In compliance with the
Canadian Privacy Legislation
some supporting forms
may have been removed from
this dissertation.

While these forms may be included
in the document page count,
their removal does not represent
any loss of content from the dissertation.

Scalability and Performance of Distributed Feature Composition

By
Jun Feng

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

Master of Science
in Information and Systems Science

Ottawa-Carleton Institute of Electrical and Computer Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6
2003

©Copyright 2003, Jun Feng

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-88768-5

Our file Notre référence

ISBN: 0-612-88768-5

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Canada

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

Scalability and Performance of Distributed Feature Composition

Submitted by Jun Feng, B.Sc., M.Eng.

**in partial fulfillment of the requirements for
the degree of Master of Science in Information and Systems Science**

Chair, Department of Systems and Computer Engineering

Thesis Supervisor

Carleton University

2003

Abstract:

Feature interaction has been identified as a major problem in the traditional telephony network and is expected to be worse in the Internet telephony network. Distributed feature composition (DFC) was proposed by AT&T as a service architecture for next generation telecommunication networks. In DFC, a feature is modularized as an independent feature box, all the features in a usage are dynamically assembled in a pipeline fashion and each feature has full control of all the call signals. This thesis studies the performance of a DFC prototype called ECLIPSE. The CPU demands for operations of DFC/ECLIPSE in different call scenarios are measured. Layered Queuing Networks (LQN) performance models for the call scenarios in DFC/ECLIPSE system are developed. Based on the LQN model, suggestions are made to improve the performance of ECLIPSE.

Acknowledgements

I would like to express my deep gratitude to my supervisor Professor Murray Woodside for his guidance, inspiration, patience and support in this research. I would also like to thank him for devoting time and energy in discussions of many difficult issues. I learned a lot from his profound knowledge and hardworking. I would like to thank AT&T for providing the thesis project and financial aid.

I would like to thank my colleagues in the Real-time and Distributed System Group for their friendship and support.

Finally, I would like to thank my family for their love, support and understanding during my study and research.

Contents

Acknowledgement.....	I
Abstract.....	II
Contents.....	III
List of Figures.....	VII
List of Tables.....	X
Chapter 1. Introduction.....	1
1.1 Background	1
1.2 Motivation and Objective	2
1.3 Thesis Outline	3
1.4 Thesis Contribution	3
1.5 Thesis Organization	4
Chapter 2. Feature Interactions in the Intelligent Network	5
2.1 Introduction to Intelligent Network	6
2.1.1 IN architecture	6
2.1.2 IN application protocol	10
2.2 Feature Interactions	10
2.2.1 Avoidance of feature interactions.....	12
2.2.2 Detection of feature interactions.....	13
2.2.3 Resolution of feature interactions.....	14
2.3 Summary	14
Chapter 3. Distributed Feature Composition (DFC)	
Architecture and ECLIPSE Project	15
3.1 DFC Architecture	15
3.1.1 Components	15
3.1.2 Routing and feature composition	17

3.2 ECLIPSE	20
3.2.1 Implementation of DFC architecture	21
3.2.2 Implementation basics	22
3.2.3 Supporting Data	26
3.2.4 Feature Boxes	27
3.2.5 Interface Boxes	30
3.2.6 Media Boxes	30
3.2.7 Routers	30
3.2.8 Provisioning Manager	31
3.3 Call Examples	32
3.3.1 Featureless call (Basic Call)	32
3.3.2 Transparent feature call	33
3.4 Comparison with IN	36
3.5 Summary	37
Chapter 4. Introduction to Layered Queuing Network (LQN)	
Models	38
4.1 Basic Ideas of LQN	38
4.2 LQN Notations	39
4.2.1 Task	39
4.2.2 Resource	40
4.2.3 Interaction	40
4.2.4 Entry	40
4.3 Building a Model	41
4.4 LQN Tools	46
4.5 Summary	46
Chapter 5. LQN model for DFC	48
5.1 Generic LQN Model of DFC/ECLIPSE System	49
5.1.1 Steps to build a LQN model for a DFC system	49
5.1.2 ECLIPSE specific LQN model details	50
5.2 Specific LQN Models for Two Typical Call Scenarios	52

5.2.1 LQN sub model for the two party featureless call	52
5.2.2 LQN model for the two party call with Transparent features	53
5.3 Summary	55
Chapter 6. Evaluation of Java Measurement Tools	56
6.1 Issues about Choosing Java Measurement Tool	56
6.2 Current Available Tools	57
6.3 Using JVM Profiler for Java Measurement	59
6.3.1 HPROF	60
6.3.2 PerfAnal	62
6.3.3 Calibration of the data from PerfAnal	63
6.4 Conclusions	63
Chapter 7. ECLIPSE Measurement	64
7.1 Eclipse Configuration for Measurement	64
7.2 Measurement	65
7.3 Measurement Results	67
7.3.1 Two-party featureless call	68
7.3.2 Two-Party feature call with transparent feature boxes	76
7.4 Summary	83
Chapter 8. Performance and Scalability Analysis of DFC	84
8.1 Performance Evaluation with Monitor System	84
8.1.1 Important system performance parameters	84
8.1.2 LQN model for the complete DFC/ECLIPSE system	85
8.1.3 LQN simulation results	89
8.1.4 Model results for two party featureless Calls without Monitor System.....	92
8.2 Detailed Performance Evaluation of DFC/ECLIPSE Network without Monitor System.....	95
8.2.1 Influence of the number of users on the performance	99
8.2.2 Influence of number of Transparent boxes	105
8.3 Sensitivity of The System to Workload Parameters	109
8.4 Discussion	112

8.4.1 Using dynamic extensible feature box pool instead of creating feature boxes at calling time	112
8.4.2 Deploying feature boxes at user terminal	113
8.5 Summary	114
Chapter 9. Conclusions and Future Work	115
9.1 Conclusions	115
9.2 Future Work	116
References	118
Appendix	121

List of Figures:

Figure 2.1 IN conceptual model.....	7
Figure 3.1. Components in the DFC architecture.....	16
Figure 3.2. A DFC usage.....	18
Figure 3.3 Components in ECLIPSE network.....	21
Figure 3.4. Box class hierarchy.....	23
Figure 3.5. DFC port class hierarchy.....	24
Figure 3.6. Relationship between the CallerPort, CalleePort, BoxPort and other objects.....	24
Figure 3.7. FSM class Hierarchy.....	25
Figure 3.8. A simple FSM.....	26
Figure 3.9. OCS-FSM.....	29
Figure 3.10. TransparentFSM.....	29
Figure 3.11 Two party featureless call.....	32
Figure 3.12 Signaling Sequence for a two-party featureless call.....	33
Figure 3.13 A usage with Transparent features.....	34
Figure 3.14 TransparentFeatureFSM.....	34
Fig. 3.15 Signaling sequence diagram for two party call with 2 transparent boxes...	35
Figure 4.1 Order tickets online line.....	39
Figure 4.2. LQN model with User and System.....	42
Figure 4.3 LQN model for Online Ticket system with User and class sub model...	42
Figure 4.4. LQN model for Online Ticket system with User and action task Submodel.....	44
Figure 4.5. Complete LQN model for Online Ticket system.....	45
Figure 5.1 Generic LQN model for DFC.....	52
Figure 5.2 LQN submodel for two party featureless call with monitor system, to fill a scenario submodel slot in Figure 5.1.....	53
Figure 5.3 LQN submodel for two party call with Transparent boxes, filling a scenario slot in the overall model of Figure 5.1.....	54
Figure 6.1 Interactions between JVM and profiler agent.....	60

Figure 6.2 jdk1.2.2 NT version consistency measurement.....	61
Figure 6.3 jdk1.3 UNIX version consistency measurement.....	62
Figure 7.1 The sequence of CPU usage for router process.....	66
Figure 7.2 CPU demand vs number of calls.....	67
Figure 7.3 CPU time relationships on caller terminal.....	69
Figure 7.4 CPU time relationships on callee terminal.....	72
Figure 7.5 CPU relationships on router.....	73
Figure 7.6 Relationship of CPU time of provisioning manager node	75
Figure 7.7 Relationship of CPU times of Router for two party call with Transparent box.....	78
Figure 7.8 CPU demand of RouterNode and RouterJVM vs number of Transparent boxes.....	82
Figure 8.1 Steps in a call cycle.....	84
Figure 8.2 Graphic LQN model for two party featureless call with monitor system.	86
Figure 8.3 Call signal handling time vs number of users for two party featureless call.....	90
Figure 8.4 Throughput vs number of users for two party featureless call with monitor system.....	90
Figure 8.5 Utilization vs number of users for two party featureless call with monitor system.....	91
Figure 8.6 Graphic LQN model for two party featureless call without monitor system.....	92
Figure 8.7 Call signal handling time vs number of users for two party featureless call without monitor system.....	93
Figure 8.8 Throughput vs number of users for two party featureless call without monitor system.....	94
Figure 8.9 Utilization vs number of users for two party feature less call without monitor system.....	94
Figure 8.10 LQN model for two party call with Transparent boxes.....	96
Figure 8.11 Call signal handling time vs number of users for two party call with Transparent boxes.....	100

Figure 8.12 Throughput vs number of users for two party call with Transparent boxes.....	102
Figure 8.13 Utilization vs number of users for the two party call with Transparent boxes.....	104
Figure 8.14 Call signal handling time vs number of Transparent boxes for two party call with Transparent.....	106
Figure 8.15 Throughput vs number of Transparent boxes for two party call with Transparent boxes.....	106
Figure 8.16 Utilization vs number of boxes for two party call with Transparent boxes.....	108
Figure 8.17 Call signal handling time vs CPU ratio for two party call with 16 Transparent boxes and 810 users.....	110
Figure 8.18 Throughput vs CPU ratio for two party call with 16 Transparent boxes and 810 users.....	110
Figure 8.19 Utilization vs CPU Ratio for two party call with 16 Transparent boxes and 810 users.....	111

List of Tables:

Table 7.1 Definition of the abbreviations of measurement results on caller host.....	70
Table 7.2 Results for caller user terminal.....	70
Table 7.3 Definition of the abbreviations of measurement results on callee host.....	72
Table 7.4 Results for callee user terminal.....	72
Table 7.5 Definition of the abbreviations of measurement results on router host..	74
Table 7.6 CPU time of router for two-party featureless call.....	74
Table 7.7 Definition of the abbreviations of the measurement results on the provisioning manager.....	75
Table 7.8 CPU time of provisioning manager for two-party featureless call.....	75
Table 7.9 Definition of the abbreviations in the router measurement results for two party call with Transparent boxes.....	78
Table 7.10 CPU used by the operations in the Transparent feature boxes.....	80
Table 7.11 The pure router measurement results.....	81
Table 8.1 Task and entry description of the Graphic LQN model for two party featureless call with monitor system.....	87
Table 8.2 CPU demand of the entries in the two party featureless call LQN model.....	88
Table 8.3 Task and entry description of Transparent box in the Graphic LQN model for two party call with transparent boxes.....	97
Table 8.4 CPU demands of the entries in the two party call with Transparent boxes LQN model.....	97

Chapter 1. Introduction

1.1. Background

Internet telephony provides text, voice and multimedia communication over the Internet. It is also called IP telephony. As compared to the end user devices used in the traditional Public Switched Telecommunication Network (PSTN), the end user devices in the IP telephony network are usually personal computers (PCs) or computing devices customized for telecommunication, and are much more intelligent. The intelligent end user devices can potentially provide new advanced services that traditional telephone end devices cannot, to the customers [Zave 00]. The increase of the number of services makes “feature interaction, which refers to situations in which instances of the same features or different features affect each other” [Keck 98], even worse than before. In this thesis, the terms, “feature” and “service” are used interchangeably as they are roughly synonymous, even though “service” is larger than “feature” in meaning in some cases. This thesis considers a system that intended to support flexible specification of advanced features and to avoid feature interactions.

Distributed feature composition (DFC) is an architecture for description of telecommunication services. It is designed for feature modularity, structured feature composition, analysis of feature interaction and separation between the signaling layer and media layer [Jackson 98, Bond 00]. In DFC, a feature is built based on the Pipe and Filter pattern illustrated in the literature [Shaw 96]. A feature has a few pipes and a finite state machine (FSM). Pipes are used to receive and send messages. A FSM processes the messages. A feature is modularized and independent from the other features. All features the caller and callee subscribe to are assembled in pipeline fashion along the call path. Each feature has complete control of the call. Each feature reacts to its own action triggering signals and lets all the other call signals pass through it. A

precedence rule is used in the assembly of the features to avoid bad feature interactions and preserve good feature interactions (feature composition). The DFC architecture is implemented in a prototype called the Extended Communication Layered on IP•Synthesis Environment (ECLIPSE) project. “The primary goal of the ECLIPSE project is to determine whether DFC can serve as the service architecture of next-generation telecommunication networks” [Bond 00].

This thesis is to study the performance of the DFC/ECLIPSE system.

1.2. Motivation and Objective

DFC significantly simplifies creation of new features, integration of them into the DFC network and avoids feature interaction through simple precedence rule. However, the modularity of the features, the pipeline fashion in assembling all the features and the fact that each feature fully touches and controls all call signals, causes system performance concerns. Furthermore, DFC is a brand new architecture for next generation telecommunication services. It is important to know the performance of DFC, to get the performance parameters metrics and to identify the performance bottleneck. The main motivation of this thesis is to study the performance of the DFC/ECLIPSE network. The objectives of the thesis are:

1. To measure the performance parameters of DFC/ECLIPSE system, mainly CPU demand for each task in the call.
2. To use LQN model and its tool to study the performance of the DFC/ECLIPSE system, get some important performance parameters such as call signal handling time, throughput and utilization of the system and identify the bottleneck of the system
3. To study the system performance after removal of the bottleneck, with the number of the users and the number of the features in each call.
4. To investigate the possibilities to improve the performance of the system through studying the sensitivity of the system performance.

1.3. Thesis Outline

In this thesis, the performance of the DFC/ECLIPSE network is investigated. The main tasks carried out in the thesis are listed below:

1. The CPU demand for operations of DFC/ECLIPSE network in different call scenarios are measured.
2. LQN models for the DFC/ECLIPSE system are developed.
3. The performance bottleneck for the system is identified as the monitor manager running on the provisioning manager host.
4. The variations of call signal handling time, throughput and utilization of the system with the number of users or the number of features in each call are computed.
5. System performance sensitivity is also studied.

Based on the results from above tasks, the following recommendations are proposed to improve the performance of the system.

1. Using dynamically extensible feature box pool instead of creating of feature boxes at the call setup stage.
2. Deploying as many feature boxes at the end user devices as possible.
3. Using C/C++ to replace Java if the performance of the real system is really unsatisfactory.
4. Using faster machine is another option if the performance of the system is unsatisfactory.

1.4. Thesis Contribution

The thesis makes the following contributions.

1. An approach to performance modeling which uses a pseudo-task (action task) to capture a scenario (Section 4.3).

2. The measurement and model of the ECLIPSE prototype as of early 2001 as a modeling case study (Chapter 7).
3. The analysis and recommendations as a performance engineering case study (Chapter 8).

1.5. Thesis organization

The remainder of the thesis is organized as follows: The second chapter gives a brief overview of intelligent network (IN) architecture, the dominant service architecture in the traditional telecommunication network and discusses the feature interactions within IN. The third chapter introduces DFC/ECLIPSE including creation of features, assembling of features, management of feature interactions, routing protocols, etc. The fourth chapter describes the LQN model and its tools. Basic concepts in LQN model and how to build LQN model for a specific system are briefly illustrated. The fifth chapter briefly reviews the Java measurement tools. In the sixth chapter, a generic LQN model for DFC/ECLIPSE system and customized models for different call scenarios are developed. In the seventh chapter, The CPU demands for the tasks of the components of the DFC/ECLIPSE network are measured in different call scenarios. In the eighth chapter, the performance and sensitivity of the DFC/ECLIPSE system is explored. The influence of the number of users and influence of the number of features in each call on the call signal handling time, throughput and utilization of the system are investigated. Sensitivity of the system performance to the CPU demand for each task in the DFC/ECLIPSE system is also studied. Finally the ninth chapter presents the conclusions of the thesis and provides suggestions for the future work.

Chapter 2. Feature Interactions in the Intelligent Network

“Feature interaction has been a major problem in the traditional public switched telephone network (PSTN) and is expected to be a worse problem in next generation telecommunication networks” [Jackson 98, Bond 00]. “Feature interaction makes service creation process long and costly and makes development of large switch extremely difficult” [Kolberg 00a]. In this chapter, we will discuss feature interactions in telecommunication networks. Intelligent network (IN) architecture is the dominant service architecture in the traditional telephone network based on switched circuits. International Telecommunication Union recommendation (ITU) H.323 and Session Initiation Protocol (SIP) developed by SIP working group within the Internet Engineering Task Force (IETF) are the current dominant IP telephony network protocols.

H.323 specifies supplementary services in the H.450.x series protocols. However it does not specify the service architecture to implement these services. H.450.x does not address feature (service) interactions. Within the framework of SIP, the Session Description Protocol (SDP) and Call Processing Language (CPL) can be viewed as a service architecture. As IP telephony is relative new, many aspects are still under development. Very few concrete feature interaction examples have been reported and no good solution to the feature interaction problem has been proposed [Lennox 00]. Many of the feature interaction examples are from IN. Only a few examples are IP telephony specific. Therefore the feature interaction is not discussed within H.323 and SIP.

Since the introduction of IN, a large number of telephone services have been developed and many feature interaction examples have been reported [Cameron 93,

Bouma 94, Cheng 95]. A great effort has been made to solve the feature interaction problem and a few solutions have been proposed. Therefore in this chapter, we will mainly discuss IN and feature interaction problems and solutions within IN.

2.1. Introduction to Intelligent Network

Telcordia (formerly Bellcore) first proposed the Advanced Intelligent Networks (AIN) concept using the signaling system number 7 (SS7) protocols with the addition of Service Control Points (SCPs) to simplify the creation, deployment and management of value-added services in the telecommunication networks [Telcordia 03]. Based on the Telcordia's original design, the International Telecommunication Union began to standardize IN in the late 1980s. Since then, several standards have been proposed in the form of capacity sets: CS-1, CS-2, CS-3 and CS-4. Each of the capacity sets describes an incremental set of service features [Green 97].

2.1.1 IN architecture

The ITU standards put the physical and logical views of the IN architecture into an IN conceptual model (INCM). The INCM is defined in the ITU recommendations. The INCM defines four different views of IN: the service plane (SP), the global functional plane (GFP), the distributed functional plane (DFP) and the physical plane (PP), as shown in Figure 2.1.

Service plane (SP)

The service plane is the top plane of the INCM that presents services from the user's point of view. The service can be a stand-alone feature or can be composed from other reusable features. For example, the one-number feature and reverse charging are stand-alone features, and free phone service is composed of these two reusable features.

The one number feature routes incoming calls to a single number to different telephones. Reverse charging feature charges the callee instead of the caller. Free phone

Figure 2.1 IN conceptual model.

service is like 1-800 number service. Usually a department, such as a sell department in a company, subscribes to this service. If a customer calls this 1-800 number, the call

will be routed to several phones in the company, and one of the clerks can pick up the phone. The company will be charged for the call.

The service plane specifies what features a service is composed of, but not the implementation of the features. IN standardizes the reusable features rather than those composite services. The composition of services is left to the network operator. Here are a few sample features defined in the CS-1 service plane: call forwarding, authentication, originating call screening, call waiting and automatic call back.

Global functional plane (GFP)

The GFP defines the services from the service provider's point of view. The GFP specifies the software components that a service provider must deploy to assemble services on the service plane. These components are called service independent building blocks (SIB). A feature defined in the SP is implemented by one or more SIBs in the GFP. This relationship is illustrated by the lines connecting the features in the SP and the SIBs in the GFP. There are two kinds of data supporting SIBs. One is Call Instance Data (CID) that includes the originator's address and the dialed number. The other is Service Support Data (SSD) that is independent of the call. Examples of SSD are call-screening list, a call-forwarding table or a charging scheme.

Among the SIBs, there is one special building block that represents the call itself. This SIB is called the Basic Call Process (BCP). The BCP is responsible for all phases to set up a telephone call from one user to another. At each of these phases, the call setup process can be halted and other services can be initiated. After the last service finishes, it returns the control to the BCP. Here are some examples of SIBs defined in the CS-1 global functional plane: algorithm, authenticate, charge, queue, screen and translate.

- An algorithm SIB is used to process CID and SSD.
- An authenticate SIB verifies a user's privilege to place and receive calls and to access particular service logic and data.

- A charge SIB is responsible to charge the user for the call.
- A queue SIB queues calls for completion to a specific telephone number.
- A screen SIB compares the input data against a list of data.
- A translate SIB translates the input parameters into output values based on a specific algorithm or table.

Distributed function plane (DFP)

The DFP provides logical distributed functions inside the network. Setup of a telephone call between a caller and a callee is the result of interactions between switches. These switches use a routing algorithm to decide how to route the call from the caller to the callee. The DFP specifies the different functional entities (FE) in the network. FEs are distributed in the network to support the BCP and other SIBs defined in the GFP. A single SIB may need more than one FEs to function.

The most important FEs defined in CS-1 are the call control function (CCF), the service switch function (SSF), the service control function (SCF), the service data function (SDF), the service management function (SMF) and the special resource function (SRF).

- A CCF keeps the state of a call.
- A SSF controls voice circuits.
- A SCF contains service logic that controls services.
- A SDF is the database that supports the services.
- A SMF contains all management functions.
- A SRF contains resource related functions, such as playing announcement and tones.

Physical plane (PP)

The physical plane deploys functions to physical locations or specific nodes. There are five Physical Entities (PE): Service Switching Point (SSP), Service Control

Point (SCP), Service Data Point (SDP), Service Management Point (SMP) and Intelligent Peripheral (IP) in this plane. There is a straightforward mapping of functional entities in distributed functional plane onto physical ones in the physical plane as shown in figure 2.1.

- A SSP hosts the switch, CCF and SSF of the distributed functional plane.
- A SCP hosts SCF.
- A SDP hosts the database for the SDF.
- A SMP hosts the SMF.
- A IP implements the SRF.

The INCM allows IN to be built in a modular way by creating a software module for each FE.

2.1.2 IN application protocol

The protocol used for communication between the FEs is called the IN Application Protocol (INAP). INAP messages contain the information exchanged between FEs. The communication between the service control point (SCP) and service data point (SDP) is an exception. It is not defined by INAP. It is based on the Directory Access Protocol (DAP) specified by the ITU in its X.500 recommendation.

2.2. Feature interactions

Feature interaction has been a major problem in the traditional circuit switched network including IN. It is expected that this problem will be much worse in the IP telephony network since intelligent end user devices, multimedia transmission and mobility will bring in a large number of new services to the IP telephony [Zave 00]. In the following part of the chapter, we will discuss features, feature interaction, the feature interaction problem and strategies to tackle it.

Feature

A feature is an optional unit or increment of functionality in a software system [Jackson 98]. Here are some well-known features in PSTN: Call Waiting (CW), Call Forwarding (CF), Voice Mail (VM), Outgoing Call Screen (OCS) and Call Party Identification (CPI).

Feature Interaction

“A feature interaction is some way in which a feature or features modify or influence another feature in defining overall system behavior” [Jackson 98]. Feature interactions can be divided into two categories: good feature interactions and bad interactions

A good feature interaction is the situation in which some features interact with each other to achieve certain desired behavior. In most cases, it is called feature composition. For example, in PSTN, Call Party Identification (CPI) feature provides the identity of the caller to Voice Mail (VM) feature so that the VM can send different messages to different callers.

“A bad feature interaction is one that causes the specification to be incomplete, inconsistent, or unimplementable, or that causes the overall system behavior to be undesirable” [Jackson 98]. A feature interaction problem is caused by a bad feature interaction. A number of bad feature interaction cases in traditional circuit switched network IN, and IP networks based on H.323 and/or SIP protocols, have been reported [Blair 00]. Here a few bad feature interaction examples are presented.

Example 1 Outgoing Call Screening (OCS) and Call Forwarding (CF)

Outgoing call screening blocks calls from the caller to a list of numbers. However, if a callee subscribes to a CF feature whose forwarding number is on the list,

even if OCS blocks the caller from calling to all numbers on the list, the call still can reach a number on the list because it is forwarded to that number by CF subscribed by the callee.

Example 2 Speed Calling (SC) and Outgoing Call Screening (OCS)

Speed Calling and Outgoing Call Screening in wireless system may have a bad feature interaction [Zave 00]. SC enables a user to dial a short code that is translated by the SC feature to a full address. OCS prevents the placing of outgoing calls to certain addresses. If the SC makes action after OCS, then OCS fails to prevent the placing of outgoing calls to certain addresses.

Example 3 Call Waiting (CW) and Call Forwarding on Busy (CFB)

Both CW and CFB are features handling the busy signal sent back from the callee. When CW receives a busy signal from callee, it sends a signal to the callee indicating a call is coming so that callee can know another call is waiting. If CFB is assigned higher priority to react to the busy signal from the callee. The CFB will absorb the busy signal from the callee, if callee is already engaged in another call, and the CFB will forward the call to the other address. In this case, an unexpected result happens, the CW feature will never receive a busy signal and never have a chance to function.

Feature interaction problems make the service creation process long and costly and causes big difficulty in development of a large switch [Kolberg 00a]. Due to the seriousness of the problems, the feature interaction problem has drawn a lot of attention in the industry and academia. Feature interactions have been intensely studied. The research has been focused in three areas for solving feature interaction problem: avoidance, detection and resolution.

2.2.1 Avoidance of feature interactions

In some cases, it is possible to design the services and their control model in such a way that feature interactions can be avoided or reduced. The modularized feature and single point control rule in IN CS-1 was defined to restrict certain feature interactions. According to this rule, only one feature can control a call at any time and which feature controls the call depends on its priority. Object oriented language is ideal candidate to implement these features. For example, each feature can be implemented as an object. The Generic Services Framework developed by Nortel is such an attempt [Gregoire 97]. It supports a variety of call models including IN and Integrated Service Digital Network (ISDN). The Distributed Feature Composition architecture (DFC) developed by AT&T is another attempt to modularize the functions based on component-oriented programming [Jackson 98]. Each feature in DFC is an independent module and is designed based on Pipe and Filter pattern. The features are assembled in a pipeline fashion. The precedence rule is used to guide the assembly of the features. DFC will be discussed in details in Chapter 3.

The performance of modularized software is often not as good as that of non-modularized software because of overhead on module interface. However, modularized software has many advantages over non-modularized software such as reusability, easiness to program and understand. Therefore, it is important to study the performance of the new service architectures to determine if the performance is good enough, and to locate the performance bottleneck.

2.2.2 Detection of feature interactions

When a new feature is added to the network, it is necessary to check its interactions with other known features in the network so that unwanted feature interactions can be reduced. This is usually done at the feature specification time saving large effort in implementation of new features with potential bad interaction with the existing features.

A large amount of work has been done in the area of feature interaction detection. A few tools for detection of feature interactions have been developed, for examples, Promela by AT&T [Jackson 98] and LOTOS by Ottawa University [Kamoun 98]. The SCF3 (Service Creation Form, Fit and Function) tool set developed by Telcordia (formerly Bellcore) has following seven steps: feature requirements definition, feature specification, feature verification, feature validation, feature interaction analysis and feature interaction management for feature creation and reduction of feature interaction [Cameron 98b].

2.2.3 Resolution of feature interactions

When a bad feature interaction between a new feature and the existing features occurs, it has to be resolved. In some cases, only a small adjustment of the precedence of the features may solve the problem. However, in some cases, the newly developed feature has to be re-engineered or both the newly developed feature and the existing features need to be re-engineered.

2.3. Summary

IN is the predominant architecture in the traditional telephone network. A large number of feature interaction cases in the above networks have been reported. Several strategies have been developed to avoid, detect and resolve feature interaction problem. DFC is an alternative to IN and goes beyond IN as DFC also accommodates those features which are specific to the IP telephony features.

Chapter 3. Distributed Feature Composition (DFC) Architecture and ECLIPSE Project

Distributed Feature Composition (DFC) is an architecture for the description of telecommunication services developed by AT&T. It was designed to modularize features, structure feature composition, analyze feature interactions, and separate the signaling layer from the media transmission layer [Jackson 98, Bond 00]. DFC architecture is implemented in Extended Communication Layered on IP•Synthesis Environment (ECLIPSE) project. “The primary goal of the ECLIPSE project is to determine whether DFC can serve as the service architecture for the next-generation telecommunication networks” [Bond 00]. To be a successful service architecture, it must accommodate various kinds of features including value added features from traditional networks and new IP telephony features, and manage them well. In this chapter we will review the DFC architecture and ECLIPSE.

3.1. DFC Architecture

3.1.1 Components

There are four kinds of components in DFC architecture: feature boxes (F), interface boxes, routers and data as shown in Figure 3.1. The boxes can also be categorized into free and bounded boxes. A bound box is a unique, persistent, addressable box. A free box is an anonymous, interchangeable copy of its type with no persistence outside its tenure in a customer call.

3.1.1.1 Feature boxes

Features are the core of the DFC architecture. They are designed based on a Pipe and Filter pattern. In the Pipe and Filter pattern, a pipe is used to pass messages and a filter processes the messages. In DFC, a feature is implemented as an independent module, feature box (F) with a number of ports, shown as black dots in Figure 3.1. A feature box can be a free box or a bound box depending on which interface it implements.

3.1.1.2 Interface boxes

There are two kinds of interface boxes: line interface (LI) boxes and trunk interface (TI) boxes. A line interface box is an interface to an end user device. A trunk interface box is an interface to the other telephone systems such as PSTN. All interface boxes are bound boxes.

Figure 3.1. Components in the DFC architecture[Bond 00].

3.1.1.3 Routers

A router in the DFC network is different from a router in the normal communication network. It not only routes calls to the destinations as any network router does, but it also assembles features on the signaling path from the caller to the callee.

3.1.1.4 Data

The process to establish a connection between the caller's LI box to the callee's LI box needs data on feature subscriptions, feature precedence, the dialing plan and configuration data as shown in double rectangles in Figure 3.1. Operational data in Figure 3.1 are used by feature boxes in their operations. Each feature can only access its own operational data preset by its subscriber and network operator. This strict partition of the operational data by features keeps feature modularity.

3.1.2 Routing and Feature Composition

Routing in DFC refers to establishing a route through boxes to process the connection setup.

3.1.2.1 Usage

In DFC, a call scenario is named as a usage in which boxes and internal calls are dynamically assembled. A usage can be presented as a graph with all the call parties involved, the features they are subscribed to and the message flows among them. The dynamic assembly of boxes and internal calls in a usage have great advantages including that feature boxes are largely independent, feature boxes can be composed freely in many different combinations, the set of features is easily extended, the

interactions between features can be analyzed with relative ease. A usage is presented in Figure 3.2.

Figure 3.2. A DFC usage.

Bound boxes are presented as double bordered boxes in Figure 3.2. Free boxes are presented as single bordered boxes in Figure 3.2.

A feature box has full control of all the signals it sends or receives. It contains all the service logic and processes the signals received without external assistance. If a feature box has no function, it behaves transparently, i.e., the feature box receives the signal and passes the signal to the next box without processing the signal. When an action triggering signal received, the related feature box absorbs and processes the signal, generates new signal (s), re-routes calls and/or processes media streams.

In Figure 3.2, DFC internal calls are expressed as arrows from the port of line interface box A (caller) to the port of line interface box B (callee) and connect all the feature boxes between them. In a usage, there are usually three logical zones: source, dialed and target, where the features are assembled. A call signal goes from the source, via the dialed and to the target along the signaling path. These zones can be physically distributed or on the same physical device. In Figure 3.2, user A subscribes to an Outgoing Call Screen (OCS) feature at the source zone and Credit Card Call (CCC) at

the dialed zone and user B subscribes to a Voice Mail (VM) and Call Waiting (CW) features at the target zone.

3.1.2.2 Precedence Rule

The features in each usage are assembled according to the precedence rule. The precedence rule sets a unique priority for each feature and is the only constraint in the DFC where the features are explicitly related to one another. The feature that has a higher priority absorbs and reacts to the triggering signal earlier.

3.1.2.3 Routing and feature assembly

In Figure 3.2, a call starts when the caller port of line interface box A sends a setup message to the DFC router on which it runs. After receiving the setup message, the router acknowledges to the sender and computes a route based on DFC routing algorithm. The setup message has five fields of interest to the router, source: address, dialed: string, target: address, command: {new, continue, update}, and route: sequence of (**seq**) routing_pair. Each routing pair has a first component of type box_type and a second component of type zone = {source, dialed, target}.

How to route a call will be explained by an example. The setup message sent by LI A has a source field containing userA, a dialed field containing the dialed string, and a command field containing new. The other two fields are empty. Upon receiving the signal, first the router extracts the target line interface box address userB from the dialed string, and puts it into the target field. Then, the router computes a route and puts it into the route field. Assume that user A subscribes to one feature OCS in the source zone, so the first one pair of the route is (OCS, source). Since Credit Card Call involves the database operations, CCC is deployed in the network where the database server runs.

The next pair of the route is (CCC, dialed). Assume that user B subscribes to two features VM and CW at the target zone. Both of them are busy signal treatment features. The order of these two features reacting to the busy signal is crucial. Here we assume that user B is already in a conversation with another user C. If VM was assigned a higher priority than CW, VM would react to the busy signal earlier than CW. VM would ask user A to leave a message to the user's mailbox. CW would be blocked causing bad feature interaction. If CW is assigned a higher priority as shown in Figure 3.2, CW will react to the busy signal earlier than VM. Upon receiving the busy signal, CW will send a signal to LI box B indicating that a call is coming so that user B can switch between user A and user C. Now a route has been built. The router routes the internal call from current box to the next box. It strips the first pair off the route, and since OCS is of the type free box, it routes the internal call to an arbitrary fresh box of that type. The OCS and CCC feature boxes Fig. 3.2 have no need to re-route the call. So each box prepares a setup message for an outgoing call by copying the entire setup message from its incoming call, and changing the command field from new to continue. The continue command tells the router not to re-compute the route. As the chain unfolds, one pair of the route is deleted and each free box is added to the usage. Finally, in the last internal call, the route is empty so the router routes to the bound box LI B.

From the call setup phase until the call teardown phase, the call has a two-way signaling channel. The call can also have any number of media channels. Each carries one medium.

3.1.2.4 Signaling/Media Separation

DFC separates the signaling path from the media path, and this work describes only the signal path. Media path is chosen as the shortest path between the caller and callee.

3.2. ECLIPSE

Extended Communication Layered on IP•Synthesis Environment (ECLIPSE) project is the implementation and evaluation of DFC architecture. This thesis considers the first ECLIPSE prototype as it existed at the end of 2000.

Figure 3.3 Components in ECLIPSE network.[Bond 00]

3.2.1 Implementation of DFC architecture

There are two layers: the signaling layer and the media layer, in the ECLIPSE implementation of DFC. Bandwidth shortage is the major concern in many multimedia Internet applications. Therefore media transmission follows the shortest possible path. Signaling is less expensive in term of use of bandwidth, therefore the signaling path may follows a different path from the media path. The signaling layer controls the media layer through well defined interfaces. The interfaces between the signaling layer and media layer are designed to be stable. The software components on signaling layer are implemented with Java. The software components on the media transmission layer are implemented using C. Figure 3.3 presents all the components in ECLIPSE network: features (F1-6), routers (A-C), line interface box (LI), trunk interface box (TI), provisioning server and media boxes (MB). The heavy dashed lines separate the nodes.

3.2.2 Implementation basics

Box class hierarchy

To simplify the design and implementation, significant abstractions are used in ECLIPSE. The box is the most important concept and basic unit in DFC/ECLIPSE. A box class hierarchy encompasses interface and feature boxes as shown in Figure 3.4. Dbox class is the super class of all box classes (here D is from DFC). Each DBox has several ports to communicate with other components in DFC. It creates one or more threads to perform call logic and port operations. A DBox will be garbage collected when all its threads have returned and all port connections have been destroyed. This situation can occur when the box port of a box receives a Reset message, or when the box's operations complete normally.

The Dbox class is extended by line interface box (LIBox), trunk interface box (TIBox), resource interface box (RIBox) and feature box (FBox) classes. Resource interface box is newly introduced in ECLIPSE. It provides an interface to media-processing resources such as announcement players, message recorders and text-voice converters.

Feature boxes are the most important part of the DFC/ECLIPSE. There are two interfaces: free feature box (FFBox) and bounded feature box (BFBox) as shown in Figure 3.4. Each feature box implements one of these two interfaces depending on the nature of feature. It can be a free feature box or a bound feature box.

For example, call forwarding on busy feature box (CFB) implements FFBox. It is a free feature box. While call waiting feature box (CW) is a bound feature box, it is bound to the subscriber's address and is not interchangeable.

Figure 3.4. Box class hierarchy.

Port class hierarchy

Ports are used in the communications between the components in the ECLIPSE network. A port class hierarchy is shown in Figure 3.5.

A port is an interface specifying the basic input and output methods. To detect errors and possible problems of the ECLIPSE system, the message flow between the software components is monitored by embedding monitor functions in these input and output activities.

Figure 3.5. DFC port class hierarchy.

- A BoxPort of the feature box is used to receive message from the router. It is the peer port of the switch port of the router.
- A SwitchPort of the router is used to receive messages from the boxes and send messages to the boxes.
- LinkablePort is a super class that holds the common characteristics of callee and caller ports. The LinkablePort has a finite state machine keeping track of the communication state and reacts to the signals received properly.
- CallerPort implements the Caller interface. It sends messages to the next box along the calling path or the callee's LI box, as shown in Figure 3.6.

- CalleePort implements the Callee interface. It sends messages to the last box along the calling path or caller's Lbox.
- Dual port implements both Caller and Callee interfaces, thus it can be used as a CallerPort or a CalleePort at different time.

Figure 3.6. Relationship between the CallerPort, CalleePort, BoxPort and other objects. LF is the last feature and NF is the next feature on the calling path.

Finite state machine class hierarchy

A finite state machine (FSM) is the core of feature boxes. The class hierarchy of the finite state machine is presented in Figure 3.7.

FSMOperations is an abstract class specifying the common operations of FSMs. Finite state machine is a next level abstract class extending FSMOperations. It supports both internal transitions and message transitions.

Figure 3.7. F S M class Hierarchy.

A message transition is triggered by receiving a triggering message. An internal transition is the transition that is not triggered by any messages. It happens automatically when certain state changes without external influence. Either type of transitions can define an optional guard condition based on the values of state variables alone and/or the value and type of the message in the case of message transitions. The guard condition for a transition must be evaluated to be true before the transition occurs. The transition can invoke any actions such as, update state variables, output messages, etc., when the transition fires. States can define actions to be performed just before the state is entered and after the state is exited.

Nested FSMs can also be defined for states. The current state get in a nested FSM after its parent state's entry action has been performed and before its exit action is performed.

Figure 3.8. A simple FSM.

Figure 3.8 shows a simple FSM. Initial state is state1. Before FSM enters this state, entry action, action1 is performed. If the triggering message arrives and guard condition evaluates to true, then exit action, action2; transition action, trans-action; and entry action, action 3 are fired consecutively and FSM enters state2. Through an internal transition, the current state changes from state2 to state3. Before exiting, the FSM fires action4.

3.2.3 Supporting Data

To support the functions of the components in ECLIPSE network, the following data are defined.

Subscription data

Subscription data is implemented as a hash table that stores the information of the customer, the features he/she subscribed to and the operational data for these features. Each router only stores subscription data for the customers associated with the router. The provisioning manager stores subscription data for all the customers.

Precedence

The features in a usage are ordered according to the precedence of the features. The precedence of a feature is depends on the characteristics of the feature and its relationship with the other features. The precedence data for a series of features is global data stored in both the router and the provisioning manager.

Dialing patterns

Dialing patterns are the few patterns that the consumers used when they are dialing for special services. For example, if we use a 1-800 number service, we need to dial a 1-800xxxxxxx. Dial patterns are also stored in both router and the provisioning manager.

Configuration data

Configuration data is the information about the association of line interface boxes and trunk interface boxes with a specific router. There is a configuration data for provisioning manager too. It stores all the information about the routers and line and trunk interface boxes.

3.2.4 Feature Boxes

Each DFC feature is modularized as a feature box with one or more FSMs, a number of ports and its operational data. Customized UML FSM diagrams are used in the feature design in ECLIPSE. Each feature box is expressed as the composition of several FSMs in sequent order and/or nested fashion. The Out-going Call Screen (OCS) is used as an example to show how a DFC feature is built. OCS feature extends FB and implements FFBox. It consists of a OCS-FSM, a BoxPort, a CallerPort and a CalleePort. The OCS-FSM is presented in Figure 3.9. OCS-FSM has four states: Start, Link, Transparent and End.

At the Start state, after the box port of the feature receives a setup message from the router, if the address is on the black list, the FSM blocks the call by sending back an busy signal, Upnack through the callee port and the current state changes to End state. If the address is not on the black list, the FSM let the call continue by sending an Upack message through the callee port to acknowledge that the setup message has been received. The caller port then sends out the “continue setup” message and the current state changes to the Link state.

At the Link state, if the caller port receives the Upnack message from the next feature on the call path or LIbox, the calleePort will pass the Upnack to the last feature and the current state moves to End state. If the caller port receives an Upack message, the current state moves into the Transparent state.

Figure 3.9. OCS-FSM.

After entering the Transparent state, the OCS-FSM creates a nested TransparentFSM, which has only one state: TransparentPass state and allows the DFC protocol messages to pass through the caller and callee ports transparently as shown in Figure 3.10.

Figure 3.10. TransparentFSM.

At the Transparent state, if either the caller port or callee port receives a teardown message, the current state changes to the End state.

In the End state, nested CallerTearDownFSM and CalleeTearDownFSM are created to tear down the caller and callee ports.

3.2.5 Interface Boxes

Line-interface and trunk-interface boxes are addressable and persistent in ECLIPSE. Each has an address and runs all the time on the router during its lifetime. They are created and destroyed independently from the usages.

3.2.6 Media Boxes

Media box is a media switch box where media is processed. It is implemented based on Megaco protocol. One media box can support one or more routers.

3.2.7 Routers

Each router supports a subset of all ECLIPSE network customers and only holds data of that subset. The routers relevant to the same customer call work together to compute the logical routes and route the call from the caller to the callee through the feature boxes on the path.

In ECLIPSE, the feature boxes are created and deleted responding to the start and end of a call by a box factory and a clean up function in the router. The code for each feature box is stored in the central server. It is dynamically loaded to the routers when the feature is needed in a call at the first time and stays there afterwards. An instance of the feature box is constructed from the loaded code using Java Reflection. Each feature box runs as an independent thread and communicates with each other using port-to-port signaling through unbounded queues.

Feature boxes are categorized into free box type or bound box type. The box factory of the router treats a free box and a bound box differently. For a free box, a new instance is created when a setup signal is received. For a bound box, the box factory checks whether the box is already instantiated; if so, the box is sent the relevant setup signal, and if not, a new box is created. After the box is created, it is put in route according to the precedence rule. Finally all the feature boxes are assembled along the call path.

3.2.8 Provisioning Manager

The provisioning manager consists of a central data server, a web server and a administration monitor.

Central data server

The central data server provides the data support to the ECLIPSE network. The data comes from the registration data submitted by the router when the router instances are created, and from the registration data submitted by the line interface boxes and trunk interface boxes. Customer's feature subscription data is another source of data when user subscribes features through the web server. Other data such as precedence and dialing pattern are global data that exist since the start of the provisioning manager. Part of the data is loaded to the router when the router needs the data.

Web server

The web server allows the customers to subscribe to the telephone features and set parameters for the features. It may also allow the customers to pay the bills on line.

Administration Monitor

Monitoring is a very important part of ECLIPSE system. It provides information to the tasks such as fault management, security enforcement, performance analysis,

visualization, billing, customer care, and data mining. ECLIPSE system has developed a single, scalable, flexible, extensible monitoring architecture.

To support scalability, the monitoring architecture is designed to be fully distributed and hierarchically structured. The monitor's probe is located in every box and the monitors are organized hierarchically, including a central monitor monitoring all the system. To support extensibility it is possible to define monitor's subclasses for monitoring and handling new events. A number of the ECLIPSE programming abstractions provided for service developers already have monitors embedded in them. For example, probes in the Box and Port abstractions notify a monitor when a box or port is created or destroyed. A network visualization subsystem is developed in ECLIPSE. This visualization system provides a dynamical picture of a DFC usage as it evolves over time. It is very important for diagnosis of the new features added to the system.

3.3. Call Examples

The call examples introduced here will be used as scenarios for creating performance models.

3.3.1 Featureless call (Basic Call)

In a featureless call, both parties do not subscribe to any features, the same as basic call in IN. Party A calls Party B. The router routes the call from Party A to Party B without connecting any features as shown in Figure 3.11.

Fig. 3.11 Two party featureless

Figure 3.12 is a sequence diagram for a basic call. For the convenience in performance study, the diagram is divided into five steps: Setup, Routing, Connecting, Talking, and Clean up. In the setup step, the dual port of Party A LI Box sends a Setup message to the router. In the routing step, the router computes a route to Party B and sends a Setup message to Party B. After receiving the Setup message, Party B LIBox returns an Upack message to Party A LIBox. In the connecting step, Party B sends an Alert message to the Party A and then sends a Connect message to confirm the connection between the two parties. In the talking step, a media conversation between Part A and Party B occurs. In the clean up step, after the media conversation, Party A hand up the phone that causes Party A's LIBox to send a Teardown message to Party B. Party B acknowledges with Downack message.

Fig. 3.12 Signaling Sequence for a two-party featureless call

3.3.2 Transparent feature call

Figure 3.13 is a usage diagram for a customer call. Party A subscribes to a Transparent feature and Party B also subscribes to a Transparent feature. Transparent feature does nothing but simply passes the signal to the other feature boxes or line interface boxes. It is the simplest feature in DFC. It does not have real value in terms of providing services for the users. However it has significant value for the performance

evaluation because it is simple and many DFC features can be implemented by slightly modification of the transparent feature, or can use the transparent feature as a part.

Figure 3.13 A usage with Transparent features.

The FSM for Transparent feature is shown in Figure 3.14. Comparing Figure 3.14 with Figure 3.9, we can see, OCS-FSM is a FSM from TransparentFeatureFSM with a slight modification.

Fig. 3.14 TransparentFeatureFSM

Figure 3.15 is a sequence diagram for a feature call with Transparent feature. For performance study convenience, the diagram is also divided into five steps: Setup, Routing, Connecting, Talking, and Clean up. In Setup step, the dual port of Party A LI

Fig. 3.15 Signaling sequence diagram for two party call with 2 transparent boxes

Box sends a Setup message to router A. In routing step, router A computes a route to Party B and sends a Setup message to Transparent box TA. After receiving the Setup message, TA sends a Upack message to Party A LIbox, then sends a Setup message router B. After router B receives the Setup message, it passes it to Transparent Box TB.

After receiving the Setup message, TB returns a Upack message to Transparent Box TA, then sends a Setup message to router B. The router B in turn sends the Setup message to Party B LIbox. Party B LIBox returns an Upack message to TB. In connecting step, Party B sends an Alert message to the Party A through TB and TA, and then sends a Connect message through TB and TA, to confirm the connection between the two parties. In the talking step, a media conversation between Part A and Party B occurs. In clean up step, after the media conversation, Party A hand up the phone that causes Party A's LIBox to send a Teardown message to TA. After TA receives the Teardown message, it returns a DownAck to Party A then passes the Teardown message to TB. After TB receives the Teardown message, it returns a DownAck message to TA and then passes the Teardown message to Party B LIbox. Party B acknowledges with Downack message.

3.4. Comparison with IN

In DFC, features communicate with each other using only featureless internal calls, that guarantee that features are independent of each other. All the signals about the signal and media channels go through each feature box in the usage, so each feature box can process its triggering signals if the box receives the signals. The feature interacts with the media channel with well defined interfaces. The feature box has not to change when its environment changes, for example, by the addition of some new features. The precedence rule is the only constraint to guide the feature composition which makes feature interaction analysis much easier and also makes the system more extensible and evolvable.

As described earlier, IN defines 4 planes, which makes the creation and integration of new features is more complicated. Another important difference between IN architecture and DFC is that features are deployed centrally in IN, while features can be distributed all over the network even on the end user devices.

3.5. Summary

A distributed, stable, flexible and media efficient IP telephony system based on DFC architecture has been developed at application level. In this system, the signaling layer and media layer are separated. The software components in the signaling layer are implemented with object-oriented language Java. The media layer is implemented with C. The media path in ECLIPSE follows the shortest possible path between the two end points.

The features are modularized as independent feature boxes. Features used in each call are assembled in a pipeline fashion. The creation and integration of features in DFC is much simpler than in IN and the features are deployed more distributive.

Even through the modularity of DFC/ECLIPSE system and pipeline fashion assembly of features has many advantages, it may also bring in certain amount of the overhead to the performance of the system. Therefore study of the performance of the system is of importance.

Chapter 4. Introduction to Layered Queuing Network (LQN) Models

The layered queuing network (LQN) model is very useful in performance analysis of software systems. A number of works on the LQN theory and its applications have been published [Franks 96, Rolia 95, Sheikh 97, Shousha 98, Woodside 95]. In this chapter, LQN is briefly introduced with a simple example. The main ideas, notations, tools of LQN, and general steps to build LQN model for a software system are presented.

4.1. Basic ideas of LQN

The Client–Server paradigm is very common in software architecture. Within this architecture, the client sends service requests to the server. The server queues the requests according to certain discipline and processes them. A typical example is Online Ticket System, as shown in Fig 4.1. Users buy tickets and check ticket status using the system. A number of user terminals may send requests to the software ticket agent in the system simultaneously. The agent processes the requests with first-come-first-serve discipline. In this case, the user terminal is the client and the ticket agent is the server. The agent may send queries to ticket database for ticket status or update ticket status database in response to the user’s requests. In this case, the agent is the client, and the database is the server.

Fig. 4.1 Order tickets online line

A Layered Queuing Network model (*LQN*) is based on the Client-Server paradigm. It provides performance parameters about the system such as response time, throughput and utilization of the system. In the LQN model, a server is named as resource and service queries are named as interactions. The basic idea of LQN model is describing the system as a set of resources that are ordered into layers by request precedence. All possible sequences of requests in a LQN model must be acyclic. For example, in the Online Ticket system, there are two resources, the ticket agent and database. We can order the ticket agent as first layer, the database as second layer in Online Ticket system

4.2. LQN Notations

There are four key notations: task, resource, interaction and entry in LQN model. They are defined as following:

4.2.1 Task

A task is any software object that provides operations and executes on its own thread. There are three kinds of task:

- Single thread task: Only one request can be served at a time. For example, the ticket database in the on line system can only have one thread to update the database and synchronize the update operation.
- Infinite-threaded task: The task has infinite copies, each with its own context, e.g. there is no resources constraint on such task. When a request is received, a thread is created to serve the request. Infinite threads will be created if there are infinite requests.
- Multi-threaded task: The task has limited number of copies, e.g. there is a thread pool for such kind of task. Each with its own context. When a request is received, if there is an idle thread in the thread pool, the request can be served immediately, if there is no idle thread available in the thread pool, the request has to be queued until a thread is available. For example, the ticket agents can have several thread process requests from different users at the same time.

4.2.2 Resource

Resources in LQN can be software resources – tasks, and hardware resources – devices. A device could be a CPU or a disk.

4.2.3 Interaction

A request in LQN is an interaction between resources. There are three kinds of interaction:

- Synchronous call: the sender is blocked for a reply. The receiver must reply.
- Asynchronous call: the sender does not wait to receive the reply. The sender sends a request and return.
- Forwarding: the first sender sends a synchronous call, but the receiver does not reply, but forwards the request to a third party, which either replies or forwards it further.

4.2.4 Entry

An entry is the interface of a task to receive a service request. In other words, a task offers services through the method called an entry. A task can have more than one entry, for example, the ticket agent task has two entries: `provideTicketStatus` and `sellTicket`.

In the case of a synchronous call, to release the sender as soon as possible, the receiver sends the reply before it finishes the service entirely, and then finishes the remaining part of the service. The entire service execution can be in some cases divided into first phase one and second phase. In the first phase, the entry service is executed before the reply. In the second phase, the remaining service is executed after the reply.

4.3. Building a Model

To make the performance analysis of a real world software system, a LQN model has to be built based on the real world software system. Here are the steps to be followed to build the LQN model:

4.3.1 Identify user tasks

Any software system in the real world must have some inputs (entries) either from users or triggering devices. These entries are called User tasks. The top level of the LQN model is always the User task. The user task in the online ticket system only has one entry – `sendQuery`. The LQN model for this step is shown in Fig. 4.2. The system is presented as system submodel in this LQN model.

Fig. 4.2. LQN model with User and System

4.3.2 Identify the services of the system

This step is to identify what kind of services the system could provide. For example, the Online Ticket system provides two kinds of services: query the status of the ticket only and buy tickets online.

In LQN model, services are categorized into classes. For example, there are two classes, QueryOnly class and Buying class in the Online Ticket system corresponding to Usage1 and Usage2. There is one submodel for each class in the LQN model. The sendQuery entry of User task (as shown in Fig. 4.2) sends a fraction of the total queries. With several classes, we can assign a fraction f_n for queries of class n , and show the queries together as in Fig. 4.3.

Fig. 4.3 LQN model for Online Ticket system with User and class sub model

4.3.3 For each class, identify the action sequences to complete the services

The purpose of this step is to give the LQN model builder the means to get the further performance matrices of some interesting part of the service process. The interesting part of the service process could be a common action shared by different classes, an action happens frequently, an action visible by the user, etc. How to divide the process for completing the service into action sequences, for example how many actions, what is each action for, is up to the LQN model builder.

For example, for the Usage1 and Usage2 in the Online Ticket system, the action sequences could be:

- Usage1:

The users query the ticket agent about ticket status (any left, price etc.) of a movie (query action).

- Usage2:

The users query the ticket agent about ticket status (any left, price etc.) of a movie (query action). The user sends Buy ticket message to the agent. (buy action)

The query action is shared by Usage1 and Usage2, and how fast it is will be evident to the user, So make querying ticket status as an individual action is a good decision.

In the LQN model, each action can be an individual task with one entry in the model as shown in Fig. 4.4. The action task is a pseudo task. The CPU demand of the entry in the action task is zero. The rounded rectangle to represent an action task is a new notation introduced in this work.

Fig. 4.4. LQN model for Online Ticket system with User and action task submodel

4.3.4 For each action task submodel, identify the queries of this action model to other entries of software tasks and hardware tasks, and the further queries of these entries

For example, QueryAction sends query to provideTicketStatus, the entry of TicketAgent task. Buying action sends query to provideTicketStatus and sellTicket, entries of TicketAgent task, and the sellTicket further sends query to the 'updateTicketStatus' entry of Database task. Finally the complete LQN model for the Online Ticket system is shown in Fig. 4.5. The tasks shown as rectangles with sharp corners represent software tasks.

Fig. 4.5. Complete LQN model for Online Ticket system

4.3.5 Gathering input data for LQN model

To get the performance metrics from the layered structure, we also need to provide average behavior of the entries by giving the resources demand for all the phases of an invocation of the entry.

- Execution demand: the average CPU demand for each entry.
- Demand for other resource services: the average number of calls to other entries.
- Waiting delay (e.g. think time, it is optional, it can be used to model any delay that does not occupy the processor.)

4.4. LQN Tools

Two LQN tools have been developed at Carleton University.

- Layered Queuing Network Solver (LQNS): solves the LQN models analytically.
- Stochastic Rendez-Vous Network Simulator (ParaSRVN): simulates LQN models using the PARASOL simulation system.

Both of the above tools use the same input and output file format. The performance metrics for each entry or task include:

- Throughput: jobs finished in a unit time.
- Service time:
 - for entry: its execution time and blocking time for single request (blocked by nested lower layer service to complete).
 - for task: average (weighted by frequency) of the entry times is the mean service time of the task.
- Utilization of resources: ratio of resources busy duration to finish a task to resource free and busy durations. For multi-threaded task, it is a ratio of the mean number of busy threads to whole number of the threads.

4.5. Summary

The LQN model closely matches to a wide range of real world software systems, and represents software and hardware resources in a natural way. The LQN model for a real world software system can be built with little difficulty. LQNS and ParaSRVN

provide important performance parameters such as service time, throughput and utilization of resources.

A new approach has been described in this research, to model the behavior of a system with “action tasks” that are artificial containers for behavior parameters. The modeling of behavior by an action task will be described in the next chapter.

Chapter 5. LQN model for DFC

As shown in Chapter 4, to get accurate performance metrics of a real software system, the LQN model for the system has to represent the system. However, the number of existing services in the telecommunication network is quite large, new services are being invented and these services are distributed inside the network. It is very difficult to give one LQN model for each specific configuration. There are some common objects in each configuration of DFC/ECLIPSE network. Therefore in this chapter, we will provide a generic LQN model for DFC in which a few sub models are left to be customized when the specific configuration is available, and the rules to customize the models for different scenarios and distributed configurations.

A practical telephony system will have many features. However, in this work, LQN models for a two party featureless call (basic call) and a two party call with transparent features will be created. The reasons behind this are listed as follows:

1. Two party featureless call or basic call is the most common call.
2. The ECLIPSE software we used has very few features.
3. The transparent feature is the most representative in DFC architecture. All the features in a usage behave transparently except the only active feature and have the same common operations to handle the call signals as the transparent feature box.

5.1. Generic LQN model of DFC/ECLIPSE system

5.1.1 Steps to build a LQN model for a DFC system

A generic model for DFC is built by following the procedures set out in section 4.3 in Chapter 4.

Step 1: Set the name for the LQN model top user task

The top level of the LQN model is always the user task. Here the user task is named as “callScenario”. The callScenario task only has one entry - sendQuery

Step 2: Identify the classes of the system

A telecommunication system provides a large number of services. There are many scenarios running at the same time corresponding to these services. For example, teleconference, three party call, and two party call can run at the same router at the same time. As mentioned in Chapter 4, a particular class can be defined for each scenario, and all the classes merge together into one LQN model.

There is one sub model for each class in the LQN model. The sendQuery entry of callScenario task sends a fraction f_i of its queries to sub model i . If there are n classes and sub models , $f_1+f_2+\dots+f_i+ f_n=1$.

Step 3: For each class, identify the actions to complete the service

As shown in Fig. 3.12 and Fig. 3.15, a call consists of several phases such as setup and teardown. Here we call these phases as actions. For each of these actions in the call processing sequence, a action task is assigned. How to divide the call processing sequence into small actions can be quite flexible and is up to the model builder. However, it is better to divide the sequence into meaningful actions. In this

thesis, the call processing sequence is divided into four actions: setup, routing, connecting and clean up. For each of these actions, an action task called a “action task” is assigned, with an entry containing an action submodel.

Step 4: For each action sub model, identify the queries of this action to other entries of software tasks and hardware tasks, and the further queries of these entries

As mentioned in Chapter 3, there are three kinds of hosts: provisioning manager host, router host and user terminal host. There are router task, feature box tasks and provisioning manager task running on these hosts. Each of these tasks has some entries. Different classes may share some entries of these tasks.

For each of the step tasks, such as call routing step task, it needs the service of router task, provisioning manager task and feature box tasks. Different classes share some entries of these tasks. The detail of the sub model for each step task changes according to different scenarios and dividing methods for the steps in the same scenario.

5.1.2 ECLIPSE specific LQN model details

Next, we will give the details for building the sub models for the two call scenarios mentioned in Chapter 3. Besides the generic 4 steps in build LQN model for DFC/ECLIPSE systems mentioned above, some important details for the model are given here.

- Currently DFC/ECLIPSE is implemented using Java. Each Java application runs on a JVM. We can measure how much CPU an application method consumed. However we cannot determine how much CPU is used by the JVM to support each application method. This part of CPU is only fairly small part of total JVM we measured. We treat JVM as a background task, set it as second phase task called

directly by CallScenario. A different task is assigned to the JVM of each host respectively.

- There is a monitor process running in the system for fault management and billing purposes. As mentioned in Chapter 3, the events for monitoring purposes in each host are sent to the administrative monitor after the host processes the incoming messages, and the host doesn't need to wait for any reply from the monitor. Therefore the CPU demand of the monitor tasks is set as second phase.
- User terminal task is modeled as a LQN infinite threaded task running on infinite user terminal host, because each call party occupies one individual user terminal host and the number of users is unlimited in DFC/ECLIPSE. This gives an open arrival process.
- Each feature box is modeled as a LQN infinite threaded task. In the DFC architecture, there is no limitation on the number of usages. In each usage (scenario), feature boxes are created and the number of feature boxes is unlimited. There are not shared feature boxes between different usages.
- In DFC, there are three zones in a call: source zone, destination zone and dialing zone between caller and callee. It is natural to assign one router to each zone physically, even though it is not absolutely necessary.

Following the discussion above, a generic model for DFC is built, as shown in Figure 5.1.

Fig. 5.1 Generic LQN Model for DFC

After building the model, the next step is to collect CPU demands for the entries in the model, network demand and some system statistic data (population of the users for each class). Network demand equals message size divided by network speed.

5.2. Specific LQN models for two typical call scenarios

5.2.1 LQN sub model for the two party featureless call scenario

In this scenario (described in section 3.3.1), the callers and callees do not subscribe to any features. The LQN model for the system with two party featureless call only is built by assigning number of classes $n = 1$ and replacing the scenario1 in Fig.5.1 with the TwoPartyFeaturelessCall submodel shown in Fig. 5.2. The call scenario is broken into a sequence of four stages (Setup, Routing, Connecting and Clean) described by action tasks and running on virtual processor resources.

Fig.5.2 LQN submodel for two party featureless call with monitor system, to fill a scenario submodel slot in Fig 5.1.

5.2.2 LQN sub model for the two party call with Transparent features:

In this scenario, callers and callees subscribe to a number of transparent boxes. The LQN model for the system with two party call with Transparent feature only is built by assigning number of classes $n = 1$ and replacing the scenario1 in Fig.5.1 with the TwoPartyCallWithTransparentBoxes submodel shown in Fig. 5.3.

Fig. 5.3 LQN submodel for two party call with Transparent boxes, filling a scenario slot in the overall model of Fig.5.1.

5.3. Summary

In this chapter, the generic model for DFC is built. Some details for building the concrete LQN model for the DFC/ECLIPSE system are discussed. Two customized LQN models for two typical call scenarios, i.e. a two party featureless call and a two party call with transparent features, are built.

The entire model has not been developed as a single figure because of its complexity. However, it consists of the overall model of Fig. 5.1, with one or more of its scenario submodels filled with submodels that follow Fig. 5.2 and Fig. 5.3. In this work, just one of these scenario submodels is used, but in general they can be combined, using work load fractions defined by $f_1, \dots, f_n$ as shown in Fig. 5.1.

Chapter 6. Evaluation of Java Measurement Tools

Reasonably accurate measurement is essential for performance modeling. As mentioned in Chapter 5, to complete the LQN model for DFC/ECLIPSE, we need to know the CPU time consumed by method, by object, and by thread, and the number of times the unit (method, object and thread) is invoked during an experiment. To get these parameters, we have to choose a proper measurement tool to probe the system in the experiment. In this chapter, we mainly discuss the issues about choosing a proper Java measurement tool to measure those parameters of DFC/ECLIPSE.

6.1. Issues about choosing a Java measurement tool

From language point of view, the tool should support Java platform and the JDK version used by the application program. Java is a relatively new language that is updated very often. As of early 2001, JDK version had changed from 1.0, 1.2 to 1.3 within 5 years. Frequent updating of Java versions overcomes the defects and fixes the bugs in the previous version, which is no doubtfully beneficial to the Java programmers. However it increases the difficulty for Java measurement tool vendors to keep their tools compatible with the latest version of JDK. It also increases the difficulty for the user of the tool to choose the proper tool, and to update the tool according to the JDK version changes. For example, JProbe, a commercial java measurement tool, supports JDK 1.3.1, but not JDK 1.4 up to now. If the JDK version used in the implementation of an application program is 1.4, using JProbe to do the measurement of the application program may be problematic.

The measurement tool should be able to measure what the user wants to with desired accuracy, and with as little as possible disturbance to the system. Examples of measures are CPU time for a line of code, a method of the application program and memory occupancy by the tool etc. Different tool has different feature and different accuracy.

6.2. Current available tools

The following commercial tools were available when the evaluation was done in early 2001:

OptimizeIt

The full description of the tool can be found at www.optimizeit.com. It is a fully featured JVMPI compliant Profiler (Works with any JVMPI compliant VM). It has been integrated into Borland JBuilder IDE seamlessly. It has a lot of functionalities besides the performance measurements, such as memory leak detection. It has been recommended to be a good tool in some white papers.

JProbe

The full description of the tool can be found at www.klggroup.com/jprobe. It is a fully featured JVMPI compliant Profiler. It can measure the CPU time (number of CPU cycles) and elapsed (wall Clock) time for NT, but only elapsed time for Solaris.

TrueTime

The full description of the tool can be found at www.numega.com/products/perf/tt_Java.shtml. It is a fully featured JVMPI compliant Profiler. It can also profile native code and nominally be used in distributed computing.

Quantify

The full description of the tool can be found at www.rational.com/products/vis_quantify/index.jtmpl. It is a fully featured JVMPI compliant Profiler. It also profiles C and C++ code. The version we have in our lab only supports a Microsoft instrumented JVM.

These non-commercial tools were also examined:

JInsight

The full description of the tool can be found at www.alphaworks.ibm.com/tech/jinsight. It is a free non-JVMPI compliant Profile for JDK1.1.x. It only works with its own instrument VM.

Apache Jmeter

The full description of the tool can be found at Java.apache.org/jmeter. It is mainly WebServer scalability tester/exerciser.

Javiz

It works only for jdk1.1.5. It has some features for RMI, which none of the tools mentioned above have.

The advantages of the above commercial tools include: 1) Keeping relatively close pace with the updating of JDK version, 2) Providing CPU time for execution of a line of code, a method, a thread, many other features, a lot of user friendly graphic user interfaces. The disadvantages of the above commercial tools include: 1) The measurement tools themselves are too big and need a lot of memory for just running those nice GUIs, which slows down the ECLIPSE system running on the host. 2) In some cases, the tool vendor modified the JVM to get more information inside the JVM

that narrows the applying area of the tool, 3) The commercial tools are expensive. The non-commercial tools are usually out of date.

As mentioned above, none of the Java measurement tools are perfect for the measurement of the ECLIPSE project. We learned a way to get a reliable and light Java measurement tool inside the JDK package and performance analysis tool (PerfAnal) in [Mehers 00]. These tools are used for measurement of performance parameters of ECLIPSE system in this thesis. The principles and how to use the tools is discussed in next section.

6.3. Using JVM profiler for Java measurement

JDK includes a Java Virtual Machine Profiler Interface (JVMPI) for tools vendors to develop profilers of JVM. The JVMPI is a two-way function call interface between the Java virtual machine and an in-process profiler agent as shown in figure 6.1. On one hand, the virtual machine notifies the profiler agent of various events, corresponding to, for example, heap allocation, thread start, etc. On the other hand, the profiler agent can issue controls and requests for more information through the JVMPI.

Figure 6.1 Interactions between JVM and profiler agent.

HPROF is a relatively simple profiler agent shipped with JDK after version 1.2. It is a dynamically-linked library that interacts with the JVMPI and writes out profiling information either to a file or to a socket in ASCII or binary format. It is capable of presenting CPU usage, heap allocation statistics and monitor contention profiles. In addition it can also report complete heap dumps and states of all the monitors and threads in the Java virtual machine.

Even though HPROF provides a lot of useful information stored in the file, the information is very difficult to understand and analyze. PerfAnal is a free performance analysis tool coming with [Mehers 00]. PerfAnal analyzes the HPROF results dumped after a profiled application program terminates and presents the CPU time for thread, method and line of code in a graphical manner.

In the following section, we will evaluate HPROF and PerfAnal in detail. How to use HPROF and PerfAnal is shown in Appendix.

6.3.1. HPROF

The output of HPROF is different from version to version of JDK. This causes difficulty of measurement. For example, for CPU time measurement, some HPROF versions give details about each method and each thread. Some versions only give CPU time for the whole process.

The CPU time gathered from HPROF is in terms of ticks. With the help of PerfAnal, we can get CPU time in ticks for each thread and method. We have measured the consistency of conversion ratio of tick to real time, for example units of milliseconds. This measurement is called the consistency measurement here. We found that consistency of conversion ratio of tick to real time of HPROF is different from version to version of JDK. All the consistency measurements were made by running a simple loop for n times, and recording the start time and ending time using Java commend `System.currentTimeMillis()`. The same measurement is repeated 25 times. Some of the results are presented in Fig. 6.2-3. The X-axis represents the case of the measurement. The x-axis displays the case number from 1 to 25.

Fig. 6.2 jdk1.2.2 NT version consistency measurement

Fig. 6.2 shows that the CPU time measured in milliseconds using Java command, and in ticks using the HPROF, is relatively stable for JDK1.2.2 NT version. The same consistency is also found for JDK1.2.2 Unix version. However, the HPROF in JDK 1.2.2 Unix version only provides total CPU time for the whole process and does not provide details about the thread and the method.

Fig. 6.3 JDK1.3 UNIX version consistency measurement.

In case of JDK1.3.0 Unix version, the consistency of CPU time measured using Java command is good, but the consistency of the CPU time measured using HPROF is very bad as shown in Figure 6.3. It is impossible to calibrate the ticks. Therefore this version is not good for CPU time measurement.

From the above results, we can conclude that the measurement results from HPROF in JDK1.2.2 NT version are more consistent. Therefore performance measurement of ECLIPSE system is carried out on a Pentium III workstation with 766 MHz running JDK1.2.2 NT version.

6.3.2. PerfAnal

PerfAnal analyzes the text file dumped by HPROF after the Java process terminates. Therefore PerfAnal does not disturb the process as the commercial tools mentioned above do. The commercial tools mentioned above mainly interact with JVM

through JVMPI and some even do their own instrumentation of the JVM. They present the performance results using their GUI in real time. Each of these commercial tools is expected to consume much larger memory and CPU than HPROF. PerfAnal is easy to use.

PerfAnal has its drawbacks. It analyzes the text file dumped by HPROF after the measurement thus it cannot provide performance measurements in real time.

6.3.3. Calibration of the data from PerfAnal

Data from PerfAnal are gathered in terms of clock ticks that must be converted to the actual time values. As shown in Fig. 6.2, one tick corresponds to 1.01 millisecond. The same measurements are also done on NT workstations with 400MHz or 133MHz CPU. The results are the same as those shown in Figure 6.2, e.g. one tick corresponds to 1.01 millisecond.

6.4. Conclusions

Selecting a proper Java measurement tool was a complicated task. The commercial measurement tools are generally big and introduce some visible disturb to the application system they probe. How they modify JVM is unknown. The non-commercial tools are usually out of date. A simple and light HPROF JVM profiler shipped with JDK and its analysis tool PerfAnal are evaluated. HPROF can measure the CPU time for method, thread and line of code that are needed for performance measurement in ECLIPSE. Among different JDK versions, JDK1.2.2 NT version, the measurement results from HPROF /PerfAnal are consistent and reasonably accurate (1.01 millisecond). PerfAnal is a free performance analysis tool.

Chapter 7. ECLIPSE Measurement

This chapter reports the results of measurement of the CPU demands of the ECLIPSE network made with HPROF and PerfAnal. The measurement is carried out in two typical call scenarios: a featureless call and a feature call with transparent features.

7.1. ECLIPSE Configuration for Measurement

In an ECLIPSE network, there were the following hosts: one provisioning manager host, several router hosts and a large number of user terminals.

A provisioning manager runs on the provisioning manager host. It supports the routers, user terminals and calls by providing feature operational data, configuration data, subscription data, precedence data, dialing patterns and feature box code to the routers, and accepting registration from routers and user terminals or LI boxes, and recording various events from the routers and LI boxes for fault management. It interacts only lightly with individual call.

A router runs on a router host. It computes a route for each call, loads feature box code from the provisioning manager, creates feature boxes and connects all features caller and callee subscribed and reports events to the provisioning manager. It has the following threads:

- A router thread
- A router monitor thread.
- Feature box threads.

A phone line interface box runs on each user terminal host. It has threads:

- A phone line interface box thread.

- A box monitor thread.

In the measurements of the thesis, the ECLIPSE network was built with a provisioning manager, a router and a caller phone LI box and a callee phone LI box. To use the machines in the laboratory efficiently without sacrificing the accuracy of the measurement, when we did measurement of one of the above components, we deployed that component solely on a separate machine and the others on another machine. All the measurement of ECLIPSE system was carried out on the same Pentium III workstation with 766 MHz.

7.2. Measurement

The objects in the ECLIPSE system, provisioning manager, routers and LI boxes run like a client –server system. Each of them starts with initialization, then waits for inputs and then serves the inputs, then more waits and serves, finally the object shuts down, as illustrated in Figure 7.1. Here the inputs are the call signals. Therefore each object undergoes four types of phases within its lifetime: Initialization, Waiting, Serving and Shut down. Waiting-Serving cycles occupy almost all the lifetime of the object. The Initialization and Shut down phases only occupy ignorable amount of the lifetime.

Fig. 7.1 The sequence of CPU usage for router process

For example, in the initialization phase, the router registers itself to the provisioning manager and loads data from the central data server in the provisioning manager host. Then in its waiting phase, it waits for the call setup requests from user terminals. When the router receives a call setup request, the router processes the call request and sets up the feature box chain for the call that is the serving phase. After the router is initialized, it stays in either the waiting phase or the serving phase, until it shuts down.

In the measurement of two party featureless calls between the caller and callee, we found that the CPU time consumed by each process in the serving phase for the same call is not constant. Figure 7.2 shows that in a succession of two party featureless calls, the first few calls are more expensive in CPU time than the later ones for the router process. After a few calls, the CPU time consumed by each call becomes stable. This suggests that the measurement should only begin after an initial “warm-up” period. Generally speaking, the difference could come from processor caching, JVM optimization, and some “once forever” optimization of each process. A good “once forever” example is that the router creates the first feature box of a type, by loading the code of that type of feature box from the central data server. After this, for the same

type of feature box in different calls, the router uses the local copy of the code. Therefore we only use the CPU time measured at the stable part.

Fig. 7.2 CPU demand vs number of calls.

CPU time consumed in waiting phase by the daemon processes is very small for all the objects, so it is ignored.

7.3. Measurement Results

We did the measurement on two scenarios: a two party featureless call and a two party feature call with transparent feature boxes, described in section 3.3 in Chapter 3. To calibrate the CPU overhead for each object in the system such as caller and callee terminal, router and provisioning manager in initialization and shut down phases shown in Fig.7.1, in each scenario, the whole system was set up and down 5 times without making any calls. The CPU consumed by each object in the system was measured. In second set of measurements, the warm-up overhead for the calls in each of the above scenarios was calibrated by making 1 call. The measurements were repeated 5 times. In

the third set of measurements, 5 sequential calls were made in each of the above scenario. The measurements were repeated 5 times. The final measurement results presented here are the average values measured in the third set of measurements minus the corresponding average values in the first and second set of measurements.

7.3.1. Two-party featureless call

There are one caller terminal, one callee terminal, one router and one provisioning manager involved in a two-party featureless call (basic call). HPROF was used in profiling of each of objects in this scenario. The CPU time consumed by each of the objects in the call is broken down into pieces according to threads and methods, plus some unlogged “others”. Here “others” represent CPU time of those threads or methods running on one of the hosts in the scenario that consume so little CPU time that the HPROF cannot allocate the threads or the methods. When we calculate the final parameters for the modeling, we take “others” into account. The results measured on each host are listed here. To make citation easier, abbreviations of the names for the measured results are also listed in the tables.

Measurement results on Caller Host

The measurement results from PerfAnal are presented a measurement result tree as shown in Figure 7.3, because each task, thread or method can create threads or call other methods. The CPU time consumed by each task, thread or method is further divided into smaller pieces consumed by tasks, threads or methods at the sub level. The CPU time consumed by each task, thread or method is the sum of all the smaller pieces. Each of the smaller pieces can be farther divided until the leaf nodes in the tree. The abbreviations of the names for the measurement results on the caller host are listed in table 7.1.

The total CPU used by PhoneCliBox on caller terminal in the two-party featureless call and the CPU used by its key component PhoneCliBoxFSM were

measured. The “fire message transition” method of PhoneCliBoxFSM was called several times. The total CPU consumed by this method is called FMT. The fire message transition method further calls the CFSM method to create BusyFSM, sends a Setup message (Dport!Setup) and sends a Teardown message (Dport!Teardown) through dual port of the PhoneCliBox. The CPU used for each of these operations was measured. The CPU used in internal transition of PhoneCliBoxFSM to tear down the dual port (Dport.TearDown) was also measured. The CPU used by getNextMessage() and CPU used by GetNextEnabledMessageTransition() were also measured. The CPU consumed by JVM running on the user terminal and the CPU consumed by LI box monitor were also measured. They are put together and ascribed in the same task called LIBoxRun. In the above measurement, there are always some operations undetected by HPROF because their CPU consumption is too low. We call them “others”. The standard deviation for the total CPU used by the PhoneCliBox on the caller terminal across the 5 runs is 7.1%.

Fig. 7.3 CPU time relationships on caller terminal

Table 7.1 Definition of the abbreviations of measurement results on caller host

Abbreviation	Definition
LiBoxRun	CPU demand to support LiBox running on user terminal
LiJVM	CPU demand for user terminal JVM
LiBoxMonitor	CPU demand for event dispatching thread on user terminal
PhoneCliBox	CPU demand for user terminal interface feature box
FSM	CPU demand for Finite State Machine
FMT	CPU demand for FireMessageTransition method
CFSM	CPU demand to create busyFSM
DPort!Setup	CPU demand for the dual port sending Setup message
DPort!Teardown	CPU demand for sending Teardown message
DportTeardown	CPU demand for tearing down the dual port of LiBox
GNM	CPU demand for GetNextMessage method
GEMT	CPU demand for GetEnabledMessageTransition method
Total	Total CPU demand for making one call

Table 7.2 Results for caller user terminal

Abbreviation	CPU demand (millisecond)
LiBoxRun	99.5
LiJVM	53.53
LiBoxMonitor	45.97
PhoneCliBox	177
FSM	176.96
FMT	168.04
CFSM	109.11
DPort!Setup	46.9
DPort!Teardown	12.02
Dport.Tear down	1.67
GNM	2.54
GEMT	4.75
Total	276.5

Measurement results on Callee Host

The measurement result tree from PerfAnal on the callee host is presented in Figure 7.3. The abbreviations of the names for the measurement results on the callee host are listed in table 7.3. The performance parameters measured on callee host are listed in table 7.4.

The total CPU used by PhoneCliBox on the callee terminal in the two-party featureless call and the CPU used by its key component PhoneCliBoxFSM were measured. The “Fire Message Transition” method (FMT) of PhoneCliBoxFSM was called several times. The total CPU consumed by this method is FMT. The “fire message transition” method further calls CFSM method to create BusyFSM, sends a UpAck message (Dport!UpAck) and sends DFC protocol messages (Dport!DFCMsg) through dual port of the PhoneCliBox. The CPU used for each of these operations was measured. Alert and Connect messages are of DFC protocol message. According to the sequence diagram for two party featureless call, the CPU consumed by DPort!DFCMsg method is expected to include sending the above two messages. The CPU used in internal transition of PhoneCliBoxFSM to tear down the dual port (Dport.TearDown) was also measured. The CPU used by getNextMessage() and CPU used by GetNextEnabledMessage - Transition() were also measured. LiBoxRun is also part of total CPU consumed by the callee terminal. The standard deviation for the total CPU used by the PhoneCliBox on the callee terminal across the 5 runs is 9.2%.

Fig. 7.4 CPU time relationships on callee terminal

Table 7.3 Definition of the abbreviations of measurement results on callee host

Abbreviation	Definition
DPort!UpAck	CPU demand for dual port sending UpAck
Dport!DFCMsg	CPU demand for dual port sending DFC protocol msg
Dport!DownAck	CPU demand for dual port sending DownAck
Dport.Teardown	CPU demand for tear down dual port
GNM	CPU demand for GetNextMessage method
GEMT	CPU demand for GetEnabledMessageTransition method

Table 7.4 Results for callee user terminal

Abbreviation	CPU demand (millisecond)
LiBoxRun	99.25

LiJVM	40.62
LiMonitor	58.63
PhoneCliBox	173.75
PhoneCliBoxFSM	173.42
FMT	156.05
CFSM	105.2
DPort!UpAck	29.81
DPort!DFC	10.47
DPort!DownAck	10.57
DPort.TearDown	11.19
GNM	1
GEMT	5.51
Total	273

Measurement results on router

The measurement result tree from PerfAnal on the router is presented in Figure 7.5. The abbreviations of the measurement results on the router are listed in table 7.5. The performance parameters measured on router are listed in table 7.6.

7.5 CPU relationships on router

The total CPU used by router in the two-party featureless call was measured. The CPU used by JVM and by RouteMonitor was also measured and is ascribed as RouterBG. The CPU for run() method of the router node thread was measured. The run() further calls handelM() to handle the message. The CPU used by each of these two methods was measured. The standard deviation for the total CPU used by the router across the 5 runs is 9.9%.

Table 7.5 Definition of the abbreviations of measurement results on router host

Abbreviation	Definition
RouterBG	CPU demand supporting the router running
RouterJVM	CPU demand for JVM running on the router
RouterMonitor	CPU demand for router monitor to dispatch events
Run	CPU demand for the run method of router thread
HandleM	CPU demand for HandleMessage method
Total	Total CPU demand for the router

Table 7.6 CPU time of router for two-party featureless call

Abbreviation	CPU demand (millisecond)
RouterBG	24.06
RouterJVM	13.1
RouterMonitor	10.96
Run	7.3
HandleM	7.3
Total	31.36

Measurement results on provisioning manager

The measurement result tree from PerfAnal on the provisioning manager is presented in Figure 7.6. The abbreviations of the measurement results only on the

router are listed in table 7.7. The performance parameters measured on router are listed in table 7.8.

The total CPU used by provisioning manager in the two-party featureless call consists of mainly the CPU used by JVM and the CPU used by monitor manager and ignorable others. The CPU of each of these was measured. The monitor manager is responsible for receiving the events from routers and LI boxes and processing the events (mainly saving the events in the log files). The standard deviation for the total CPU used by provisioning manager across the 5 runs is 6.6%.

Fig. 7.6 Relationship of CPU time of provisioning manager node

Table 7.7 Definition of the abbreviations of the measurement results on the provisioning manager

Abbreviation	Definition
ProvJVM	CPU demand for provisioning manager node JVM
MoniMgr	CPU demand for provisioning manager node JVM monitor thread
Total	Total CPU demand for making one call

Table 7.8 CPU time of provisioning manager for two-party featureless call

Abbreviation	CPU demand in millisecond
ProvJVM	158
MoniMgr	57
Total	215

7.3.2. Two-party feature call with transparent feature boxes

There are one caller terminal, one callee terminal, one router and one provisioning manager and a number of transparent feature boxes involved in this two-party feature call. To examine the effect of the number of feature boxes on the performance parameters of the system, different number transparent boxes were launched on the router. The measurement results on the caller terminal and callee terminal in the current call are the same as those in the basic call since the operations on the caller terminal and callee terminal in both scenarios are the same. Therefore, only the measurement results on the router and the transparent boxes are presented here.

Measurement results on router

Figure 7.7 shows the measurement result tree from PerfAnal on the router. The notations used measurement results tree on the router are explained in table 7.9. Different number feature boxes were launched in the measurements on the router. In each measurement, a certain number of feature boxes were launched and a set of results were recorded. The total volume of results is large, so only selected values needed for the LQN models are listed here. The monitoring subsystem was not included in the measurement because it is very CPU intensive and is mostly used for debugging. It is too CPU intensive to be deployed into present form, so it was not studied.

The total CPU used by router in the two-party feature call with transparent feature boxes was measured. The standard deviation for the total CPU used by the router, across the 5 runs, with 2 transparent boxes, 10 transparent boxes and 20 transparent boxes is 3.7%, 1.8% and 1.1% respectively. The CPU used by JVM and by RouteMonitor was also measured and is ascribed as RouterBG. The CPU for run() method of the router node thread was measured and named as RouterRun. The run() further calls handleM() to handle the message from the message queue of the router. The CPU used by the handleM() was measured. The handleM() calls setup() method to setup the feature box

chain, the setup further calls createFBox() to create feature boxes needed for the feature box chain.

The CPU consumed by all transparent feature boxes was measured. Each transparent feature box has a TransparentBoxFSM and a house keeping method, boxRemove() to do the clean up job when the usage in which the transparent feature box was created, finishes. Both of the CPU used by the TransparentBoxFSM and the CPU used by boxRemove() are measured. The “fire message transition” method was called for several times to facilitate the following actions: sending a UpAck to the last box (Libox or Fbox) by the calleePort of the transparent feature box, sending a continue Setup message to the router by the calleePort and creating a TransparentFSM with CFSM method(). The CPU used by the “fire message transition” method (FMT) was measured and so are the CPU used by those actions and the FSM created within the method.

The CPU for sending DFC messages through callee port (CalleeProt!DFCMsg) by TransparentFSM was also measured. The DFC protocol messages are expected to be Alert and Connect messages according to Figure 3.15 In Chapter 3. RunNextInternalTransitionSequence() (RNITS) which further invokes teardown() method on caller port and callee port when the call finishes. The CPU for getNextMessage() method (GNM), and the CPU used in BoxRemove method that is called to do the cleaning up job after the call finishes, was also measured.

Fig. 7.7 Relationship of CPU times of Router for two party call with Transparent box.

Table 7.9 Definition of the abbreviations in the router measurement results for two party call with Transparent boxes.

Abbreviation	Defination
RouterBG	CPU demand supporting the router running

RouterJVM	CPU demand for JVM running on the router
RouterMonitor	CPU demand for router monitor to dispatch events
RouterRun	CPU demand for the run method of router thread
HandleM	CPU demand for HandleMessage method
Setup	CPU demand for Setup the feature box chain
CreateFBox	CPU demand for creating feature box method
TransparentBoxes	CPU demand for transparent box threads
TransparentBoxFSM	CPU demand for TransparentBoxFSM
FMT	CPU demand for FireMessageTransition method
Callee!UpAck	CPU demand for callee port of the TransparentBox sending UpAck
Caller!Setup	CPU demand for caller port of the TransparentBox sending Setup
CFSM	CPU demand for createFSM method creating TransparentFSM
TransparentFSM	CPU demand for TransparentFSM
Callee!DFCMsg	CPU demand callee port sending DFC messages
RNITS	CPU demand for RunNextInternalTransitionSequence method
Caller.Teardown	CPU demand for tearing down the caller port
CalleeTeardown	CPU demand for tearing down the callee port
GNM	CPU demand for GetNextMessage method
GEMT	CPU demand for GetEnabledMessageTransition method
BoxRemove	CPU demand for removing the feature box
Total	Total CPU demand for making one call

Measurement results on calls with transparent boxes

A number of calls with different number of transparent feature boxes were measured. We find that the CPU time consumed by each transparent box depends on its position in the feature box chain. The CPU consumed by the first feature box in the feature box chain running on the router and the CPU consumed by the last is different from the rest feature boxes as shown in table 7.10. They are much bigger. The rest feature boxes consume almost the same amount of CPU in the same operation. All the feature boxes run on the router and connect with each other. However the first and last feature boxes need to connect to the remote LI boxes besides connecting to the other

feature boxes running on the router. Connecting to a LI box running on another JVM, communicating with it and terminating the connection with it, the first and the last feature box consume more CPU time than the other feature boxes.

Table 7.10 CPU used by the operations in the Transparent feature boxes

Task	Position	CPU demand in (millisecond)
TransparentBox	First	23.8
	Last	8.0
	Middle	2.59
TransparentBoxFSM	First	22.4
	Last	7.3
	Middle	2.5
FMT	First	13.4
	Last	1.95
	Middle	1.54
Callee!UpAck	First	6.05
	Last	0.33
	Middle	0.5
Caller!Setup	First	0.34
	Last	0.66
	Middle	0.08
CFSM	First	1.67
	Last	0.3
	Middle	0.125
TransparentFSM	First	5.34
	Last	0.66
	Middle	0.833
Callee!DFCMsg	First	5.34
	Last	0.66
	Middle	0.833
RNITS	First	8
	Last	3.2
	Middle	0.54
Caller.Teardown	First	3.33

	Last	2.48
	middle	0.125
CalleeTeardown	First	4.67
	Last	0.72
	middle	0.415
GNM	First	1
	Last	2.34
	middle	0.42
BoxRemove	First	1.4
	Last	0.7
	middle	0.09

Measurement results on the pure router

The CPU demands for the basic operations of the router were measured and listed in Table 7.11.

Table 7.11 The pure router measurement results

	With 2 Transparent boxes	With 10 Transparent boxes	With 20 Transparent boxes
RouterJVM	85.46	90.28	96.1
RouterRun	18.96	55.68	93.93
HandleM	15.4	48.6	80.32
Setup	8.7	38.55	67.67
CreateFBox	9	34.3	59.33

Fig 7.8 CPU demand of RouterNode and RouterJVM vs numberof Transparent boxes.

Fig 7.8 shows the CPU demand of RouterRun and RouterJVM vs number of transparent boxes according to Table 7.11. Due to the limited size of the memory of the computer used in the measurement, it is impossible to run experiment with many more feature boxes. We have to use extrapolation of the current measurement results. According to the above figure, the CPU demand of RouterRun and RouterJVM increases almost linearly with the number of transparent boxes. Therefore linear approximation is used. We get the following formula, where K is the number of Transparent boxes located between the first Transparent box and the last Transparent box in the Transparent box chain, i.e., the total number of the Transparent boxes in the usage minus two. RouterRunDemand and RouterJVMDemand are the CPU demand of RouterRun and RouterJVM respectively.

$$\text{RouterRunDemand} = 18.96 + 4.16 * K \text{ (millisecond)} \dots\dots\dots (7.1)$$

$$\text{RouterJVMDemand} = 85.46 + 0.59 * K \text{ (millisecond)} \dots\dots\dots (7.2)$$

RouterRun demand increases relatively quickly because the RouterRun task is responsible for creating these feature boxes.

7.4. Summary

This chapter describes how to use PerfAnal to do the measurements for the ECLIPSE system, according to the characteristics of the ECLIPSE system and PerfAnal. Measurement results for the user terminal, router process and Transparent box are collected for the further modeling. The measurement results for Transparent boxes show that the CPU demand for the same kind of feature boxes varies according to the position of the feature boxes. If the feature box communicates with a box running on different JVM and communicates via RMI, the CPU demand of the feature boxes is higher.

Chapter 8. Performance and Scalability Analysis of DFC

The ECLIPSE prototype was developed to explore whether DFC can serve as the service architecture of next-generation telecommunication networks. One of the major concerns is the performance and scalability of DFC. In this chapter, the performance and scalability of DFC/ECLIPSE network are evaluated using a LQN model. The evaluation is carried out in two phases. In phase one, a brief performance evaluation is carried out on the existing state of DFC/ECLIPSE network as of early 2001 to identify the performance bottleneck of the network (the monitoring manager, as shown later in this chapter). Then the monitoring system is removed, and a more detailed performance evaluation is carried out.

8.1. Performance Evaluation with Monitoring System

8.1.1. Important system performance parameters

Fig. 8.1. Steps in a call cycle.

Before we study the DFC/ECLIPSE system performance, four important system performance parameters used in this thesis are introduced.

- Call signal handling time: Fig. 8.1 shows the steps in a call cycle. Call signal handling time is the total time to complete signaling in Setup, Routing, Connecting and Clean up steps.
- Thinking time in this work includes the periods of time used in talking between the calling parties and waiting for another call.
- Throughput: Throughput is the number of jobs finished in the unite time. Call throughput is the number of calls finished per hour.
- Utilization: Utilization is the ratio of resource (for example, CPU) busy period to the whole period of the resource including the busy period and waiting period.

8.1.2 LQN model for the existing DFC/ECLIPSE system

Performance evaluation of the complete DFC/ECLIPSE system is made with the two-party featureless call scenario. The detailed scenario is described in Chapter 3. The usage diagram and signaling sequence diagram are shown in Fig 3.2 and Fig. 3.3.

LQN model for the DFC/ECLIPSE system in the two-party featureless call scenario is shown in Fig. 8.2. It is built by adding the two featureless call sub model (Fig. 5.2) to the generic LQN model for DFC (Fig.5.1). As there is only one scenario in the system, we merge the “TwoPartyFeaturelessCall” task into the “CallScenario” task. In this study, we assume that the users’ thinking time distribution and CPU demand distribution for each entry in the model are exponential, and the number of nested requests at each entry is random with a geometric distribution and a stated mean. All above assumptions are the default settings for the ParaSRVN. Here user’s thinking time is set to 20 seconds.

Fig.8.2 Graphic LQN model for two party featureless call with monitor system.

The CPU demand for each entry in the model

Table 8.1 gives a brief description of each task and each entry shown in Figure 8.2 and Table 8.2 lists the CPU demand for each entry described in table 8.1. As HPROF can only measure total CPU a method consumed but not the CPU the method

consumed in one time call, to determine the CPU the method consumed in one time call, we have to check the call sequence diagram, the source code about that method, and how it is used. Therefore, there are some divisions in the table 8.2.

Table 8.1 Task and entry description of the Graphic LQN model for two party featureless call with monitor system

Task Name	Entry Name	Description
CallScenario		In the LQN models of the two scenarios (two party featureless call and two party call with transparent boxes) modeled in this thesis, there is one top task – <i>CallScenario</i> .
	Call	There is only one entry of <i>CallScenario</i> task, named <i>call</i> . The CPU demand for this entry is the user thinking time. Here the user thinking time is 20 seconds.
JVMP		Provisioning Node JVM
	Jvmp	JVMP task has one entry - <i>jvmp</i>
JVMR		Router Node JVM
	Jvmr	JVMR task has one entry - <i>jvmr</i>
JVMA		Caller terminal node JVM
	Jvma	JVMA task has one entry - <i>jvma</i>
JVMB		Callee terminal node JVM
	Jvmb	JVMB task has one entry - <i>jvmb</i>
Setup		In Each call scenario, there are four phases: Setup, Routing, Talking and Clean. Corresponding to Setup phase is the <i>Setup</i> task.
	Send	<i>Setup</i> task has one entry named <i>setup</i> . The CPU demand for <i>setup</i> entry is always 0.
Routing		Corresponding to Routing phase in a call scenario is the <i>routing</i> task.
	Routing	<i>Routing</i> task has one entry named <i>routing</i> . The CPU demand for <i>routing</i> entry is always 0.
Connecting		Corresponding to Connecting phase in a call scenario is the <i>connecting</i> task.
	Connecting	<i>Connecting</i> task has one entry named <i>connecting</i> . The CPU demand for <i>connecting</i> entry is always 0.
Clean		Corresponding to clean phase in a call scenario is the <i>clean</i> task.
	Clean	<i>Clean</i> task has one entry named <i>clean</i> . The CPU demand for <i>clean</i> entry is always 0.
Router		Router task represents the routing service thread who is responsible to setup the feature boxes for the call.
	Handlem	<i>Router</i> task has one entry <i>handlem</i> to handle the ‘Setup’ message
RouterM		<i>RouterM</i> task represent the monitor thread on the router node.
	Monir	<i>RouterM</i> has one entry <i>monir</i> .
PartyAM		<i>PartyAM</i> task represents the monitor thread on the caller node.
	Monia	<i>PartyAM</i> has one entry <i>monia</i>
PartyALI		<i>PartyALI</i> task represents the line interface box thread on

		the caller user terminal.
PartyALI	Setupa	Corresponding to the four phase of call, there are four entry of the ParyALI to serve these four phase. <i>Setupa</i> serves Setup phase.
	Routinga	<i>Routinga</i> serves Routing phase.
	Connectinga	<i>Connectinga</i> serves Connecting phase.
	Cleana	<i>Cleana</i> serves Clean phase.
PartyBM	Monib	<i>PartyBM</i> task represents the monitor thread on the callee node.
		<i>PartyBM</i> has one entry <i>monib</i> .
PartyBLI		<i>PartyBLI</i> task represents the line interface box thread on the callee user terminal.
	Setupb	Corresponding to the four phase of call, there are four entry of the ParyBLI to serve these four phase. <i>Setupb</i> serves Setup phase.
	Routingb	<i>Routingb</i> serves Routing phase.
	Connectingb	<i>Connectingb</i> serves Connecting phase.
	Cleanb	<i>cleanb</i> serves Clean phase.
MoniMgr	Minimgr	Task <i>MoniMgr</i> represents the monitor manager thread on the provisioning node.
	Minimgr	<i>MoniMgr</i> has one entry <i>monimgr</i> .

Table 8.2 CPU demand of the entries in the two party featureless call LQN model

Task Name	Entry Name	CPU demand (second)	Formula
CallScenario	Call	20	
JVMP	Jvmp	0.158	ProvJVM
JVMR	Jvmr	0.0131	RouterJVM
JVMA	Jvma	0.05353	LIJVM (caller)
JVMB	jvmb	0.04062	LIJVM (callee)
Setup	send	0	
Routing	routing	0	
Connecting	connecting	0	
Clean	clean	0	
Router	HandleM	0.0073	HandleM
RouterM	monir	0.00548	RouterMonitor/2 1*
PartyAM	monia	0.00766	LiBoxMonitor(caller)/6 2*
PartyALI	setupa	0.15889	CFSM + DPort!Setup + DportTeardown + GNM/6 + GEMT/6 3*
	routinga	0.00121	GNM/6 + GEMT/6 4*
	connectinga	0.00242	2(GNM/6 + GEMT/6) 5*
	cleana	0.01444	2(GNM/6 + GEMT/6) + DPort!Teardown 6*
PartyBM	monib	0.009772	LiBoxMonitor(callee)/6 7*
PartyBLI	routingb	0.14832	CFSM + DPort!UpAck + DPort.Teardown + GNM/4 + GEMT/4 8*
	connectingb	0.01372	DPort!DFC + GNM/2 + GEMT/2 9*
	cleanb	0.01219	DPort!DownAck + GNM/4 + GEMT/4 10*
MoniMgr	minimgr	0.00407	MoniMgr/14 11*

1*: As shown in Fig. 3.12, the router sends 2 monitoring events to the monitor manager for receiving and sending out a Setup message. Because PerfAnal only give the sum of the CPU used by each method, so the total CPU used is divided by 2 for each invocation of the method to send the event.

2*: PartyA's PhoneGuiLIbox sends 6 monitoring events to the monitor manager for receiving and sending out 6messages.

3*: According to the source code, the following actions occurred: 1) PartyA's PhoneGuiLIbox receives a Setup signal from the caller. 2) The LI box cleans up the dualport of the LI box. 3) The dualport of the LI box sends out a Setup message. 4) The LI box creates a nested BusyFSM. 5) getNextMessage (GNM) retrieves the message once. 6) getNextEnabledMessageTransaction(GEMT) is invoked once. The GNM and GEMT have been invoked 6 times in the whole diagram.

4*: In the routing step, GNM and GEMT are invoked once.

5*: In the connecting step, PartyA's PhoneGuiLIbox receives two messages: Alert and Connect, therefore GNM and GEMT are invoked twice.

6*: In the cleanup step, PartyA 's PhoneGuiLIbox sends a Teardown message and receives a Downack message, so the GNM and GEMT are invoked twice. The LI box also invokes a teardown() method to teardown the dual port.

7*: PartyB's PhoneGuiLIbox sends 6 monitoring events to the monitor manager for receiving and sending out 6messages.

8*: In the routing step, PartyB's PhoneGuiLIbox receives a Setup message and sends out a Upack message then creates a nested BusyFSM. The GNM and GEMT have been invoked 4 times in the whole diagram. In this step, GNM and GEMT are invoked once.

9*: In the connecting step, The dualport of PartyB's PhoneGuiLIbox sends out two DFC messages. GNM and GEMT have been invoked twice.

10*: In the clean up step, dual port of PartyB's PhoneGuiLIbox sends out a DownAck to PartyA. GNM and GEMT are invoked once.

11*: The monitor manager receives total 14 monitoring events in the whole scenario.

8.1.3 LQN simulation results

Figure 8.3 and Figure 8.4 present the call signal handling time and throughput of the system vs the number of users using the system respectively. It is assumed that each user, after making a call, waits 20 seconds (the thinking time), and then makes another call. Figure 8.4 show the utilization of the provisioning manager host and router host.

Fig. 8.3 Call signal handling time vs number of users for two party featureless call.

Fig. 8.4 Throughput vs number of users for two party featureless call with monitor system

Figure 8.3 shows that the call signal handling time has little variation, as the number of the users increases. 99% confidence intervals (CI) in the simulation is less than $\pm 1\%$ of the mean of the call signal handling time. Figure 8.4 shows that the

throughput of the system (router) is linearly increasing with the increase of the number of users in the system. 99% CI in the simulation is less than $\pm 0.1\%$ of the mean of the throughput. Figure 8.5 shows that in the experimental range, the utilization of the router host is lower than 15%, which means the router host is quite free. 99% CI in the simulation on the router host is less than $\pm 8\%$ of the mean of the utilization. However the utilization of the provisioning manager host is higher than 95% when the number of the users increases to 100 and the provisioning manager is saturated. 99% CI in the simulation on the provisioning manager host is less than $\pm 4\%$ of the mean of the utilization. Thus the performance bottleneck of the system is the provisioning manager host. As shown in Table 8.2, the CPU demand for monitoring manager task is very large. It makes the provisioning manager host saturated while the other hosts in the system are quite free. In DFC/ECLIPSE system, the monitoring events are sent and processed in the second phase. Therefore the monitoring host is busy while other hosts in the system are quite free.

Many events are sent to the monitor manager. They keep monitor manager busy that makes further study of the system performance impossible. Therefore we explored

Fig. 8.5 Utilization vs number of users for two party featureless call with monitor system

the system performance if the monitor manager and those monitor probes in the router and the LI boxes were removed.

8.1.4 Model Results for two party featureless Calls without Monitor System

In this part, we study how many users can saturate the system or the router host after removing the monitor system including monitor manager and monitor probes in the router and LI boxes. The LQN model for the two-party featureless calls without monitor manager is shown in Fig. 8.6. In this model, all the components in the Figure 8.2 remain the same, except monitor system is removed.

Fig.8.6 Graphic LQN model for two party featureless call without monitor system.

The CPU demand for each entry in the model

The description and the CPU demand for each entry are the same as Table 8.1 and 8.2.

Fig. 8.7 Call signal handling time vs number of users for two party featureless call without monitor system.

Figures 8.7 to Figure 8.9 show the system performance vs the number of users in different aspects. In Figure 8.7, the call signal handling time does not change much with the increase in the number of users at the initial stage. Above about 950 users, the call signal handling time increases rapidly. 99% CI in the simulation is less than $\pm 0.5\%$ of the mean of the call signal handling time. In Figure 8.8, the throughput increases linearly with the number of users up to about 900 users, matching the call injection rate by the users. After that the increase rate slows down. With 1000 users, the throughput of the system is about 170000 calls/hour. It is much higher than in the system with monitoring system (16200/hour) at saturated stage as shown in Figure 8.4. 99% CI in the simulation is less than $\pm 0.5\%$ of the mean of the throughput. With 1000 users the utilization approaches to 100% as shown in Figure 8.9. The system is saturated at the router. 99% CI in the simulation is less than $\pm 0.5\%$ of the mean of the utilization.

Fig. 8.8 Throughput vs number of users for two party featureless call without monitor system.

Fig. 8.9 Utilization vs number of users for two party feature less call without monitor system.

The above simulation results show that the monitor system introduces too much performance overhead. In this version of DFC/ECLIPSE, a large part of events sent to the monitor manager are debugging related. In the future version, this part of events will be removed. Therefore we removed the monitoring subsystem, including monitor manager and monitor probes in the router and the LI boxes, from the system.

8.2. Detailed Performance Evaluation of DFC/ECLIPSE

Network without Monitor System

In this section, the DFC/ECLIPSE network without the monitoring system is evaluated in more detail. The influence of the number of users and the number of feature boxes on the performance of the DFC/ECLIPSE system and the performance sensitivity of the system on the workload parameters are studied. The number of users and the number of feature boxes directly affect the performance of the system. The sensitivity study of the system indicates how much the performance of the system changes if the system workload parameters change. The scenario is described in Chapter 3. The usage diagram and signaling sequence diagram are shown in Fig 3.1 and Fig. 3.3.

Fig. 8.10 LQN model for two party call with Transparent boxes

The graphical LQN model is shown in Fig. 8.10, and is formed using the LQN model for two party call with transparent boxes in Fig. 5.3, replacing a scenario submodel in the generic LQN model for DFC in Fig. 5.1. Here we merge the “TwoPartyCallWithTransparentBox” into the “CallScenario” task.

The CPU demand for each entry in the model

The description and the CPU demand of each entry are the same as Table 8.1, except there are more tasks and entries for Transparent box. The CPU demands for caller and callee are the same as in Table 8.2.

Table 8.3 Task and entry description of Transparent box in the Graphic LQN model for two party call with transparent boxes

TransparentBox		Task <i>TransparentBox</i> represents the transparent box thread on the router node. It has three entry: <i>routingt</i> , <i>connectingt</i> and <i>cleant</i> to serve the <i>Routing</i> , <i>Connecting</i> and <i>Clean</i> phase of a call respectively.
	routingt	<i>routingt</i> serves <i>Routing</i> phase.
	connectingt	<i>connectingt</i> serves <i>Connecting</i> phase.
	cleant	cleant serves <i>Clean</i> phase.

Table 8.4 CPU demands of the entries in the two party call with Transparent boxes LQN model

Task Name	Entry Name	CPU demand (second)	Formula
CallScenario	call	200	
JVMR	jvmr	$0.08546 + 0.00059 * K$	RouterJVM
JVMA	jvma	0.05353	LiJVM(caller)
JVMB	jvmb	0.04062	LiJVM(callee)
Setup	send	0	
Routing	routing	0	
Connecting	connecting	0	
Clean	clean	0	
Router	handlem	$0.01896 + 0.004135K$	RouterNode
PartyALI	setupa	0.15889	$CFSM + DPort!Setup + DportTeardown + GNM/6 + GEMT/6$ 1*
	routinga	0.00121	$GNM/6 + GEMT/6$ 2*
	connectinga	0.00242	$2(GNM/6 + GEMT/6)$ 3*
	cleana	0.01444	$2(GNM/6 + GEMT/6)$ 4*
PartyBLI	routingb	0.148.32	$CFSM + DPort!UpAck + DPort.Teardown + GNM/4 + GEMT/4$ 5*
	connectingb	0.01372	$DPort!DFC + GNM/2 + GEMT/2$ 6*
	cleanb	0.01219	$DPort!DownAck + GNM/4 + GEMT/4$ 7*
Transparent Box	routingt	$0.01046 + 0.000845K$	$Callee!UpAck + Caller!Setup + CFSM + GNM/3$ 8*
	connectingt	$0.00711 + 0.000973K$	$Callee!DFCMsg + GNM/3$ 9*
	cleant	$0.01441 + 0.00077K$	$BoxRemove + RNITS + GNM/3$ 10*

1*: According to the sequence diagram in Fig. 4.15 and the source code, the following actions occurred: 1) PartyA's PhoneGuiLibox receives a Setup signal from the caller. 2) The LI box cleans up the dualport of the LI box. 3) The dual port of the LI box sends out a SetUp message. 4) LI box creates a nested BusyFSM. 5) getNextMessage (GNM) retrieves the message once. 6) getNextEnabledMessage-Transaction (GEMT) is invoked once. The GNM and GEMT have been invoked 6 times in the whole diagram.

2*: In the routing step, GNM and GEMT are invoked once.

3*: In the connecting step, PartyA's PhoneGuiLibox receives two messages: Alert and Connect, therefore GNM and GEMT are invoked twice.

4*: In the cleanup step, PartyA 's PhoneGuiLibox sends a Teardown message and receives a Downack message, so the GNM and GEMT are invoked twice. The LI box also invokes a teardown() method to teardown the dual port.

5*: In the routing step, PartyB's PhoneGuiLibox receives a Setup message and sends out a Upack message then creates a nested BusyFSM. The GNM and GEMT have been invoked 4 times in the whole diagram. In this step, GNM and GEMT are invoked once.

6*: In the connecting step, The dualport of PartyB's PhoneGuiLibox sends out two DFC messages. GNM and GEMT have been invoked twice.

7*: In the clean up step, dual port of PartyB's PhoneGuiLibox sends out a DownAck to PartyA. GNM and GEMT are invoked once.

8*: In the routing step, the Transparent box receives 1 Seup message therefore GNM is invoked once. GNM is invoked 3 times in the whole diagram. A TransparentFSM is created. The callee port of the box sends back a UpAck message and the caller port passes a Setup message.

9*: In the connecting step, the callee port passes DFC messages. GNM is invoked once.

10*: In the cleaning up step, each of runNextInternalTransitionSequence(RNITS) and GNM is invoked once.

For convenience of the analysis, the CPU demands in the model shown in Table 8.4 are organized as following (here K is the total number of feature boxes in a usage minus two, N is the number of users):

- CPU demands for entries in phase one:

Dt - The total CPUdemand of caller and callee user terminals to make a call in phase one.

$Dt = \text{setupa} + \text{routinga} + \text{connectinga} + \text{cleana} + \text{routingb} + \text{connectingb} + \text{cleanb} = 0.35119$
(second)

Dr – The total cpu demand on router node to complete a call in phase one.

$$Dr = \text{handlem} + \text{routingt} + \text{connectingt} + \text{cleant} = (0.05094 + 0.006723K) \text{ (second)}$$

- CPU demands for entries in phase two:

Djvmt – Caller JVM cpu time plus Callee JVM cpu time.

$$Djvmt = jvma + jvmb = 0.09415$$

Djvmr – Router node JVM cpu time.

$$Djvmr = 0.08546 + 0.00059 * K$$

8.2.1 Influence of the number of users on the performance

The simulation results of call signal handling time, throughput and utilization of the system and the number of user are shown in Fig. 8.11, 8.12 and 8.13. In each figure, there are three call scenarios with 16, 22 and 34 Transparent boxes respectively. The User thinking time Z in each of the calls equals 200 seconds.

In Fig. 8.11, there are three curves for call signal handling time vs number of users. 99% CI in this simulation is less than $\pm 1\%$ of the mean of the call signal handling time. There are three parts on each curve. The dashed lines roughly describe asymptotic behavior.

The first part of the curve is a straight line paralleled with the horizontal axis. This is because that the number of users is small and CPU is fast enough to process each call immediately after the call is injected in the system without queuing of the call in the system. In this chapter the call injection into the system is interchangeable with the user inject into the system since two users are in one call and all users are assumed to make calls after a period of thinking time.

Fig. 8.11 Call signal handling time vs number of users for two party call with Transparent boxes.

The dashed lines display asymptotic results. In light traffic, the call signals are processed immediately after they are received by the CPU of the system without any delay. The throughput of the system is determined by the call injection rate. The call signal handling time is the sum of the total CPU demand for signaling in a call on caller and callee, and the total CPU demand for signaling in a call on the router node. From the model parameters:

$$\begin{aligned}
 \text{Light-traffic call signal handling time} &= D_t + D_r \\
 &= 0.35119 + 0.05094 + 0.006723K \\
 &= 0.40213 + 0.006723K
 \end{aligned}
 \tag{8.1}$$

Here K is the total number of feature boxes minus two. The dashed line d in Fig.8.11 is the call signal handling time for a call with 34 transparent boxes. In the real system, there is always some amount queuing effect, the call signal handling time is

always larger than the above value. There are similar horizontal lines, somewhat lower for fewer boxes as shown in Fig. 8.14.

For a large number of users, the curves rise rapidly because the calls are queued and wait for being processed.

The heavy traffic asymptotes are the dashed line a, b, c. Here the system is saturated. The throughput of the system is determined by the CPU call process speed. Those CPU demand for the second phase tasks, such as CPU demand for the JVM, which is not taken into account when system is not busy, has to be taken into account now. From the system parameters:

$$\begin{aligned}
 \text{Heavy traffic call signal handling time} &= Dt + (D_{jvmr} + Dr) * N - Z \\
 &= 0.35119 + (0.08546 + 0.00059 * K + 0.05094 + 0.006723K) * N - 200 \\
 &= ((0.1364 + 0.007313K) * N - 199.64881) \text{ (second)} \quad (8.2)
 \end{aligned}$$

Here K is the total number of the transparent boxes minus two, N is the number of users and Z is user's thinking time. Line a, b, c in Fig.8.11 are with 16 boxes, 22 boxes and 34 boxes respectively. The model calculation of the call signaling handling time in heavy traffic lies on or below the asymptotic values because it ignores a component of delay due to JVM overhead.

There is a “knee” between the first part and the last part of each curve. This part of each curve represents the transition from the first part to the last part.

Fig. 8.12 Throughput vs number of users for two party call with Transparent boxes.

Figure 8.12 shows the relationship between the throughput of the system and number of the users of the system. 99% CI in this simulation is less than $\pm 0.5\%$ of the mean of the throughput. In Fig. 8.12, each of the curves can also be divided into three parts as those curves in figure 8.11.

In light traffic, the throughput is just the call injection rate. The throughput of the system can be expressed as:

$$\text{Throughput} = N/Z \quad (8.3)$$

The dashed line d in Fig.8.12 is corresponding to this formula. The slope of line d equals $1/(\text{User Thinking time}) = 18/\text{hr}$.

At the last part in Fig. 8.12, the throughput of the system does not change much with the increase of the number of users. This is because when the system is saturated,

each call is queued in the system. The throughput is determined by the CPU call processing speed. Even though the user injection rate to the system is increased, the throughput of the system does not change much, it only makes the queue longer and increases the call signal handling time for each call tremendously as shown in Figure 8.12.

When the system is completely saturated, the throughput is determined by the call processing speed of the CPU.

$$\begin{aligned}\text{Throughput} &= 1/(D_{jvnr} + D_r) \\ &= 1/(0.1364 + 0.007313K)\end{aligned}\quad (8.4)$$

Lines a, b, c in Fig.8.11 are computed based on the above formula with 16 boxes, 22 boxes and 34 boxes respectively. The throughput limits for calls with 16 feature boxes, 22 feature boxes and 34 feature boxes, are 15076 calls/hr, 12736 calls/hr and 9719 calls/hr respectively.

There is a “knee” between the first part and the last part. At this part of the curves, effect of queuing calls starts to show its influence on the performance.

Figure 8.13 shows the variation of utilization of the system with the number of users. 99% CI in this simulation is less than $\pm 0.5\%$ of the mean of the utilization. Each of the curves in Fig. 8.13 can be divided into three parts as the curves in Fig. 8.12. The first part of the curves is a straight line. At this part of the curve, there is no queuing effect in the system yet.

Fig. 8.13 Utilization vs number of users for the two party call with Transparent boxes.

When there is no queuing effect in the system, the Utilization can be expressed:

$$\text{Utilization} = \frac{Dr \cdot N}{Dr + Z} \quad (8.5)$$

Lines a, b, c in Fig.8.13 are computed based on the above formula with 16 boxes, 22 boxes and 34 boxes respectively.

The last part of the curve is approaching to 100% utilization of the system as shown as the line d in Fig.8.12. The system is saturated.

There is a “knee” between the first part and the last part of the curve. At this part of the curve, calls that users introduce in the system are queued.

The above three figures are strongly related. Each curve in the figures has three stages. At the first stage, there are fewer users in the system. The CPU can process all the calls immediately without queuing the calls. The call signal handling time keeps constant with the increase of the number of users, which the throughput and the

utilization of the system increases linearly with the increase of the number of users. We call this stage the linear stage. At the last stage, the CPU process speed is less than the call insertion speed by the users, the calls are queued. The throughput is controlled by the CPU process speed and becomes constant. At this stage, the CPU is always busy. The utilization is approaching to 100%. We call this stage a saturated stage. The increase of call injection to the system does not increase the throughput of the system, but increases the length of the waiting queue, thus increases the call signal handling time tremendously. Between the linear stage and the saturated stage there is a transitional stage. There is a knee on each curve at this stage. At this stage, queuing effect becomes prominent in the system. The increase of the user number increases the system workload thus increases the queuing effect. Then the further increase of the number of the users finally makes the system saturated. The more the Transparent box for each call, the more workload for each call, the earlier transitional stage and the saturated stage appear.

8.2.2 Influence of number of Transparent feature boxes

The simulation results of call signal handling time, throughput and utilization of the system with the number of Transparent boxes are shown in Fig. 8.14, 8.15 and 8.16. In each figure, there are three curves with 410, 610 and 810 users respectively. The user thinking time Z equals 200 seconds.

Fig. 8.14 Call signal handling time vs number of Transparent boxes for two party call with Transparent

Fig. 8.15 Throughput vs number of Transparent boxes for two party call with Transparent boxes.

Fig. 8.14 shows the relationship between the call signal handling time and the number of transparent boxes in each call. 99% CI in this simulation is less than $\pm 1\%$ of the mean of the call signal handling time. In Figure 8.14, the first part of all curves is a straight line. It is close to line a which is computed using formula 8.1 for the system with 810 users.

The last part of the curves with 610 users and 810 users has a large slope. Lines b and c are computed based on formula 8.2 for the system with 610 and 810 users. The model calculation of the call signaling handling time in heavy traffic lies on or below the asymptotic values because it ignores a component of delay due to JVM overhead as same as that in Fig. 8.11. There is a “knee” between the first part and the last part on the curves with 610 users and 810 users.

Figure 8.15 shows the relationship between the throughput and the number of transparent boxes in each call. 99% CI in the simulation is less than $\pm 0.5\%$ of the mean of the throughput. In Fig. 8.15, the first part of each curve is a straight line paralleled with the number of transparent box axis. These lines are very close to lines a, b and c computed with formula 8.3 for the system with 810, 610 and 410 users respectively.

At the last part of the curves with 610 users and 810 users, the throughput decreases as the number of transparent boxes increase. These lines are close to line d. Line d is computed using formula 8.4. There is a transitional part between the first part and last part of the curves.

Figure 8.16 shows the relationship between the utilization of the router and the number of the transparent boxes in each call. 99% CI in the simulation is less than $\pm 0.5\%$ of the mean of the utilization. In Fig. 8.16, the curves with 610 users and 810 users can be divided into three parts. The first part of the curves is a straight line. These curves are close to the lines computed with formula 8.5.

The last part of the curves approaches to 100% that means that CPU is constantly busy and the system is saturated.

Fig. 8.16 Utilization vs number of boxes for two party call with Transparent boxes.

There is a “knee” between the first part and the last part. There is a strong correlation between the curves the above three figures. Each figure represents one aspect of the same system. There are three stages for the influence of the number of transparent boxes on the system. At the first stage, there are fewer transparent boxes in each call. CPU process speed is fast enough to process all the calls the system receives immediately. The throughput is control by the call injection rate. The increase of the number of transparent boxes in each call does not cause the increase of the call injection rate. The throughput keeps constant, so does the call signal handling time. As the increase of the number of transparent boxes in each call, the workload of the system increases linearly. Therefore the utilization increased linearly with the increase of the number of transparent boxes. At the last stage, calls are queued in the system. The utilization of the system approaches to 100% and system is saturated. Between the linear stage and the saturated stage there is a transitional stage presented as a knee on the curve. At this stage, there is queuing effect in the system. The increase of the

number of transparent boxes increases the system load and increases the queuing effect. The more the users in the system, the more workload for the system, the earlier transitional stage and the saturated stage appear.

8.3. Sensitivity of the system to workload parameters

In this section, the sensitivity of the system to its workload is studied. It describes what will happen if the workload is changed due to the hardware or software reasons. For example, if the CPU demand for each entry of the system is reduced in a certain ratio, sensitivity describes how much the call signal handling time of the system will be reduced, how much throughput will be increased and how much utilization will be reduced. The CPU reduction can be due to the improvement of hardware, such as using higher speed CPU or the improvement of software, such as JVM performance optimization, or new version of JVM, or using another language such as C instead of JAVA, or program optimization or combination of the above all. Any improvement of hardware or software means cost, so to know the sensitivity of the system performance to the workload is important. It tells at what condition, the system need to be improved and how much reward can be gained. The results from sensitivity study might help us in making decisions to buy a faster CPU system or modify the software.

The simulation results of the system i.e., call signal handling time, throughput and utilization vs different percents of original CPU demands are shown in Fig. 8.17, 8.18 and 8.19. There are 810 users in the system and 16 transparent boxes in each call. The user thinking time is 200 second. 99% CI in the simulation shown in Figure 8.17 is less than $\pm 0.5\%$ of the mean of the call signal handling time. By changing the CPU demand ratio, we can see the sensitivity of call signal handling time changes. If we reduce the CPU demand for each entry at a ratio = 0.9 (90%), the call signal handling time will be reduced significantly. Maybe we can reduce the CPU demand for each entry a little more to 80%. But if we further reduce the CPU demand for each entry, the service time does not improve much. This means that if further improvement of the hardware or software does not gain much but waste more effort and money.

Fig.8.17 Call signal handling time vs CPU ratio for two party call with 16 Transparent boxes and 810 users

Fig. 8.18 Throughput vs CPU ratio for two party call with 16 Transparent boxes and 810 users.

In Figure 8.18, the throughput of the system increases dramatically with the reduction of the CPU demand for each entry of the system from 100% to 80% of the original CPU demand. The throughput does not increase much with further reduction of the CPU demand for each entry of the system. 99% CI in the simulation is less than $\pm 0.5\%$ of the mean of the throughput.

As shown in Fig. 8.17 and Fig. 8.18, there are three stages in the curves. At the first stage, the call signal handling time is not very sensitive to the CPU demand ratio. At this stage, the system is far away from being saturated (as shown in Fig 8.19), and there is a minimal queuing effect. Workload reduction of the system does not improve system performance much. At the last stage, the call signal handling time and throughput improves rapidly with the workload reduction. At this time, the workload reduction relieves the system from saturated condition and improves the system performance tremendously. Between the first stage and the last stage is the transition stage. There is a knee on the transition stage. At this stage, the workload reduction improves the system performance obviously, but not as much as at the last stage. 99% CI in the simulation in Figure 8.19 is less than $\pm 0.5\%$ of the mean of the utilization.

Fig. 8.19 Utilization vs CPU Ratio for two party call with 16 Transparent boxes and 810 users

From the above results, it is found that the sensitivity of the system performance changes according to the conditions the system is in. A saturated system is more sensitive to workload reduction. It means that improvement of hardware or software of a saturated system has more impact.

8.4. Discussion

To improve the performance of the ECLIPSE system, we have following suggestions:

8.4.1 Using a dynamic extensible feature box pool instead of creating feature boxes at calling time

There are three advantages of using a dynamic extensible feature box pool.

The first advantage is saving the box creation time. As shown in Table 8.4, the CPU demand for creation of a Transparent box is 0.004135 seconds. The CPU demand for running the Transparent box in a call is 0.002588 seconds. If we can eliminate the box creation time, we can save 60% CPU demand for each box. From Fig.8.17 – Fig.8.18, system performance can be significantly improved. Of course, there will be some overhead for maintaining the feature box pool.

The second advantage is creating feature boxes in second phase. Although this does not reduce the system workload, it reduces the call signal handling time when the system is not saturated.

The third advantage is that the feature box pool is extensible. When 90% of feature boxes in the feature box pool are used, the feature box pool manager creates 10% more feature boxes in the second phase and adds them to the pool.

8.4.2 Deploying feature boxes at user terminals

As the IP telephony end user terminal hosts have substantial processing power, it is possible to deploy some features at the end user terminal hosts within a DFC/ECLIPSE network. The user terminal's CPU utilization is usually very low, moving feature boxes from the router to end user hosts does not affect the call signal handling time much when there is light traffic on the router. However, when the traffic on the router is heavy, the reduction in the number of feature boxes on the router will significantly improve the performance of the system.

For example, if both caller and callee subscribe to 17 feature boxes respectively, and all the boxes run on the router node, the performance of the system can be presented as curves with 34 feature boxes in Fig.8.14 –Fig.8.16. If we move 6 feature boxes from the router node to the end user nodes for caller and callee respectively, the throughput and utilization vs number of users curves will be as the same as those with 22 feature boxes in Fig.8.15 and Fig.8.16.

8.5. Summary

In this chapter, the performance of the DFC/ECLIPSE network system is studied. First we identify that the monitor manager is the performance bottleneck of the system. As significant monitoring events can be reduced in the future version of DFC, we remove the monitor system from the system so that we can study the performance of the system further.

In the modified system, the influence of the number of users and influence of number of feature boxes in each call on the system performance are studied. The throughput of the system depends on the number of the feature boxes in each call from 15076 calls/hr with 16 feature boxes, 12736 calls/hr with 22 feature boxes to 9719 calls/hr with 34 feature boxes. The sensitivity of the system performance to the

workload of each entry of the LQN model for the system is studied. The results show that when router is at saturated stage, significant performance gain can be obtained by reduction of the workload. When router is at the saturated stage, if we improve software architecture such as using dynamic extensible feature pool, deploying some features on the end user hosts, and using faster language such as C or C++, and hardware such as faster CPU machine, the performance of the system will improve greatly.

Chapter 9. Conclusions and Future Work

This chapter contains conclusions concerning the research carried out in this thesis, as well as possible future work.

9.1. Conclusions

This thesis has studied the new DFC telephony architecture. It applied basic Java measurement tools to obtain a detailed picture of the workload of the ECLIPSE prototype DFC system, in its state as of late 2000. We found that the creation of feature boxes counts more than 60% total CPU demand on router host. The throughput of the system is limited to 15076 calls/hr with 16 feature boxes, 12736 calls/hr with 22 feature boxes and 9719 calls/hr with 34 feature boxes.

A layered (LQN) model was created and used to study scalability and sensitivity. System performance sensitivity to the workload for each task of the system shows that, reduction of the work load of each entry of the system can significantly improve the performance of the system especially, the limit of the throughput. Base on the measurement results and simulation results, we propose following suggestions to improve the performance of the system:

1. Using dynamically extensible feature box pool instead of creating of feature boxes at the call setup stage.
2. Deploying of feature boxes at the end user devices as many as possible.
3. Using C/C++ replacing Java if the performance of the real system is unsatisfactory.
4. Using a faster machine is another option if the performance of the system is unsatisfactory.

The limitations of the study are as follows:

1. No validation of the LQN model for DFC/ECLIPSE in a fully loaded real system, because we have not got the test patch from AT&T that enables us to do the test.
2. The CPU demands by the operations of the components in the system were measured under the following conditions:
 - 1.) The host machine was at low utilization.
 - 2.) The calls are injected in the system sequentially. There is only one call in the system at any time.

Under the above conditions, the context switch between the component threads is light. Therefore the CPU demands by the components measured in the study are lower than those demands when the system runs in the working condition in which there are a large number of calls running simultaneously and utilization of the system is higher.

9.2. Future Work

The DFC/ECLIPSE is still under development. A number of new characters may be added to the system in the later version DFC/ ECLIPSE. In term of system performance, the following works are interesting.

In the DFC/ECLIPSE version used in thesis, all features are deployed on the router. It is possible to deploy features at the user terminals too within the DFC architecture. It would be interesting to see the performance improvement if we deploy some of the features on the user terminals.

The performance parameters of DFC/ECLIPSE system are measured on a workstation in the laboratory at Carleton University instead of the real network node, The real network node has much better performance parameters. Therefore, to get more

simulation results for prediction of the real DFC/ECLIPSE system performance, we need to do measurement on the real system.

Processing voice and multimedia requires a lot of CPU time. The DFC/ECLIPSE version used in thesis does not include the hardware and software to support these media. In the future version, we can do the measurement with these media.

As ECLIPSE project evolves, more features will be added in the system. With some reasonable statistics of users for each feature, we can get more accurate measurement results and make more accurate simulation.

REFERENCES

- [Blair 00] L. Blair and J.Pang, “ Feature Interactions-Life Beyond Traditional Telephony”, in “Feature Interactions in Telecommunications and Software Systems VI”, eds M. Calder and E. Magil, IOS Press, pp293, 2000.
- [Bond 00] Gregory Bond, Andrew Forrest, Eric Cheung, Michael Jackson, Hal Purdy, Chris Ramming, and Pamela Zave, "DFC as the basis for ECLIPSE, an IP Communications software platform", IPTS 2000, pp19, Sept. 2000.
- [Bouma 94] “Feature interaction in Telecommunication Systems I”, eds L. G. Bouma and H. Velthuijsen, IOS Press, 1994.
- [Cameron 93] E. J. Cameron, N. D. Griffeth, Y. Lin, H. Velthuijsen, “ A Feature Interaction Benchmark for IN and Beyond”, IEEE Communication Magazine, V31, p64, 1993.
- [Cameron 98a] Jane Cameron and F. Joe Lin, “ Feature Interactions in the New World”, in “Feature Interactions in Telecommunications and Software Systems V”, eds K. Kimbler and L.G. Bouma, IOS Press, pp3, 1998.
- [Cameron 98b] J. Cameron, K. Cheng, S. Gallagher, F. J. P. Russo, and D. Sobirk, “ Next Generation Service Creation: Process, Methodology, and Tool Integration”, in “Feature Interactions in Telecommunications and Software Systems V”, eds K. Kimbler and L.G. Bouma, IOS Press, pp3, 1998.
- [Cheng 95] “Feature interaction in Telecommunication Systems II”, eds K. E. Cheng and T. Ohta, IOS Press, 1995.
- [Dini 97] “Feature interaction in Telecommunication Systems IV”, ed P. Dini and L. Logrippo, IOS Press, 1997.
- [Franks 96] G. Franks, S. Majumdar, J. Neilson, D. Petriu, J. Rolia, and M.

- Woodside, "Performance Analysis of Distributed Server Systems," in The Sixth International Conference on Software Quality (6ICSQ), Ottawa, Ontario, 1996, pp. 15-26.
- [Green 97] The Irwin handbook of telecommunications, ed J. H. Green, Irwin Professional Publishing, 1997.
- [Gregoire 97] J. Gregoire and M. J. Ferguson, "Neglected Topics of Feature Interaction Mechanisms, Architectures, Requirements", "Feature interaction in Telecommunication Systems", ed P. Dini and L. Logrippo, IOS Press, pp3, 1997.
- [Jackson 98] Michael Jackson and Pamela Zave, "Distributed feature composition: A virtual architecture for telecommunication services", IEEE Transactions on Software Engineering, XXIV(10), pp831, October 1998.
- [Kamoun 98] J. Kamoun and L. Logrippo, "Goal-Oriented Feature Interaction Detection in the Intelligent Network Model", in "Feature Interactions in Telecommunications and Software Systems V", eds K. Kimbler and L.G. Bouma, IOS Press, pp173, 1998.
- [Keck 98] Dirk O. Keck and Paul J. Kuehn, "The Feature and Service Interaction Problem in Telecommunications Systems: A Survey", IEEE Transactions on Software Engineering, vol.24, no.10, 1998.
- [Kolberg 00a] M. Kolberg, E.agil, D. Marples and S. Reiff, "Second Feature Interaction Contest", in "Feature Interactions in Telecommunications and Software Systems VI", eds M. Calder and E. Magil, IOS Press, pp293, 2000.
- [Lennox 00] Jonathan Lennox and Henning Schulzrinne, "Feature Interaction in Internet Telephony", in "Feature Interactions in Telecommunications and Software Systems VI", eds M. Calder and E. Magil, IOS Press, 2000.
- [Mehers 00] N. Mehers, "Java Programming on Linux", Waite Group press, 2000.
- [Rolia 95] J. A. Rolia and K. C. Sevcik, "The Method of Layers," IEEE Trans. on Software Engineering, vol. 21, no. 8 pp. 689-700, August 1995.

- [Shaw 96] Mary Shaw and David Garlan, "Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.
- [Sheikh 97] F. Sheikh and C. M. Woodside, "Layered Analytic Performance Modelling of Distributed Database Systems," in Proc. Int. Conf. on Distributed Computer Systems, Baltimore, U.S.A., May 1997, pp. 482-490.
- [Shousha 98] C. Shousha, D. C. Petriu, A. Jalnapurkar, and K. Ngo, "Applying Performance Modelling to a Telecommunication System," in Proc. of First International Workshop on Software and Performance (WOSP98), October 1998, pp. 1-6.
- [Telcordia 03] www.telcordia.com/training.
- [Woodside 95] 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 Computer, vol. 44, no. 1 pp. 20-34, January 1995.
- [Zave 00] Pamela Zave and Michael Jackson, "New Feature Interactions in Mobile and Multimedia Telecommunication Services", in "Feature Interactions in Telecommunications and Software Systems VI", eds M. Calder and E. Magil, IOS Press, pp 51, 2000.

APPENDIX

HPROF and PerfAnal usage

Sample HPROF usage:

Synopsis:

java -Xrunhprof:<agent-args> ...

If we want to profile a Java application program named “demo”, we type in:

Workingdir\$ java -Xrunhprof:cpu=samples,depth=12,file=results.txt \
-Djava.compiler= demo

Here cpu=samples - uses profiling to collect stack traces showing where code is spending its time.

depth=12 – Depth of stack trace to use for cup=sample options.

file=results.txt – Output profile data to the results.txt.

Sample PerfAnal usage:

Synopsis:

Java PerfAnal <file>

If we want to analyze the profile for the demo program mentioned above, we type in: java PerfAnal results.txt