configuration is reproduced in table 5.7.

| Variable | Parameter Estimate | Standard Error | 95% Confidence Limits | |
|---|---|---|---|---|
| $\hat{\beta}_1$ | 0.00162 | 1.11E-05 | 0.00159 | 0.00164 |
| $\hat{\beta}_2$ | 0.000781 | 2E-05 | 0.000735 | 0.00082709 |

**Root MSE:** 2.81795

Table 5.7 output for heavy workload for four-node configuration

$\hat{\beta}_1$ in table 5.5 and in table 5.6 are very close. The confidence intervals for $\hat{\beta}_1$ in table 5.5, table 5.6 and 5.7 are around $\pm 4\%$. But we see $\hat{\beta}_2$, the regression results are not as good as $\hat{\beta}_1$, especially in table 5.5. The lower boundary of the confidence limits is negative. The conclusion from table 5.5 is also not acceptable. The receiving part is reading a packet from the socket and switching it. The sending part is dequeuing it and writing to the socket. The difference between them, we infer, is not so big as table 5.5 described. We may notice the

Root MSE of table 5.7 is the smallest one among three tables. Thus we choose the parameters $a_{SW}$ and $a_{SD}$ from $\hat{\beta}_1$ and $\hat{\beta}_2$ in table 5.7.

The regression theory assumes the variance of measurement errors is constant (Assumption of homoscedasticity [Mendenhall81]). We may violate the equal variances assumption and that may affect the accuracy of the prediction. That would be a reason for us to take $\hat{\beta}_1$ and $\hat{\beta}_2$ of table 5.7 only. Hence we choose $a_{SW}$ and $a_{SD}$ that $a_{SW}$ = 0.000781sec, $a_{SD}$ = 0.000839sec for four node configuration.

## 5.6 Obtaining the Parameters for the Five-Node Configuration

The case study configuration is more complex than the simple one mentioned in section 5.5 and the topology is shown in Figure 4.1. There are more generators, traffic sinks, links, and more sending/sinking threads within one node than the simple four-node configuration.

We perform the procedure of measurement described in section 5.3 and estimation shown in section 5.4. Table 5.8 presents a collection of sample data for the five-node configuration. We did a regression analysis as was done in section 5.5 and tabulated the results of the output from SAS in table 5.9 for all cases, table 5.10 for no packet loss case, and table 5.11 for case with packet losses.

From table 5.9 the multiple regression model equation (5-8) gave the negative value prediction for $\hat{\beta}_2$. The confident limits almost reach zero. The reason for the problem could be that the model cannot capture everything properly or the overhead is important. The cost of the thread switching may drop when there are packet losses. We may assume $\hat{\beta}_2$ as zero and give the parameters of the model.

| **TP (secs)** | $n_S$ **(packets)** | $n_L$ **(packets)** |
|---:|---:|---:|
| 91.12765 | 22430 | 0 |
| 113.113 | 31050 | 0 |
| 149.31465 | 43914 | 0 |
| 175.37086 | 51897 | 0 |
| 219.31372 | 64011 | 0 |
| 299.64372 | 85237 | 0 |
| 375.59602 | 113009 | 0 |
| 155.39386 | 43261 | 0 |
| 218.89731 | 59725 | 0 |
| 310.10803 | 83883 | 0 |
| 364.29541 | 99217 | 0 |
| 421.62298 | 122198 | 0 |
| 549.46126 | 162499 | 0 |
| 730.53637 | 220080 | 0 |
| 901.07699 | 290276 | 0 |
| 121.2862 | 32729 | 0 |
| 161.5628 | 44885 | 0 |
| 244.48669 | 63302 | 0 |
| 279.42707 | 74777 | 0 |
| 348.78466 | 92007 | 0 |
| 441.84533 | 122603 | 0 |
| 550.05091 | 162129 | 0 |
| 160.23081 | 48635 | 0 |
| 238.5233 | 67355 | 0 |
| 329.3871 | 95013 | 0 |
| 390.25624 | 112740 | 0 |
| 450.44263 | 138880 | 0 |
| 583.39743 | 185277 | 0 |
| 261.90265 | 65218 | 0 |
| 371.31325 | 90192 | 0 |
| 493.8099 | 127277 | 0 |
| 570.07184 | 150824 | 0 |
| 661.5285 | 186111 | 0 |
| 854.09688 | 248026 | 0 |

(a) the light workload without packet loss

| TP (secs) | $n_S$ (packets) | $n_L$ (packets) |
|---|---|---|
| 368.90198 | 137510 | 2607 |
| 428.99966 | 159523 | 20753 |
| 458.93117 | 161261 | 38735 |
| 469.97419 | 153657 | 51964 |
| 530.99505 | 178503 | 75851 |
| 587.42672 | 197404 | 96949 |
| 673.61537 | 212969 | 122641 |
| 1027.0594 | 371497 | 11416 |
| 1095.5077 | 407260 | 26431 |
| 608.95458 | 202723 | 559 |
| 695.68501 | 248365 | 17404 |
| 765.67243 | 267607 | 35027 |
| 718.54679 | 243736 | 48001 |
| 831.59089 | 288781 | 70444 |
| 933.25368 | 323871 | 90729 |
| 1036.8922 | 357500 | 116137 |
| 633.86122 | 235106 | 11974 |
| 747.99291 | 263602 | 56698 |
| 918.26021 | 318420 | 95021 |
| 973.92031 | 337178 | 125424 |
| 892.36915 | 297714 | 144955 |
| 986.79767 | 343384 | 185388 |
| 1075.0217 | 370967 | 207307 |

(b) the heavy workload with packet loss

Table 5.8 Collection of sample data for five-node configuration

| Variable | Parameter Estimate | Standard Error | 95% Confidence Limits | |
|---|---|---|---|---|
| $\hat{\beta}_1$ | 0.00306 | 0.00003498 | 0.00299 | 0.00313 |
| $\hat{\beta}_2$ | -0.00038554 | 0.00014389 | -0.00067266 | -0.00009843 |

**Root MSE:** 58.09625

Table 5.9 output for all cases for five-node configuration

Here we also could investigate the data without packet loss and with packet losses. In table 5.11 $a_{SW}$ ($\hat{\beta}_2$) is still small as in table 5.5 and $a_{SD}$ ($\hat{\beta}_1$-$\hat{\beta}_2$) is bigger than in other cases. It is puzzling to us. This is possibly due to more thread switching in the sending process when there are multiple destinations in a node. We notice $\hat{\beta}_2$ is not very significant and its confidence interval is around $\pm 60\%$. We can choose the parameter value of zero for $\hat{\beta}_2$ or a non- zero value, 0.00028019. Choosing zero is for simplicity, but the evidence is not strong. Table 5.11 and table 5.9 shows that the LQN model with 2 or more sending tasks would have different parameters for $a_{SD}$ compared with table 5.5 and 5.7. The difference can be accounted for increased thread switching.

| **Variable** | **Parameter Estimate** | **Standard Error** | **95% Confidence Limits** | |
|---|---|---|---|---|
| $\hat{\beta}_1$ | 0.00343 | 0.00003799 | 0.00335 | 0.00351 |

**Root MSE:** 29.65452

Table 5.10 output for light workload for five-node configuration

| **Variable** | **Parameter Estimate** | **Standard Error** | **95% Confidence Limits** | |
|---|---|---|---|---|
| $\hat{\beta}_1$ | 0.00278 | 0.00003320 | 0.00271 | 0.00284 |
| $\hat{\beta}_2$ | 0.00028019 | 0.00009301 | 0.00008731 | 0.00047308 |

**Root MSE:** 24.89353

Table 5.11 output for heavy workload for five-node configuration

The values of $\hat{\beta}_1$ in table 5.9, 5.10, 5.11 are higher than those in table 5.5, 5.6, 5.7. We think the reason is overhead. Part of the overhead comes from thread switching. Here we infer the possible overhead including thread switching in a little detail as follows:

- Thread switching: The operating system is forced to reload cache; it will poll the available thread when one sending thread can be blocked for emulating network delay. The original configuration requires running more sending/sinking threads for each outgoing queue than the simple configuration.

- Socket polling: The main process should poll among the incoming sockets after it has enqueued one packet. There are more incoming sockets and the main process should monitor and poll among the larger set of incoming sockets of the five-node configuration than of the four-node configuration.
- Vain socket reading: Once the main process detects the packet available in one socket, it will read the incoming socket one by one. For original configuration, there could be more chances to read a socket but no data within the socket. But for the simple configuration, this problem doesn't exit because there is only one incoming socket.
- Routing table and related expense of CGNet: In fact both configurations are very simple. For a five-node configuration, there are five entries in the routing table. But there is only one entry in the routing table of a simple configuration. Each node spends more time to keep the link alive because there are more links each node connects with than the simple configuration.

We do not know the exact amount of overhead for each task and propose to distribute them evenly to the parameters. Table 5.10 is chosen as a source in this thesis for parameters and we assume the ratio of $a_{SW} / a_{SD}$ is the same as that in section 5.5 after overhead distribution. So $\hat{\beta}_1$ is 0.00343 sec and $a_{SW} / a_{SD} = \frac{0.000781}{0.000839}$. After calculations, we have $a_R = 0$, $a_{SW} = 0.0016536$ sec, $a_{SD} = 0.0017764$ sec, and $a_{SK} = 0.0017764$ sec. The complete LQN model for CGNet with these parameters is shown in appendix C.

# Chapter 6 Solving and Validating the Performance Model

In this chapter we solve the model with the parameters derived in the chapter 5 so that we can make the validation of the performance model. The validation is on three aspects: utilization, throughput, and packet loss. We begin by the description of the experiment of the real system in section 6.1. Measurement procedures should be performed as in chapter 5 to obtain the CPU time so that we can derive the utilization of each node. Section 6.2 validates the performance model in three aspects of performance characteristics: throughput, utilization, and packet loss and present discussions on them. Section 6.3 summarizes the validation.

## 6.1 Description of Experiments for Validation

The case study is CGNet for a five-node network with links between them. Each node has one generator, one traffic sink and one statistics sink connected. We chose six SUN SPARCstation 2 machines as the experimental environment. The operating system on these machines is Sun Solaris 5.7. The NIC and Hub speeds are 10Mbps and all machines are in an isolated local area network. Each node ran on its own machine and all the generators, traffic sinks and traffic statistics sinks were executed on the sixth machine.

We performed the measurement procedure described in chapter 5 for each node on each machine so we could obtain the CPU time for each node. A bunch of the *compulmt* processes with different execution time had been prepared before we performed measurement for validation purpose. Thus it is convenient to pick up the suitable *compulmt* processes with different execution time. The experiments were setup for different workloads. We used the *multiplier* to vary the workload of CGNet and carried out the observations on the performance characteristics of CGNet for each case.

## 6.2 Validation of the model

Validation is the most critical step to gain confidence of the model's correctness. Comparing the predicted performance characteristics with the real measurement is the best validation, although some performance characteristics are difficult to obtain. The measurement approach with "Displacement" technique for the CPU time of node executable can be employed in the validation stage of the performance model.

Plenty of experiments with different workloads have been made and some performance characteristics of the system were collected. We chose the workload in the original configuration of CGNet and varied it by *multiplier*. The comparisons between the prediction from model and the measurement of the real system were made on the following three aspects: throughput, utilization, and packet loss. The detailed comparison and discussions have been broken into three subsections: subsection 6.2.1 focuses on utilization for each node; subsection 6.2.2 is on throughput for each node; and packet loss discussion is shown in subsection 6.2.3.

We use SPEX to solve the performance model and the solver in spex input file is *parasrvn*
We controlled the external arrival rate of each node by the *multiplier* to solve the model in the format of SPEX (shown in appendix C). The summary of the predicted utilization and throughput from the LQN model was collected from the file generated from SPEX and reproduced in figure 6.1 and figure 6.2.

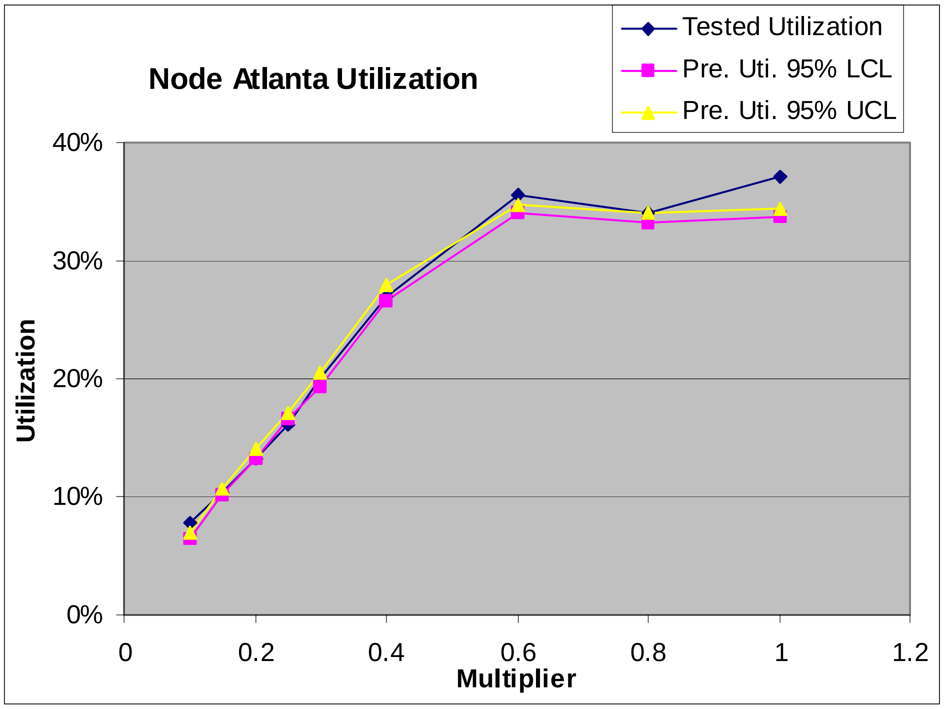
### 6.2.1 Utilization Validation

The utilization of each node well indicates the situation of the node with the current configuration. We chose *multiplier*s 0.1, 0.15, 0.2, 0.25, 0.3, 0.4, 0.6, 0.8, and 1.0 in adjustment of the speed of the generator.

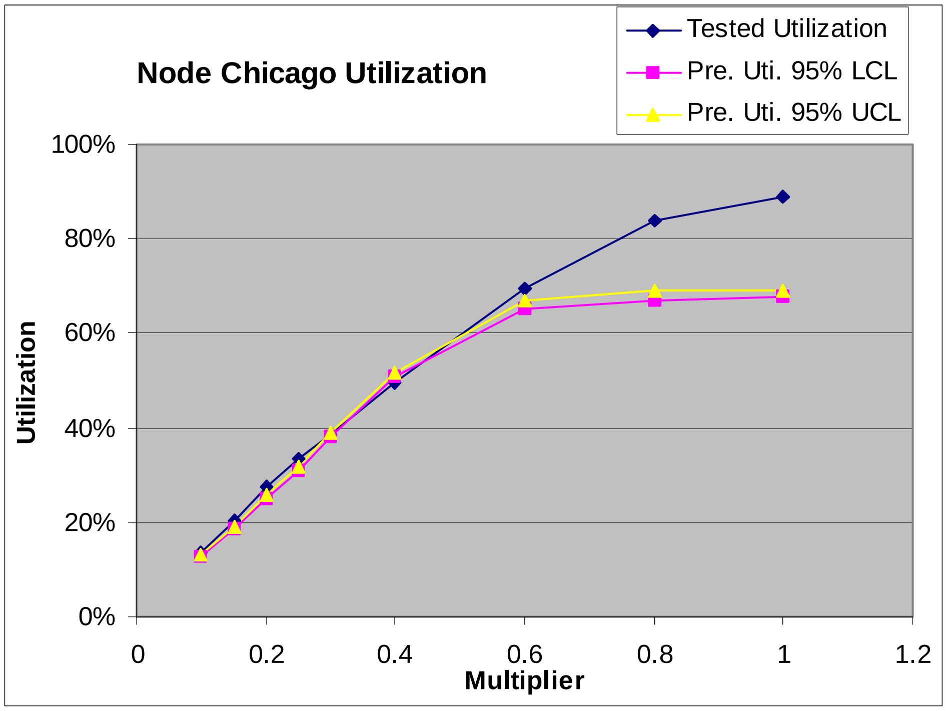
The *time* command is used to obtain the execution time $T_{XX}$ of node XX. At the same time we employed the measurement approach of the CPU time proposed in section 5.3 to obtain the CPU time TP for each nodeXX. The utilization can be calculated by TP over $T_{XX}$. The predicted utilization for each node can be obtained by the utilization of the corresponding host processor, which includes all tasks' utilization running on the host processor.

The predicted utilization of the simulation technology in the confidence interval with confidence coefficient 95% compared to the measured utilization for each node is plotted in the graphs on Figure 6.1 against *multiplier* for the speeds of generators in the network within the range from 0.1 to 1. Figure 6.1 (a), (b), (c), (d) and (e) are for node Atlanta, Chicago, Dallas, New York, and Washington respectively. For each node we find the curve of the predicted utilization is close to the curve of the measured utilization within the range from 0.1 to 0.6. The LQN model provides consistent predictions in the utilization. When the *multiplier* for the speeds of the generator exceeds 0.6, the node Washington was saturated. The predicted utilization and measured utilization were inconsistent. The trend is that the measured utilization can not be predicted by LQN model when the *multiplier* is greater than 0.6.

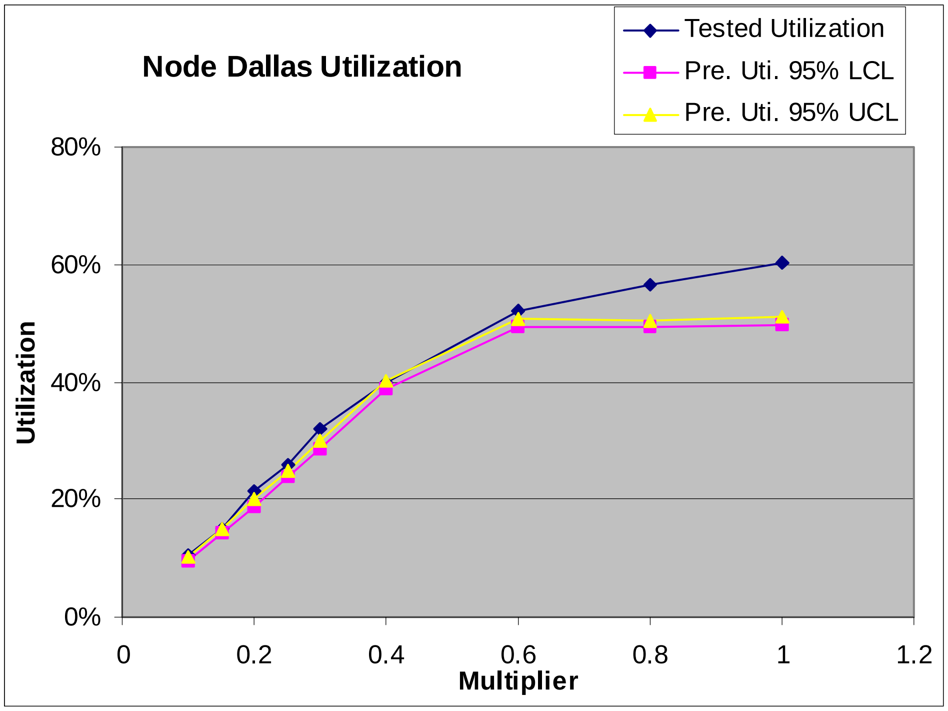
The performance characteristics of the Sun Solaris System in time-sharing mode, the goal to achieve high throughput, are demonstrated in the measured cases where the *multiplier* is greater than 0.6. The extra CPU time, compared to the prediction, can be explained as the system's efforts in time-sharing mode to achieve the high throughput. But it is different for the prediction from model. If one node is saturated, the utilization of other nodes will not increase any more.



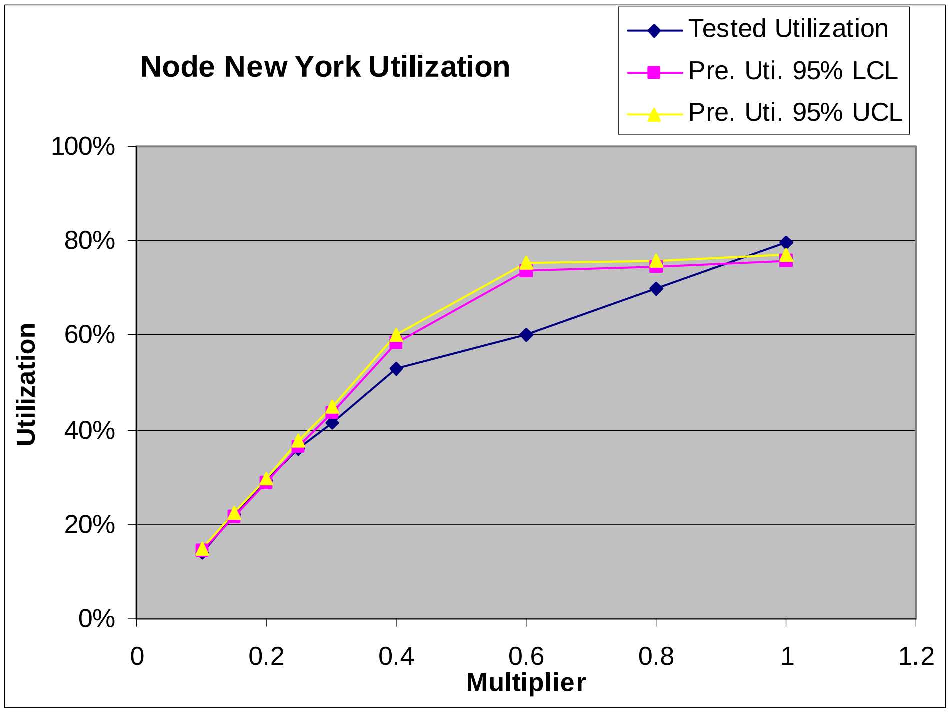
(a) Utilization of Atlanta for prediction and measurement against *multiplier* of workload



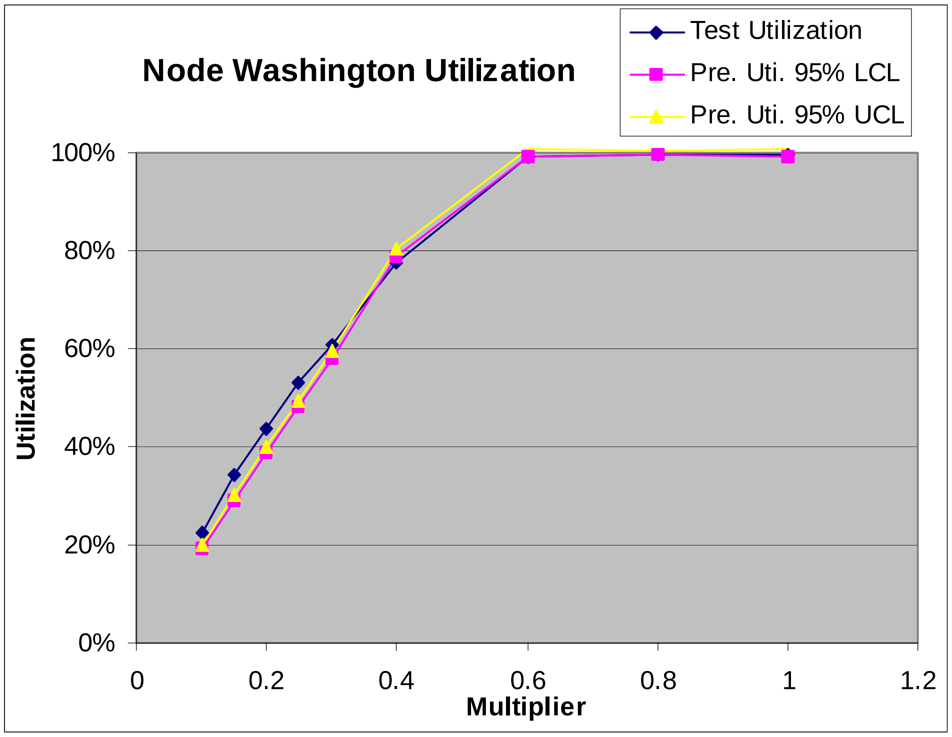
(b) Utilization of Chicago for prediction and measurement against *multiplier* of workload



(c) Utilization of Dallas for prediction and measurement against *multiplier* of workload



(d) Utilization of New York for prediction and measurement against *multiplier* of workload

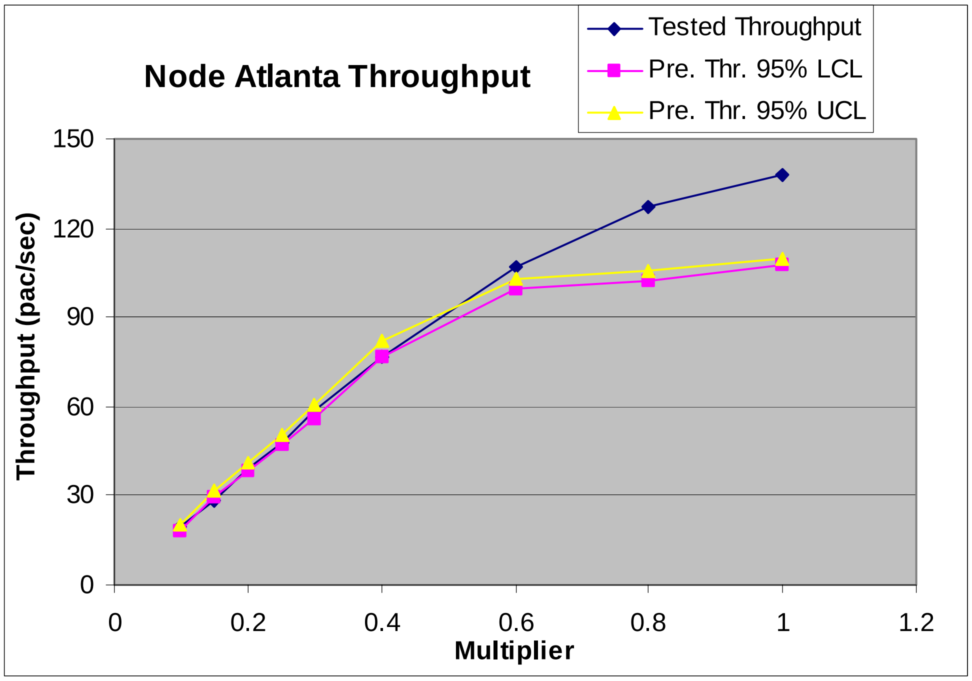


(e) Utilization of Washington for prediction and measurement against *multiplier* of workload

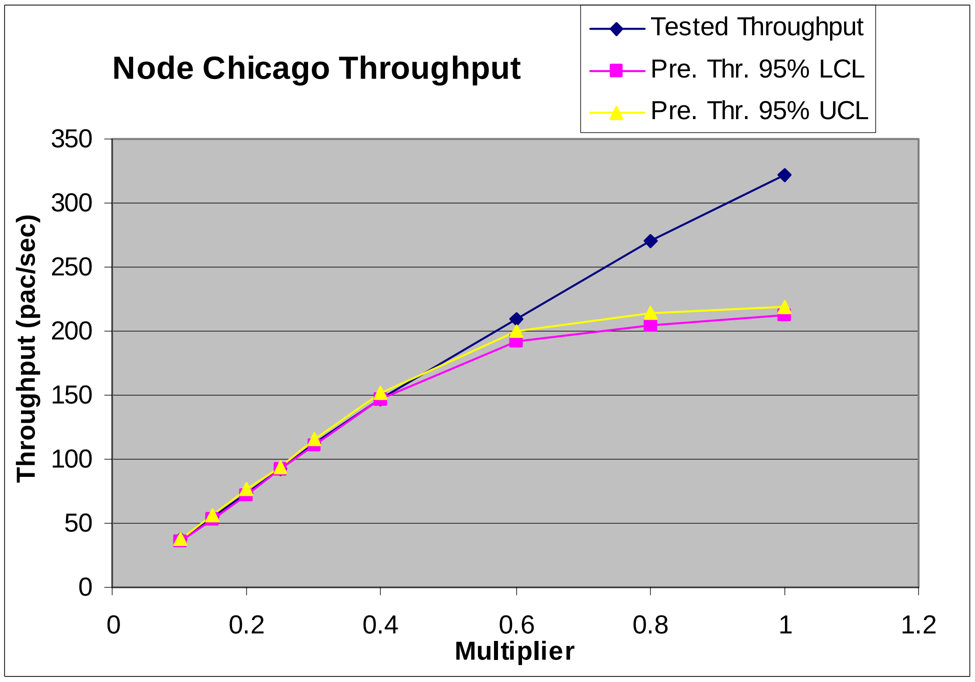
Figure 6.1 Comparison between predicted and measured utilization for each node

### 6.2.2 Throughput Validation

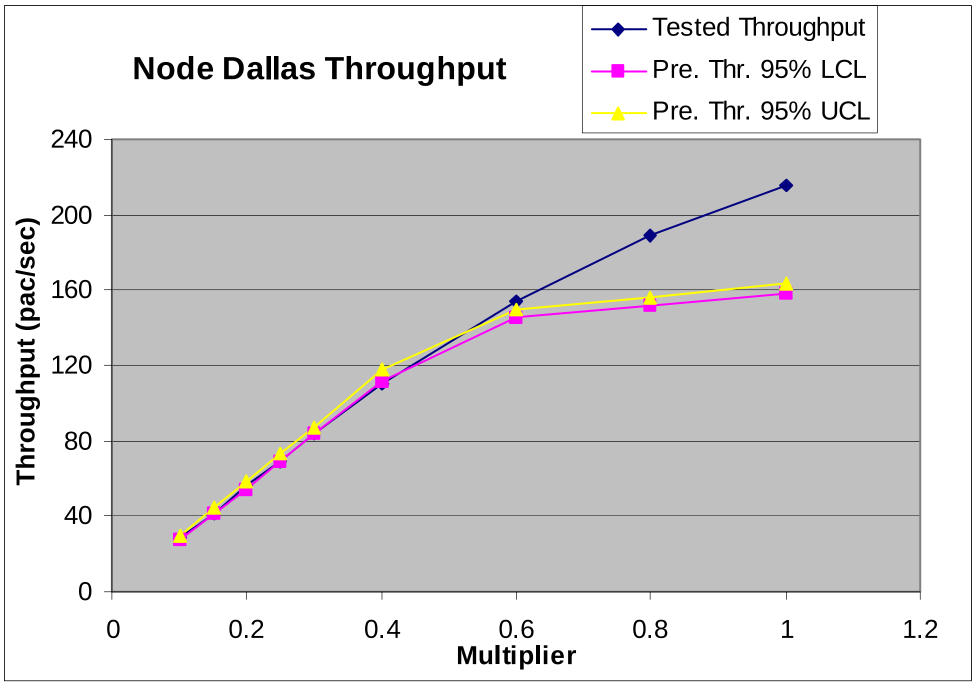
Throughput is an important factor in performance. We can get to know the capacity of node through the throughput of the node and what kind of workload the system can handle. We still chose *multiplier*s with 0.1, 0.15, 0.2, 0.25, 0.3, 0.4, 0.6, 0.8, and 1.0 to vary the speed of the generator.



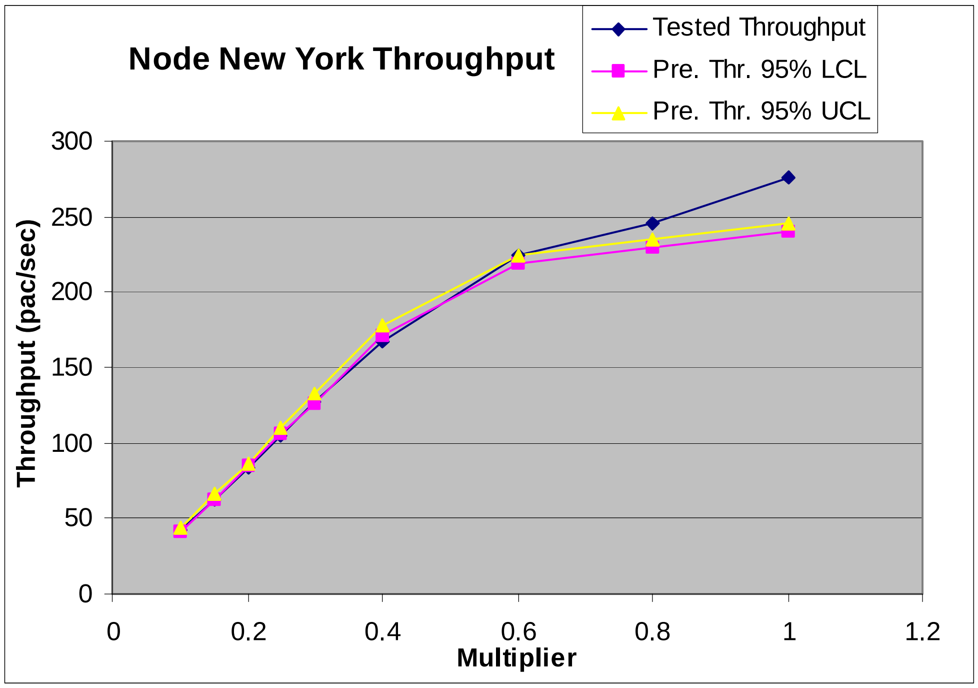
(a) Throughput of Atlanta for prediction and measurement against *multiplier* of workload



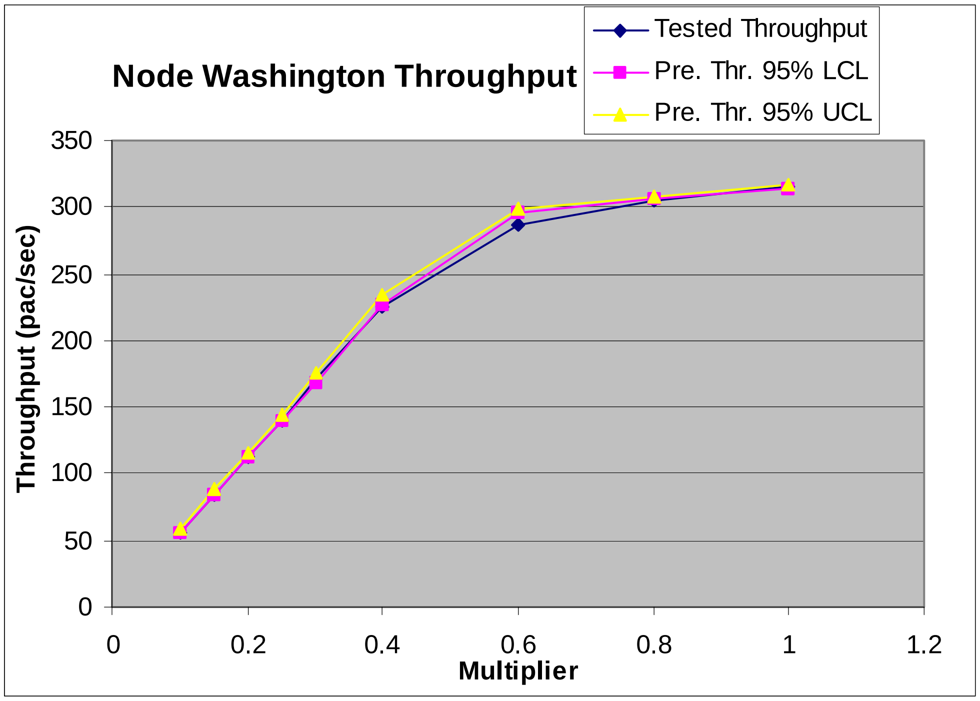
(b) Throughput of Chicago for prediction and measurement against *multiplier* of workload



(c) Throughput of Dallas for prediction and measurement against *multiplier* of workload



(d) Throughput of New York for prediction and measurement against *multiplier* of workload



(e) Throughput of Washington for prediction and measurement against *multiplier* of workload

Figure 6.2 Comparison between predicted and measured throughput for each node

The measured throughput of the node here is defined as the number of packets $n_R$ handled by node in a time unit, involving not only the packets sent/sunk but also the packet loss, which means the throughput of the switching step. The execution time of each node can be determined by the *time* command as shown in section 6.2.1. We can obtain the packets received and switched from the network statistics report when CGNet has been executed. Following the definition of the measured throughput of the node, we can map it to the throughput of the switching task in the LQN model.

Figure 6.2 show the comparison between the predicted throughput of the LQN model in the confidence interval with confidence coefficient 95% and the measured throughput against the rate of generators in the network within the range from 0.1 to 1. The throughputs for all nodes, Atlanta, Chicago, Dallas, New York, and Washington for comparison are plotted in Figure 6.2 (a), (b), (c), (d) and (e) respectively.

From the figure for each node, we find the LQN model provided a good prediction in the throughput for the experiment with the *multiplier* for the speed of the generator in the range 0.1- 0.6. The predicted throughput and measured throughput were quite similar and the difference between them was less than 5%. When the *multiplier* exceeded 0.6, there was a gap between the prediction throughput and the real system.

This trend of consistence in light load and inconsistency in heavy loads is the same as the utilizations discussed in subsection 6.3.1. The inconsistency in the heavy load is due to the time-sharing mode of the operating system with the goal to achieve high throughput.

### 6.2.3 Packet Loss Validation

Packet loss is an annoying factor in the performance of the network. The network designs and implementations try to avoid packet loss. There is no packet loss in the network under the specific design and configuration and this kind of design and configuration, can be accepted by us. In this subsection we focus on the packet loss and present the prediction of the LQN model on this aspect. The range of *multiplier*s is still 0.1, 0.15, 0.2, 0.25, 0.3, 0.4, 0.6, 0.8, and 1.0

In CGNet there are two types of buffer defined, one is a hidden buffer and another is an exposed buffer. The former is the incoming sockets and all the sources such as the local generator and other nodes connected to this node can send packets to the sockets. The latter is defined as outgoing queueing. Every send/sinking thread dequeues the packet and sends it. With either buffer, hidden buffer or exposed buffer, all the packets can line up in the queue.

Once the queue is full, it could lead to packet loss. Here we name them as receiving loss and link loss respectively.

| | **Node Atlanta** | | **Node Chicago** | | **Node Dallas** | |
|---|---|---|---|---|---|---|
| *multiplier* | $n_S$ | $n_L$ | $n_S$ | $n_L$ | $n_S$ | $n_L$ |
| 0.1 | 22430 | 0 | 43261 | 0 | 32729 | 0 |
| 0.15 | 31050 | 0 | 59725 | 0 | 44885 | 0 |
| 0.2 | 43914 | 0 | 83883 | 0 | 63302 | 0 |
| 0.25 | 51897 | 0 | 99217 | 0 | 74777 | 0 |
| 0.3 | 64011 | 0 | 122198 | 0 | 92007 | 0 |
| 0.4 | 85237 | 0 | 162499 | 0 | 122603 | 0 |
| 0.6 | 113009 | 0 | 220080 | 0 | 162129 | 0 |
| 0.8 | 137510 | 2607 | 290276 | 0 | 202723 | 559 |
| 1 | 159523 | 20753 | 371497 | 11416 | 248365 | 17404 |

| | **Node New York** | | **Node Washington** | |
|---|---|---|---|---|
| *multiplier* | $n_S$ | $n_L$ | $n_S$ | $n_L$ |
| 0.1 | 48635 | 0 | 65218 | 0 |
| 0.15 | 67355 | 0 | 90192 | 0 |
| 0.2 | 95013 | 0 | 127277 | 0 |
| 0.25 | 112740 | 0 | 150824 | 0 |
| 0.3 | 138880 | 0 | 186111 | 0 |
| 0.4 | 185277 | 0 | 248026 | 0 |
| 0.6 | 235106 | 11974 | 301038 | 0 |
| 0.8 | 263602 | 56698 | 326810 | 0 |
| 1 | 318420 | 95021 | 363203 | 0 |

Table 6.1 Statistics of a node for packets sent and loss

GNet records the link loss in network statisitics report during the execution. We tabulated the link loss and the packets that each node sent or sinked in Table 6.1. We found that node NewYork started to drop a packet when the *multiplier* exceeded 0.6, node Atlanta started to drop packet at 0.8, and node Chicago and Dallas began to drop packets at 1.0. There is no link loss in node Washington.

| | Atlanta | Chicago | Dallas | New York | Washington | Summary |
|---|---|---|---|---|---|---|
| *multiplier* | $n_{RG}$ | $n_{RG}$ | $n_{RG}$ | $n_{RG}$ | $n_{RG}$ | $\sum n_{RG}$ |
| 0.1 | 11304 | 21422 | 16236 | 24553 | 21265 | 94780 |
| 0.15 | 15582 | 29625 | 22403 | 33897 | 29408 | 130915 |
| 0.2 | 21925 | 41775 | 31560 | 47865 | 41455 | 184580 |
| 0.25 | 25926 | 49450 | 37322 | 56628 | 49093 | 218419 |
| 0.3 | 31986 | 60975 | 46085 | 69756 | 60526 | 269328 |
| 0.4 | 42702 | 81145 | 61476 | 92860 | 80603 | 358786 |
| 0.6 | 60465 | 114724 | 86914 | 131431 | 77453 | 470987 |
| 0.8 | 82815 | 157335 | 119203 | 180199 | 66019 | 605571 |
| 1 | 112295 | 213371 | 161604 | 244392 | 72697 | 804359 |

(a) Summary of received packets $n_{RG}$ from generator for each case

| | Atlanta | Chicago | Dallas | New York | Washington | Summary |
|---|---|---|---|---|---|---|
| *multiplier* | $n_{SK}+n_L$ | $n_{SK}+n_L$ | $n_{SK}+n_L$ | $n_{SK}+n_L$ | $n_{SK}+n_L$ | $\sum n_{SK}+n_L$ |
| 0.1 | 11130 | 21835 | 16493 | 24121 | 21156 | 94735 |
| 0.15 | 15504 | 30099 | 22532 | 33460 | 29242 | 130837 |
| 0.2 | 21990 | 42118 | 31778 | 47214 | 41307 | 184407 |
| 0.25 | 26003 | 49773 | 37472 | 56112 | 48988 | 218348 |
| 0.3 | 32103 | 61206 | 46016 | 69199 | 60560 | 269084 |
| 0.4 | 42593 | 81347 | 61212 | 92525 | 80863 | 358540 |
| 0.6 | 52588 | 105413 | 75409 | 128543 | 107372 | 469325 |
| 0.8 | 60751 | 132757 | 84660 | 197868 | 124751 | 600787 |
| 1 | 89329 | 181650 | 122650 | 264993 | 139261 | 797883 |

(b) Summary of sinked packets $n_{SK}$ and lost packets $n_L$ for each case

Table 6.2 Comparison for total received and total sinked and lost packets for each case

The receiving loss is not recordable in the network statisitcs report because the packets have been lost when they try to enter the incoming socket. The network statistics report is based on the incoming socket. Although the receiving loss has not been recorded, we still can get to know where it is lost. For the packets from the generator, we know how many packets generators generated and the packets $n_{RG}$ that the nodes really received. For the packets from other nodes, we can get to know the number of packets the node receives from upstream nodes and the number of packets the upstream nodes have sent in the network statistics report.

For the receiving loss from the *upstream* routers, we tabulated all packets $n_{RG}$ received from the generators in table 6.2 (a), and the packets sinked $n_{SK}$ and lost $n_L$ in each node in table 6.2 (b). Comparing the last column in table 6.2 (a) and table 6.2 (b), we know they are almost the same. The slight difference is due to CGNet performing statistics reports every 30 seconds. Some packets can be received but have not been accounted in the last 30 seconds. We can infer there is no packet loss in the receiving loss from the *upstream* router.

For the receiving loss from the generator, we know how long CGNet nodes executable runs for the *time* command when we performed the meausurements, and the capacity of the generator. We can calculate the number of the packets $n_{GS}$ the local generator has sent and the node receives $n_{RG}$. The difference is $g_{XX} = n_{GS} - n_{RG}$ for each node. In addition, we also know every node performs statistics every 30 seconds and writes to statistics reports to a statistics file. We assume CGNet needs an extra 5 seconds to write a file. That means the maximum packets $G_{XX}$ could enter the node but the node did not make statistics on them. They should not count as link loss. We have $G_{XX} = r_{XX} * 35 / (85 * 8)$ packets. We can tabulate the data collected from network statistics reports and the calculations based on configurations for each case in Table 6.3. The comparison between $g_{XX}$ and $G_{XX}$ is also shown in the last two column for each node in table 6.3. If there is a $g_{XX} > G_{XX}$, there must be receiving loss in that node at that case. From table 6.3 we can find there is receiving loss from the generator in node Washington since the *multiplier* is 0.6. There is no receiving loss from the generator in the other nodes.

| | **Node Atlanta** | | | | **Node Chicago** | | | |
|---|---|---|---|---|---|---|---|---|
| *multiplier* | $n_{GS}$ | $n_{RG}$ | $g_{AT}$ | $G_{AT}$ | $n_{GS}$ | $n_{RG}$ | $g_{CH}$ | $G_{CH}$ |
| 0.1 | 11479 | 11304 | 175 | 345 | 21808 | 21422 | 386 | 656 |
| 0.15 | 16095 | 15582 | 513 | 517 | 30548 | 29625 | 923 | 984 |
| 0.2 | 22346 | 21925 | 421 | 690 | 42417 | 41775 | 642 | 1311 |
| 0.25 | 26751 | 25926 | 825 | 862 | 50773 | 49450 | 1323 | 1639 |
| 0.3 | 32337 | 31986 | 351 | 1035 | 61321 | 60975 | 346 | 1967 |
| 0.4 | 43747 | 42702 | 1045 | 1379 | 83035 | 81145 | 1890 | 2623 |
| 0.6 | 62428 | 60465 | 1963 | 2069 | 118482 | 114724 | 3758 | 3934 |
| 0.8 | 85366 | 82815 | 2551 | 2759 | 161273 | 157335 | 3938 | 5246 |
| 1 | 113900 | 112295 | 1605 | 3449 | 216393 | 213371 | 3022 | 6557 |

(a) table for packets from generator in for Node Atlanta and Node Chicago

| | **Node Dallas** | | | | **Node New York** | | | |
|---|---|---|---|---|---|---|---|---|
| *multiplier* | $n_{GS}$ | $n_{RG}$ | $g_{DA}$ | $G_{DA}$ | $n_{GS}$ | $n_{RG}$ | $g_{NY}$ | $G_{NY}$ |
| 0.1 | 16487 | 16236 | 251 | 496 | 24910 | 24553 | 357 | 751 |
| 0.15 | 23093 | 22403 | 690 | 744 | 34823 | 33897 | 926 | 1126 |
| 0.2 | 32067 | 31560 | 507 | 992 | 48490 | 47865 | 625 | 1502 |
| 0.25 | 38312 | 37322 | 990 | 1240 | 57931 | 56628 | 1303 | 1877 |
| 0.3 | 46357 | 46085 | 272 | 1489 | 70096 | 69756 | 340 | 2253 |
| 0.4 | 62717 | 61476 | 1241 | 1985 | 94835 | 92860 | 1975 | 3004 |
| 0.6 | 89567 | 86914 | 2653 | 2977 | 135301 | 131431 | 3870 | 4506 |
| 0.8 | 121918 | 119203 | 2715 | 3969 | 184177 | 180199 | 3978 | 6008 |
| 1 | 163313 | 161604 | 1709 | 4962 | 246957 | 244392 | 2565 | 7510 |

(b) table for packets from generator in Node Dallas and Node New York

| | Node Washington | | | |
|---|---|---|---|---|
| *multiplier* | $n_{GS}$ | $n_{RG}$ | $g_{WA}$ | $G_{WA}$ |
| 0.1 | 21561 | 21265 | 296 | 651 |
| 0.15 | 30165 | 29408 | 757 | 977 |
| 0.2 | 42005 | 41455 | 550 | 1302 |
| 0.25 | 50135 | 49093 | 1042 | 1628 |
| 0.3 | 60664 | 60526 | 138 | 1953 |
| 0.4 | 82076 | 80603 | 1473 | 2604 |
| 0.6 | 117199 | 77453 | 39746 | 3907 |
| 0.8 | 159539 | 66019 | 93520 | 5209 |
| 1 | 213934 | 72697 | 141237 | 6511 |

(c) table for packets from generator in Node Washington

Figure 6.3 Receiving loss of each node from the local generator

The LQN model can capture the packet loss. It defines the asynchronous call to each entry of the receiving task and sending task and the corresponding buffer that is associated with the task. We can also define the buffer size mentioned in section 5.2 when we invoke the simulator with "messages=917" (see appendix C). The results of the LQN model use the term as an *Asynchronous Message Loss* to indicate the packet loss as that we defined in the real system. If there is packet loss for the receiving task, there is receiving loss; and the sending task is for link loss. Comparing the output files from the LQN model, we know that the LQN model provides the prediction of the packet loss when the *multiplier* is 0.6. That is consistent with the measurement.

The LQN model also offers which node is the busiest one. From the results, node Washington's utilization is 0.99895 and that is the cause for the receiving loss from the generator. Packet loss is the critical factor to evaluate the design and configuration of the network. The attractiveness of the LQN can be used to predict what kind of design and configuration will cause the packet loss and why there is packet loss.

Certainly, there are limitations of the LQN model to capture packet loss for CGNet. Once there is the asynchronous message loss in the LQN model, it treats all the task fairly. It distributes the asynchronous message loss to each tasks. Thus, the LQN model results indicate that there is an asynchronous message loss for every task whose entries receive the asynchronous call. We cannot use it to make further prediction of the packet loss proportion.

## 6.3 Discussion

The LQN model has provided the consistent prediction of CGNet in utilzation and throughput in the low workload case. It effectively pointed out when packet loss happens and showed the reason which causes the packet loss in the link level and receiving level.

We noticed that the LQN model can not predict the performance factors of CGNet with a *multiplier* more than 0.6. That makes the modeling exercise difficult especially when one node is saturated. In the following we summarize the assumptions in building the model to help understand the modeling exercise, and discuss some points addressed in chapter 5.

- Cost per packet in CGNet depends strongly on the number of threads and overhead from polling among the sockets. (secton 5.6).
- The UNIX system in time sharing mode does not obey the simple assumption that the CPU spends the same time in communication tasks with high workload as that with low workload. (secton 5.5, section 5.6)
- Only the CPU time can be derived without more accurate profiling tools because of the ISR for socket communication processing. (section 5.3)
- Regression modelling is effective to predict the parameter. (secton 5.4)
- Packet loss mechanisms in UNIX suystems is different from the assumption of the LQN model.(section 6.2.3)

# Chapter 7 Converter Tool

In chapter 4, we have developed the compositional approach to building the performance model through detailed analysis for a typical network. The structure of the performance model can be constructed from the configuration. Combined with the approach to measurement and parameter estimation in chapter 5, we can derive the parameters for the performance model. The converter tool integrating the compositional approach to the construction of the performance model structure and parameters obtained from measurements has been developed in the thesis. The tool makes it possible to generate an LQN model from the configuration files automatically if we assume the execution demands are the known variables. This chapter focuses on the automated tool and it covers the overview of the automated tool in section 7.1, the algorithm of the automated tool for gathering information in section 7.2, the algorithm of the automated tool for outputting the spex input file in section 7.3 and the validation of the automated tool in section 7.4.

## 7.1 Overview of Converter Tool

CGNet is a configuration-oriented tool for a test network. Once CGNet starts up, the CGNet executables read network description files in order to configure them and connect to each other during the initialization. The routing table is constructed according to the configuration. Hence the packets traverse the network according to the routing table derived from the configuration. The compositional model building approach based on packet class along the path has been well described in chapter 4. This approach makes it possible to automate the procedure of building the performance model, which integrates the execution demands derived in chapter 5. The idea can be illustrated in Figure 7.1.

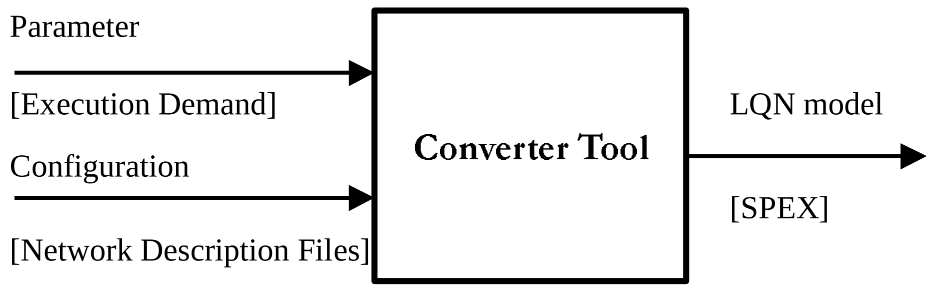


Figure 7.1 An approach to building the LQN model from the configuration of network.

All description files are formatted as simple ASCII text with lines formatted as follows:

*<keyword>:<parameter1>;<parameter2>;....;<parameterN>*

Each line consists of a keyword followed by a colon and a series of parameters separated by semicolons. The converter tool reads each line in the network description files and defines the characteristics of the nodes, traffic generators, traffic sinks, and links. The routing tables for all nodes are generated in the same way as in CGNet. We assume that we have known execution demands for the LQN model, we will now generate the performance model to implement the compositional approach described in chapter 4. The output of the converter tool generates the LQN model in the format of the spex input file. The converter tool was written in Java.

## 7.2 Algorithm of Converter Tool in Information Collection

In this section we focus on the algorithm of the converter tool in gathering information and building the routing table. When we gather information, the corresponding host processors, and tasks are defined together. After building the routing table as CGNet, the entries are defined for the LQN model.

The procedure of the converter tool may be formalized in the following steps:

1. Initialize the converter tool;
2. Get node information from *nodeinfo* and process the information;
3. Get generator information from *generatorinfo* and process the information;
4. Get sink information from *sinkinfo* and process the information;
5. Get link information from *linkinfo* and process the information;
6. Build routing table for each node;
7. Add entries to the corresponding task from the routing table;
8. Output the initial information such as Model Title and Description, Setting Pragmas, Controls, and Parameters and the Global Information to the LQN model for SPEX;
9. Output the processor information;
10. Output the task information;
11. Output the entry information;
12. Output the report information;

We discuss steps 1 through 7 in the following subsections. The remaining will be shown in section 7.3. Before we get into the detailed discussion, we define the following collection of objects as array in the converter tool:

a set of nodes **N** = $\{n_1, n_2, n_3, \ldots.\}$;

a set of generators **G** = $\{ g_1, g_2, g_3, \ldots\ldots\}$;

a set of links **L** = $\{ l_1, l_2, l_3, \ldots\ldots\}$;

a set of sinks **S** = $\{ s_1, s_2, s_3, \ldots\ldots\}$;

a set of processor **P** = $\{ p_1, p_2, p_3, \ldots\ldots\}$;

a set of routing table **RT** = $\{ rt_1, rt_2, rt_3, \ldots\ldots\}$;

for each node $n_i \in$ **N**, we define:

a set of user tasks, **Usr** $n_i$ = $\{ usrni_1, usrni_2, usrni_3, \ldots\ldots\}$;

a set of reveiving tasks, **Rcv** $n_i$ = $\{rcvni_1, rcvni_2, rcvni_3, \ldots\ldots\}$;

a set of switching tasks, **Swi** $n_i$ = $\{ swini_1, swini_2, swini_3, \ldots\ldots\}$;

a set of sending tasks, **Snd** $n_i$ = { sndni $_1$, sndni $_2$, sndni $_3$,……};

a set of sinking tasks, **Snk** $n_i$ = { snkni $_1$, snkni $_2$, snkni $_3$,……};

a set of network delay tasks, **Net** $n_i$ = { netni $_1$, netni $_2$, netni $_3$,……};

for each node $n_i \in$ **N** or routing table $rt_i \in$ RT , we define:

a set of routing entry, **RE** $n_i$ = { reni $_1$, reni $_2$, reni $_3$,……};

### 7.2.1 Initialize the Converter Tool

This subsection describes the initial step of the converter tool and it initializes the variables that will be used in the following steps.

1. Set a set of nodes **N** $=\varnothing$; a set of generators **G** $=\varnothing$; a set of links **L** $=\varnothing$; a set of sinks **S** $=\varnothing$

### 7.2.2 Get Node Information from *nodeinfo*

This subsection deals with the file *nodeinfo* and collects information of nodes. The switching task of the LQN model should be created in this step.

```
1.  Open the file nodeinfo
2.  While getnextline() != null do
    2.1     if line is effective
        2.1.1   Create node
            2.1.1.1 Set this node a set of user tasks Usr =∅; a set of receiving tasks Rcv
                    =∅; a set of switching tasks Swi =∅; a set of sending tasks Snd =∅; a
                    set of sinking tasks Snk =∅
        2.1.2   Parse the line for characteristics and set the node variable
        2.1.3   Add the switching task to a set of switching tasks Swi for this node
        2.1.4   Add the node to the set of nodes N
3.  Close the file nodeinfo
```

### 7.2.3 Get Generator Information from *generatorinfo*

This subsection handles the file *generatorinfo* and collects information of the generators. The user and receiving tasks of the LQN model should be created and the entry for different traffic destinations should be added in the receiving task and switching task for the node that this generator is connected to.

```
1.  Open the file generatorinfo
2.  While getnextline() != null do
    3.1     if line is effective
        3.1.1   Create generator
        3.1.2   Parse the line for characteristics and set the generator variable
        3.1.3   Add the user task to a set of user tasks Usr for the node which the generator
                is connected
        3.1.4   Add the receiving task to a set of receiving tasks Rcv for the node which the
                generator is connected
        3.1.5   Parse the line for traffic destinations
            3.1.5.1     for each destination do
                3.1.5.1.1   Add entry with this destination for the receiving task Rcv for the
                            node which the generator is connected
                3.1.5.1.2   Add entry with this destination for the switching task Swi for the
                            node which the generator is connected
        3.1.6   Add the generator to the set of generators G
```

```
3.  Close the file generatorinfo
```

### 7.2.4 Get Sink Information from *sinkinfo*

This subsection deals with the file *sinkinfo* and collects information of the traffic sinks. The switching task of the LQN model should be created in the following steps.

```
1.  Open the file sinkinfo
2.  While getnextline() != null do
    3.2     if line is effective
        3.2.1   Create sink
        3.2.2   Parse the line for characteristics and set the sink variable
        3.2.3   Add the sink task to a set of sinking tasks Snk for this node that this traffic
                sink is connected to
        3.2.4   Add the entry with the destination (its self) for the sink task for this node
                that this traffic sink is connected to
        3.2.5   Add the entry with the destination (its self) for the switching task for this
                node that this traffic sink is connected to
        3.2.6   Add the sink to the set of sinks S
3.  Close the file sinkinfo
```

### 7.2.5 Get Link Information from *linkinfo*

This subsection describes the collection of information for traffic links from the file *linkinfo*. The receiving and sending tasks of the LQN model should be created for each node the link is connected to.

```
1.  Open the file linkinfo
2.  While getnextline() != null do
    3.3     if line is effective
        3.3.1   Create link
        3.3.2   Parse the line for characteristics and set the link variable
        3.3.3   Add the receiving task to a set of receiving tasks **Rcv** for each node that this
                link is connected to
        3.3.4   Add the sending task to a set of sending tasks **Snd** for each node that this
                link is connected to
        3.3.5   Add the network delay task to a set of network delay tasks **Net** for each node
                that this link is connected to
        3.3.6   Add the link to the set of links **L**
3.  Close the file linkinfo
```

### 7.2.6 Build Routing Table for Each Node

The following algorithm fragment focuses on building the routing table following the same way as in CGNet. We put the routing tables of all nodes in a set of routing tables **RT** and one element in **RT** is defined for one node. The current routing policy is OSPF, Open Shortest Path First. For each node we can know the next hop for a packet class from the routing table and the shortest path to the destination is chosen for the packet class.

```
1.  for each node $n_i \in$ the set of Node **N**
    1.1     Create routing table $rt_i$ for node $n_i$
    1.2     Add routing table $rt_i$ to the set of routing table **RT**
    1.3     for each sink $s_{ij} \in$ the set of Sink **S**
        1.3.1   if the sink is connect the node $n_i$
```

```
            1.3.1.1 add routing table entry for this destination with cost 0
        1.3.2   else
            1.3.2.1 add routing table entry for this destination with cost $\infty$
2. do
    2.1     for each node $n_i \in$ the set of Node **N**
        2.1.1   for each link $l_k \in$ the set of Link **L** (link $l_k$ cost = $c_k$)
            2.1.1.1   for each entry $re_j \in$ the set of routing entry **RE** for node $n_i$
                      (destination is $d_j$ and cost is $c_j$ )
                2.1.1.1.1   if link $l_k$ is connected to node (the node on the other end of link
                            is $n_s$)
                    2.1.1.1.1.1   if ($c_m + c_k < c_j$) (the cost with the destination $d_j$ in node $n_s$)
                        2.1.1.1.1.1.1   Change entry $re_j$ next hop as $n_s$
                        2.1.1.1.1.1.2   Change entry $re_j$ new cost
                        2.1.1.1.1.1.3   Recording the change in routing table
3. while there is change in routing table
```

### 7.2.7 Add Entries to the Task from Routing Table

This subsection focuses the entries in the tasks in the LQN model. The algorithm generates the entries from the routing table and assigns the entries to the tasks. We traverse all routing entries in the routing table and define the corresponding entries in the LQN model. Then we assign the entry to the corresponding task.

```
1. for each node $n_i \in$ the set of nodes **N**
    1.1     for each routing entry $re_j \in$ the set of Routing Entries **RE** $n_i$
        1.1.1   Add the entry of LQN model to the network delay task in node $n_i$
        1.1.2   Add the entry of LQN model to the sending task in node $n_i$
```

1.1.3 Add the entry of LQN model to the receiving task in next hop node

1.1.4 Add the entry of LQN model to the switching task in next hop node

## 7.3 Algorithm of Converter Tool in Model Output

This section outputs the host processors, tasks, entries generated in section 7.3. Some necessary information including the parameters for the spex input file are output together. Thus we output the whole model.

### 7.3.1 Output the Initial Information

Now we have defined all the tasks and entries for the LQN model. The converter tool arrives at the stage of outputting the input file for the SPEX tools. This subsection begins to output the lines of text, which will form the initialization of the model. The initialization of the model involves the solver the SPEX chooses, the control statements, parameters and expressions, and the global information. The following algorithm is related to create the initialization of the spex input file.

```
1. Open the spex input file
2. Write the solver and related information
3. Write the control statements
4. Write the spex parameters
   4.1 Write the speed of generators
   4.2 Write the speed of the proportion for different destination
   4.3 Write the parameters for execution demand
5 Write the global information
6 Close the spex input file
```

### 7.3.2 Output the Processor Information

In this subsection comes the processor information. The algorithm fragment writes the information of the processor to the spex input file and the host processors of the nodes and the network processor should be included.

```
1. Open the spex input file
2. for each node $n_i \in$ the set of Node $\mathbf{N}$
    2.1     Write the user host processor
    2.2     Write the node host processor
3. Write the network host processor
4. Close the spex input file
```

### 7.3.3 Output the Task Information

The task information is in this subsection. The algorithm fragment outputs the task information in the spex input file. All the tasks such as user task, receiving task, switching task, sending task, sinking task and network delay task will be written into the spex input file.

```
1. Open the spex input file
2. for each node $n_i \in$ the set of Node $\mathbf{N}$
    2.1     Write the user task
    2.2     Write the receiving task
    2.3     Write the switching task
    2.4     Write the sending task
    2.5     Write the sinking task
    2.6    Write the network delay task
3.  Close the spex input file
```

### 7.3.4 Output the Entry Information

The entry information follows the task information in the spex input file and is described below in this subsection. The algorithm fragment implements the entry information in the spex input file. It not only handles the entries in the task, but also indicates the call between entries.

```
1.  Open the spex input file
2.  for each node $n_i \in$ the set of Node **N**
    2.1    Write the entries in the user task **User**
    2.2    Write the entries in the receiving task **Rcv**
    2.3    Write the calls from user task to receiving task
    2.4    Write the entries in the switching task **Swi**
    2.5    Write the calls from receiving task to switching task
    2.6    Write the entries in the sending task **Snd**
    2.7    Write the entries in the sinking task **Snk**
    2.8    Write the calls from switching task to the sending task and the sink task
    2.9    Write the entries in the network delay task **Net**
    2.10   Write the call from the sending task to the network delay task
    2.11   Write the call from the network delay task to the receiving task
3.  Close the spex input file
```

### 7.3.5 Output the Report Information

The final section of the spex input file is the report section. The purpose of this section is to specify which variable values are to be printed in the spex result file. The algorithm fragment outputs the report information in the utilization of processor and throughput. We can define the different report information according to different requirement.

```
1. Open the spex input file
2. for each node $n_i \in$ the set of Node $\mathbf{N}$
    2.1 Write the utilization of the node
    2.2 Write the throughput of the switching tasks
3. Close the spex input file
```

## 7.4 Validation of the Converter Tool

This section describes the example CGNets used to validate the converter tool algorithm. The examples are a linear unidirectional configuration of CGNet, a linear bidirectional configuration of CGNet and a five-node configuration of CGNet. These configurations are used to validate the converter tool for different purposes

The current LQN model generated by the converter tool is in the format of the input file for SPEX. SPEX checks both the syntax and the semantics of the LQN models by solver, LQNS or ParaSRVN tools, which is specified in the spex input file. SPEX generates the LQN models by variable substitutions, solves them by LQNS or the ParaSRVN tools, and then collect the data for performance characteristics. The LQN models generated by SPEX can be used as the input file for JlqnDef, which makes it possible to check the LQN model by JlqnDef. JlqnDef not only can perform a syntax check, but also can perform a visual check of the LQN model composition. So it can generate a graphical view of the LQN model and that makes it different from LQNS and the ParaSRVN tools.

### 7.4.1 The Linear Unidirectional Configuration Example

We chose the configuration with four nodes. There is only one generator and one sink. The packets, generated by the generator, traverse the network through four nodes and arrive at the sink. The topology is shown the Figure 5.5.

This configuration is very simple but is good for testing the converter tool. We get to know there are three bidirectional traffic links in the network and the links exist between Vancouver and Calgary, between Calgary and Toronto, and between Toronto and Montreal. From the current configuration, there is only unidirectional traffic that is from Vancouver to Montreal along the links. That could be a trick in the converter tool.

The converter tool generates the LQN model for the linear unidirectional configuration. Figure 7.2 is the visual output from JlqnDef for the LQN model generated by SPEX with the variable substitutions. The LQN model is solvable by LQNS and ParaSRVN, which shows that the LQN model generated by the converter tool is both syntactically and semantically correct.

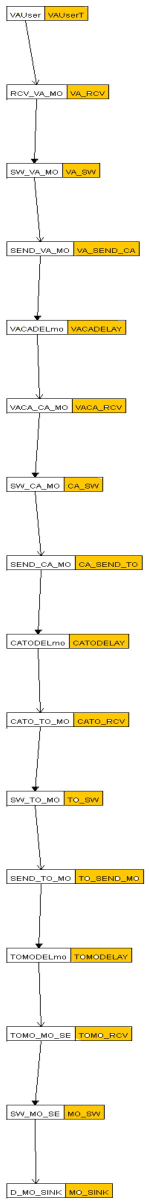


Figure 7.2 The LQN model generated by the converter tool from the linear unidirectional configuration.

### 7.4.2 The Linear Bidirectional Configuration Example

We chose another configuration with four nodes and it is still a linear configuration. There is two traffic generators located in the end of the line nodes, and two traffic sinks are in each end of the line. The traffic generated by a generator connected to the edge router traverses four node and arrives at a traffic sink connected to the other edge router. The topology is shown in the Figure 5.5.

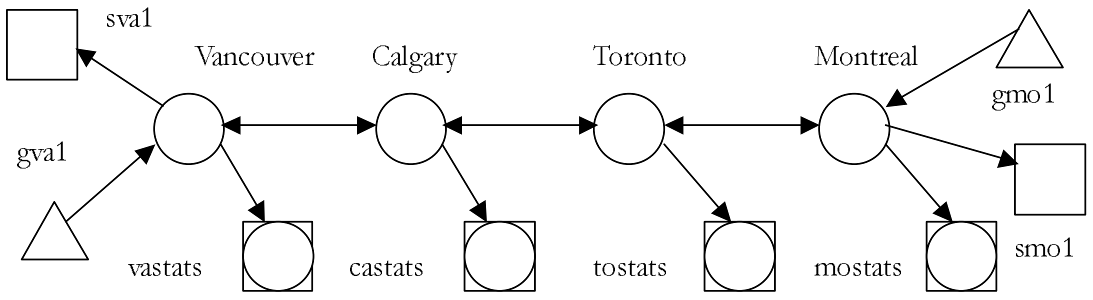


Figure 7.3 Linear Bidirectional Configuration of CGNet with four nodes

This configuration is still simple but it well deploys the bidirectional characteristics of the traffic links. The traffic generated by two generators follows the links between Vancouver and Calgary, between Calgary and Toronto, and between Toronto and Montreal. Some nodes lack traffic generators and traffic sinks. For example in the linear configuration, there are no traffic generators and traffic sinks that connect to nodes Calgary and Toronto.

We use the converter tool to generate the LQN model for the linear bidirectional configuration. JlqnDef performs a syntax check for the LQN model generated by SPEX with variable substitutions. Figure 7.4 shows the visual output for the LQN model generated by SPEX. The LQN model of the converter tool can be solved with LQNS as well as simulated with ParaSRVN. The syntactical and semantic correctness has been demonstrated for the LQN model generated by the converter tool.

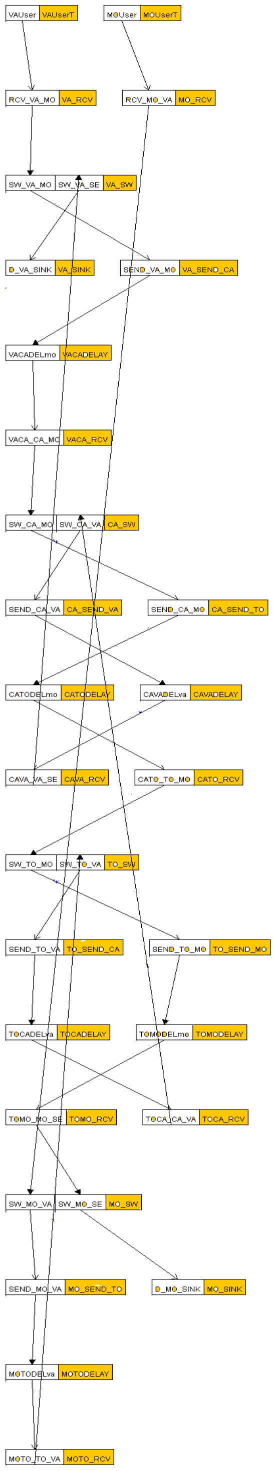


Figure 7.4 The LQN model generated by the converter tool from the linear bidirectional configuration.

### 7.4.3 The Five-Node Configuration Example

We chose the complicated five-node configuration for a network: there are five nodes, and there are traffic generators and traffic sinks, which connect to each node. Every bidirectional traffic links are fully used for data traffic. This configuration is more complex than the two previous configurations. The topology is shown in Figure 4.1

The converter tool generates a large number of entries, tasks, and call relationships that correspond to the spex input file. This results in the LQN model with lots of entries. The graphical view is not presented here because it is constrained by the page size. The spex input file generated by the converter tool is shown in Appendix D.

The LQN model for the complex configuration can be solved by the LQNS analytic solver as well as by the ParaSRVN simulator. We can solve it and derive the same result as the model created by hand. This demonstrates that the output from the converter tool is both syntactically and semantically correct.

# Chapter 8 Conclusions

We draw the conclusions of the thesis in this chapter. This chapter can be broken into the following sections: firstly, section 8.1 summarizes the thesis; secondly, section 8.2 discusses the conclusions of thesis; thirdly, section 8.3 provides suggestions for performance purpose; fourthly, section 8.4 outlines the contributions in detail; next, section 8.5 addresses the limitations of the research; and finally, in section 8.6, we propose some future works which focus on the limitations of the research.

## 8.1 Summary

It is the original motivation to predict the performance of the software system, identify the performance problems and solve the problems for performance critical systems. The performance modeling approach is the cornerstone of performance predictions and it also provides the basis for performance problem detection and performance optimization in the future. The challenges in the performance modeling approach have been well addressed.

Based on the research for CGNet, We chose the LQN model as the performance model. One of the characteristics of CGNet is that it can well emulate the behaviour of operational network. The LQN model for CGNet can bridge the gap between the operational network and performance analysis. The LQN model is constructed to predict the performance characteristics and identify performance problems.

During the construction of the performance model, we highlighted the packet class in CGNet and traced the scenarios of the packet class. With our understanding of CGNet, we built the template node-path sub models for the scenario fragments. The compositional approach to merging the sub model has been proposed especially for the application CGNet.

We can derive the node-path sub models by substitution for template node-path sub model within one node and merge them to the node sub models. We can then acquire the entire model through the composition of the node sub model for each node.

We parameterized the performance model through the collection of the configuration information, the measurement of the execution information and the estimation of parameters. Measurements of CGNet where the model is fully busy with communication processing were the most challenging in the thesis. We made use of the "Displacement" technique to obtain the CPU cost of processing for the model. Simple regression and multi regression models have been used for the least square estimations. Therefore, we completed the performance model with parameters with our option.

We built the LQN model based on the packet class. In fact, we can derive the packet class from the configuration of CGNet, and the network description file. That makes the converter tool possible, which generate the LQN model from the configuration. The thesis offers a detailed description for the algorithm of the converter tool and validates the converter tool with three typical configurations.

We can derive the predicted performance by solving the LQN model and obtain the performance characteristics by collection and measurements of the execution information. Thus we can validate the performance model in the performance characteristics of CGNet such as throughput, utilization, and packet loss. This has proven that the LQN model can be deployed in the prediction of network performance effectively.

From the predicted performance, we can make the analysis and provide the suggestions in the architecture redesign and reconfiguration for high performance in the network system.

## 8.2 Conclusions

The compositional framework for the constructing model approach is to be capable of modeling arbitrary large configurations. The converter tool was developed and it makes modeling of CGNet convenient and efficient.

Adequate measurements in CGNet for model parameters are difficult. The "Displacement" technique was good enough to give repeatable measurements for the total execution time. Parameters were found by regressions; however the values were not stable for different cases, and sometimes were not reasonable.

A notable discovery in the measurements was the much larger packet-handling cost in the five-node configuration. It was almost nearly double as that in the four-node configuration. The most likely cause of this phenomenon is thread switching overhead, as there are more threads (one per link for sending).

The model structure may not be adequate, in particular due to it ignoring the thread-switching overhead. The layered queueing framework has not addressed this problem; but it can be simulated. However, measurement of the thread switching would be necessary.

The contribution of the measurement is not so much for the model, but is the discovery of these problems during the model building.

## 8.3 Suggestions for Performance Purpose

Performance optimization is what people are concerned the most with and that meets the objective of the traffic-engineering framework described in chapter 1. This section discusses the optimization of performance after the modeling exercise.

Through constructing the LQN model, measuring and estmating parameters for the model, we derived the behaviour of CGNet from one configuration and understand its execution. The results of the LQN model and the discussions of the modeling exercise provided the necessary evidence for the software architecture redesign and network reconfiguration. Some suggestions are listed as follows:

1. Decreasing the number of threads in the node: The CPU cost in processing a packet in CGNet depends strongly on the number of threads. We can reduce the number of threads to decrease the overhead of thread switching.. Currently, there is one thread for each outgoing link. We can consider to use a reduced number of threads to manage all the outgoing links.
2. Batching for socket communications: For the main thread, we can change it to read all the data in the socket through one operation of reading the socket. For the sending/sinking thread, we can write several packets with the same next hop to the socket through one operation of writing the socket. The batching operation can save the overhead of I/O socket operation.
3. Load redistribution: We know there are link level and receiving level packet loss. The LQN model can predict what kind of load can lead to packet loss. If the arrival rate exceeds the capacity of the link, that will lead to link loss. For OSPF/IS-IS it could implement load balance through adjusting the cost of some specific links for its use, we can refer the paper [Fortz02]. For MPLS [Xiao00], it looks simple and we add an LSP to reroute the traffic. That means that we can add one packet class on CGNet and add one request class in the LQN model. We also need to update the converter tool in favor of widen usage .

## 8.4 Contributions

One of the contributions of this thesis is that the compositional model building approach based on assembling the sub models has been proposed. This approach is used in CGNet for constructing performance models and the sub models describe the network operations based on the packet class. The packet class can be derived from the configuration of CGNet. The routing table can be used to determine the path of the packet class.

The converter tool is the glue between the configuration of CGNet and the performance analysis using Layered Queueing Network (LQN). The converter tool bridges the configuration to the performance characteristics and enhances the prediction of the compositional strategy by its automated nature. The output of the tool can be analyzed by existing performance model tools such as the LQNS analytic solver and the ParaSRVN simulator. Although the tool is developed for OSPF, it can be changed to fit any source-destination pair routing algorithm.

The other contribution of the thesis is that the investigation of parameters estimation has been deployed for the concrete CGNet system. The parameters estimation approach integrates the "Displacement" technique in measurement and the least square estimation techniques for parameters from execution information. A notable finding is that the thread-switching overhead could be significant in a multithreaded application and cannot be ignored in the performance model building, which could lead to the unstable values of predicted parameters from the regression models.

## 8.5 Limitations

There are limitations to the research in the performance model of the communication system.

The thread-switching overhead has been ignored in the LQN model structure. Actually the layered queueing framework has not addressed this problem. The solution of this thesis is to distribute thread-switching overhead evenly; but that may lead to the inconsistency between predicted and measured performance characteristics in the validation stage. In particular, there is inconsistency for high loads.

The parameters derived from measurements and estimation may be too coarse. We only obtained the total CPU execution demand of the process from "Displacement" technique. Then we broke it into two parts, the receiving part and the sending part per packet. We detected the overhead of CPU scheduling among threads in our measurement but we still can not figure out how much the overhead is.

## 8.6 Future Work

Future work should be done to address the limitations in section 8.5. We can focus on the improvement of the Layered Queueing framework and make it simulate the overhead of thread switching.

Kernel measurements can be performed to detect the amount of overhead of thread switching and the overhead of polling among receiving sockets.

# Reference

[Awduche02] D. Awduche, A. Chui, A. Elwalid, I. Widjaja, and X. Xiao, "Overview and Principles of Internet Traffic Engineering", Request for Comments 3272, Internet Engineering Task Force, May. 2002.

[Bolch98] Gunter Bolch, Stefan Greiner, Hermann de Meer, Kishor S. Trivedi "Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications" John Wiley & Sons, August 1998

[Buhr96] R. J. A. Buhr, R. S. Casselman "Use Case Maps for Object-oriented Systems" Prentice Hall, Inc 1996

[Cary99] Cary "SAS OnlineDoc version eight" http://v8doc.sas.com/sashtml/ SAS Institute Inc, 1999

[Ciardo89] G. Ciardo, K. S. Trivedi, and J. Muppala, "SPNP: stochastic Petri net package", Proceedings of the Third International Workshop on Petri Nets and Performance Models (PNPM'89) Kyoto, Japan, 1989.

[Cruz91] R. L. Cruz, "A Calculus for Network Delay, Part II: Network Analysis", IEEE Transactions on Information Theory, vol. 37, Jan 1991.

[Elwalid01] Anwar Elwalid, Cheng Jin, Steven Low, and Indra Widjaja, "MATE: MPLS Adaptive Traffic Engineering", IEEE INFOCOM 2001, Apr 2001

[Fenlason97] Jay Fenlason and Richard Stallman "*GNU gprof*: The GNU Profiler" Manual, Free Software Foundation Inc. Sep. 1997

[Fink02] Jason R. Fink, Matt Sherer, Kurt Wall, "Linux Performance Tuning and Capacity Planning" SAMS, 2002, pp. 61

[Follenweider93] R.Follenweider, R.Karcich, G.J.Knafl "A Systematic Approach to Software Reliability Modeling" IEEE 1993

[Fortz02] B. Fortz, J. Rexford, and M. Thorup, "Traffic Engineering with Traditional IP Routing Protocols", *IEEE Communications*, Oct, 2002, pp.118-124.

[Frank98] H.Frank, Cervone "Solaris Performance Administration: Performance Measurement, Fine Tunning, and Capacity Planning for Releases 2.5.1 and 2.6" McGraw-Hill, 1998

[Franks00] G. Franks, "Performance Analysis of Distributed Server Systems", Report OCIEE-00-01, Ph. D. Thesis, Carleton University, Ottawa, Canada, Jan 2000.

[Gomaa00] Hassan Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML" Addison-Wesley, August 2000

[Graham82] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A Call Graph Execution Profiler" Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, Association for Computing Machinery, Jun 1982

[Hobbs01] C. Hobbs, G. Young, “CGNet: A User’s guide & designer’s manual” Nortel internal publications, Jun 2001

[Jacobson92] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. “Object-Oriented Software Engineering: A Use Case Driven Approach” Addison-Wesley, 1993.

[Jain91] R. Jain, "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling," John Wiley & Sons, Apr 1991.

[McCanne93] S. McCanne, C. Torek “ A Randomized Sampling Clock for CPU Utilization Estimation and code profiling” 1993 Winter USENIX conference, Jan. 1993

[Mendenhall81] W. Mendenhall, J. T. McClave “A second course in business statistics : regression analysis” Dellen Pub. Co., 1981

[Miller95] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam and T. Newhall. “The Paradyn Parallel Performance Measurement Tools” *IEEE Computer*, Nov 1995

[Marcotty 86] M. Marcotty & H. Ledgard, *The World of Programming Languages*, Springer-Verlag, Berlin 1986.

[Neilson95] J.E. Neilson, C.M. Woodside, D.C. Petriu and S. Majumdar, "Software Bottlenecking in Client-Server Systems and Rendez-vous Networks", *IEEE Transactions On Software Engineering*, Vol. 21, No. 9, Sep 1995

[Rolia95] J.A. Rolia, K.C. Sevcik,"The Method of Layers", IEEE Transactions on Software Engineering, Vol. 21 No. 8, Aug 1995,

[Scheaffer86] Richard L. Scheaffer, James T. McClave "Probability and Statistics for Engineers" Second Edition, Duxbury Press, 1986

[Smith90] C.U. Smith, "Performance Engineering of Software Systems", Addison-Wesley, 1990.

[Smith99] C. U. Smith, Murray Woodside, "Performance Validation at Early Stages of Development", Position paper, Performance 99, Istanbul, Turkey, October 99.

[Smith02] C. U. Smith and L. G. Williams, "Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software", Addison-Wesley, 2002.

[Takagi90] Hideaki Takagi, "Stochastic analysis of computer and communication systems" IFIP, 1990

[Veran85] M. V'eran and D. Potier. "QNAP2: a portable environment for queueing systems modeling". Modelling Techniques and Tools for Performance Analysis (ed. by D. Potier), North Holland, 1985.

[Woodside95A] C.M. Woodside, J.E. Neilson, D.C. Petriu and S. Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software", *IEEE Transactions on Computers*, Vol. 44, No. 1, Jan 1995

[Woodside95B] C.M. Woodside, "A Three-View Model for Performance Engineering of Concurrent Software", *IEEE Transactions On Software Engineering*, Vol. 21, No. 9, Sept. 1995.

[Woodside97] C. M. Woodside, Marc Courtois, Cheryl Schramm, “A "Displacement" Technique for Robust Portable Measurement of Communications Processing Overheads”, Internal Report, Real-time And Distributed Systems (RADS) Lab, Carleton University, May 5, 1997

[Xiao00] X. Xiao, A. Hannan, B. Bailey, “Traffic Engineering with MPLS in the Internet” IEEE Network, March/April 2000

# APPENDIX A BNF Description of Naming Notation of LQN Model

## Rule of Naming Notation of LQN model for CGNet

<host_node_name> ::=

The first two capital letter of the name for the node

<destination_node_name> ::=

The first two capital letter of the name for the node

<from_node_name> ::=

The first two capital letter of the name for the node

<to_node_name> ::=

The first two capital letter of the name for the node

<Hostprocessor_def> ::=

<User_processor> | <node_host_processor> | <network_processor>

<User_processor> ::= <host_node_name>**UserProc**

<node_host_processor> ::= <host_node_name>**Server**

<network_processor> ::= **Network**

<Task_def> ::= <User_task> | <Receiving_task> | <Switching_task> | <Sending_task> |

<Sinking_task> | <NetworkDelay_task>

<User_task> ::= <host_node_name>**UserT**

Based on generator connected to node

**<**entry_def**>** ::= <host_node_name>**User**

<Receiving_task> ::= <Rcv_generator_task> | <Rcv_node_task>

<Rcv_generator_task> ::= <host_node_name>**_RCV**

Based on generator

**<**entry_def**>** ::= **RCV_**<host_node_name>_<destination_node_name>

<Rcv_node_task> ::= <from_node_name><host_node_name>**_RCV**

Based on link connected to node

**<**entry_def**>** ::=

<from_node_name><host_node_name>_<host_node_name>_<destination_node_name> |

<from_node_name><host_node_name>_<host_node_name>**_SE**

(if <host_node_name> = <destination_node_name>)

<Switching_task> ::= <host_node_name>**_SW**

**<**entry_def**>** ::= **SW_**<host_node_name>_<destination_node_name> |

**SW_**<host_node_name>**_SE**

(if <host_node_name> = <destination_node_name>)

<Sending_task> ::= <host_node_name>**_SEND_**<to_node_name>

Based on link associated to node

**<**entry_def**>** → **SEND_**<host_node_name>_<destination_node_name>

<Sinking_task> ::= <host_node_name>**_SINK**

Based on sink associated to node

**<**entry_def**>** ::= **D_**<host_node_name>**_SINK**

<NetworkDelay_task> ::= <host_node_name><to_node_name>**DELAY**

Based on link associated to node

**<**entry_def**>** ::= <host_node_name><to_node_name>**DEL**<destination_node_name>*

(*<destination_node_name> are in the small letter)

# Appendix B the Code of *compulmt* Process

```
#include <stdio.h>
#include <time.h>
#include <math.h>
#define ITERATIONS 60000000

int do_something(int CPU_loops)
{

double temp, temp2;
int i, max_i;
static int flip_flop = 0;

max_i = CPU_loops;

for (i=0; i< max_i; i++) {

temp = 0.0;

temp = sin(1.4);
temp = cos(temp);
temp = sin(temp);
temp = cos(temp);
temp = sin(temp);
temp2 = sin(1.4);
temp2 = cos(temp2);
temp2 = sin(temp2);
temp2 = cos(temp2);
temp2 = sin(temp2);
}
return 1;
}

double output_timespec(struct timespec *t)
{
double res;
res=(double)(t->tv_sec) + (double)(t->tv_nsec)/1.000000000e9;
// printf("\n The time is %d.%09ld =  %.9lf% ", (double) t->tv_sec, t->tv_nsec,
res);
return res;
}

int main()
{
int i;
struct timespec start, end;
double starttime, endtime, elaptime;

clock_gettime(CLOCK_REALTIME, &start);
do_something(ITERATIONS);
clock_gettime(CLOCK_REALTIME, &end);

starttime = output_timespec(&start);
endtime = output_timespec(&end);

elaptime = endtime - starttime;

printf("\nelapsed from \t %.9lf\t to \t %.9lf\t and difference is \t %.9lf\n.",
starttime, endtime, elaptime);
}
```

# Appendix C LQN Model Constructed by Hand for Five-Node Network

```
$solver = parasrvn -B 10,100000,150000 -P messages=917

$factor1=0.2:2,0.2
$ATrate=0.0985294*$factor1   #67000/(85*8)=98.5294
$CHrate=0.1873529*$factor1   #127400/(85*8)=187.3529
$DArate=0.1417647*$factor1   #96400/(85*8)=141.7647
$NYrate=0.2145588*$factor1   #145900/(85*8)=214.5588
$WArate=0.1860294*$factor1   #126500/(85*8)=186.0294

$factor2=1
$Delay075=9.067*$factor2     #8*85/75000=0.009067
$Delay105=6.476*$factor2     #8*85/105000=0.006476
$Delay120=5.667*$factor2     #8*85/120000=0.005667
$Delay150=4.533*$factor2     #8*85/150000=0.004533

$RCVp1=0
$RCVp2=0
$ATSWIp1=1.6536
$ATSWIp2=0
$CHSWIp1=1.6536
$CHSWIp2=0
$DASWIp1=1.6536
$DASWIp2=0
$NYSWIp1=1.6536
$NYSWIp2=0
$WASWIp1=1.6536
$WASWIp2=0
$SNDp1=1.7764
$SNDp2=0
$SNKp1=1.7764
$SNKp2=0



G "Router Software System" .000001 100 1 0.9 -1

P 0
p ATUserProc f i
```

```
p ATServer f %u $ATRU

p DAUserProc f i
p DAServer f %u $DARU

p CHUserProc f i
p CHServer f %u $CHRU

p NYUserProc f i
p NYServer f %u $NYRU

p WAUserProc f i
p WAServer f %u $WARU

p NETProc f i
-1

T 0
t ATUserT n ATUser -1 ATUserProc  # Pseudo AT Task
t AT_RCV n RCV_AT_CH RCV_AT_DA RCV_AT_NY RCV_AT_WA -1 ATServer %f $ATGThr %pu $ATGU
# Task for AT generator

t DAAT_RCV n DAAT_AT_SE -1 ATServer %pu $ATDARU   # Task for ATDA link (rcv)
t WAAT_RCV n WAAT_AT_SE -1 ATServer %pu $ATWARU   # Task for ATWA link (rcv)

t AT_SW n SW_AT_SE SW_AT_CH SW_AT_DA SW_AT_NY SW_AT_WA -1 ATServer %f $ATAThr %pu
$ATSWU

t AT_SINK n D_AT_SINK -1 ATServer %f $ATSThr %pu $ATSINKU   # Thread Task for AT sink
t AT_SEND_DA n SEND_AT_DA -1 ATServer %f $ATDAThr %pu $ATDASU
# Thread Task for ATDA link (send)
t AT_SEND_WA n SEND_AT_CH SEND_AT_NY SEND_AT_WA -1 ATServer %f $ATWAThr %pu $ATWASU
# Thread Task for ATWA link (send)

t CHUserT n CHUser -1 CHUserProc   # Pseudo CH Task
t CH_RCV n RCV_CH_AT RCV_CH_DA RCV_CH_NY RCV_CH_WA -1 CHServer %f $CHGThr %pu $CHGU
# Task for CH generator

t DACH_RCV n DACH_CH_SE -1 CHServer %pu $CHDARU   # Task for CHDA link (rcv)
t NYCH_RCV n NYCH_CH_SE -1 CHServer %pu $CHNYRU   # Task for CHNY link (rcv)
t WACH_RCV n WACH_CH_SE -1 CHServer %pu $CHWARU   # Task for CHWA link (rcv)
```

```
t CH_SW n SW_CH_SE SW_CH_AT SW_CH_DA SW_CH_NY SW_CH_WA -1 CHServer %f $CHAThr %pu
$CHSWU

t CH_SINK n D_CH_SINK -1 CHServer %f $CHSThr %pu $CHSINKU  # Thread Task for CH sink
t CH_SEND_DA n SEND_CH_DA -1 CHServer %f $CHDAThr %pu $CHDASU  # Thread Task for CHDA
link (send)
t CH_SEND_NY n SEND_CH_NY -1 CHServer %f $CHNYThr %pu $CHNYSU  # Thread Task for CHNY
link (send)
t CH_SEND_WA n SEND_CH_AT SEND_CH_WA -1 CHServer %f $CHWAThr %pu $CHWASU
# Thread Task for CHWA link (send)

t DAUserT n DAUser -1 DAUserProc   # Pseudo DA Task
t DA_RCV n RCV_DA_AT RCV_DA_CH RCV_DA_NY RCV_DA_WA -1 DAServer %f $DAGThr %pu $DAGU
# Task for DA generator

t ATDA_RCV n ATDA_DA_SE -1 DAServer  %pu $DAATRU   # Task for ATDA link (rcv)
t CHDA_RCV n CHDA_DA_SE -1 DAServer  %pu $DACHRU   # Task for CHDA link (rcv)
t WADA_RCV n WADA_DA_SE -1 DAServer  %pu $DAWARU   # Task for WADA link (rcv)

t DA_SW n SW_DA_SE SW_DA_AT SW_DA_CH SW_DA_NY SW_DA_WA -1 DAServer %f $DAAThr %pu
$DASWU

t DA_SINK n D_DA_SINK -1 DAServer %f $DASThr %pu $DASINKU   # Thread Task for DA
sink
t DA_SEND_AT n SEND_DA_AT -1 DAServer %f $DAATThr %pu $DAATSU
# Thread Task for ATDA link (send)
t DA_SEND_CH n SEND_DA_CH -1 DAServer %f $DACHThr %pu $DACHSU
# Thread Task for CHDA link (send)
t DA_SEND_WA n SEND_DA_NY SEND_DA_WA -1 DAServer %f $DAWAThr %pu $DAWASU
# Thread Task for DAWA link (send)

t NYUserT n NYUser -1 NYUserProc  # Pseudo NY Task
t NY_RCV n RCV_NY_AT RCV_NY_CH RCV_NY_DA RCV_NY_WA -1 NYServer %f $NYGThr %pu $NYGU
# Task for NY generator

t CHNY_RCV n CHNY_NY_SE -1 NYServer  %pu $NYCHRU   # Task for CHNY link (rcv)
t WANY_RCV n WANY_NY_SE -1 NYServer  %pu $NYWARU   # Task for NYWA link (rcv)

t NY_SW n SW_NY_SE SW_NY_AT SW_NY_CH SW_NY_DA SW_NY_WA -1 NYServer %f $NYAThr %pu
$NYSWU
```

```
t NY_SINK n D_NY_SINK -1 NYServer %f $NYSThr  %pu $NYSINKU   # Thread Task for NY
sink
t NY_SEND_CH n SEND_NY_CH -1 NYServer %f $NYCHThr %pu $NYCHSU
# Thread Task for CHNY link (send)
t NY_SEND_WA n SEND_NY_AT SEND_NY_DA SEND_NY_WA -1 NYServer %f $NYWAThr %pu $NYWASU
# Thread Task for NYWA link (send)

t WAUserT n WAUser -1 WAUserProc  # Pseudo WA Task
t WA_RCV n RCV_WA_AT RCV_WA_CH RCV_WA_DA RCV_WA_NY -1 WAServer %f $WAGThr %pu $WAGU
# Task for WA generator

t ATWA_RCV n ATWA_WA_CH ATWA_WA_NY ATWA_WA_SE -1 WAServer %pu $WAATRU # Task for
ATWA link (rcv)
t CHWA_RCV n CHWA_WA_AT CHWA_WA_SE -1 WAServer %pu $WACHRU # Task for ATWA link (rcv)
t DAWA_RCV n DAWA_WA_NY DAWA_WA_SE -1 WAServer %pu $WADARU # Task for ATWA link (rcv)
t NYWA_RCV n NYWA_WA_AT NYWA_WA_DA NYWA_WA_SE -1 WAServer %pu $WANYRU # Task for
ATWA link (rcv)

t WA_SW n SW_WA_SE SW_WA_AT SW_WA_CH SW_WA_DA SW_WA_NY -1 WAServer %f $WAAThr %pu
$WASWU

t WA_SINK n D_WA_SINK -1 WAServer %f $WASThr %pu $WASINKU   # Thread Task for WA
sink
t WA_SEND_AT n SEND_WA_AT -1 WAServer %f $WAATThr %pu $WAATSU
# Thread Task for WAAT link (send)
t WA_SEND_CH n SEND_WA_CH -1 WAServer %f $WACHThr %pu $WACHSU
# Thread Task for WACH link (send)
t WA_SEND_DA n SEND_WA_DA -1 WAServer %f $WADAThr %pu $WADASU
# Thread Task for WADA link (send)
t WA_SEND_NY n SEND_WA_NY -1 WAServer %f $WANYThr %pu $WANYSU
# Thread Task for WANY link (send)

t ATDADELAY n ATDADELAYda -1 NETProc
t DAATDELAY n ATDADELAYat -1 NETProc
t ATWADELAY n ATWADELAYch ATWADELAYny ATWADELAYwa -1 NETProc
t WAATDELAY n ATWADELAYat -1 NETProc
t CHDADELAY n CHDADELAYda -1 NETProc
t DACHDELAY n CHDADELAYch -1 NETProc
t CHNYDELAY n CHNYDELAYny -1 NETProc
t NYCHDELAY n CHNYDELAYch -1 NETProc
```

```
t CHWADELAY n CHWADELAYwa -1 NETProc
t WACHDELAY n CHWADELAYat CHWADELAYch -1 NETProc
t DAWADELAY n DAWADELAYny DAWADELAYwa -1 NETProc
t WADADELAY n DAWADELAYda -1 NETProc
t NYWADELAY n NYWADELAYat NYWADELAYda NYWADELAYwa -1 NETProc
t WANYDELAY n NYWADELAYny -1 NETProc

-1

E 0
s ATUser 0 0 0 -1
#Z ATUser 0 0 0 -1
a ATUser $ATrate
z ATUser RCV_AT_CH 0.276 0 -1  #185/(185+103+204+178)=185/670=0.2761
z ATUser RCV_AT_DA 0.154 0 -1  #103/(185+103+204+178)=103/670=0.1537
z ATUser RCV_AT_NY 0.304 0 -1  #204/(185+103+204+178)=204/670=0.3045
z ATUser RCV_AT_WA 0.266 0 -1  #178/(185+103+204+178)=178/670=0.2657

y RCV_AT_CH SW_AT_CH 1 0 -1
y RCV_AT_DA SW_AT_DA 1 0 -1
y RCV_AT_NY SW_AT_NY 1 0 -1
y RCV_AT_WA SW_AT_WA 1 0 -1

z SW_AT_SE D_AT_SINK 1 0 -1
z SW_AT_CH SEND_AT_CH 1 0 -1
z SW_AT_DA SEND_AT_DA 1 0 -1
z SW_AT_NY SEND_AT_NY 1 0 -1
z SW_AT_WA SEND_AT_WA 1 0 -1

s RCV_AT_CH $RCVp1 $RCVp2 -1
s RCV_AT_DA $RCVp1 $RCVp2 -1
s RCV_AT_NY $RCVp1 $RCVp2 -1
s RCV_AT_WA $RCVp1 $RCVp2 -1
s DAAT_AT_SE $RCVp1 $RCVp2 -1
s WAAT_AT_SE $RCVp1 $RCVp2 -1

s SW_AT_SE $ATSWIp1 $ATSWIp2 -1
s SW_AT_CH $ATSWIp1 $ATSWIp2 -1
s SW_AT_DA $ATSWIp1 $ATSWIp2 -1
s SW_AT_NY $ATSWIp1 $ATSWIp2 -1
s SW_AT_WA $ATSWIp1 $ATSWIp2 -1
```

```
s D_AT_SINK $SNKp1 $SNKp2 -1
s SEND_AT_CH $SNDp1 $SNDp2 -1
s SEND_AT_DA $SNDp1 $SNDp2 -1
s SEND_AT_NY $SNDp1 $SNDp2 -1
s SEND_AT_WA $SNDp1 $SNDp2 -1


s CHUser 0 0 0 -1
#Z CHUser 0 0 0 -1
a CHUser $CHrate
z CHUser RCV_CH_AT 0.145 0 -1  #185/(185+314+450+325)=185/1274=0.1452
z CHUser RCV_CH_DA 0.247 0 -1  #314/(185+314+450+325)=314/1274=0.2465
z CHUser RCV_CH_NY 0.353 0 -1  #450/(185+314+450+325)=450/1274=0.3532
z CHUser RCV_CH_WA 0.255 0 -1  #325/(185+314+450+325)=325/1274=0.2551

y RCV_CH_AT SW_CH_AT 1 0 -1
y RCV_CH_DA SW_CH_DA 1 0 -1
y RCV_CH_NY SW_CH_NY 1 0 -1
y RCV_CH_WA SW_CH_WA 1 0 -1

z SW_CH_SE D_CH_SINK 1 0 -1
z SW_CH_AT SEND_CH_AT 1 0 -1
z SW_CH_DA SEND_CH_DA 1 0 -1
z SW_CH_NY SEND_CH_NY 1 0 -1
z SW_CH_WA SEND_CH_WA 1 0 -1

s RCV_CH_AT $RCVp1 $RCVp2 -1
s RCV_CH_DA $RCVp1 $RCVp2 -1
s RCV_CH_NY $RCVp1 $RCVp2 -1
s RCV_CH_WA $RCVp1 $RCVp2 -1
s DACH_CH_SE $RCVp1 $RCVp2 -1
s NYCH_CH_SE $RCVp1 $RCVp2 -1
s WACH_CH_SE $RCVp1 $RCVp2 -1

s SW_CH_SE $CHSWIp1 $CHSWIp2 -1
s SW_CH_AT $CHSWIp1 $CHSWIp2 -1
s SW_CH_DA $CHSWIp1 $CHSWIp2 -1
s SW_CH_NY $CHSWIp1 $CHSWIp2 -1
s SW_CH_WA $CHSWIp1 $CHSWIp2 -1
s D_CH_SINK $SNKp1 $SNKp2 -1
s SEND_CH_AT $SNDp1 $SNDp2 -1
```

```
s SEND_CH_DA $SNDp1 $SNDp2 -1
s SEND_CH_NY $SNDp1 $SNDp2 -1
s SEND_CH_WA $SNDp1 $SNDp2 -1


s DAUser 0 0 0 -1
#Z DAUser 0 0 0 -1
a DAUser $DArate
z DAUser RCV_DA_AT 0.107 0 -1  #103/(103+314+295+252)=103/964=0.1068
z DAUser RCV_DA_CH 0.326 0 -1  #314/(103+314+295+252)=314/964=0.3257
z DAUser RCV_DA_NY 0.306 0 -1  #295/(103+314+295+252)=295/964=0.3060
z DAUser RCV_DA_WA 0.261 0 -1  #252/(103+314+295+252)=252/964=0.2614

y RCV_DA_AT SW_DA_AT 1 0 -1
y RCV_DA_CH SW_DA_CH 1 0 -1
y RCV_DA_NY SW_DA_NY 1 0 -1
y RCV_DA_WA SW_DA_WA 1 0 -1

z SW_DA_SE D_DA_SINK 1 0 -1
z SW_DA_AT SEND_DA_AT 1 0 -1
z SW_DA_CH SEND_DA_CH 1 0 -1
z SW_DA_NY SEND_DA_NY 1 0 -1
z SW_DA_WA SEND_DA_WA 1 0 -1

s RCV_DA_AT $RCVp1 $RCVp2 -1
s RCV_DA_CH $RCVp1 $RCVp2 -1
s RCV_DA_NY $RCVp1 $RCVp2 -1
s RCV_DA_WA $RCVp1 $RCVp2 -1
s ATDA_DA_SE $RCVp1 $RCVp2 -1
s CHDA_DA_SE $RCVp1 $RCVp2 -1
s WADA_DA_SE $RCVp1 $RCVp2 -1

s SW_DA_SE $DASWIp1 $DASWIp2 -1
s SW_DA_AT $DASWIp1 $DASWIp2 -1
s SW_DA_CH $DASWIp1 $DASWIp2 -1
s SW_DA_NY $DASWIp1 $DASWIp2 -1
s SW_DA_WA $DASWIp1 $DASWIp2 -1
s D_DA_SINK $SNKp1 $SNKp2 -1
s SEND_DA_AT $SNDp1 $SNDp2 -1
s SEND_DA_CH $SNDp1 $SNDp2 -1
s SEND_DA_NY $SNDp1 $SNDp2 -1
```

```
s SEND_DA_WA $SNDp1 $SNDp2 -1


s NYUser 0 0 0 -1
#Z NYUser 0 0 0 -1
a NYUser $NYrate
z NYUser RCV_NY_AT 0.140 0 -1  #204/(204+450+295+510)=204/1459=0.1398
z NYUser RCV_NY_CH 0.308 0 -1  #450/(204+450+295+510)=450/1459=0.3084
z NYUser RCV_NY_DA 0.202 0 -1  #295/(204+450+295+510)=295/1459=0.2022
z NYUser RCV_NY_WA 0.350 0 -1  #510/(204+450+295+510)=510/1459=0.3496

y RCV_NY_AT SW_NY_AT 1 0 -1
y RCV_NY_CH SW_NY_CH 1 0 -1
y RCV_NY_DA SW_NY_DA 1 0 -1
y RCV_NY_WA SW_NY_WA 1 0 -1

z SW_NY_SE D_NY_SINK 1 0 -1
z SW_NY_AT SEND_NY_AT 1 0 -1
z SW_NY_CH SEND_NY_CH 1 0 -1
z SW_NY_DA SEND_NY_DA 1 0 -1
z SW_NY_WA SEND_NY_WA 1 0 -1

s RCV_NY_AT $RCVp1 $RCVp2 -1
s RCV_NY_CH $RCVp1 $RCVp2 -1
s RCV_NY_DA $RCVp1 $RCVp2 -1
s RCV_NY_WA $RCVp1 $RCVp2 -1
s CHNY_NY_SE $RCVp1 $RCVp2 -1
s WANY_NY_SE $RCVp1 $RCVp2 -1

s SW_NY_SE $NYSWIp1 $NYSWIp2 -1
s SW_NY_AT $NYSWIp1 $NYSWIp2 -1
s SW_NY_CH $NYSWIp1 $NYSWIp2 -1
s SW_NY_DA $NYSWIp1 $NYSWIp2 -1
s SW_NY_WA $NYSWIp1 $NYSWIp2 -1
s D_NY_SINK $SNKp1 $SNKp2 -1
s SEND_NY_AT $SNDp1 $SNDp2 -1
s SEND_NY_CH $SNDp1 $SNDp2 -1
s SEND_NY_DA $SNDp1 $SNDp2 -1
s SEND_NY_WA $SNDp1 $SNDp2 -1
```

```
s WAUser 0 0 0 -1
#Z WAUser 0 0 0 -1
a WAUser $WArate
z WAUser RCV_WA_AT 0.141 0 -1  #178/(178+325+252+510)=178/1265=0.1407
z WAUser RCV_WA_CH 0.257 0 -1  #325/(178+325+252+510)=325/1265=0.2569
z WAUser RCV_WA_DA 0.199 0 -1  #252/(178+325+252+510)=252/1265=0.1992
z WAUser RCV_WA_NY 0.403 0 -1  #510/(178+325+252+510)=510/1265=0.4032

y RCV_WA_AT SW_WA_AT 1 0 -1
y RCV_WA_CH SW_WA_CH 1 0 -1
y RCV_WA_DA SW_WA_DA 1 0 -1
y RCV_WA_NY SW_WA_NY 1 0 -1

z SW_WA_SE D_WA_SINK 1 0 -1
z SW_WA_AT SEND_WA_AT 1 0 -1
z SW_WA_CH SEND_WA_CH 1 0 -1
z SW_WA_DA SEND_WA_DA 1 0 -1
z SW_WA_NY SEND_WA_NY 1 0 -1

s RCV_WA_AT $RCVp1 $RCVp2 -1
s RCV_WA_CH $RCVp1 $RCVp2 -1
s RCV_WA_DA $RCVp1 $RCVp2 -1
s RCV_WA_NY $RCVp1 $RCVp2 -1
s ATWA_WA_CH $RCVp1 $RCVp2 -1
s ATWA_WA_NY $RCVp1 $RCVp2 -1
s ATWA_WA_SE $RCVp1 $RCVp2 -1
s CHWA_WA_AT $RCVp1 $RCVp2 -1
s CHWA_WA_SE $RCVp1 $RCVp2 -1
s DAWA_WA_NY $RCVp1 $RCVp2 -1
s DAWA_WA_SE $RCVp1 $RCVp2 -1
s NYWA_WA_AT $RCVp1 $RCVp2 -1
s NYWA_WA_DA $RCVp1 $RCVp2 -1
s NYWA_WA_SE $RCVp1 $RCVp2 -1

s SW_WA_SE $WASWIp1 $WASWIp2 -1
s SW_WA_AT $WASWIp1 $WASWIp2 -1
s SW_WA_CH $WASWIp1 $WASWIp2 -1
s SW_WA_DA $WASWIp1 $WASWIp2 -1
s SW_WA_NY $WASWIp1 $WASWIp2 -1
s D_WA_SINK $SNKp1 $SNKp2 -1
s SEND_WA_AT $SNDp1 $SNDp2 -1
```

```
s SEND_WA_CH $SNDp1 $SNDp2 -1
s SEND_WA_DA $SNDp1 $SNDp2 -1
s SEND_WA_NY $SNDp1 $SNDp2 -1


s ATDADELAYda $Delay075 0 -1         #8*85/75000=0.009067
s ATDADELAYat $Delay075 0 -1

s ATWADELAYch $Delay105 0 -1         #8*85/105000=0.006476
s ATWADELAYny $Delay105 0 -1
s ATWADELAYwa $Delay105 0 -1
s ATWADELAYat $Delay105 0 -1

s CHDADELAYda $Delay105 0 -1         #8*85/105000=0.006476
s CHDADELAYch $Delay105 0 -1

s CHNYDELAYny $Delay120 0 -1         #8*85/120000=0.005667
s CHNYDELAYch $Delay120 0 -1

s CHWADELAYat $Delay150 0 -1         #8*85/150000=0.004533
s CHWADELAYwa $Delay150 0 -1
s CHWADELAYch $Delay150 0 -1

s DAWADELAYny $Delay120 0 -1         #8*85/120000=0.005667
s DAWADELAYwa $Delay120 0 -1
s DAWADELAYda $Delay120 0 -1

s NYWADELAYat $Delay120 0 -1         #8*85/120000=0.005667
s NYWADELAYda $Delay120 0 -1
s NYWADELAYwa $Delay120 0 -1
s NYWADELAYny $Delay120 0 -1



y SEND_AT_DA ATDADELAYda 1 0 -1
z ATDADELAYda ATDA_DA_SE 1 0 -1
y ATDA_DA_SE SW_DA_SE 1 0 -1
y SEND_AT_CH ATWADELAYch 1 0 -1
z ATWADELAYch ATWA_WA_CH 1 0 -1
y ATWA_WA_CH SW_WA_CH 1 0 -1
y SEND_AT_NY ATWADELAYny 1 0 -1
```

```
z ATWADELAYny ATWA_WA_NY 1 0 -1
y ATWA_WA_NY SW_WA_NY 1 0 -1
y SEND_AT_WA ATWADELAYwa 1 0 -1
z ATWADELAYwa ATWA_WA_SE 1 0 -1
y ATWA_WA_SE SW_WA_SE 1 0 -1

y SEND_CH_DA CHDADELAYda 1 0 -1
z CHDADELAYda CHDA_DA_SE 1 0 -1
y CHDA_DA_SE SW_DA_SE 1 0 -1
y SEND_CH_NY CHNYDELAYny 1 0 -1
z CHNYDELAYny CHNY_NY_SE 1 0 -1
y CHNY_NY_SE SW_NY_SE 1 0 -1
y SEND_CH_AT CHWADELAYat 1 0 -1
z CHWADELAYat CHWA_WA_AT 1 0 -1
y CHWA_WA_AT SW_WA_AT 1 0 -1
y SEND_CH_WA CHWADELAYwa 1 0 -1
z CHWADELAYwa CHWA_WA_SE 1 0 -1
y CHWA_WA_SE SW_WA_SE 1 0 -1

y SEND_DA_AT ATDADELAYat 1 0 -1
z ATDADELAYat DAAT_AT_SE 1 0 -1
y DAAT_AT_SE SW_AT_SE 1 0 -1
y SEND_DA_CH CHDADELAYch 1 0 -1
z CHDADELAYch DACH_CH_SE 1 0 -1
y DACH_CH_SE SW_CH_SE 1 0 -1
y SEND_DA_NY DAWADELAYny 1 0 -1
z DAWADELAYny DAWA_WA_NY 1 0 -1
y DAWA_WA_NY SW_WA_NY 1 0 -1
y SEND_DA_WA DAWADELAYwa 1 0 -1
z DAWADELAYwa DAWA_WA_SE 1 0 -1
y DAWA_WA_SE SW_WA_SE 1 0 -1

y SEND_NY_CH CHNYDELAYch 1 0 -1
z CHNYDELAYch NYCH_CH_SE 1 0 -1
y NYCH_CH_SE SW_CH_SE 1 0 -1
y SEND_NY_AT NYWADELAYat 1 0 -1
z NYWADELAYat NYWA_WA_AT 1 0 -1
y NYWA_WA_AT SW_WA_AT 1 0 -1
y SEND_NY_DA NYWADELAYda 1 0 -1
z NYWADELAYda NYWA_WA_DA 1 0 -1
y NYWA_WA_DA SW_WA_DA 1 0 -1
```

```
y SEND_NY_WA NYWADELAYwa 1 0 -1
z NYWADELAYwa NYWA_WA_SE 1 0 -1
y NYWA_WA_SE SW_WA_SE 1 0 -1

y SEND_WA_AT ATWADELAYat 1 0 -1
z ATWADELAYat WAAT_AT_SE 1 0 -1
y WAAT_AT_SE SW_AT_SE 1 0 -1
y SEND_WA_CH CHWADELAYch 1 0 -1
z CHWADELAYch WACH_CH_SE 1 0 -1
y WACH_CH_SE SW_CH_SE 1 0 -1
y SEND_WA_DA DAWADELAYda 1 0 -1
z DAWADELAYda WADA_DA_SE 1 0 -1
y WADA_DA_SE SW_DA_SE 1 0 -1
y SEND_WA_NY NYWADELAYny 1 0 -1
z NYWADELAYny WANY_NY_SE 1 0 -1
y WANY_NY_SE SW_NY_SE 1 0 -1
-1

R 0
$0=$factor1
$factor2

$ATRU1 = $ATGU + $ATDARU + $ATWARU + $ATSWU + $ATDASU + $ATWASU + $ATSINKU
$CHRU1 = $CHGU + $CHDARU + $CHNYRU + $CHWARU + $CHSWU + $CHDASU + $CHNYSU + $CHWASU
+ $CHSINKU
$DARU1 = $DAGU + $DAATRU + $DACHRU + $DAWARU + $DASWU + $DAATSU + $DACHSU + $DAWASU
+ $DASINKU
$NYRU1 = $NYGU + $NYCHRU + $NYWARU + $NYSWU + $NYCHSU + $NYWASU + $NYSINKU
$WARU1 = $WAGU + $WAATRU + $WACHRU + $WADARU + $WANYRU + $WASWU + $WAATSU + $WACHSU
+ $WADASU + $WANYSU + $WASINKU
\

$ATGThr
$ATAThr
$ATSThr
$ATDAThr
$ATWAThr

\

$CHGThr
```

```
$CHAThr
$CHSThr
$CHDAThr
$CHNYThr
$CHWAThr

\

$DAGThr
$DAAThr
$DASThr
$DAATThr
$DACHThr
$DAWAThr

\

$NYGThr
$NYAThr
$NYSThr
$NYCHThr
$NYWAThr

\

$WAGThr
$WAAThr
$WASThr
$WAATThr
$WACHThr
$WADAThr
$WANYThr
-1
```

## Appendix D LQN Model Generated by Tool for Five-Node Network

```
#Define the solver of LQN model with specified option.
$solver = parasrvn -B 10,100000,150000 -P messages=917

#Define the speed of generator,proportion of traffic to destinaiton.
#and the networkdelay here.

$factor1=0.05:0.4,0.05

$ATrate =6700*$factor1/(85*8*1000)

$ATNEP =204/(204+178+185+103)
$ATWAP =178/(204+178+185+103)
$ATCHP =185/(204+178+185+103)
$ATDAP =103/(204+178+185+103)

$CHrate =12740*$factor1/(85*8*1000)

$CHNEP =450/(450+185+325+314)
$CHATP =185/(450+185+325+314)
$CHWAP =325/(450+185+325+314)
$CHDAP =314/(450+185+325+314)

$DArate =9640*$factor1/(85*8*1000)

$DANEP =295/(295+252+314+103)
$DAWAP =252/(295+252+314+103)
$DACHP =314/(295+252+314+103)
$DAATP =103/(295+252+314+103)

$NErate =14590*$factor1/(85*8*1000)

$NEATP =204/(204+510+295+450)
$NEWAP =510/(204+510+295+450)
$NEDAP =295/(204+510+295+450)
$NECHP =450/(204+510+295+450)

$WArate =12650*$factor1/(85*8*1000)
```

```
$WANEP =510/(510+178+252+325)
$WAATP =178/(510+178+252+325)
$WADAP =252/(510+178+252+325)
$WACHP =325/(510+178+252+325)


$ATDA= 8*85*1000/75000
$ATWA= 8*85*1000/105000
$CHDA= 8*85*1000/105000
$CHNY= 8*85*1000/120000
$CHWA= 8*85*1000/150000
$DAWA= 8*85*1000/120000
$NYWA= 8*85*1000/120000

#Define the CPU demand for each operation on each node.
$RCVp1=0
$RCVp2=0
$ATSWIp1=1.597822
$ATSWIp2=0
$CHSWIp1=1.537822
$CHSWIp2=0
$DASWIp1=1.897822
$DASWIp2=0
$NESWIp1=1.637822
$NESWIp2=0
$WASWIp1=1.437822
$WASWIp2=0
$SNDp1=1.732178
$SNDp2=0
$SNKp1=1.732178
$SNKp2=0

G "Router Software System" .000001 100 1 0.9 -1


#Define the hostprocessors in LQN model.
P 0
p ATUserProc f i
p sputnik f %u $ATRU
```

```
p CHUserProc f i
p alouette f %u $CHRU

p DAUserProc f i
p mariner f %u $DARU

p NEUserProc f i
p helicon f %u $NERU

p WAUserProc f i
p mira f %u $WARU

p NETProc f i
-1


#Define the tasks in LQN model.
T 0
t ATUserT n ATUser -1 ATUserProc # Pseudo AT Task

t AT_RCV n RCV_AT_NE RCV_AT_WA RCV_AT_CH RCV_AT_DA -1 sputnik %pu $ATRCVU0
t DAAT_RCV n DAAT_AT_SE -1 sputnik %pu $ATRCVU1
t WAAT_RCV n WAAT_AT_SE -1 sputnik %pu $ATRCVU2

t AT_SW n SW_AT_NE SW_AT_WA SW_AT_CH SW_AT_DA SW_AT_SE -1 sputnik %pu $ATSWIU0 %f
$ATThr0

t AT_SINK n D_AT_SINK -1 sputnik %pu $ATSNKU0

t AT_SEND_DA n SEND_AT_DA -1 sputnik %pu $ATSNDU0
t AT_SEND_WA n SEND_AT_CH SEND_AT_WA SEND_AT_NE -1 sputnik %pu $ATSNDU1

t ATDADELAY n ATDADELda -1 NETProc
t ATWADELAY n ATWADELch ATWADELwa ATWADELne -1 NETProc


t CHUserT n CHUser -1 CHUserProc # Pseudo CH Task

t CH_RCV n RCV_CH_NE RCV_CH_AT RCV_CH_WA RCV_CH_DA -1 alouette %pu $CHRCVU0
t DACH_RCV n DACH_CH_SE -1 alouette %pu $CHRCVU1
t NECH_RCV n NECH_CH_SE -1 alouette %pu $CHRCVU2
```

```
t WACH_RCV n WACH_CH_SE -1 alouette %pu $CHRCVU3

t CH_SW n SW_CH_NE SW_CH_AT SW_CH_WA SW_CH_DA SW_CH_SE -1 alouette %pu $CHSWIU0 %f
$CHThr0

t CH_SINK n D_CH_SINK -1 alouette %pu $CHSNKU0

t CH_SEND_DA n SEND_CH_DA -1 alouette %pu $CHSNDU0
t CH_SEND_NE n SEND_CH_NE -1 alouette %pu $CHSNDU1
t CH_SEND_WA n SEND_CH_WA SEND_CH_AT -1 alouette %pu $CHSNDU2

t CHDADELAY n CHDADELda -1 NETProc
t CHNEDELAY n CHNEDELne -1 NETProc
t CHWADELAY n CHWADELwa CHWADELat -1 NETProc


t DAUserT n DAUser -1 DAUserProc # Pseudo DA Task

t DA_RCV n RCV_DA_NE RCV_DA_WA RCV_DA_CH RCV_DA_AT -1 mariner %pu $DARCVU0
t ATDA_RCV n ATDA_DA_SE -1 mariner %pu $DARCVU1
t CHDA_RCV n CHDA_DA_SE -1 mariner %pu $DARCVU2
t WADA_RCV n WADA_DA_SE -1 mariner %pu $DARCVU3

t DA_SW n SW_DA_NE SW_DA_WA SW_DA_CH SW_DA_AT SW_DA_SE -1 mariner %pu $DASWIU0 %f
$DAThr0

t DA_SINK n D_DA_SINK -1 mariner %pu $DASNKU0

t DA_SEND_AT n SEND_DA_AT -1 mariner %pu $DASNDU0
t DA_SEND_CH n SEND_DA_CH -1 mariner %pu $DASNDU1
t DA_SEND_WA n SEND_DA_WA SEND_DA_NE -1 mariner %pu $DASNDU2

t DAATDELAY n DAATDELat -1 NETProc
t DACHDELAY n DACHDELch -1 NETProc
t DAWADELAY n DAWADELwa DAWADELne -1 NETProc


t NEUserT n NEUser -1 NEUserProc # Pseudo NE Task

t NE_RCV n RCV_NE_AT RCV_NE_WA RCV_NE_DA RCV_NE_CH -1 helicon %pu $NERCVU0
t CHNE_RCV n CHNE_NE_SE -1 helicon %pu $NERCVU1
```

```
t WANE_RCV n WANE_NE_SE -1 helicon %pu $NERCVU2

t NE_SW n SW_NE_AT SW_NE_WA SW_NE_DA SW_NE_CH SW_NE_SE -1 helicon %pu $NESWIU0 %f
$NEThr0

t NE_SINK n D_NE_SINK -1 helicon %pu $NESNKU0

t NE_SEND_CH n SEND_NE_CH -1 helicon %pu $NESNDU0
t NE_SEND_WA n SEND_NE_WA SEND_NE_AT SEND_NE_DA -1 helicon %pu $NESNDU1

t NECHDELAY n NECHDELch -1 NETProc
t NEWADELAY n NEWADELwa NEWADELat NEWADELda -1 NETProc


t WAUserT n WAUser -1 WAUserProc # Pseudo WA Task

t WA_RCV n RCV_WA_NE RCV_WA_AT RCV_WA_DA RCV_WA_CH -1 mira %pu $WARCVU0
t ATWA_RCV n ATWA_WA_CH ATWA_WA_SE ATWA_WA_NE -1 mira %pu $WARCVU1
t CHWA_RCV n CHWA_WA_SE CHWA_WA_AT -1 mira %pu $WARCVU2
t DAWA_RCV n DAWA_WA_SE DAWA_WA_NE -1 mira %pu $WARCVU3
t NEWA_RCV n NEWA_WA_SE NEWA_WA_AT NEWA_WA_DA -1 mira %pu $WARCVU4

t WA_SW n SW_WA_NE SW_WA_AT SW_WA_DA SW_WA_CH SW_WA_SE -1 mira %pu $WASWIU0 %f $WAThr0

t WA_SINK n D_WA_SINK -1 mira %pu $WASNKU0

t WA_SEND_AT n SEND_WA_AT -1 mira %pu $WASNDU0
t WA_SEND_CH n SEND_WA_CH -1 mira %pu $WASNDU1
t WA_SEND_DA n SEND_WA_DA -1 mira %pu $WASNDU2
t WA_SEND_NE n SEND_WA_NE -1 mira %pu $WASNDU3

t WAATDELAY n WAATDELat -1 NETProc
t WACHDELAY n WACHDELch -1 NETProc
t WADADELAY n WADADELda -1 NETProc
t WANEDELAY n WANEDELne -1 NETProc


-1


#Define the entries of tasks in LQN model.
```

```
E 0
s ATUser 0 0 -1
a ATUser $ATrate

z ATUser RCV_AT_NE $ATNEP 0 -1
z ATUser RCV_AT_WA $ATWAP 0 -1
z ATUser RCV_AT_CH $ATCHP 0 -1
z ATUser RCV_AT_DA $ATDAP 0 -1

s RCV_AT_NE $RCVp1 $RCVp2 -1
s RCV_AT_WA $RCVp1 $RCVp2 -1
s RCV_AT_CH $RCVp1 $RCVp2 -1
s RCV_AT_DA $RCVp1 $RCVp2 -1

y RCV_AT_NE SW_AT_NE 1 0 -1
y RCV_AT_WA SW_AT_WA 1 0 -1
y RCV_AT_CH SW_AT_CH 1 0 -1
y RCV_AT_DA SW_AT_DA 1 0 -1


s DAAT_AT_SE $RCVp1 $RCVp2 -1

y DAAT_AT_SE SW_AT_SE 1 0 -1


s WAAT_AT_SE $RCVp1 $RCVp2 -1

y WAAT_AT_SE SW_AT_SE 1 0 -1


s SW_AT_NE $ATSWIp1 $ATSWIp2 -1
s SW_AT_WA $ATSWIp1 $ATSWIp2 -1
s SW_AT_CH $ATSWIp1 $ATSWIp2 -1
s SW_AT_DA $ATSWIp1 $ATSWIp2 -1
s SW_AT_SE $ATSWIp1 $ATSWIp2 -1

s D_AT_SINK $SNKp1 $SNKp2 -1

z SW_AT_SE D_AT_SINK 1 0 -1

s SEND_AT_DA $SNDp1 $SNDp2 -1
```

```
z SW_AT_DA SEND_AT_DA 1 0 -1
y SEND_AT_DA ATDADELda 1 0 -1
z ATDADELda ATDA_DA_SE 1 0 -1
s SEND_AT_CH $SNDp1 $SNDp2 -1
s SEND_AT_WA $SNDp1 $SNDp2 -1
s SEND_AT_NE $SNDp1 $SNDp2 -1

z SW_AT_CH SEND_AT_CH 1 0 -1
y SEND_AT_CH ATWADELch 1 0 -1
z ATWADELch ATWA_WA_CH 1 0 -1
z SW_AT_WA SEND_AT_WA 1 0 -1
y SEND_AT_WA ATWADELwa 1 0 -1
z ATWADELwa ATWA_WA_SE 1 0 -1
z SW_AT_NE SEND_AT_NE 1 0 -1
y SEND_AT_NE ATWADELne 1 0 -1
z ATWADELne ATWA_WA_NE 1 0 -1

s ATDADELda $ATDA 0 -1
s ATWADELch $ATWA 0 -1
s ATWADELwa $ATWA 0 -1
s ATWADELne $ATWA 0 -1



s CHUser 0 0 -1
a CHUser $CHrate

z CHUser RCV_CH_NE $CHNEP 0 -1
z CHUser RCV_CH_AT $CHATP 0 -1
z CHUser RCV_CH_WA $CHWAP 0 -1
z CHUser RCV_CH_DA $CHDAP 0 -1

s RCV_CH_NE $RCVp1 $RCVp2 -1
s RCV_CH_AT $RCVp1 $RCVp2 -1
s RCV_CH_WA $RCVp1 $RCVp2 -1
s RCV_CH_DA $RCVp1 $RCVp2 -1

y RCV_CH_NE SW_CH_NE 1 0 -1
y RCV_CH_AT SW_CH_AT 1 0 -1
y RCV_CH_WA SW_CH_WA 1 0 -1
y RCV_CH_DA SW_CH_DA 1 0 -1
```

```
s DACH_CH_SE $RCVp1 $RCVp2 -1

y DACH_CH_SE SW_CH_SE 1 0 -1


s NECH_CH_SE $RCVp1 $RCVp2 -1

y NECH_CH_SE SW_CH_SE 1 0 -1


s WACH_CH_SE $RCVp1 $RCVp2 -1

y WACH_CH_SE SW_CH_SE 1 0 -1


s SW_CH_NE $CHSWIp1 $CHSWIp2 -1
s SW_CH_AT $CHSWIp1 $CHSWIp2 -1
s SW_CH_WA $CHSWIp1 $CHSWIp2 -1
s SW_CH_DA $CHSWIp1 $CHSWIp2 -1
s SW_CH_SE $CHSWIp1 $CHSWIp2 -1

s D_CH_SINK $SNKp1 $SNKp2 -1

z SW_CH_SE D_CH_SINK 1 0 -1

s SEND_CH_DA $SNDp1 $SNDp2 -1

z SW_CH_DA SEND_CH_DA 1 0 -1
y SEND_CH_DA CHDADELda 1 0 -1
z CHDADELda CHDA_DA_SE 1 0 -1
s SEND_CH_NE $SNDp1 $SNDp2 -1

z SW_CH_NE SEND_CH_NE 1 0 -1
y SEND_CH_NE CHNEDELne 1 0 -1
z CHNEDELne CHNE_NE_SE 1 0 -1
s SEND_CH_WA $SNDp1 $SNDp2 -1
s SEND_CH_AT $SNDp1 $SNDp2 -1

z SW_CH_WA SEND_CH_WA 1 0 -1
```

```
y SEND_CH_WA CHWADELwa 1 0 -1
z CHWADELwa CHWA_WA_SE 1 0 -1
z SW_CH_AT SEND_CH_AT 1 0 -1
y SEND_CH_AT CHWADELat 1 0 -1
z CHWADELat CHWA_WA_AT 1 0 -1

s CHDADELda $CHDA 0 -1
s CHNEDELne $CHNY 0 -1
s CHWADELwa $CHWA 0 -1
s CHWADELat $CHWA 0 -1


s DAUser 0 0 -1
a DAUser $DArate

z DAUser RCV_DA_NE $DANEP 0 -1
z DAUser RCV_DA_WA $DAWAP 0 -1
z DAUser RCV_DA_CH $DACHP 0 -1
z DAUser RCV_DA_AT $DAATP 0 -1

s RCV_DA_NE $RCVp1 $RCVp2 -1
s RCV_DA_WA $RCVp1 $RCVp2 -1
s RCV_DA_CH $RCVp1 $RCVp2 -1
s RCV_DA_AT $RCVp1 $RCVp2 -1

y RCV_DA_NE SW_DA_NE 1 0 -1
y RCV_DA_WA SW_DA_WA 1 0 -1
y RCV_DA_CH SW_DA_CH 1 0 -1
y RCV_DA_AT SW_DA_AT 1 0 -1


s ATDA_DA_SE $RCVp1 $RCVp2 -1

y ATDA_DA_SE SW_DA_SE 1 0 -1


s CHDA_DA_SE $RCVp1 $RCVp2 -1

y CHDA_DA_SE SW_DA_SE 1 0 -1
```

```
s WADA_DA_SE $RCVp1 $RCVp2 -1

y WADA_DA_SE SW_DA_SE 1 0 -1


s SW_DA_NE $DASWIp1 $DASWIp2 -1
s SW_DA_WA $DASWIp1 $DASWIp2 -1
s SW_DA_CH $DASWIp1 $DASWIp2 -1
s SW_DA_AT $DASWIp1 $DASWIp2 -1
s SW_DA_SE $DASWIp1 $DASWIp2 -1

s D_DA_SINK $SNKp1 $SNKp2 -1

z SW_DA_SE D_DA_SINK 1 0 -1

s SEND_DA_AT $SNDp1 $SNDp2 -1

z SW_DA_AT SEND_DA_AT 1 0 -1
y SEND_DA_AT DAATDELat 1 0 -1
z DAATDELat DAAT_AT_SE 1 0 -1
s SEND_DA_CH $SNDp1 $SNDp2 -1

z SW_DA_CH SEND_DA_CH 1 0 -1
y SEND_DA_CH DACHDELch 1 0 -1
z DACHDELch DACH_CH_SE 1 0 -1
s SEND_DA_WA $SNDp1 $SNDp2 -1
s SEND_DA_NE $SNDp1 $SNDp2 -1

z SW_DA_WA SEND_DA_WA 1 0 -1
y SEND_DA_WA DAWADELwa 1 0 -1
z DAWADELwa DAWA_WA_SE 1 0 -1
z SW_DA_NE SEND_DA_NE 1 0 -1
y SEND_DA_NE DAWADELne 1 0 -1
z DAWADELne DAWA_WA_NE 1 0 -1

s DAATDELat $ATDA 0 -1
s DACHDELch $CHDA 0 -1
s DAWADELwa $DAWA 0 -1
s DAWADELne $DAWA 0 -1
```

```
s NEUser 0 0 -1
a NEUser $NErate

z NEUser RCV_NE_AT $NEATP 0 -1
z NEUser RCV_NE_WA $NEWAP 0 -1
z NEUser RCV_NE_DA $NEDAP 0 -1
z NEUser RCV_NE_CH $NECHP 0 -1

s RCV_NE_AT $RCVp1 $RCVp2 -1
s RCV_NE_WA $RCVp1 $RCVp2 -1
s RCV_NE_DA $RCVp1 $RCVp2 -1
s RCV_NE_CH $RCVp1 $RCVp2 -1

y RCV_NE_AT SW_NE_AT 1 0 -1
y RCV_NE_WA SW_NE_WA 1 0 -1
y RCV_NE_DA SW_NE_DA 1 0 -1
y RCV_NE_CH SW_NE_CH 1 0 -1


s CHNE_NE_SE $RCVp1 $RCVp2 -1

y CHNE_NE_SE SW_NE_SE 1 0 -1


s WANE_NE_SE $RCVp1 $RCVp2 -1

y WANE_NE_SE SW_NE_SE 1 0 -1


s SW_NE_AT $NESWIp1 $NESWIp2 -1
s SW_NE_WA $NESWIp1 $NESWIp2 -1
s SW_NE_DA $NESWIp1 $NESWIp2 -1
s SW_NE_CH $NESWIp1 $NESWIp2 -1
s SW_NE_SE $NESWIp1 $NESWIp2 -1

s D_NE_SINK $SNKp1 $SNKp2 -1

z SW_NE_SE D_NE_SINK 1 0 -1

s SEND_NE_CH $SNDp1 $SNDp2 -1
```

```
z SW_NE_CH SEND_NE_CH 1 0 -1
y SEND_NE_CH NECHDELch 1 0 -1
z NECHDELch NECH_CH_SE 1 0 -1
s SEND_NE_WA $SNDp1 $SNDp2 -1
s SEND_NE_AT $SNDp1 $SNDp2 -1
s SEND_NE_DA $SNDp1 $SNDp2 -1

z SW_NE_WA SEND_NE_WA 1 0 -1
y SEND_NE_WA NEWADELwa 1 0 -1
z NEWADELwa NEWA_WA_SE 1 0 -1
z SW_NE_AT SEND_NE_AT 1 0 -1
y SEND_NE_AT NEWADELat 1 0 -1
z NEWADELat NEWA_WA_AT 1 0 -1
z SW_NE_DA SEND_NE_DA 1 0 -1
y SEND_NE_DA NEWADELda 1 0 -1
z NEWADELda NEWA_WA_DA 1 0 -1

s NECHDELch $CHNY 0 -1
s NEWADELwa $NYWA 0 -1
s NEWADELat $NYWA 0 -1
s NEWADELda $NYWA 0 -1


s WAUser 0 0 -1
a WAUser $WArate

z WAUser RCV_WA_NE $WANEP 0 -1
z WAUser RCV_WA_AT $WAATP 0 -1
z WAUser RCV_WA_DA $WADAP 0 -1
z WAUser RCV_WA_CH $WACHP 0 -1

s RCV_WA_NE $RCVp1 $RCVp2 -1
s RCV_WA_AT $RCVp1 $RCVp2 -1
s RCV_WA_DA $RCVp1 $RCVp2 -1
s RCV_WA_CH $RCVp1 $RCVp2 -1

y RCV_WA_NE SW_WA_NE 1 0 -1
y RCV_WA_AT SW_WA_AT 1 0 -1
y RCV_WA_DA SW_WA_DA 1 0 -1
y RCV_WA_CH SW_WA_CH 1 0 -1
```

```
s ATWA_WA_CH $RCVp1 $RCVp2 -1
s ATWA_WA_SE $RCVp1 $RCVp2 -1
s ATWA_WA_NE $RCVp1 $RCVp2 -1

y ATWA_WA_CH SW_WA_CH 1 0 -1
y ATWA_WA_SE SW_WA_SE 1 0 -1
y ATWA_WA_NE SW_WA_NE 1 0 -1


s CHWA_WA_SE $RCVp1 $RCVp2 -1
s CHWA_WA_AT $RCVp1 $RCVp2 -1

y CHWA_WA_SE SW_WA_SE 1 0 -1
y CHWA_WA_AT SW_WA_AT 1 0 -1


s DAWA_WA_SE $RCVp1 $RCVp2 -1
s DAWA_WA_NE $RCVp1 $RCVp2 -1

y DAWA_WA_SE SW_WA_SE 1 0 -1
y DAWA_WA_NE SW_WA_NE 1 0 -1


s NEWA_WA_SE $RCVp1 $RCVp2 -1
s NEWA_WA_AT $RCVp1 $RCVp2 -1
s NEWA_WA_DA $RCVp1 $RCVp2 -1

y NEWA_WA_SE SW_WA_SE 1 0 -1
y NEWA_WA_AT SW_WA_AT 1 0 -1
y NEWA_WA_DA SW_WA_DA 1 0 -1


s SW_WA_NE $WASWIp1 $WASWIp2 -1
s SW_WA_AT $WASWIp1 $WASWIp2 -1
s SW_WA_DA $WASWIp1 $WASWIp2 -1
s SW_WA_CH $WASWIp1 $WASWIp2 -1
s SW_WA_SE $WASWIp1 $WASWIp2 -1

s D_WA_SINK $SNKp1 $SNKp2 -1
```

```
z SW_WA_SE D_WA_SINK 1 0 -1

s SEND_WA_AT $SNDp1 $SNDp2 -1

z SW_WA_AT SEND_WA_AT 1 0 -1
y SEND_WA_AT WAATDELat 1 0 -1
z WAATDELat WAAT_AT_SE 1 0 -1
s SEND_WA_CH $SNDp1 $SNDp2 -1

z SW_WA_CH SEND_WA_CH 1 0 -1
y SEND_WA_CH WACHDELch 1 0 -1
z WACHDELch WACH_CH_SE 1 0 -1
s SEND_WA_DA $SNDp1 $SNDp2 -1

z SW_WA_DA SEND_WA_DA 1 0 -1
y SEND_WA_DA WADADELda 1 0 -1
z WADADELda WADA_DA_SE 1 0 -1
s SEND_WA_NE $SNDp1 $SNDp2 -1

z SW_WA_NE SEND_WA_NE 1 0 -1
y SEND_WA_NE WANEDELne 1 0 -1
z WANEDELne WANE_NE_SE 1 0 -1

s WAATDELat $ATWA 0 -1
s WACHDELch $CHWA 0 -1
s WADADELda $DAWA 0 -1
s WANEDELne $NYWA 0 -1


-1


#Define the report in LQN model for SPEX.
R 0
$0=$factor1
$ATU = $ATRCVU0 + $ATRCVU1 + $ATRCVU2 + $ATSNDU0 + $ATSNDU1 + $ATSNKU0 + $ATSWIU0
$CHU = $CHRCVU0 + $CHRCVU1 + $CHRCVU2 + $CHRCVU3 + $CHSNDU0 + $CHSNDU1 + $CHSNDU2
+ $CHSNKU0 + $CHSWIU0
$DAU = $DARCVU0 + $DARCVU1 + $DARCVU2 + $DARCVU3 + $DASNDU0 + $DASNDU1 + $DASNDU2
+ $DASNKU0 + $DASWIU0
$NEU = $NERCVU0 + $NERCVU1 + $NERCVU2 + $NESNDU0 + $NESNDU1 + $NESNKU0 + $NESWIU0
```

```
$WAU = $WARCVU0 + $WARCVU1 + $WARCVU2 + $WARCVU3 + $WARCVU4 + $WASNDU0 + $WASNDU1
+ $WASNDU2 + $WASNDU3 + $WASNKU0 + $WASWIU0

$ATThr0
$CHThr0
$DAThr0
$NEThr0
$WAThr0

$ATRU
$CHRU
$DARU
$NERU
$WARU
-1
```