Multilevel Performance Analysis of Scenario Specification for a Presence System

By

Huai Liu

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

Master of Science

In Information and Systems Science

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

October 8, 2002

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

Multilevel Performance Analysis of Scenario Specification for a Presence System

submitted by Huai Liu, B.Eng, M.Eng.
in partial fulfillment of the requirements for
the degree of Master of Science in Information Systems Science

_____

Department Chair

_____

Thesis Supervisor

Carleton University
October 8, 2002

Abstract

In the development of a large, complex, distributed and Internet based software application, the performance is an important quality to be considered by the developers. A methodology called Multilevel Specification and Performance Analysis (MSPA) is proposed for tracking the performance of a specification during its development through successive levels of abstraction. The MSPA captures the specification of a system using Use Case Maps (UCM) and constructs a performance model as a Layered Queuing Network (LQN), which is then solved to obtain the performance analysis results. The use of tools and automation makes the performance analysis easier to accomplish.

The MSPA methodology is evaluated on the specification of a presence system on the Internet with interesting performance results. The scalability and the sensitivity of the system are studied and the bottleneck of the system is identified. The important performance results for the simpler, more abstract models are still valid for the more complex and more detailed models.

Acknowledgements

Firstly, I would like to express my deepest gratitude and thanks to my supervisor, Professor C. Murray Woodside, for his invaluable guidance, continuous support, and numerous encouragements through this research.

I would also like to thank the members of RADS Lab for creating a wonderful working environment and for their support. Special thanks goes to our system administrator Narendra Mehta, and RADS Lab research engineer Dorin Petriu for their technical support. And very special thanks also goes to Tom Gray, Remiro Liscano, Denial Amyot, Gunter Mussbacher, Natalia Balaba, Jun Zhao and Rushabh Gudka for their great help in this thesis research.

I would like to thank Mitel for the financial support to the Chair of RADS Lab. The financial assistance from Communications and Information Technology of Ontario (CITO) and Carleton University are greatly appreciated.

Finally, I am very grateful to my family for their support, understanding and love during my study and research.

Table of Contents

List of Figures

List of Glossary

COTS	Commercial off-the-shelf
CPIM	Common Presence and Instant Messaging
DNS	Domain Name System
IETF	Internet Engineering Task Force
IMPP	Instant Messaging and Presence Protocol
LQN	Layered Queuing Networks
LQNS	Layered Queuing Network Solver
MSC	Message Sequence Chart
MSPA	Multilevel Specification and Performance Analysis
PA	Presence Agent
PI	Presence Information
PUA	Presence User Agent
RFC	Request for Comments
SIP	Session Initiation Protocol
SPE	Software Performance Engineering
TPB	Time/Performance Budgeting
UA	User Agent
UCM	Use Case Maps
UML	Unified Modeling Language
URI	Uniform Resource Indicator
URL	Uniform Resource Locator
WUA	Watcher User Agent
XML	extensible Markup Language

Chapter 1 Introduction

1.1 Motivation

In the development of a software application, especially an application used on the Internet, one important quality to be considered by the developers of the application is the system's performance. If a user has to wait a long time to get results when interacting with the application, the person will not be satisfied with the services provided by the application even if other functions of the system are satisfied. A related quality is the scalability of the application, because the Internet based application should be used by many users at the same time.

Analysis of performance during the design phase is important when developing this kind of system. With the help of performance analysis, it is possible to evaluate the response time and throughput of the developing system and the scalability of the system and other design alternatives. Many researches have shown that early performance analysis and modelling have definite advantages in the software development processes [Smith90]. Performance analysis methodologies have been proposed to solve the performance problems of Internet based applications [Smith00].

To make quantitative performance analysis, "performance parameters" of the software must be obtained. These parameters are the resource usage demands of the program, and they provide the key parameters of a performance model. These parameters must be measured, estimated by judgement, or budgeted.

The Time/Performance Budgeting for Software Designs (TPB) approach [Siddiqui01] has been proposed to give the earlier performance analysis from the high level scenario based design model using budgeting to provide performance

parameters.

For a large and complex software system, developers usually divide the system into subsystems and the subsystem into components [Brown00]. The system description evolves in stages and the components' decomposition evolves too. For example, a web application could be decomposed into two subsystems, one is client subsystem and another is server subsystem. The server subsystem is further divided into web server subsystem/component and database server subsystem/component.

Despite lacking the precise information for all the subsystems/components during the early development stage, some performance parameters of the subsystems/components may be estimated earlier before the implementation and integration of the whole system. For instance, the performance parameters of the COTS subsystems/components or the prototype of a designing system can be obtained from vendors or by measurement. If we can obtain the performance parameters for the early, more abstract specification, then we can analyse its performance to understand the scalability of the architecture, location of the bottleneck, impact of design alternatives, and the resource budgets for the whole system. As we will see, the analysis may be useful even if the parameters are only roughly known.

As the specification is developed in more detail, the performance modelling can be repeated. This research studies the evolution of the performance analysis as the specification is developed from one level of detail to the next, using a scenario specification technique. This is called " Multilevel Specification and Performance Analysis" or MSPA.

1.2 Objectives

This research is motivated by the need for techniques for achieving high

performance of Internet based software system that are large, complex, distributed, integrated with COTS subsystem/components and have time and scalability constraints. An excellent example is presence system that will be discussed in detail as a case study. The performance analysis methodology should be easy to use and integrated with the early software development phase [Smith99].

The objectives of the thesis are summarized in the following:

- Define a Multilevel Specification and Performance Analysis (MSPA) methodology to track the performance of successive levels of specifications of a software application during its development stage.
- Apply this methodology to a substantial case study, which is a presence system.
  - Describe and create a series level of abstract models of a presence system.
  - Do performance analysis at each level and give the performance analysis results at the correspondence level of the performance models.
- Evaluate the MSPA methodology to observe how the performance results at one level related to the precious level.

1.3 Thesis Contributions

The contributions of this thesis are as follows.

- Propose a MSPA methodology to solve the large, complex, distributed and Internet based system. The MSPA tracks the performance of a series level of abstract models from the earliest developing stage of a software system. The use of tools such as UCMNav, UCM2LQN and LQN solvers makes the performance analysis easier for the system designers. The performance analysis process is integrated with the software development stage and earlier performance predictions can be obtained.

- Evaluate the MSPA methodology to a presence system case study. The thesis study shows that the MSPA is easy to use with the automatic tools.
  - Five level abstract specifications and the related UCM models are constructed.
  - Five level correspondent LQN models are constructed from the UCM models.
  - Five level performance analysis outputs are obtained. The scalability and the sensitivity of the system are studied and the bottleneck of the system is identified.
- The study shows the important performance conclusions from the earlier levels are still valid at more detailed levels of specifications, even though the system's structure and performance parameters are altered.

1.4 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 is a review of the existing techniques and approaches in literatures on the performance analysis and performance modelling. It describes the software performance engineering methodology, software design description language UCM and performance modelling method LQN and a converter tool UCM2LQN. Chapter 3 describes the presence system in detail. It introduces an abstract model and related presence protocols from IETF documents. It also gives the presence system overview and a general architecture and basic entities of the presence system. Consideration about the performance issues is also described in this chapter. Chapter 4 introduces the concept of Multilevel Specification and Performance Analysis (MSPA) approach. Application of the methodology is illustrated in the example study of the first two abstract models of the presence system. Chapter 5 describes the third abstract presence system model and the application of the MSPA approach. It analyses the model and gives discussion

to the design of the presence system. Chapter 6 applies the MSPA to the fourth level abstraction on the presence system that implements using SIP entities. Chapter 7 applies the MSPA to the fourth level abstraction on the presence system that using IBM T Spaces framework. Chapter 8 describes a presence system prototype and the measurement of the performance parameters. It also shows the performance analysis of the prototype using MSPA methodology. Chapter 9 summarizes the conclusions and gives the directions for future research.

Chapter 2 Background

This chapter describes the basic background for this thesis research. First we review Software Performance Engineering (SPE) and its approaches. Then Layered Queuing Network (LQN) performance model is described. After that in Section 2.3, a scenario based notation for software system specification called Use Case Maps (UCM) is described. In Section 2.4, the Time/Performance Budgeting approach is introduced. Finally, the concept of Tuplespace and IBM T Spaces network middleware, which is used in some of the more detailed specifications, is introduced.

2.1 Software Performance Engineering

Software Performance Engineering (SPE) is a technique introduced in [Smith90]. Smith defines the SPE as "a systematic, quantitative approach to constructing software systems that meet performance objectives." According to IEEE definition, **performance** refers to the degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage. [IEEE Std 610.12-1990] In software performance engineering, the **performance** refers to the response time or throughput as seen by the users [Smith90].

In order to assess the performance effects of different design and implementation alternatives for a system, SPE proposes the use of quantitative methods and performance models from the earliest stages of software development throughout the whole lifecycle. Performance models can be defined during the early system analysis and can be quickly solved [Smith99]. The output of the performance models can give the performance values such as mean response time, throughput and the utilization for each resource in the system and the whole system. The models' solution can be compared with the requirements and

the performance deadlines. If the results do not meet the requirements, the design architectures and parameters may be modified, and the performance models can be regenerated and resolved. This cycle continues until the requirements are met.

In the early development stages, the parameter values of a performance model are estimated based on the previous experience of a similar system, the measurement of the reusable components, and the budgeting constraints allocated to different components. These early parameters may be inaccurate. However, as the development progresses and more components are implemented and measured, the parameters become more accurate. And the performance analysis results will become more and more accurate.

Many SPE examples have shown that early performance modelling has advantages despite the inaccurate parameters and results [Smith90]. One example is described in [Smith00]. This case study is about building a responsive and scalable web application. The SPE techniques are applied to construct and evaluate performance models of various architectural alternatives to select the architecture that will meet performance objectives.

When applying SPE to analyse the performance of a system, the following main jobs must be done: [Smith90]

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

2.2 Layered Queuing Network (LQN) Model

Layered Queuing Network (LQN) was developed as an extension of the Queuing Networks for performance estimates of the complex and distributed systems [Rolia95] [Woodside95]. A LQN model describes the system architecture by the sets of resources that are used by its operations. Every operation requires one or more resources. Resources include logical resources such as process threads, critical sections or locks and control tokens as well as shared buses, processors, disks and interface controllers. It is very suitable for modelling distributed systems with features such as parallel processes, multi-threads and multiprocessors.

In a LQN model, entity “Task” represents a hardware and software object, which may execute concurrently. A task can be classified as “Client Task”, which only requests service from a server and “Server Task”, which is both a server that serves its clients and a client that requests service from other servers. Tasks execute some work on their host processors when they have acquired access to the processors. Each task can serve different requests, which are represented by entities “entries”. All the requests of one task are put into the same queue. The order in which the requests are served is dependent on the queuing discipline. The LQN supports disciplines such as FIFO, LIFO and PS.

There are three types of calls or messages between tasks and they are synchronous, asynchronous, and forwarding calls. An asynchronous call means a client task makes a request to a server task, and does not wait for a response from the server task. A synchronous call means a client task makes a request to the server task, and the client is blocked until the server task has responded back with a reply. The forwarding call is similar to a synchronous call, except that after the server task finishes servicing the request, it will transfer the request to another server task, which will be responsible for sending the reply back to the client task.

A server entry may be decomposed into activities if the service requires more detailed description of executions [Franks99]. The sequence of activities may have an AND or OR fork branches. The AND fork represents parallel execution of threads of control, while the OR fork represents randomly choice between different branches. Each activity has its own execution time demand on the resources, and can make calls to other entries of other tasks.

A LQN model can be represented in a graph. A task is represented as a row of rectangles with task name attached to the right-hand end, and other boxes representing entries with entry names inside the box. The hardware device is represented as ovals. An asynchronous call is represented as a line with an open arrow, and a synchronous call is represented as line with a closed arrow. A forwarding call is represented as a dash line with a closed arrow. The number of calls can be labelled beside the arrow if the calls are more than 1. The activities are represented with rectangles under the task. The precedence relationship of the activities are represented by a text annotation with arrow "->" to show the sequence and "+" to show the OR fork and " & " to show the AND fork. An activity may trigger a reply to a request message, and this is indicated by showing the entry as "[entry]".

A simple example of a LQN model is illustrated in figure 2.1. In this figure, the Client task only requests service while the Server task is both a server that serves the Client and a client that requests service from the Database. Task Server has two entries E_S1 and E_S2, which serve two different requests from the Client. Each entry has its own execution time and demand for other services of Database. The entry E_S1 has three activities A1, A2 and A3 and they all have their execution times on the ServerP processor. The activities' precedence is defined in the text note beside the LQN model. After execution the A1, A2 and A3 execute in parallel, and A2 makes request to E_DB entry of Database task. Then A2 sends reply to the entry that asks the service of entry E_S1.

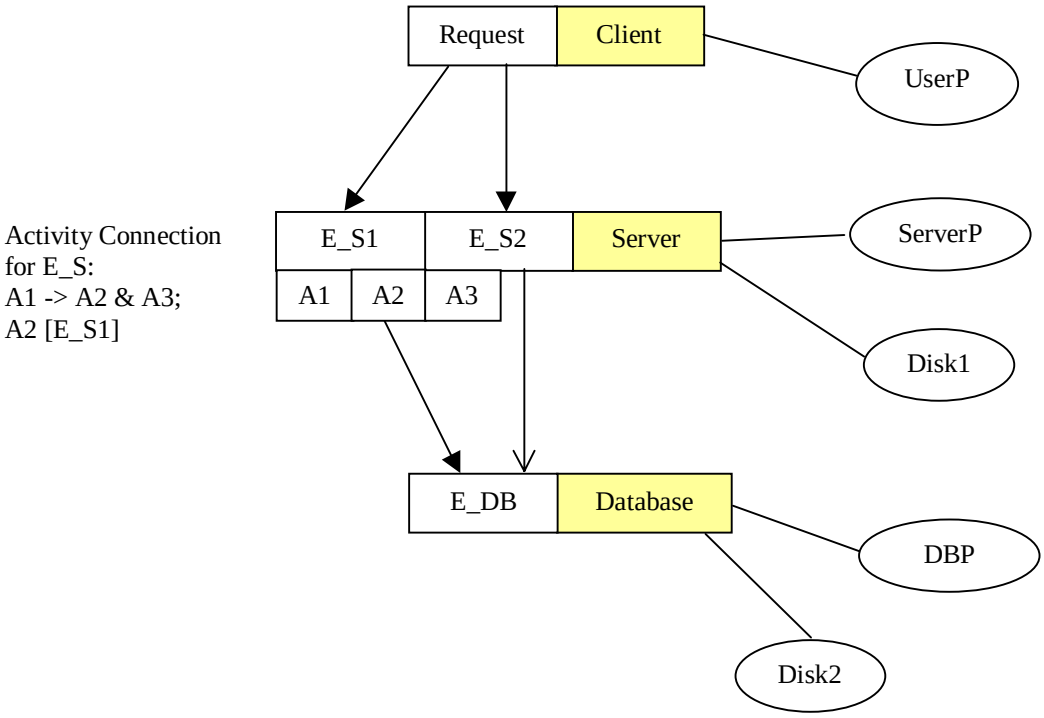

Figure 2.1 Example of LQN Model

2.2.1 LQN Tools

LQN model can be built from a conceptual analysis and a design of the system, or by analyzing data from an existing system or a prototype. In this thesis, we use UCM2LQN [Petriu01] tool to generate a LQN model from UCM model. Once a LQN model is created, it can be solved by LQN solvers [Franks99]. One is an analytic solver named LQNS, and another is a simulator named ParaSRVN. Moreover, an experiment controller named SPEX [Spex97] can be used to execute parameterised experiments over parameter ranges. It is used to experiment on a set of models with specified parameters and can get a set of output results. The solvers give performance output in the form of mean values, mean values with a given variance, or as percentile values of throughputs, service time of the system, tasks, and entries and the utilization of each task, entry on each processor.

In this thesis, we use UCM2LQN tool to generate a basic LQN file, and then add some parameter values to generate a set of input parameters of the model, and using SPEX controller to execute the experiment to get a set of outputs of the model.

2.3 Use Case Maps (UCM)

Use Case Maps (UCM) was developed by Professor R.J.A.Buhr[Bhur96] at Carleton University. It is used to describe and understand a large and complex software system. With UCM, designers and users can easily understand the behaviour and architecture of a system at a high level of abstraction. A UCM is a scenario-based architecture description language that explicitly and visually links scenarios and architecture together. It can model static as well as dynamic behaviour and structure of a system. These features make the evaluation of architectural alternatives early in the design phase very easy.

2.3.1 UCM Notation

The basic UCM notation has paths, components and responsibilities. The basic path has responsibilities that causally are linked in sequences, alternatives, or in parallel. The advanced path can have stubs, which is used for structuring hierarchical UCM and for representing dynamical behaviour.

The path has a start point and an end point, represented by a circle and a bar respectively. Starting point represents pre-conditions or triggering causes. End point represents post-conditions or resulting effects.

Components represent entities or objects composing the system. Components may be of different natures, such as software or hardware, objects or modules, packages or servers. They are represented using boxes in the UCM model.

Responsibilities represent places where the system does some operations, actions and

tasks and changes its state. Responsibilities are represented using crosses in the UCM model.

A UCM may have alternative paths and shared segments of paths. An OR-join merges two (or more) overlapping paths and an OR-fork splits a path into two (or more) alternatives. Alternatives may be guarded by conditions represented as labels between square brackets. Concurrent and synchronized segments of paths are represented through the use of a vertical bar. An AND-join synchronizes two paths and an AND-fork splits a path into two (or more) concurrent segments.

A Loop in a UCM represents a loop path that splits by an OR-fork from a main path and join the main path later by an OR-join. The loop number can be specified to state that the loop path executes the loop number times and then join the main path. The example of Loop is shown in Figure 4.2 in chapter 4.

UCM can be derived from system's requirements, or use cases. Responsibilities need to be inferred from the requirements or use cases. If the components are not specified during the early design process, an **unbound UCM** can be created with only responsibilities. When the components are defined and responsibilities are allocated to components, then a **bound UCM** can be created.

Figure 2.2 is a simple example of a UCM model.

In Figure 2.2, the system has three components and they are **Client**, **Server** and **Database**. The path has one start point and two end points. The responsibilities along the path are represented with label **rx**. Here x is a digit. The UCM is a bound map and the responsibilities are performed by the bound components. In the system, the Client sends a request to the Server and the Server processes it. The Server's responsibility is represented by **r1**. After that, the Server forks two alternative paths with one path executes responsibility **r2** and replies to the Client directly and the other path executes

responsibility **r3** and then makes a call to the Database and waits for the Database reply. The Database processes it and the responsibility is represented by **r4**. Then the Database forks two parallel paths, which do not join together later. One AND path executes responsibility **r5** and replies to the Server and the other AND path executes responsibility **r6** and ends the execution inside the component.

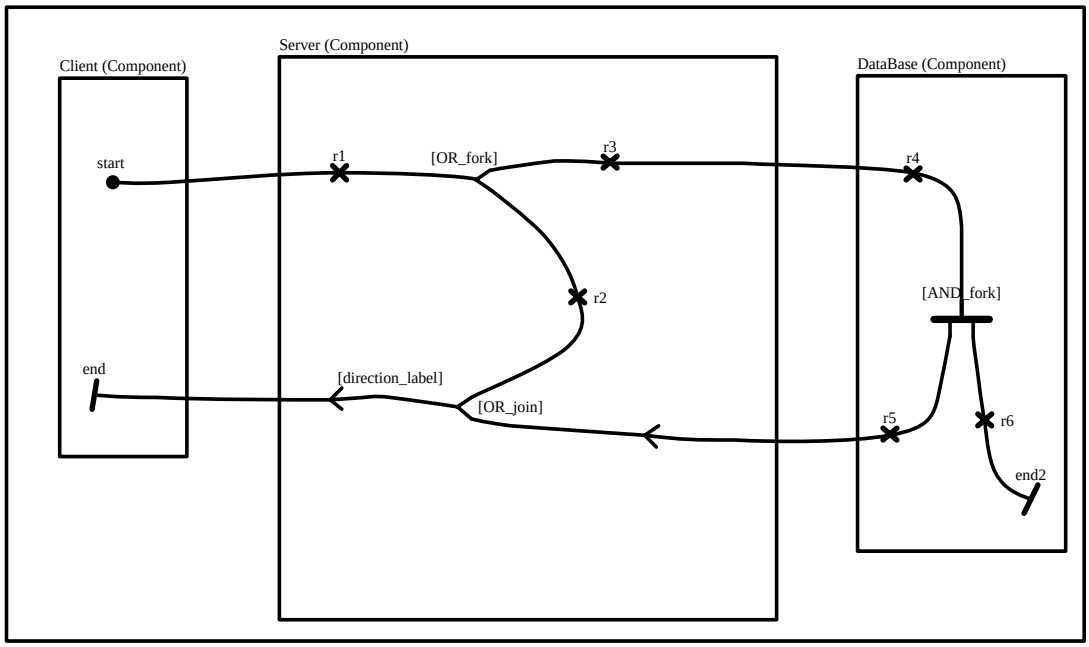

Figure 2.2 Example of UCM Model

When maps become too complex to be represented as a single UCM, a stub is used to define the sub-maps, called **plug-ins**. A top level UCM, called the root map, can include stubs for plug-ins and the plug-ins can also have stubs. Stubs are represented as diamonds with solid line, called a **static stub** or dashed line, called a **dynamic stub**. The static stub contains only one plug-in and the dynamic stub may contain several alternative plug-ins, which is determined at run-time according to pre-conditions. Furthermore, multiple plug-ins can be selected sequentially or in parallel. A dynamic stub example can be found in Figure 3.1, in which the map contains several plug-ins, such as Login, Logout, Subscribe,

Unsubscribe, UpdatePI, etc. and the selection can be made at run-time when users simulate different triggers.

More advanced UCM notations include: timer, abort, failure points, and shared responsibilities. Please refer UCM web site for further information [UCMWeb].

2.3.2 UCM Navigator

The UCM Navigator (UCMNav) is a UCM editing tool developed by Andrew Miga [Miga98] at Carleton University. Users of the UCMNav can draw and modify UCM easily. UCMNav also supports adding comments and descriptions for UCM and for the UCM elements.

Moreover, UCMNav allows users to specify the performance parameters of the design model. For example, designers can enter the following parameters into the UCM model.

- Arrival rate for an open system with special distribution type such as Exponential, Deterministic, Uniform or Erlang.
- Number of users for a closed system.
- Processors' speeds and multiplicity.
- Disks' access time.
- Task Multiplicity for a component.
- Devices are used by which components.
- CPU demands and demands for other server for each responsibility.
- Loop number
- Probability of a Fork path

Other advanced features of UCMNav include generating Message Sequence Charts (MSC) [Miga01] and Layered Queueing Network (LQN) Model [Petriu01].

Further information about how to use and the research information can be found in "UCM Tutorial" [AmyotT99] or on Web site http://www.UseCaseMaps.org [UCMWeb].

2.3.3 From UCM to LQN

A major barrier to the use of SPE is the cost of effort and time in creating the performance model from the design description. Many efforts have been done to integrate performance analysis into the software development process throughout all design phases [Balsamo01] [Petriu02]. The UCM2LQN converter is one example of automated conversion tool that generates performance models from software design model.

The UCM2LQN converter was developed by D.B. Petriu [Petriu01], [Woodside02]. It bridges the gap between a design description in the form of the UCM and a performance model in the form of LQN. It takes input of a UCM model with responsibilities, components, devices, services and related performance parameters, produces a LQN model with activities, tasks, devices and requests from some tasks to other tasks. One component of a UCM model corresponds to one task of a LQN model, and responsibilities correspond to activities of a task. The device of UCM corresponds to a hardware device of LQN and service of UCM corresponds to a processor, on which tasks execute operations. Three types of calls are created for different UCM paths traversal patterns. When a UCM path crosses from one component to another and returns back to the original component, a synchronous call is made from one task to an entry in the called task in LQN model. When a UCM path crosses from one component to another and does not return back to the original component, an asynchronous call is made from one task to an entry in the called task in LQN model. When a UCM path crosses from one component to another and then to several others, before returning back to the original component, a forwarding call is made, the original call is synchronous call, and the forwarding is asynchronous for the

other components.

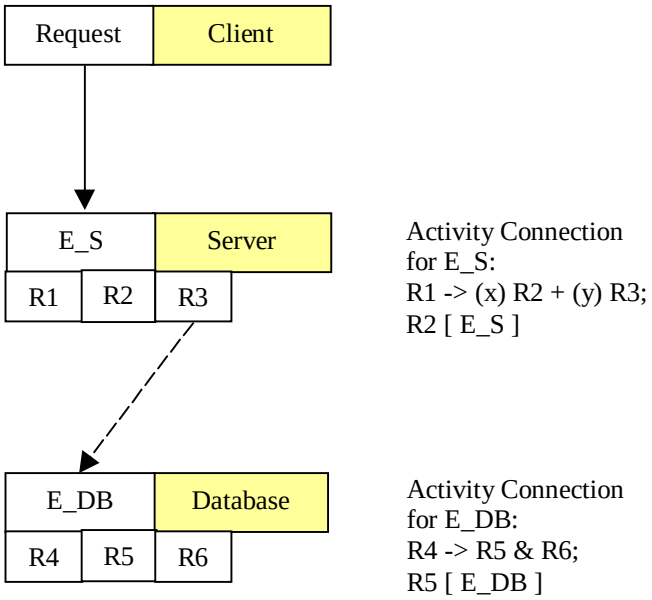

Figure 2.3 Example of LQN Model Converted from UCM Model

Figure 2.3 is an example of LQN model created using UCM2LQN converter from UCM model M_Example in the previous section. There are three tasks and each task has one entry. The Client task makes a synchronous call to the Server task. And the Server's entry E_S has activities R1, R2 and R3 and their relationship is represented in the Activity Connection's specification beside the LQN map. The activity R1 is followed either by R3 with probability of x or by R4 with probability of y and (x+y) is 100%. If R2 executes, entry E_S does not make call to the Database and replies to the Client directly. If R3 executes, R3 makes a forwarding call to the Database task and the entry E_DB replies the call to the Client task. For entry E_DB of the Database, the sequence of activities R4, R5 and R6 is described in Activity Connection's specification. R4 executes first, and then R5 and R6 execute in parallel, and only R5 replies.

UCM2LQN converter tool has already been integrated into the UCM Navigator tool.

The performance parameters such as service demands of responsibilities, arrival rates at start points, branching probabilities, loop repetition number, and device speed factors can be entered from the UCMNav and the UCM2LQN converter generates LQN model using these performance values if they exist or using default values if they don't exist.

2.4 Time/Performance Budgeting Approach

The Time/Performance Budgeting (TPB) for Software Designs [Siddiqui01] approach is proposed to achieving performance targets for a complex, distributed software system. Budget values are created for the demands of individual responsibilities. These values are based on the estimated values, using experience and on measurement, using prototyping or testing. These budgets are evaluated based on the performance model of the system and the analysis is performed to compare the consequences of the budgets with the performance specification.

The performance budget analysis road map is illustrated in Figure 2.4, which is taken from [Siddiqui01].

In the Budget Analysis Road Map, the boxes represent design artifacts or libraries used in the analysis, and the arrows show operations carried out either by the analysis, or by some automated transformation.

There are seven steps in the road map and they are listed below:

Step1: Begin from the designer UCM that describes the system requirements as a set of scenarios at a suitable abstract level. The UCM usually defines software components and responsibilities of the system.

Step2: Budget workload demands by the responsibilities in the UCM by estimating, budgeting or guessing on experience.

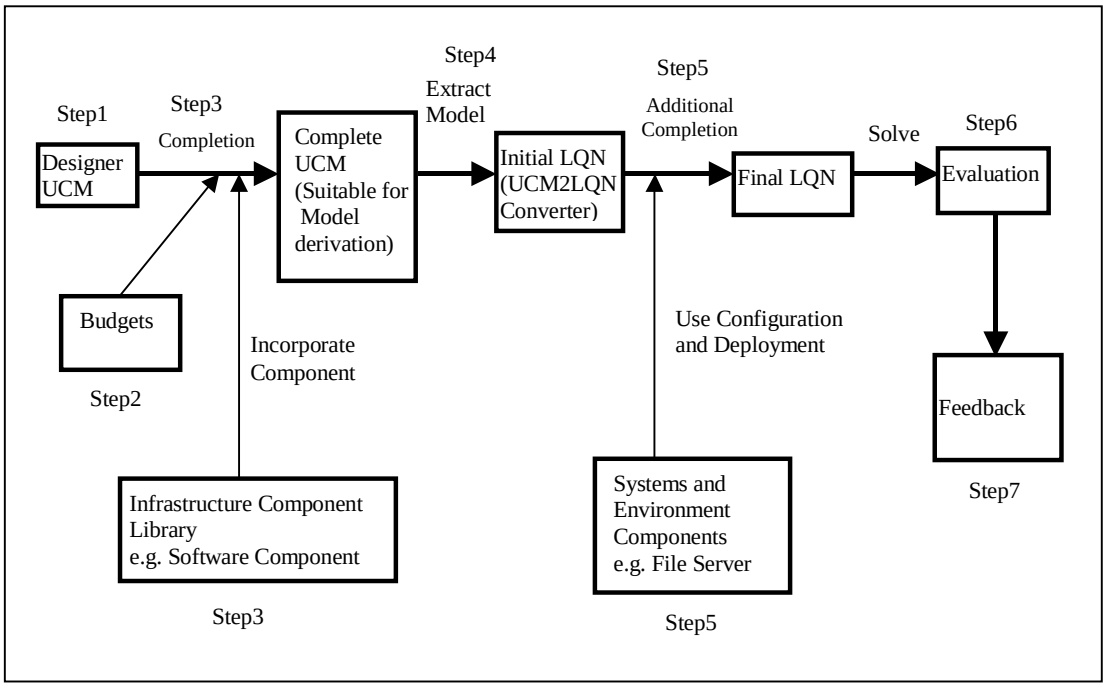

Figure 2.4 Budget Analysis Road Map Taken From [Siddiqui01]

Step3: Perform Completion by adding missing details to facilitate performance analysis. The Completion can be added either to the UCM model level or the LQN model level in later step 5.

Step4: Create the LQN performance model from the above “Complete” UCM.

Step5: Complete the LQN model by adding details of the execution environment such as hardware and software. Some examples of this LQN level “Completion” are adding file servers, web servers, or network latencies.

Step6: Evaluate the performance model and compare the results with the performance requirements.

Step7: Give feedback on the evaluated results. Some suggestions such as alternatives in design UCM, or in implementation options, or environment parameters.

2.5 Tuplespace

The concepts of Tuplespaces [Gelernter82] was first developed as part of the Linda System at Yale University in the 1980s. A **Tuplespace** is a globally shared memory space that is organized as a collection (bag) of tuples. The basic element of a Tuplespace is a **Tuple**, which is an ordered sequence of typed values called **Field**s. A **field** is the most basic component of the Tuplespace data structure hierarch. A field contains **Type, Value**, and optional **Field Name**. A filed can be a **formal field** or an **actual field.** The formal field has no value but a type that only describes the value, while the actual field has a value. A **Template** tuple is a tuple that used for matching. One or more of the Fields in a template may consist of only the type with no value. A template matches a tuple if they have an equal number of fields and each template field matches the corresponding tuple fields. Here is a example of a tuple: <2.5, “hello”, 123> means a tuple with three fields, a float with value 2.5, a string with the value “hello” and a integer with the value 123. A template <float, String, 123> matches the tuple <2.5, “hello”, 123>, while a template <int, “hello”, 123> does not match the tuple <2.5, “hello”, 123>.

A tuple is created by a process and placed in the Tuplespace via the **write** operation. Tuples are read or removed with **read** and **take** operation, which take a template and return the fist matching tuple. Tuplespace provides a simple and powerful mechanism for inter-process communication and synchronization and is very suitable for parallel and distributed programming. A process with data to share “generates” a tuple and places it into the Tuplespace. A process requiring data simply requests a tuple form the space. The receiver and sender of the message don’t require to know each other. The communication is fully anonymous. This feature is called “destination uncoupling”. Another feature is “space uncoupling”, meaning Tuplespace is able to provide a globally shared data space to all processes, regardless of machine or platform boundaries. The third feature is “time

uncoupling", meaning Tuples have their own life span, independent of processes that generated them, and processes that may read them.

2.5.1 IBM T Spaces

IBM T Spaces [Wyckoff98] is a network middleware system that facilitates communication between distributed processes.

T Spaces is a Java-based communication framework developed by IBM. The combination of the T Spaces communication model and the flexibility, portability, and strong typing of Java results in a framework that provides a lightweight computation environment, and a secure, easy-to-use application system.

T Spaces system has T Spaces client and T Spaces server and they communicate with each other via tuples. A T Spaces client issues operations that perform a variety of functions. The basic operations supported by the T Spaces are:

- write(tuple) add a Tuple to the space.
- take(template_tuple) Performs an associative search for a tuple that matches the template. When found, the tuple is removed from the space and returned. In none is found, returns null.
- waitToTake(template_tuple) Performs an associative search for a tuple that matches the template. Blocks until match is found. Removes and returns the matched tuple from the space.
- read(template_tuple) Like "take" above, except that the tuple is not removed from the tuple space.
- waitToRead(template_tuple) Like "waitToTake" above, except that the tuple is not removed form the tuple space.

Besides above basic operations, T Spaces also provides powerful event register and

event notification services. The client applications can register to be notified of the event in which they are interested. An event is the execution of a “write” or “delete” on some tuples. The selection of which tuple that is interested in is via a template tuple.

The client application must implement a Callback interface to be notified. The Callback interface in the T Space framework defines a **call** method that calls an application back with a tuple it requested. This feature can be used to implement presence information notification mechanism.

Method eventRegister(commandName, Tuple, callback) of Tuplespace registers the interested tuple for a client application. The parameter commandName identifies the operation such as “write” or “delete” to be on the watch for. In the presence system that implements the T Space event register and notification mechanism, only the “write” operation is used. The parameter Tuple specifies the tuple. The parameter callback specifies which class object contains the call() method that will be called when the event happens.

Figure 2.5 shows the T Spaces event notification mechanism UCM model.

In this UCM model, there are three components. they are Tuplesapce server, Event Notifier (EventNotifier) and Callback Application (CallbackApp). The CallbackApp registers to the Tuplespace server for the tuple it has interest. And the EventNotifier writes tuples to the Tuplespace server. When the registered tuple is written into the Tuplespace server, the call method of the CallbackApp is called and the CallbackApp is notified of the event that the tuple is written into the Tuplespace server.

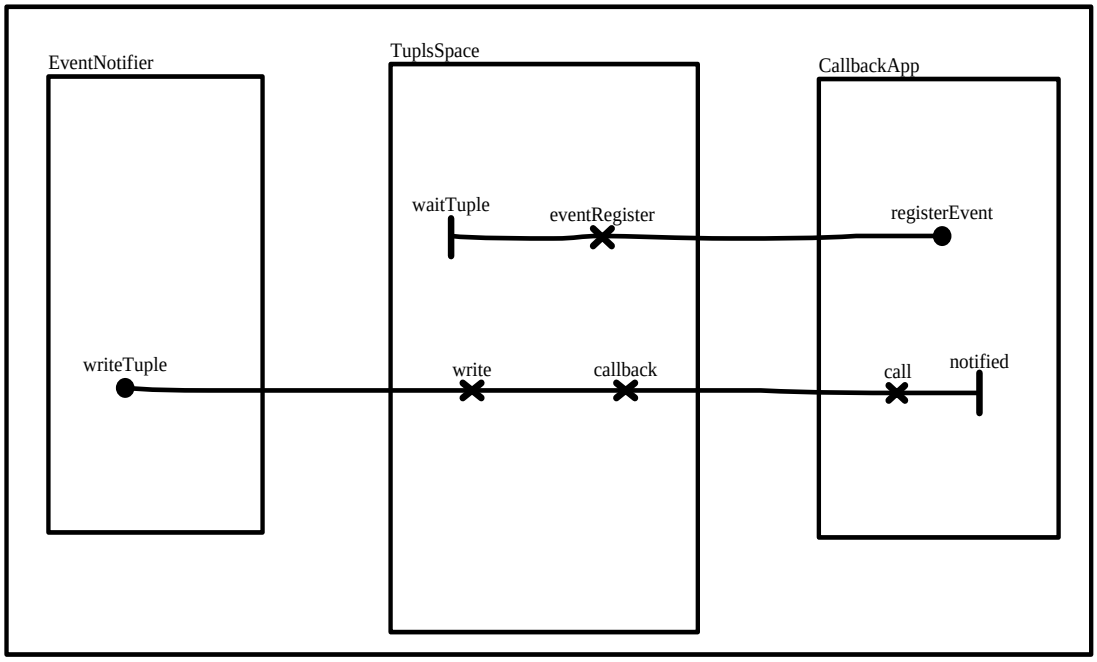

Figure 2.5 T Spaces Event Notification Mechanism UCM Model

Chapter 3 Presence System Background

Presence has recently emerged “as a new medium of communications over the Internet” [RFC2779]. Presence system “allows users to subscribe to each other and be notified of changes in state” [RFC2778]. In a presence system, presence information is published by users to the interested subscribers for their current availability or their willingness to engage in various types of communications. The availability includes their current locations and preferred communication device or media types such as telephones, mobile phones, PCs, or text, email and instant message. By using a presence system, users will know when and how they can reach their targeted contacts before they want to make a connection or a call. Many new applications can be invented and many new features can be added to the presence system.

In this chapter, the background of a presence system is presented. Section 3.1 describes the presence system specification. It includes an abstract model and the entities defined by RFC2778. The main scenarios and the performance issues of the presence system are also introduced in this section. Section 3.2 gives an introduction of presence protocols of IETF. Overview about SIP protocol and its extension for presence system are introduced. SIP components and architecture for presence are briefly described.

3.1 Presence System Specification

The presence systems are usually integrated from computer and telephony technologies, are composed of products of various vendors, and are used across the Internet. In order to facilitate the development of the presence systems and to allow interoperation of a wide variety different vendors’ applications working together on the

Internet, IETF defines a set of documents to specify the requirements, protocols, and abstract models of the presence system. The “Instant Messaging/Presence Protocol Requirements” [RFC2779] defines a minimal set of the requirements that the presence system protocol should meet. “A Model for Presence and Instant Messaging” [RFC2778] defines an abstract presence and instant messaging model and common terminology for the system. The “Common Presence and Instant Messaging (CPIM) ” [CPIM] defines a common profile which enables users of presence systems to exchange presence information with users of the services, which may use different protocols compatible with CPIM.

3.1.1 Entities of the Abstract Presence Model in RFC2778

The abstract model of RFC2778 defines two services: a presence service and an instant message service. The presence service accepts PI, stores it, and distributes it. The instant message service accepts and delivers instant messages to instant inboxes. These two services do not necessarily work together and both services can work independently. In this thesis research, we only focus on the presence service and use the entities and terminology defined by the RFC2778.

- **Presence Information (PI)** consists of an arbitrary number of presence tuples. Each tuple consists of a status marker, an optional communication address and optional other presence states. A marker might convey information such as online, offline, busy, away, and do not disturb.
- **Presentity** is a presence entity that provides presence information to a presence service. It is identified using the Universal Resources Identifier (URI) scheme. The syntax is user@domain [CPIM]. The local part “user” is assigned by the domain part of the identifier. Presence domains are defined by an Internet domain name.

- **Presence Service** accepts, stores, and distributes PI. It may have complex structure containing multiple Servers and/or Proxies.
- **Watcher** is an entity that requests PI about a **Presentity** from the presence service. Two types of Watchers are classified as **Fetchers** and **Subscribers**.
- **Fetcher** requests the current value of some PI from the presence service. A special kind of Fetcher is a **Poller** who fetches information on a regular basis.
- **Subscriber** requests notification from the presence service whenever there are changes in some PI.
- **Presence User Agent (PUA)** is a component that manipulates presence information for a Presentity. Each user can have many devices such as a mobile phone and PDA, each of which has its own PUA. This means a user may have multiple PUAs.
- **Watcher User Agent (WUA)** is a component that manipulates Watchers controlled by that user.
- **Proxy** is a server that communicates presence information, subscription and notifications to another server. The Proxy is need when presence servers are located in different domain.

3.1.2 Main Scenarios of Presence System

A presence system lets users to subscribe to a Presentity (Subscribe), unsubscribe from a Presentity (UnSubscribe) and cancel a subscription (CancelSubscribe). It also provides services for users to update presence information (UpdatePI), and the updated presence information is notified to all subscribers immediately. The system also provides other functions such as adding a new user (Register) to the system, removing an existing user (RemoveUser) from the system, and managing the users's identifications and locations. It also provides group (buddy list) management functions such as creating,

modifying or removing a group (Create/Modify/Remove Group). Users can log in and log out of the presence system, and can change their access rules and notification policies. Many new features can also be added to the presence system if invented.

In this thesis study, we are interested in the scenarios with close relations with subscription and notification of presence information in the presence system. They are Subscribe, UnSubscribe, CancelSubscribe, and UpdatePI scenarios. Other scenarios are less executed and not important compared with above scenarios from the performance perspective. Following are the description of these scenarios.

Subscribe: When a user wants to receive the presence information of associated with a Presentity, it invokes the subscribe operation of the presence system. The presence system responds by executing a subscribe operation. If the operation is success, the presence system will immediate send a notification of the PI to the Watcher whenever there are any changes to the Presentity's PI.

Unsubscribe: When a Watcher is no longer interested in receiving notifications for changes in PI of a Presentity, he/she may prematurely remove a subscription by invoking the unsubscribe operation.

CancelSubscribe: When a user who controls a Presentity wants to cancel the subscription access to a current Watcher, The user invokes the CancelSubscribe operation and the presence system removes the Watcher from the subscribers' list and sends a message to the Watcher and lets it know that its subscription has been cancelled.

UpdatePI: A user can change its PI at any time. When a user invokes the UpdatePI operation, the presence system updates the PI and stores the current Presentity's PI status, then notifies all Watchers about the change immediately.

3.1.3 Abstract Model of Presence System

For the main scenarios of the presence system described in above section, we can derive the system's UCM by defining components and responsibilities that are allocated to the components. Figure 3.1 shows the highest level of abstract model using UCM notation. In this abstract model, there are two components: User and Presence Service (PresenceService). User component enters user's requests and waits for the reply from the PresenceServiece. The PresenceService stores the information about all users and manipulating the PI data as well as notifies the updated PI to all the subscribers when the presence information is changed.

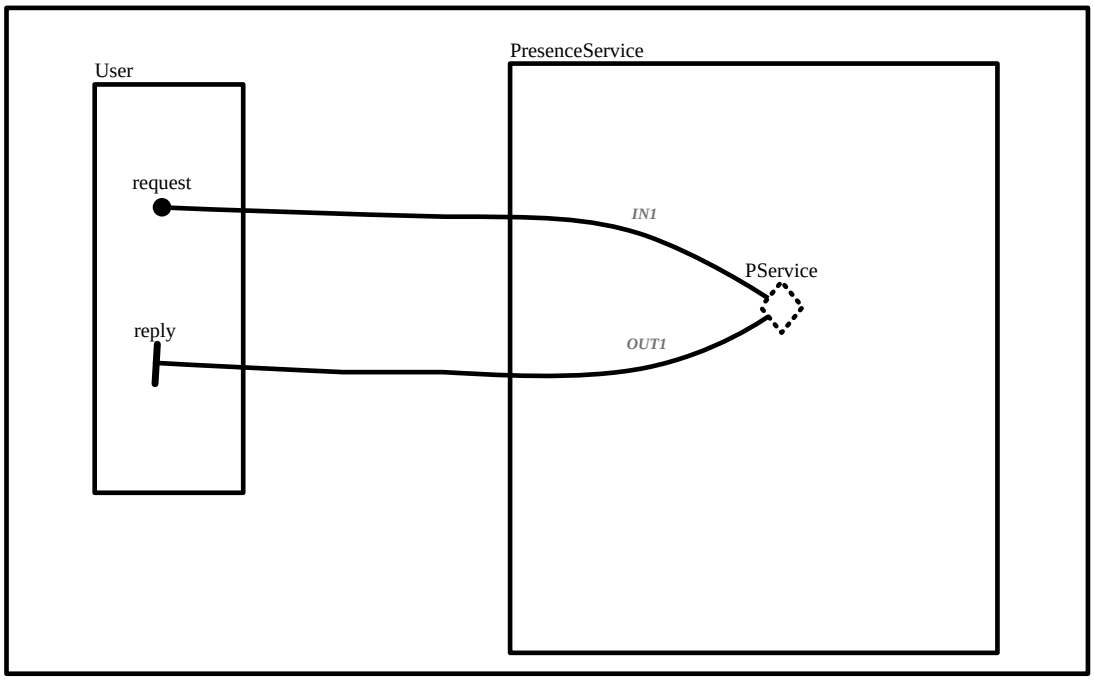

Figure 3.1 Highest Level Abstract UCM Model of Presence System

The stub PService is a dynamic stub that can represent different plug-ins for the scenarios described in section 3.1.2. They can be Subscribe, Unsubscribe, CancelSubscribe and UpdatePI. The choice of which plug-in is selected is determined at the run-time by the users who initiate the scenarios.

From the performance perspective, scenarios Subscribe, Unsubscribe, and CancelSubscribe are less executed comparing with the UpdatePI scenario. After the presence system is initialised and the users already log in the system and subscribe to some Presentity, the workload of the presence system is mainly determined by the execution of scenario UpdatePI because users may update their PI frequently. Therefore, in this thesis, we focus our concern on the UpdatePI scenario and study the scenario in detail. Exceptions such as checking failure and time out path are neglected in this thesis research.

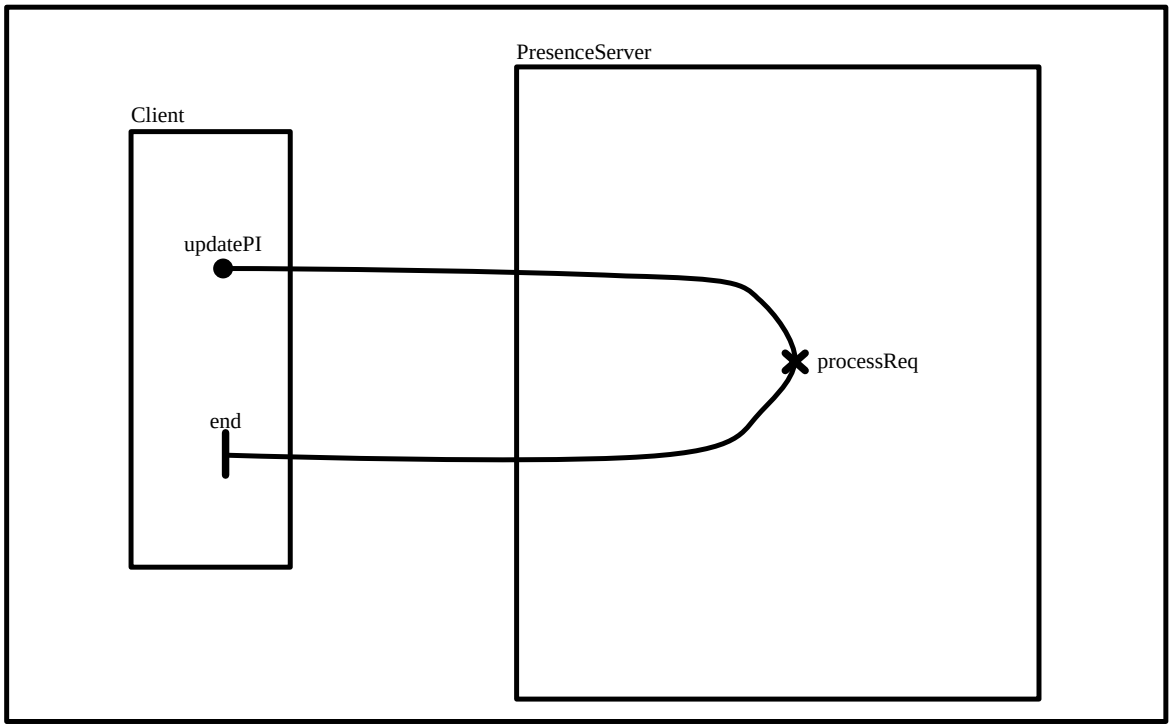

Figure 3.2 UpdatePI plug-in

Figure 3.2 is an UpdatePI plug-in for the stub PService in Figure 3.1. This gives the abstract UCM model for UpdatePI scenario. In this UCM model there are two components, one is Client and the other is Presence Server (PresenceServer). The Client is an interface to the user, and it may be any terminal that interacts with the users. User starts the UpdatePI scenario by entering the requests and the Client sends the requests to the

PresenceServer. In this figure, the PresenceServer component executes the responsibility **processReq,** which may include the operations such as updating the presence information, getting the subscribers' list according to the access policy, and sending the updated presence information to all the subscribers/Watchers. The more detailed UCM model will be discussed in section 4.3.1

3.1.4 Performance Issues of Presence System

In the presence system, the mean response time is counted from the start of the UpdatePI scenario to the end of the reply. The mean notification time is counted from the start to notify the first subscriber to the end of the last subscriber receive the notification. The throughput of the system is defined as responses per second the system can process.

According to RFC2779, "when a Presentity changes its presence information, any subscriber to that information must be notified of the changed information **rapidly**"[RFC2779]. This means that the mean notification time should be rapid. On the Internet, usually up-to-the minute knowledge of users' status should be notified. Then the mean response time and mean notification time should be within minutes. Following performance issues of the presence system should be considered when design the system.

Efficient: The system must be able to rapidly publish presence notifications to large numbers of subscribers simultaneously. There are response time and notification time constraints for UpdatePI scenario. We assume the mean delay should be within several minutes.

Scalable: The system should be scale to the entire Internet and support a large number of users, for example, hundreds or even thousands users. When the user population increases, the response time should still be within the given delay requirements. The users' number is assumed in the range of 0 to 1000.

The design of the presence system must also consider the following requirements.

Extensible: The system is extensible to make addition of new features easily. Users can specify personal preferences. For example, user can chose media type of service (text, voice, data or video), contact list (where can I be reached), contact zone, contact time, where to receive calls (phones, PCs, Palms, Pager, fax, voice, mail, e-mail, in conference)

Interoperable: the system is interoperable among different vendors, versions and among different protocols.

Independent of devices: The system is configurable for a variety of the devices such as PC, workstation, PDA, laptop, mobile phone, phone etc. The system is accessible from web browsers and should be lightweight and portable on most workstations.

3.2 Presence Protocols

A Protocol is a special set of rules governing the format and meaning of the telecommunication frames, packets and messages that are exchanged by the peer entities within a communication layer. Protocols exist at several levels or layers in a telecommunications connection. For example, there are seven layers in ISO layered model. Entities use protocols in order to implement their service definitions. A service is a set of operations that a layer provides to the layer above it.

"SIP Extensions for Presence" [SIP-pres] is an extension of using the Session Initiation Protocol (SIP) [SIP] for presence service. It uses an event notification framework and two message types SUBSCRIBE and NOTIFY defined in [SIP-event]. The SIP and SIP Extensions for Presence protocols are application layer protocols. In the following sections, SIP and SIP extension for Presence are briefly described in section 3.2.1. Then, the SIP entities are introduced in section 3.2.2. Finally, an architecture using SIP entities for presence is described in section 3.2.3.

3.2.1 Overview of SIP and SIP Extensions for Presence

3.2.1.1 SIP Overview

SIP was first proposed in March1999 as RFC 2543 [SIP] and the Internet Draft is continuously updated. The latest version is RFC 3261 [RFC3261], which was published in June 2002. The SIP is an application layer signalling protocol for creating, modifying and terminating sessions with one or more participants. These sessions include Internet multimedia conferences, Internet telephone calls and multimedia distribution. SIP works independently of underlying transport protocols and independent on the type of session that is being established. SIP is a text based protocol and similar to Hypertext transfer protocol (HTTP). SIP reuses many header fields in HTTP such as entity headers, authentication headers that allow integration with web servers easily.

SIP identifiers are email like identifiers of the form user@doamain, the same form as presence identifiers. SIP also has other forms of identifiers such as user@host, user@ip-address, or phone-no@gateway. SIP uses these addresses as part of SIP URIs such as sip:alice@example.com. SIP can use Domain Name System (DNS), mail exchange (MX) records or Lightweight Directory Access Protocol (LDAP) as means of delivering SIP invitations if email address of the user itself is published as a SIP address.

SIP defines six messages or methods: INVITE, ACK, CANCEL, BYE, REGISTER, OPTION. The functions of SIP messages are listed below:

- INVITE message is used to establish media sessions between endpoints.
- REGISTER is used by an endpoint or other SIP agent to notify a SIP server of its current IP address and the SIP URLs for which it would like to receive calls.
- BYE message terminates an established connection between two users in a call.
- ACK message is used to acknowledge final responses to INVITE requests.

- CANCEL message is used to terminate a pending search or call attempt.
- OPTION message is used to query about capabilities and preferences.

3.2.1.2 SIP Extensions for Presence Overview

To support the presence service, IETF defines SIP-Specific Event Notification [SIP-event] and SIP Extensions for Presence protocol [SIP-pres]. SIP-Specific Event Notification provides an extensible framework by which SIP nodes can request notification from remote nodes indicating that certain events have occurred. Two additional messages SUBSCRIBE and NOTIFY are defined and the operations are described.

- SUBSCRIBE is used to request current state and state updates from a remote node. It contains the logical identifiers that define the originator and recipient of the subscription. In the SUBSCRIBE message body; the request URI identifies the Presentity.
- NOTIFY message is used to inform subscribers of changes in state to which the subscriber has a subscription. It contains body that describes the state of the subscribed resource.

3.2.2 SIP Entities

SIP supports two kinds of entities: User Agent (UA) and Server.

A **UA** is an end device or endpoint. The UA can work as a client as well as a server. SIP gives the definition when UA client and UA server separate.

- **User Agent Client** (UAC) initiates requests to a server.
- **User Agent Server** (UAS) accepts requests from a client and sends reply to the client.

SIP also supports three kinds of intermediate servers:

- **Proxy Server** (Proxy) receives SIP request and, based on information it has available, forwards the request to one or more next SIP entities. For example, a proxy may forward a message to a proxy, a redirect server or an UAS. Proxy can be further classified as Stateless Proxy that doesn't know the call states, and Stateful Proxy that maintains the call states.
- **Redirect Server** (Redirect) receives SIP request, accesses a database or a location server, and tells the client of the address to contact next.
- **Registrar** accepts REGISTER messages from a client. It is typically co-located with a proxy or a redirect server to provide location and availability information of users.

SIP extension for Presence adds a new SIP entity named Presence Agent.

- **Presence Agent** (PA) accepts subscriptions, stores subscription state, and generates notifications when there are changes in users' presence status. Presence agent knows the presence data manipulated by PUAs for the Presentity. It can be co-located with the PUA or with the proxy/registrar. There can be multiple PAs for a particular Presentity, each of which handles some subset of the total subscriptions currently active for the Presentity.

SIP uses the client/server architecture concept and defines the physical entities that may compose of one or more SIP components.

- **Presence Server** (PS) acts as a server to accept requests and serves presence information to clients. It can be a PA or can be a server combined with PA, Proxy, and Registrar together.
- **Presence Client** (PC) acts as a client with a PA that is co-located with PUA. It is aware of the presence information of the Presentity.

The abbreviations such as PUA and PA will be used in the text.

3.2.3 SIP for Presence Architecture

SIP's messages and basic entities are all individually rather simple, but when used in combination become quite powerful and can be used to implement a variety of features.

Because a SIP UA may communicate with other UA directly or communicate with other UA through intermediate servers, so there are many design choices.

The simplest architecture is Client to Client subscription that does not involve a presence server. The Watcher subscribes to the Presentity, and the subscription is authorized. When the Presentity changes state, a notification is sent to the Watcher directly.

In this section, we show a notification architecture that is implemented using SIP components. Figure 3.3 illustrates the UCM model of the notification system. There are two scenarios in this example, one is Subscribe and another is UpdatePI. Three components are PUA, Presence Server (PresenceServer) and a set of Watchers. The Presence Server is a PA combined with a Proxy and a Redirect/Registrar.

For scenario Subscribe, one Watcher first sends subscribe message to the PresenceServer to subscribe to a target Presentity. Then the PresenceServer sends reply message to the Watcher as well as sends the current PI of the Presentity to the Watcher. Here assuming the Watcher is allowed to subscribe to the Presentity.

For scenario UpdatePI, the PUA first updates its PI, and informs the PresenceServer about the updated PI state. Then the PresenceServer notifies all the Watchers that already subscribe to the Presentity immediately. At last, when Watchers receive the notification, they send reply message to the PresenceServer.

In figure 3.3, multiple Watchers are shown, which indicate there may be multiple Watchers subscribing to the PressenceServer and multiple notification messages to watchers. Multiple Watchers will be modelled more completely in the next chapter.

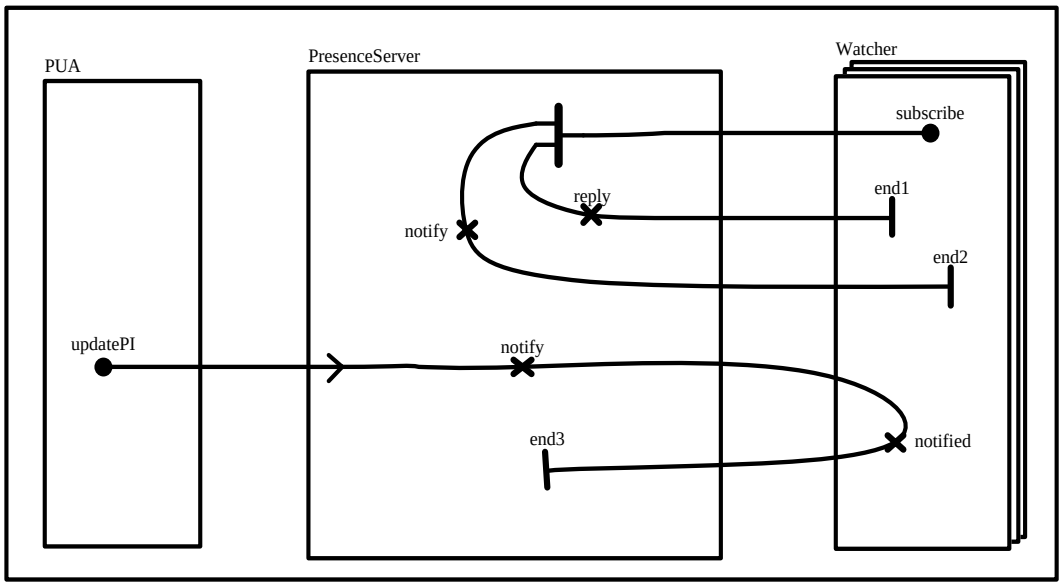

Figure 3.3 Notification System Implemented with SIP UCM Model-1

The Presence Server can be further refined by three components: a PA, a Proxy, and a Redirect/Registrar. We assume only one Proxy is used here. In practice, any number of proxies can be implemented in the system. The Registrar and Redirect servers are located together. In the UCM model, we use the responsibilities to represent the executions that the components process. We capture the main process flow and neglect some information messages between the components PUA, PA, Proxy and Redirect/Registrar.

Following figure shows a notification system implemented with these additional SIP components.

In Figure 3.4, the PUA informs the PA about the updated PI. And the PA sends the notification message to the Proxy. The Proxy sends message to the Redirect/Registrar server and the Redirect/Registrar searches the current address of the Watcher and replies the current address to the Proxy. Then the Proxy sends the notification to the Watchers. Here again multiple Watchers are represented to show multiple notification messages are sent to all subscribers.

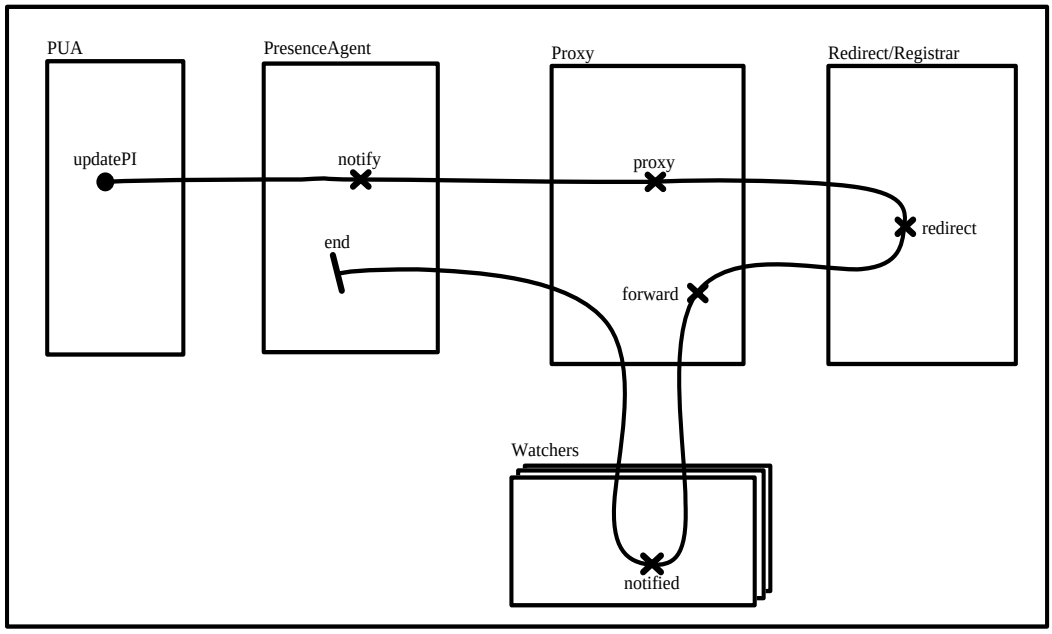

Figure 3.4 Notification System Implemented with SIP UCM Model-2

Chapter 4 Multilevel Specification and Performance Analysis of Software Architecture

This chapter presents a Multilevel Specification and Performance Analysis (MSPA) methodology. Section 4.1 presents the concept and steps of MSPA. Section 4.2 compares the proposed methodology with the TPB approach [Siddiqui01]. The rest sections 4.3 to 4.4 apply the methodology to the first two abstract models of a presence system. Section 4.3 presents the highest level of abstraction UCM model and the performance analysis on this level. Section 4.4 presents the second level specification and performance analysis.

4.1 Multilevel Specification and Performance Analysis (MSPA) Methodology

It is normal to develop a specification through a series of levels of abstraction [Bruin00]. In this thesis research, a performance analysis is attached to each level, and this is called Multilevel Specification and Performance Analysis (MSPA).

In some situations, the further decomposition of the systems or components is not needed and unnecessary in order to simplify the analysis process. For example, a COTS component is usually treated like a black box and no analysis and modification inside the component are needed. The budgeted demands of these kinds of components can be estimated as a whole to do performance analysis.

Therefore, The feedbacks at the earlier abstract level about the performance problems can be obtained at the earlier design stage. The potential performance problems based on the correspondent level of specification can be identified and suggestions on the system

design such as the architecture, bottleneck resource, and design alternatives may be proposed as early as possible.

Multilevel Specification and Performance Analysis (MSPA) methodology combines the iterative procedure of abstraction with Time/Performance Budgeting (TPB) approach for Software Designs described in Section 2.4 [Siddiqui01]. It applies the performance budgeting analysis to the multilevel abstractions of the predicted system to gain earlier and easier analysis results for large, complex and distributed systems using lots of COTS during the system development cycle.

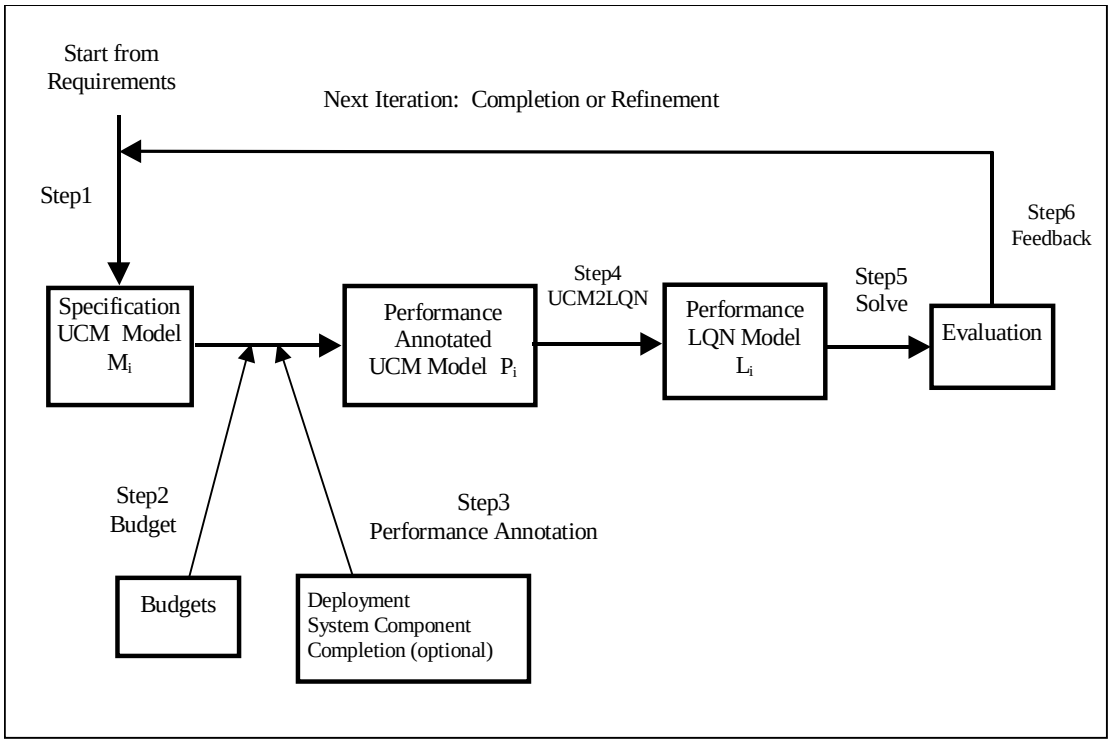

Figure 4.1 Multilevel Specification and Performance Analysis Road Map

Figure 4.1 illustrates the process in Multilevel Specification and Performance Analysis methodology. The boxes represent design artifacts or components used in the analysis, and the arrows represent the operations. There are six steps in the methodology.

Step 1: Define the UCM model at level i of abstraction. If this is the most abstract level model, the model is specified directly from the requirements, otherwise the model is a further refinement or completion model from the earlier abstract model.

The UCM model M_i may be focused on the performance perspective, and may omit paths, which are less important for performance analysis. The specification may include the performance requirements that the system must meet.

Step 2: Estimate budgets for all the components. When doing budgets, we may identify four kinds of components [Brown00].

- For off-the-shelf components (COTS), because they already exist in the market, related performance parameters can be obtained from the vendors. When a certain COTS component is chosen, the known performance parameters can be used to do performance analysis of the system. Or in a reverse way, use permitted budget values to chose a component that meet the given budget expectation.
- For "full-experience" components, because developers have experience in the past similar projects, the performance parameters can be estimated based on developers' experience and the budget values are relatively accurate to some degree.
- For "partial-experience" components, because developers have part experience in the similar projects but not the full experience, the estimated parameters based on partial experience can be guessed. And when new unit testing data are obtained, the values can be used to do performance analysis. Budget analysis may give suggestions on the improvement of the components.
- For "new-development" components, because developers don't have any experience about these kinds of components, performance parameters can be budgeted in the performance analysis. If the unit testing and prototyping data are obtained, the measured values are used to perform the analysis. Budgeting analysis can give

suggestions on the parameters' range, architecture alternatives, or process algorithm to satisfy the performance constraints.

Step 3: Add performance annotation to make the UCM M_i a performance annotated model P_i. This step adds details of the execution environment including the hardware and software aspects, such as processor speeds, components' allocation to processors and number of users or arrival rate of the system.

Step 4: From the performance annotated UCM model P_i, create the correspondent LQN model L_i using the UCM2LQN converter. Some execution controlling values must then be added to the LQN model in order to use SPEX controller to get a series of outputs of the LQN model.

Step 5: Solve the above model L_i to obtain the performance results, such as throughput, mean response time and system scalability. In this step, a series of experiments are executed to give the throughput and mean delays versus different numbers of users and different service demands of operations.

Step 6: Analyse the output to give feedback about the specification model M_i , and compare the results with the performance requirements defined in step 1 if they exist. Compare the performance results with the previous level results. If the performance results are inconsistent, reasons of the difference should be investigated. The current level of performance budget analysis is based on a further refined performance model and thus can give more specific results than the previous analysis.

When the system development evolves into the next design stage, the system is further refined. New components are added and stubs may be defined in plug-ins. Then the Specification UCM model is incremented to M_{i+1}. Repeating the steps 1 to 6 performs the iterative analysis process.

4.2 Comparison of MSPA with TPB Approach

The MSPA methodology presented in this thesis combines the TPB approach with the iterative refinement procedure. Most steps are same for both MSPA and TPB approaches, while the main difference between the two is the position of the Completion step on the UCM level step. The TPB approach processes the Completion on the UCM level step (step 3) to create a complete UCM model and on the Additional Completion step (step 5) on the LQN model. The MSPA approach processes one Completion step (Step 3) on the UCM model to make a performance annotated model, while the additional completion and the refinement are performed in the next iteration when new specification UCM model is built in step 1.

The benefit is getting performance results earlier than TPB approach. With TPB, it is not clear when the specification is ready to be evaluated. There is a temptation to wait until the UCM is quite complete. This delays evaluation.

MSPA encourages the designer to begin to evaluate the model early, and to maintain and improve it as design evolves.

The following sections give an example of applying the MSPA methodology to the first two levels of the presence system.

4.3 Apply MSPA to Presence System UCM model M1

4.3.1 Define the Specification UCM Model M1

According to the Presence System's requirements described in the section 3.1.3, the most abstract UCM model M1 has client/server architecture.

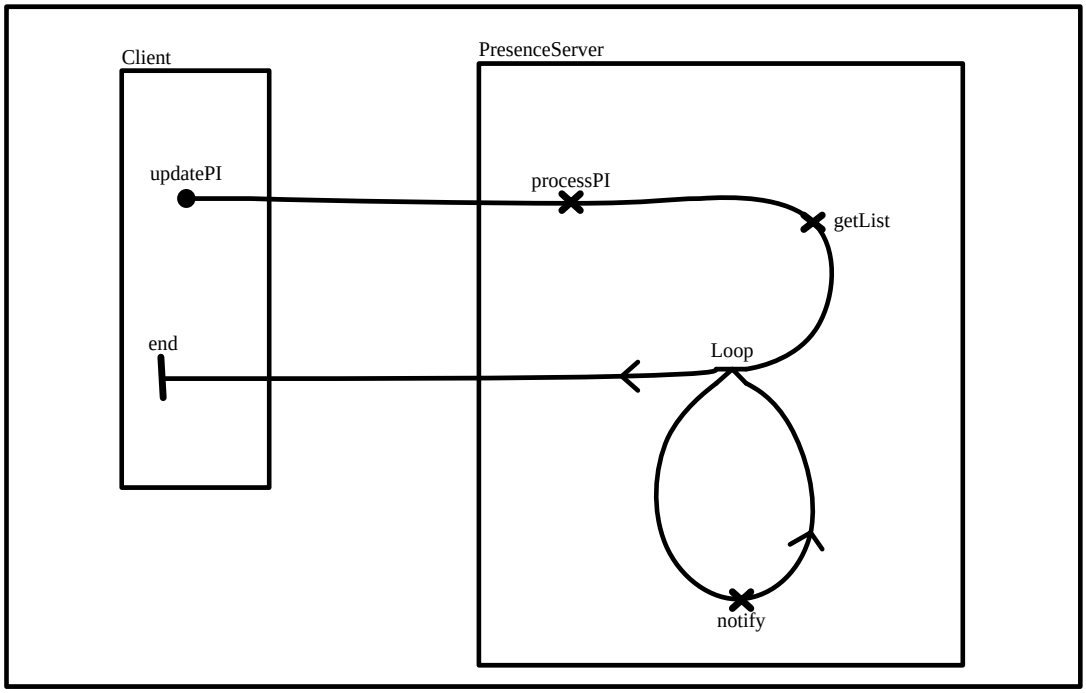

Figure 4.2 Presence System UCM Model M1

In the UCM model M1, there are two components in the model. One is Client and another is Presence Server (PresenceServer). The Client's responsibility is accepting the user's input and sending the user's request to the PresenceServer. The PresenceServer's accepts the request and processes the request and then sends the reply to the Client. In UCM model M_1, only UpdatePI scenario is considered due to the performance importance of this scenario. Other scenarios are less executed compared with this scenario.

For the presence system, the PresenceServer's activities can be further specified as processing request (processPI), getting subscribers' list (getList) and notifying presence information (notify) to all the subscribers. The details of how and where to get the list and the communication mechanism between the PresenceServer and watchers are not considered at this abstract model. Those design details may be added as completion in the next abstract model.

In this level, the performance concerns are scalability of the system when a lot of users access the system at the same time and relationship between the system throughput

and mean delay and total users number and subscribers' number.

The system can be modeled as a "closed model" if a fixed number of end users interact with the system. Each user submits a new request to the system and waits for the reply. The system processes the request and sends the response to the user. The user spends time examining the response and enters the next request. The closed model needs the number of users and the "thinking time", i.e. the average delay between the receipt of a response and submission of the next request.

The performance metrics are throughput, system response time and processors' utilization; we may also be interested in the utilization of software components.

Following are some assumptions when we make performance analysis.

- For simplicity, the only scenario to be analyzed for the presence system is UpdatePI. Scenarios such as time-out and failure exceptions are not considered in the model.
- Z is end user's thinking time. We assume Z equals to 10 minute, i.e. 600000 ms. This is assume that users change their presence status every 10 minutes.
- The budgeting CPU demands of all operations are initially set to 10ms and then are increased to 100ms to test the system's sensitivity to each operation.
- N is the number of total users. In the experiment, N is varied from 21 to 921 in steps of 100.
- Nsub is the subscriber's number to a Presentity. The more subscribers to be notified, the more workload the presence server will have. The maximal Nsub is N-1, which means every user subscribe to each other. In the following experiment, Nsub is set as a constant, for example 20, 10% of N and 20% of N.
- All data are stored in main memory; so there are no disk operations.

4.3.2 Create Performance Annotated UCM Model P1

A performance annotated UCM model can be created by doing step 2 and step 3. The budgeted demands are added to the UCM model with UCMNav editing tool. In the model, all responsibilities' demands are initially set to 10ms. Two processors are defined, UserP and PresenceSP. And the component Client is allocated to UserP while PresenceServer is allocated to PresenceSP. The processors' processing rate is set to 1.0.

The system is a closed system and the total users number N is defined in the UCM model. In the PresenceServer component, there is a loop which represents the subscribers' number Nsub. The notify responsibility is repeated for Nsub times to notify subscribers that subscribe to the updated Presentity. With its performance parameters, the UCM model becomes a performance annotated model P1.

4.3.3 Create LQN Model L1

Following step 4 of MSPA, a LQN model L1 is created using UCM2LQN converter. After a converted LQN file is created, some values are added to get a series of test results (See Appendix A: L1.xlqn). For example, the total users' number N, subscribers' number Nsub.

Figure 4.3 is the LQN model L1.

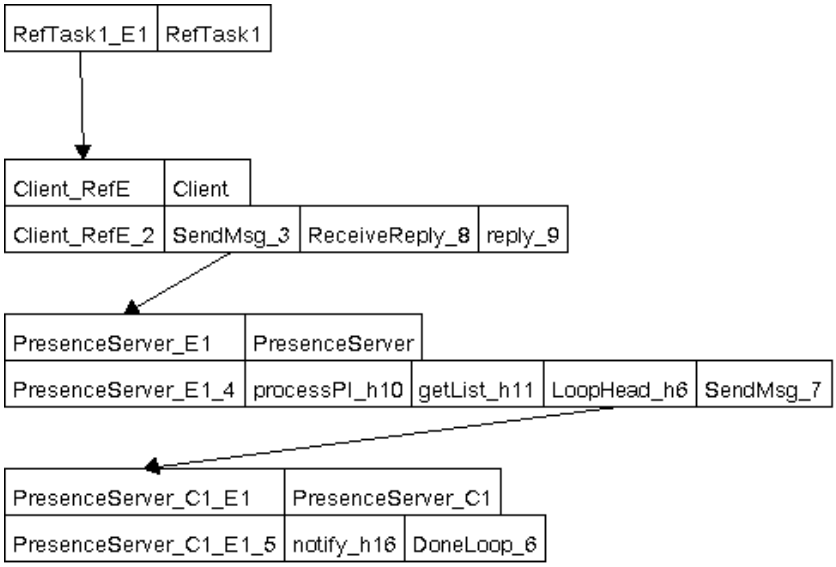

Figure 4.3 Presence System LQN Model L1

RefTask1 is an artificial reference task and creates workload of the presence system with multiplicity N. It makes a synchronous call to the Client task and waits the reply and then thinks for 10 minutes and makes a next request. The entry Client_RefE of task Client has four activities that are created by UCM2LQN tool. The default activities' CPU demands are set to 0 if they are not specified in the model P_1. The Client task then makes a synchronous call to PresenceServer task. The task then makes a number Nsub of synchronous calls to task PresenceService_C1, which is a pseudo task to represent the loop construct.

4.3.4 Performance Analysis Results

The model L1 is solved by LQNS or by ParaSRVN with a set of experiments. The performance analysis concerns about the throughput and mean response time change with the total users' number, subscribers' number and sensitive CPU demand values of some operations.

4.3.4.1 Performance Analysis for Scalability

Following is the list of experiments for system scalability.

- Test L1-1: the end users' number N ranges from 21 to 921 with increment number 100. The subscribers' number Nsub is fixed as 20.
- Test L1-2: Subscriber's number Nsub is 0.1*N. The total end users are from 21 to 921, and the subscribers ranges from 2 to 92.
- Test L1-3: The subscriber's number Nsub is 0.2*N. The total end users are from 21 to 921, and the subscribers ranges from 4 to 184.

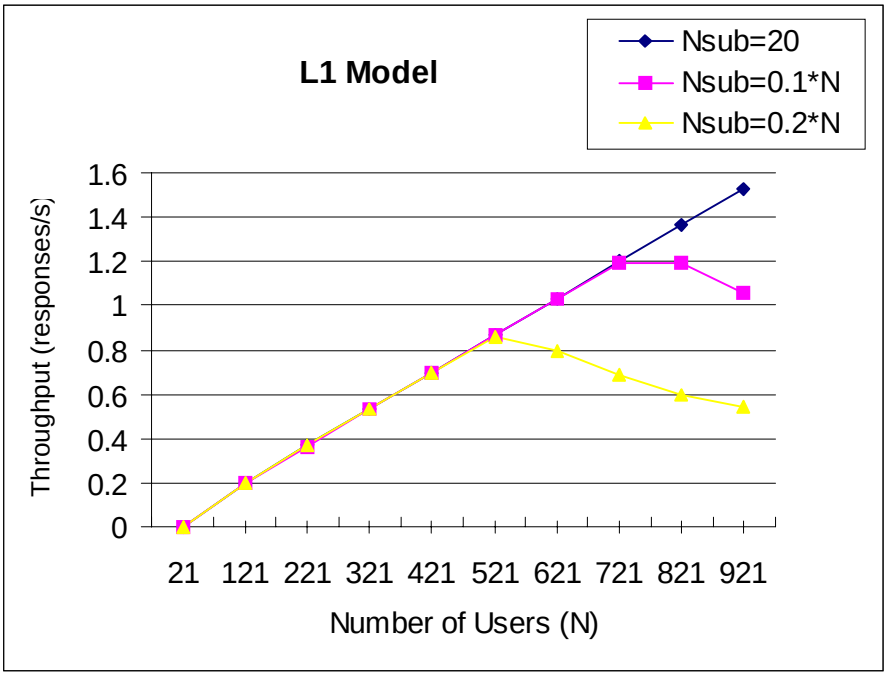

Figure 4.4 Scalability Tests: Throughputs for L1 Model

Figure 4.4 shows the output of throughputs versus the number of users. From the Figure4.4, we can see that when the users' number increases, the throughputs increase generally. However, the different cases of number of subscribers Nsub have effect on the throughputs. When subscribers Nsub is fixed as 20, the throughput increases linearly with

the number of users N. When Nsub increases with the total number of users N, the throughputs have upper limits. For Nsub=0.1*N case, the throughput has an upper limit of 1.2 responses/second when users number N is greater than about 700. And for Nsub=0.2*N case, the throughput has an upper limit of 0.85 responses/second when users number N is greater than about 500. The above tests results show that the subscribers' number Nsub and total users' number N have big effect on the throughputs of the presence system. The more subscribers per user, the less the throughputs.

Figure 4.4 also shows that when the users' number N is greater than the saturation limit, for example, 700 for case Nsub=0.1*N, the throughput declines when the users' number N increases. This is because subscribers' number Nsub is increased at a 10% rate, then the number of notification call is increased also, then the PresenceServer's service demand increases, which makes the throughput decrease.

Figure 4.5 shows the mean response time (ms) versus the total number of users N. Response time is from 200ms to 300 ms for the fixed 20 subscribers. This is because the notification processing time is the same for different number of users N. For Nsub=0.1*N case, the response time increases rapidly when users number N is above 700 and subscribers number Nsub is above 70. For Nsub=0.2*N case, the response time increases dramatically when the number of users N is above 500 and subscribers number Nsub is above 100. The reason is that in above two cases, system is saturated and the response time increases explosively at saturation situation. The response time becomes unacceptable for the presence system.

From the Figure 4.6, we can see that the presence processor's utilization is 1 when throughputs reach their upper limits for both cases. The presence processor saturates when N is above 700 and 500 respectively for case L1-2 and case L1-3.

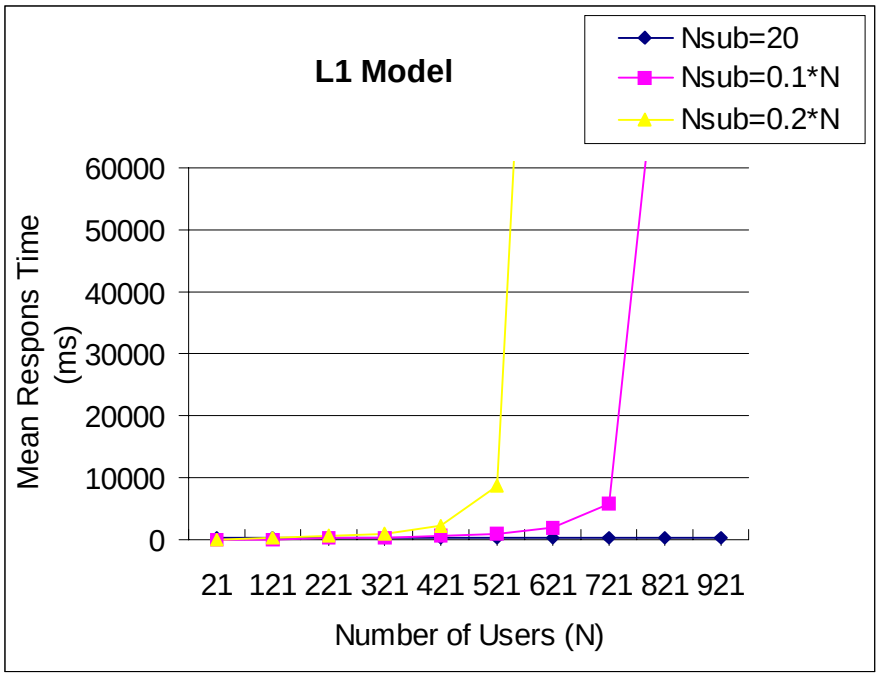

Figure 4.5 Scalability Tests: Mean Response Time for L1 Model

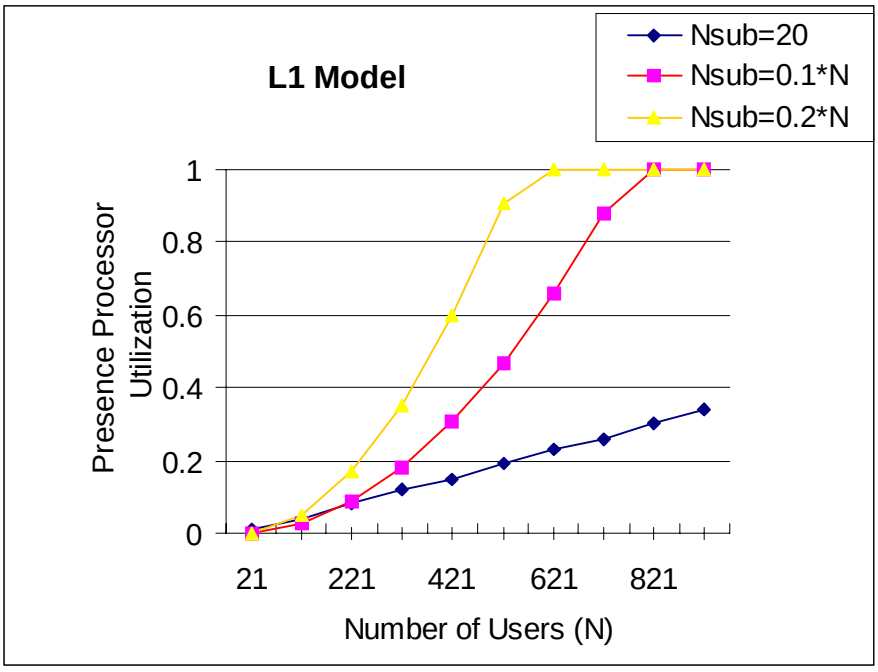

Figure 4.6 Scalability Tests: Processor Utilization For L1 Model

4.3.4.2 Performance Analysis for Sensitivity

Now assume the subscribers number Nsub is 0.1*N. We will increase the CPU

demand of each operation in the PresenceService task to see the sensitivities of the influences of the budgeted demands.

- Test L1-4: increase **processPI** demand to 100ms.
- Test L1-5: increase **getList** demand to 100ms.
- Test L1-6: increase **notify** demand to 100ms.

From Figure 4.7 to Figure 4.9, four curves are plotted for above three test cases and the case L1-2 (called L1 model base case). We can see for cases L1-4 and L1-5, the increased budgets for processPI and getList operations do not change the performance outputs very much. The curves are almost the same compared with the base case when the budgeted demands are 10 ms. However, for test case L1-6, when the budgeted value of notify is increased from 10 ms to 100ms, the response time increases greatly when the number of users N is greater than about 200. The system saturates at about 200 users and 20 subscribers per user in the system. The throughput's upper limit is 0.4 instead of 1.2 for base case. The presence processor is saturated at 200 users point.

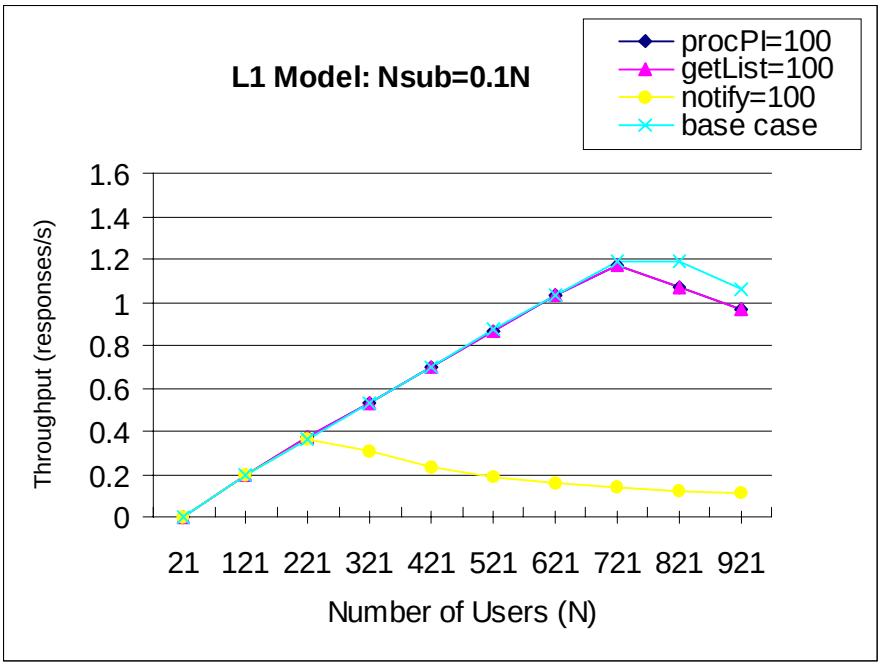

Figure 4.7 Sensibility Tests: Throughput for L1 Model

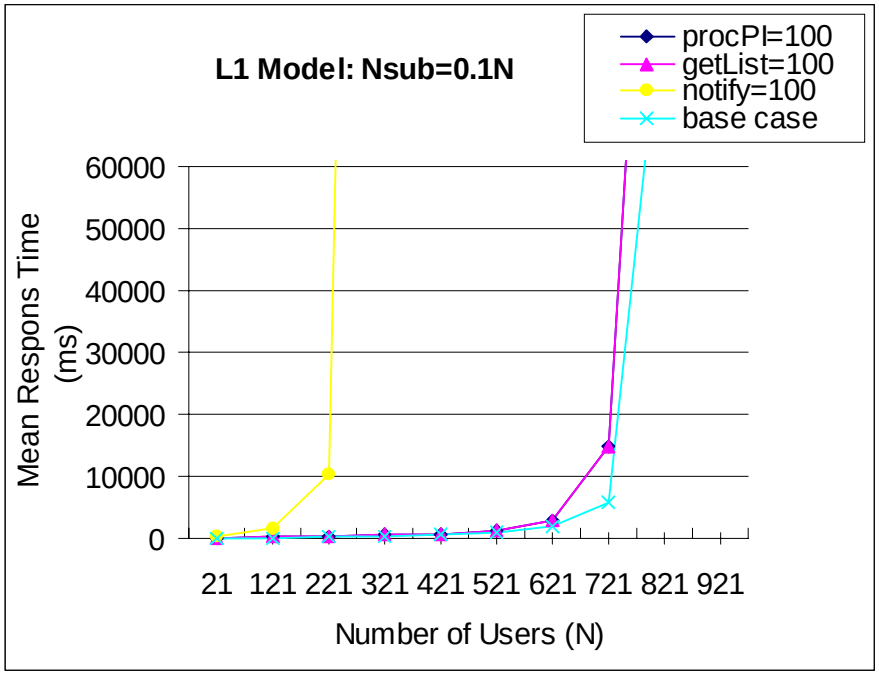

Figure 4.8 Sensibility Tests: Mean Response Time for L1 Model

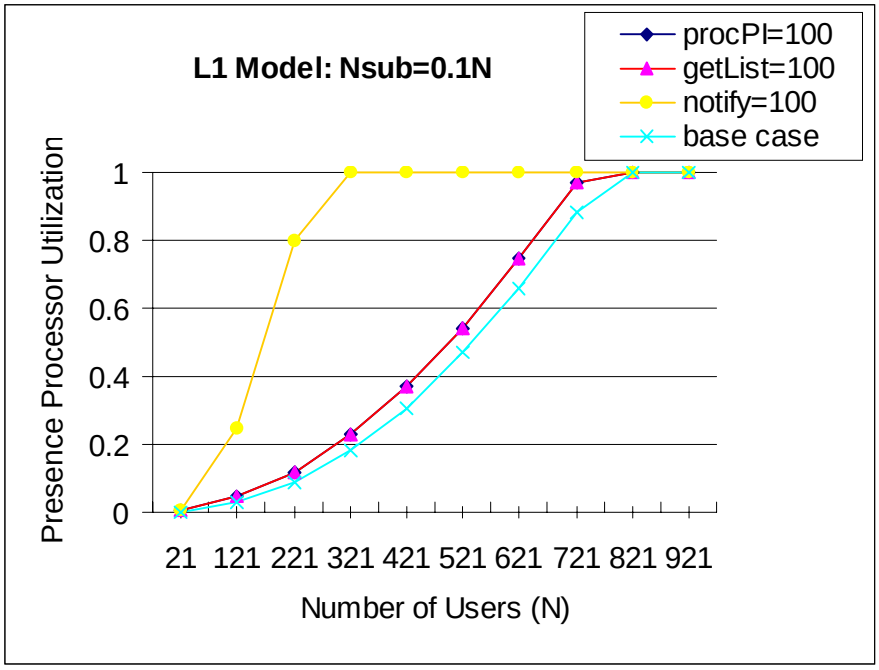

Figure 4.9 Sensibility Tests: Processor Utilization for L1 Model

4.3.5 Discussion of Model L1

The tests results tell us that the scalability of the presence system is sensitive to the total users number and subscribers' number per user. When the total number of users is fixed, the less subscribers per user, the better performance response of the system.

The sensibility concern is the budgeted demand of notify operation. Under the same condition of the number of total users and subscribers per user, the notify operation's CPU demand is more sensitive than processPI and getList operations. This means that the notify mechanism should be a main concern when doing further development.

4.4 Apply MSPA to Second Level Model M2

We know that the presence system should support a large numbers of heterogeneous end users like PCs, PDAs and IP telephones. They work on different devices to interact

with the presence system through Internet. A common architecture uses a web server to communicate with different kinds of end point devices. The end point devices use browsers to interact with the end users. The HyperText Transport Protocol (HTTP) is used to transmit the request and response between the browsers and the web server. The web server receives the request and acts upon it. When dynamic response contents are needed, web server interacts with an application server such as presence server to get the dynamic information and then replies to the users' applications.

4.4.1 Define the Specification UCM Model M2

In the presence system specification UCM model, we can refine the model M1 by adding a Communication component, which might be a web server. The PresenceService component is decomposed as Communication and Presence Server (PresenceServer) components. The Communication component receives requests from the Client component and forwards the requests to the PeresenceServer component. When the PresenceServer component sends reply to Client, the messages are relayed by the Communication component.

Figure 4.10 shows the architecture of the UCM model M2. The system has three components: Client, Communication and PresenceServer. The Client's responsibilities are the same as in model M1. The responsibilities of the PresenceServer are **processPI, getList** and **notify**. A new responsibility **communication** is added to the Communication component.

At this abstract level of performance analysis, the performance concerns are the effect of new component Communication. And the scalability of the system versus total users' number and subscribers' number are studied also to compare with the performance results of level 1.

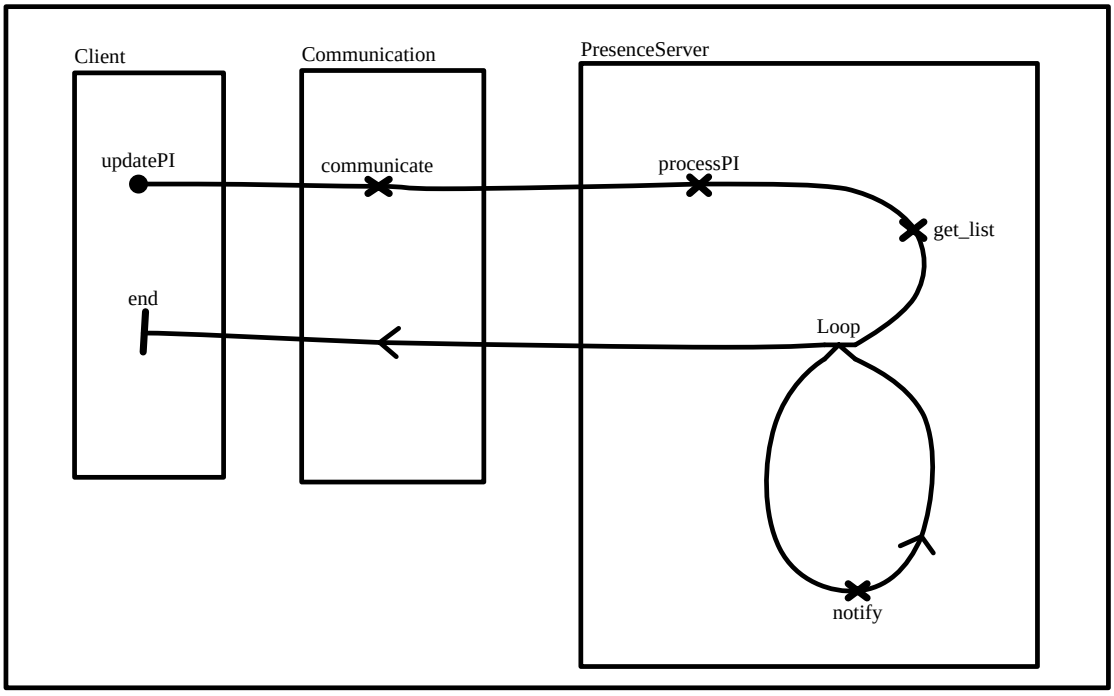

Figure 4.10 Presence System UCM Model M2

4.4.2 Create Performance Annotated UCM Model P2

Following the steps 2 and 3 in MSPA methodology, a performance annotated UCM model is created with the budgeted performance parameters in the UCM model M2. For

step 2, demands are budgeted and added to the UCM model using the assumption described in section 4.3.1 and new assumptions described below.

In this model, we assume the Communication component is a COTS component, such as the Apache HTTP Server, or other HTTP Server. And the budgets can be obtained from third party. The performance analysis on the Communication should concern many aspects such as number of clients, request message size, workload characteristics, architecture, multiprocessors, caching, and threading etc. In this thesis research, only the execution demand of the Communicate component is considered. We assume the Communication component is multithreaded on the working processor for increased throughput, reduced response time and better CPU utilization. The CPU demand of the Communicate component is estimated as 10 ms for light communication delay and as 500 ms for heavy communication delay.

For step 3, two system configurations can be chosen, one is Communication and PresenceServer executing on same processor and another is Communication and PresenceServer executing on two different processors.

4.4.3 Create LQN Model L2

LQN Model L2 can be created by UCM2LQN and is shown in Figure 4.11(See Appendix B: L2.xlqn).

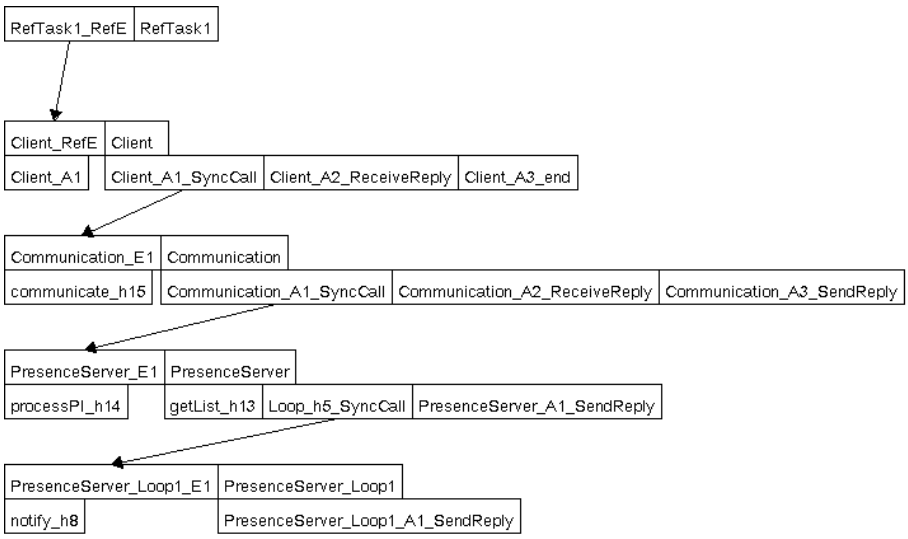

Figure 4.11 Presence System LQN model L2

The LQN model adds new task Communication between the Client and PresenceServer tasks. The Client makes a synchronous call to the Communication and the Communication makes a synchronous call to the PresenceServer task. Other tasks and entries are the same as level 1 model.

4.4.4 Performance Analysis Results

4.4.4.1 Performance Analysis for 1P Cases

For Communication and PresenceServer allocated on the same processor, the following tests are performed. For all responsibilities in the tests, the budget CPU demands are set as 10 ms. We call the following tests 1P cases.

- Test L2-1: the end users' number N ranges from 21 to 921 with increment number 100. The subscribers' number per user Nsub is fixed as 20.
- Test L2-2: the Nsub is 0.1*N, and the Nsub ranges from 2 to 92.
- Test L2-3: the Nsub is 0.2*, and the Nsub ranges from 4 to 184.

Figure 4.12 shows the throughputs against the number of users N. Like level 1 model, when the users number N increases with fixed number of subscribers 20, the throughputs increase linearly. When the subscribers' number per user Nsub increases with the total number of users N, the throughputs have upper limits 1.2 for case L2-2 and 0.85 for case L2-3. The system is saturated and the number of users number N is about 700 and 500 for case L2-2 and L2-3 respectively.

From the Figure 4.14, we can see the utilization of the presence processor is 100%. The saturated device is the presence processor. This is because the Communication and PresenceServer work on same processor. To lessen the workload of the presence processor, we allocate the Communication task to the communication processor. Several tests were performed to see the influence of different budgeted values of communication demand on the system.

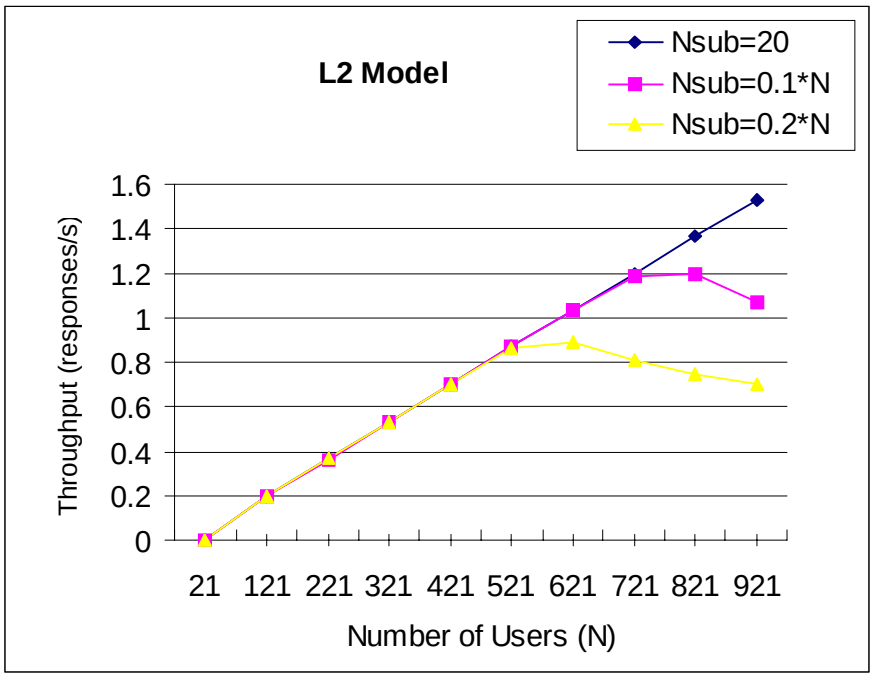

Figure 4.12 Throughput for L2 Model 1P Cases

Figure 4.13 Mean Response Time for L2 Model 1P Cases

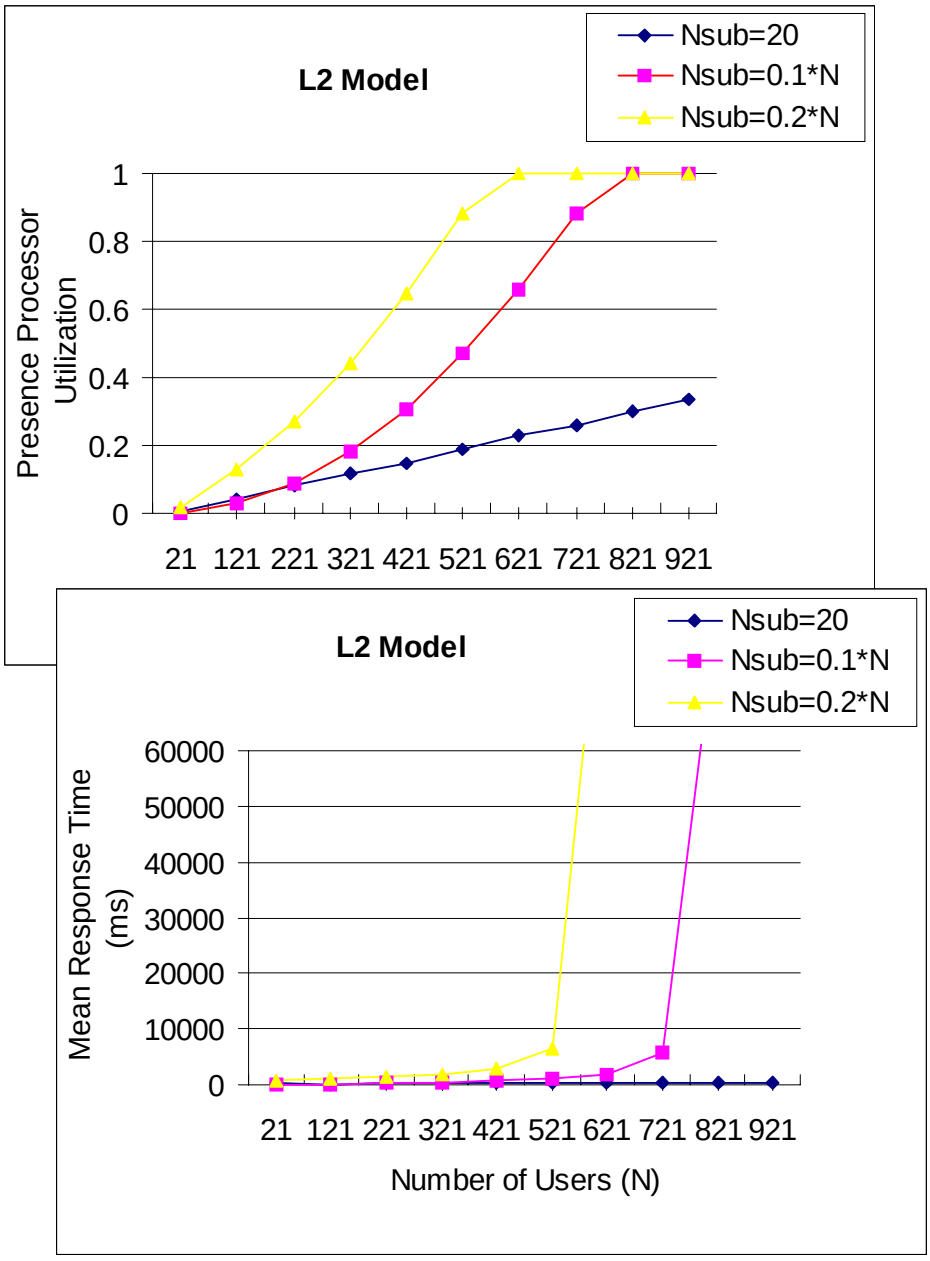

Figure 4.14 Presence Processor Utilization for L2 Model 1P Cases

4.4.4.2 Performance Analysis for Heavier Communication Demand

In the following tests, the Nsub is 0.1*N.

- Test L2-4: budget value of communication demand is 100ms, Communication and PresenceServer work on same processor. This is called 1P case.
- Test L2-5: budget value of communication demand is 100 ms. Communication and PresenceServer work on different processors. This is called 2P case.
- Test L2-6: budget value of communication demand is 500ms. Communication and PresenceServer work on different processors. This is called 2P case.

Figure 4.15 to Figure 4.18 show the test results of above cases.

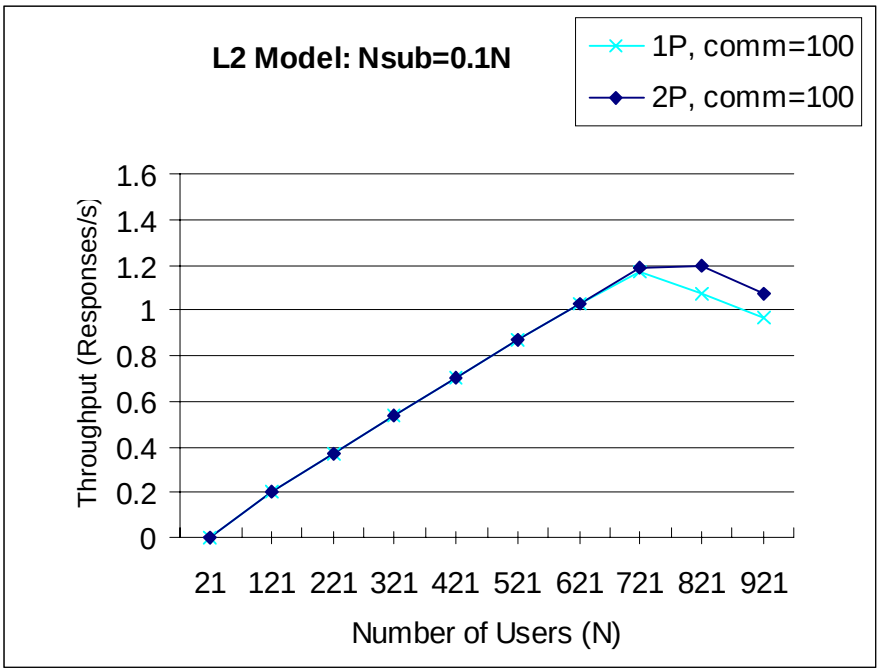

Figure 4.15 Comparison of Throughput for L2 Model

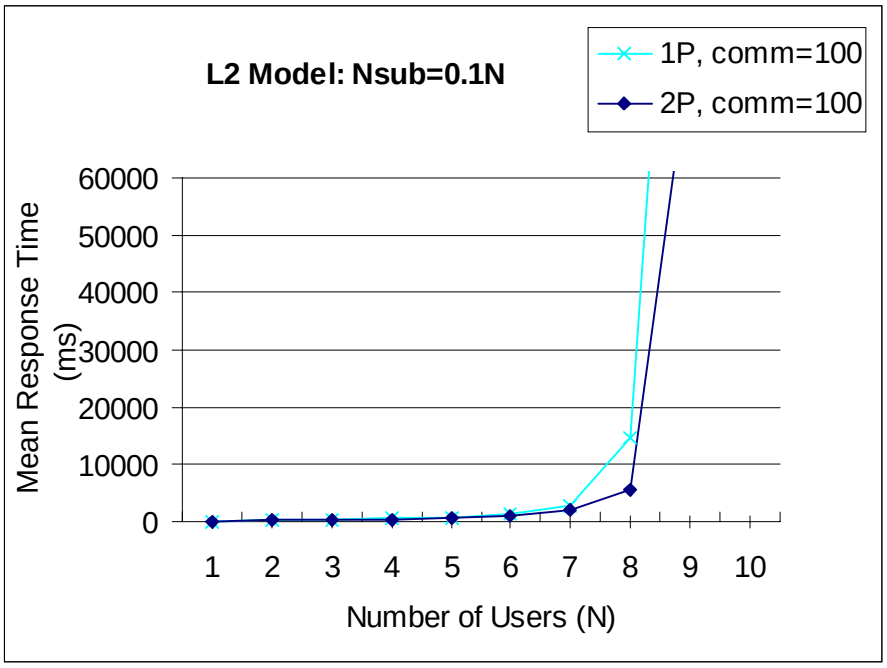

Figure 4.16 Comparison of Mean Response Time for L2 Model

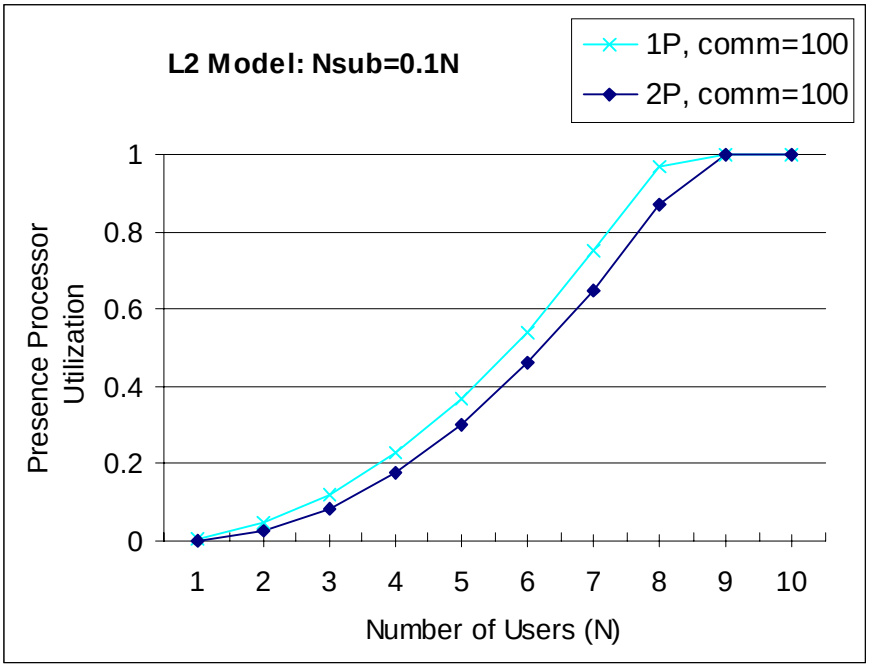

Figure 4.17 Comparison of Presence Processor Utilization for L2 Model

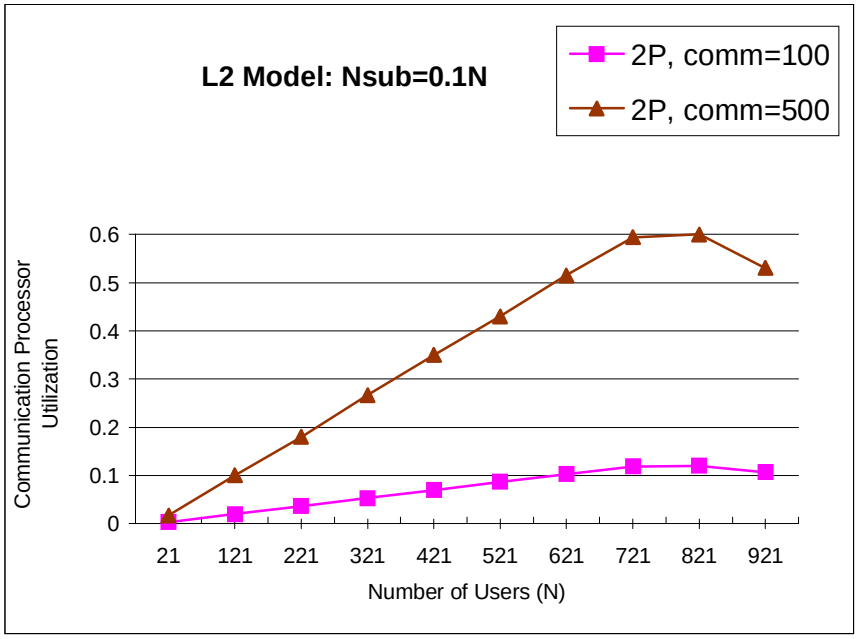

Figure 4.18 Comparison Communication Processor Utilization for L2 Model

The test results show that using two processors for PresenceServer and Communication components improve the performance. When one processor is used for PresenceServer and Communication components, the presence processor is saturated when the users number N is above 700. When two processors are used, the presence processor is saturated when the users number N is above 800 (please refer Figure 4.17). The throughput increases for 2P case (please refer Figure 4.15) and the mean response time decreases (please refer Figure 4.16). The bottleneck device of the system is still presence server when communication demand is in the range of 10ms to 500 ms for both 1P case and 2P cases (please refer Figure 4.17, Figure 4.18).

4.4.5 Discussion of Model L2

The test results tell us that when two processors are used, the performance of the system is improved.

The scalability of the presence system is sensitive to the total users number, the subscribers' number per user and the notification demand. The results are exactly like the level 1 model.

Figure 4.19 and Figure 4.20 illustrate the throughput and mean response time of two different level models L1 and L2. Two curves show the results of test case L1-2 for model L1 and test case L2-2 for model L2. We can see that the performance results are almost identical.

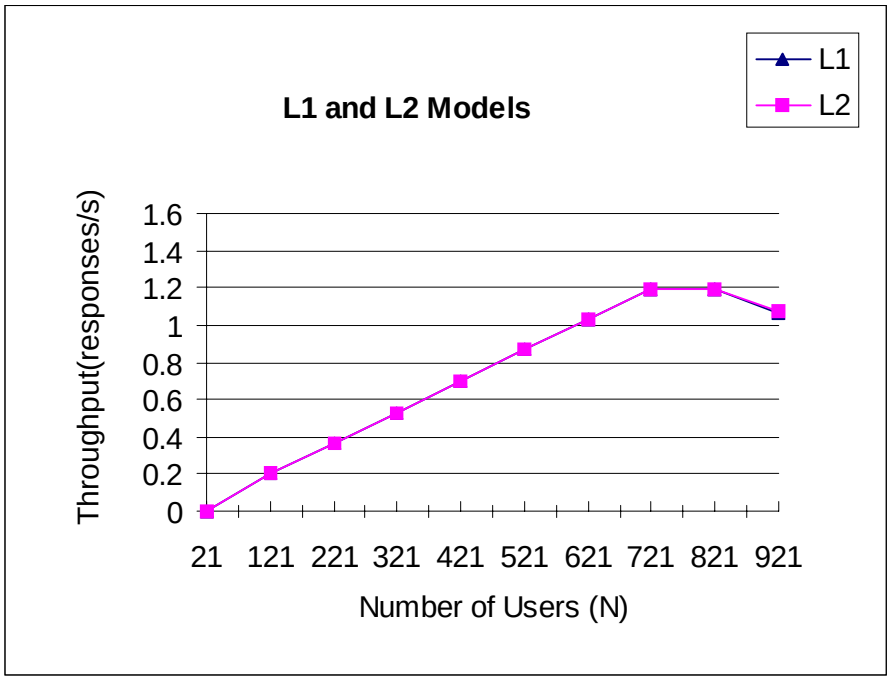

Figure 4.19 Throughput of L1 and L2 Models

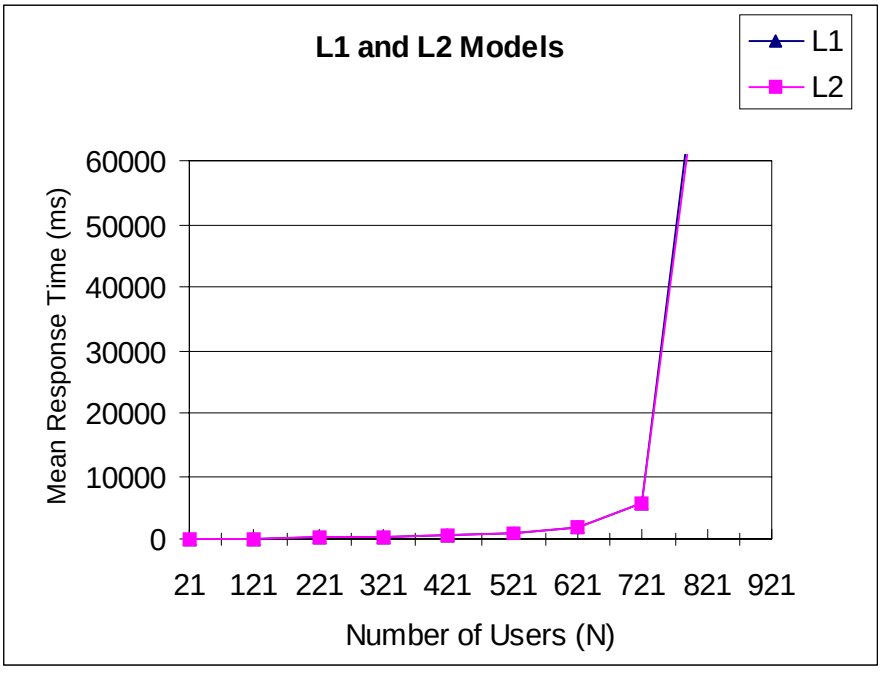

Figure 4.20 Mean Response Time of L1 and L2 Models

Chapter 5 Apply MSPA to Presence System Model M3

In chapter 4, the multilevel specification and performance analysis (MSPA) methodology is proposed and applied to the first two abstract models of the presence system. In this chapter, the iteration process of MSPA is performed to the presence system model M3. Section 5.1 defines the specification UCM model M3. Section 5.2 describes how to create performance annotated UCM model P3. In section 5.3, the LQN model is created by UCM2LQN converter. In section 5.4, some experiments were performed and analysis results are presented. Finally, discussion is presented in section 5.5.

5.1 Define the Specification UCM Model M3

In this iteration process, the presence system is further refined to represent more detailed design and implementation choices. The component PresenceServer in model M2 is refined to Presence User Agent (PUA) and Presence Agent (PA) components. Additional component Watcher is also added to represent the subscribers. The UCM model M3 is illustrated in Figure 5.1.

In Figure 5.1, the scenario UpdatePI is started by the end user entering the current presence information to the Client. The Client sends the updated PI to PUA through Communication component. The PUA processes the PI, and the responsibility is named as **processPI**. After processing PI, the PUA forks two parallel execution paths, one path sends reply message to Client through Communication, and another path sends updated PI to PresenceAgent. The PUA does not wait for the reply from the PA, and it makes an asynchronous call to PA. The PA receives the call and gets the subscribers' list according to the stored subscription information and access policy. This responsibility is named as

getList. Then PA notifies all the subscribers/Watchers. The responsibility is named as **notify**. In the UCM model a Loop constructor is used to represent how many calls the PA makes to Watchers. The count of Loop number is the subscribers' number. The Watchers receive the notification and send reply message to PA and the responsibility is named as **notified**.

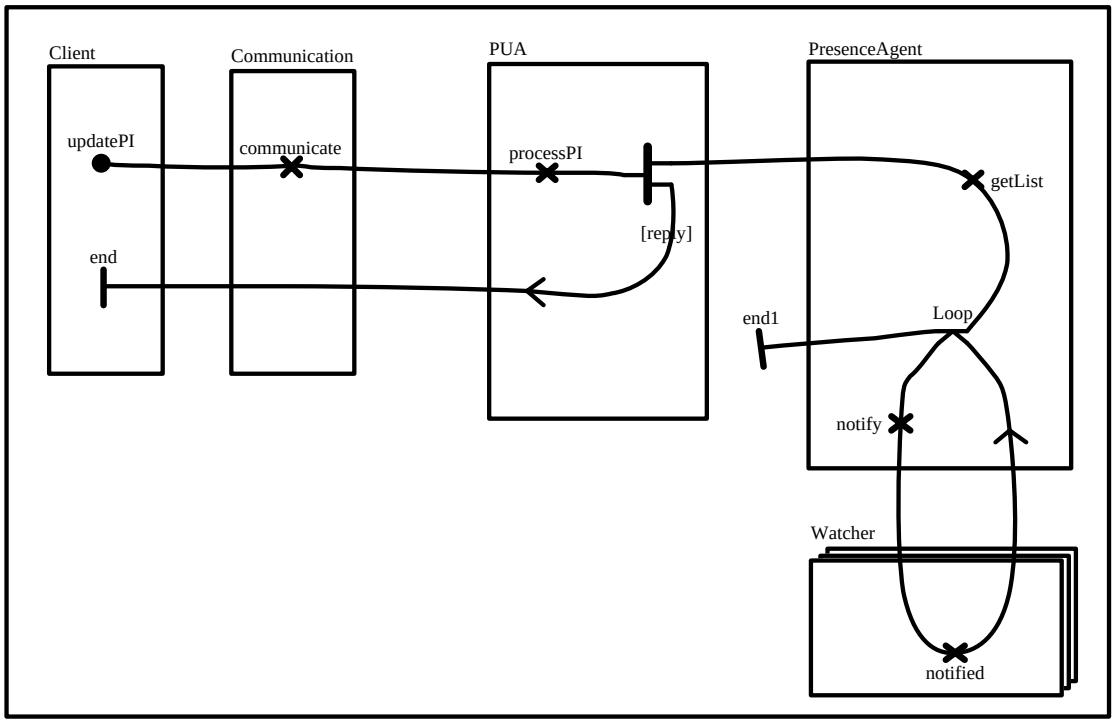

Figure 5.1 Presence System UCM Model M3

The details of notification mechanism and the underline network connection between PA and Watchers will be defined in the next level. The details of how to get the list of all subscribers and get the current contact addresses of all subscribers are not considered in this model either. The subscription information and watchers information may be kept in a database, or in another server such as a redirect, registrar server. Further completion and decomposition will be made in the next iteration model in later chapters.

In this level's performance analysis, the performance concerns are put on the

deployment of PUA and PA and sensitivity of the operations in the two components. The scalability of the system with the users' number and subscribers' number are also analyzed and compared with level 2 results.

The model described above, in which PUA does not block and wait for PA, will be considered as a base case and identified also as an "unblock" case. Later, it will be compared with an alternative case in which PUA is blocked and waits for the reply from PA. This case is called "block" case.

5.2 Create Performance Annotated UCM Model P3

Following MSPA approach step 2 and step 3, the performance annotated UCM model P3 can be created with the performance parameters and the deployment parameters.

The base deployment choice of the model P3 is using three processors: User processor (UserP), Communication Processor (CommP), and Presence Server processor (PresenceSP). The components Client and Watchers are allocated to UserP, and each user has its own executing threads. Communication component is allocated to CommP and has infinite threads working on it. PUA and PresenceAgent components are allocated to PresenceP.

The budgeted demands are added to the UCM model according to assumption described in section 4.3.1. The new responsibility **notified** is budgeted as 2 ms.

5.3 Create LQN Model L3

LQN Model L3 can be created using UCM2LQN from P3 (please refer Figure 5.2). Some execution controlling values are added to make the model L3 be suitable for SPEX execution controller. (See Appendix C: L3.xlqn).

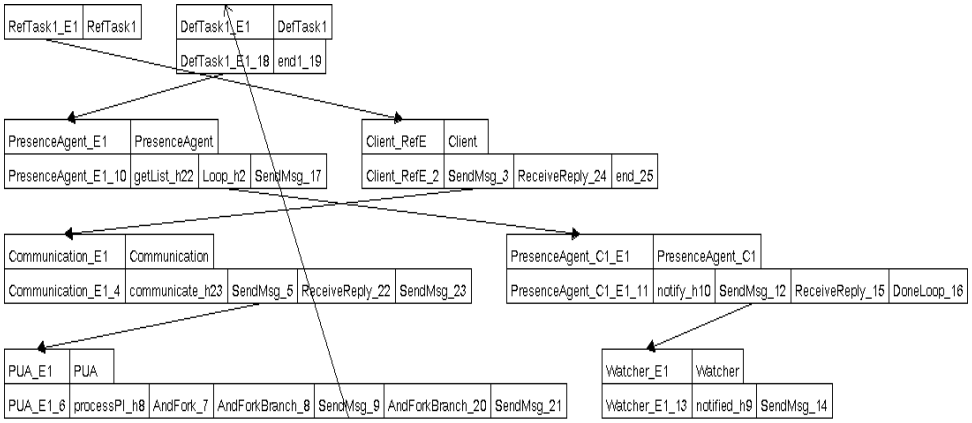

Figure 5.2 Presence System LQN model L3：Unblock case

5.4 Performance Analysis Results

5.4.1 Performance Analysis for Unblock Cases

Several tests were performed with different test parameters. For all tests below, the subscribers' number Nsub increases with the total user number N: Nsub = 0.1*N.

- Test L3-1: All budgeted demands are set to 10 ms. This is called base case.
- Test L3-2: the budgeted demand of **communication** responsibility increases to 100ms and other conditions are same as base case.
- Test L3-3: the budgeted demand of **getList** responsibility increases to 100ms and all other conditions are same as base case.
- Test L3-4: the budgeted demand of **notify** responsibility increases to 100ms and other conditions are same as base case.

Figure 5.3 to Figure 5.7 show the test results for above experiments.

Figure 5.3 shows the output of throughputs against the number of users N. When the users number increase, the throughputs increase linearly for all four cases.

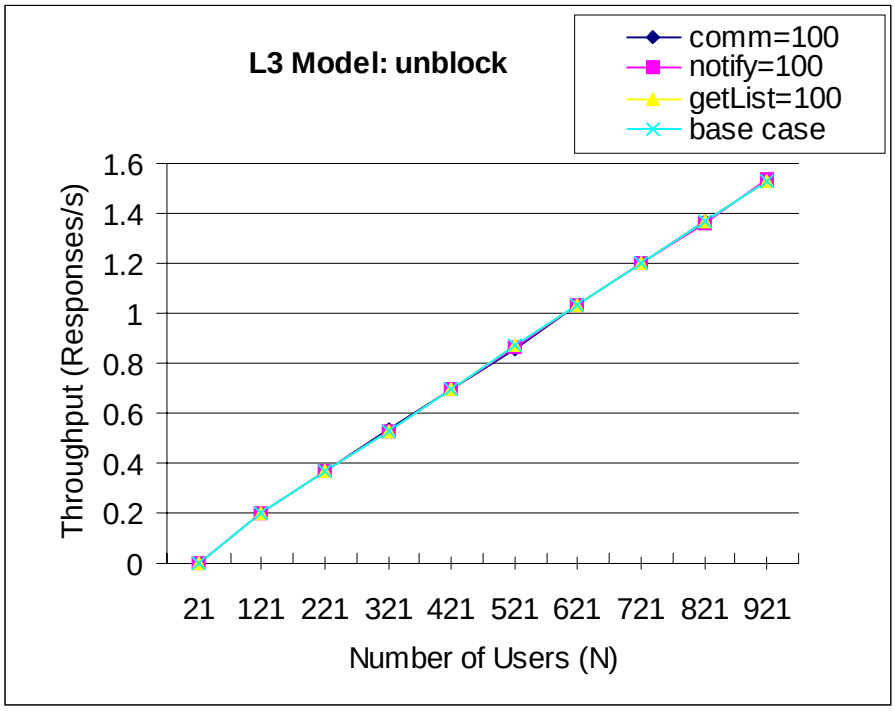

Figure 5.3 Comparison of Throughput for Model L3 Unblock Cases

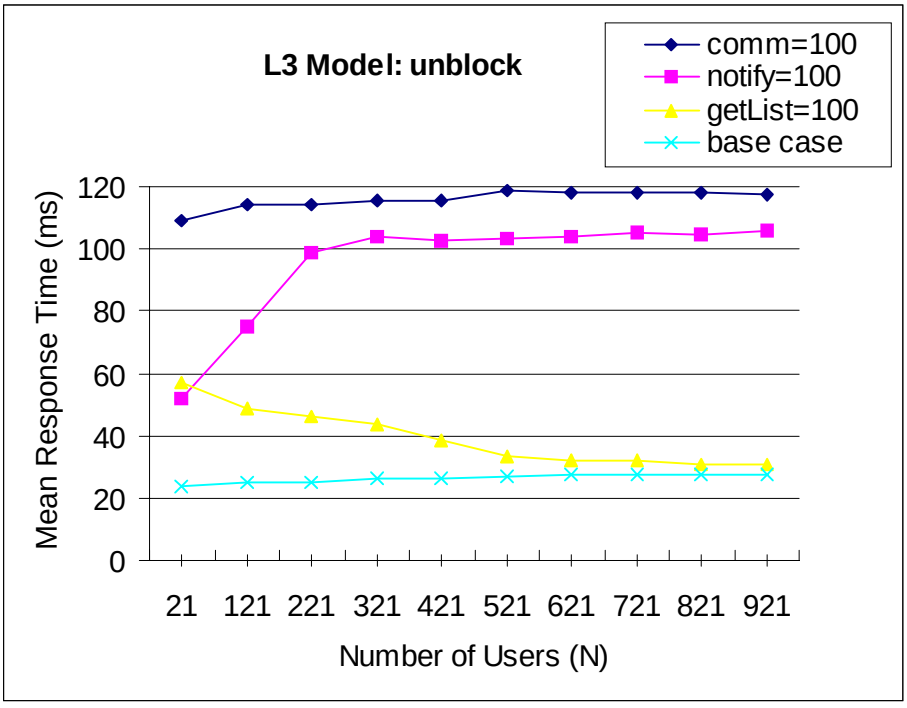

Figure 5.4 Comparison of Response Time for Model L3 Unblock Cases

Figure 5.4 shows the Mean Response Time results for four cases. When increasing the budgeted demands at different responsibilities, the effects are different on the system mean response time. The communication delay has the biggest effect. When communication demand is 100ms, the mean response time is increased from about 25ms to above 115ms. The notify demand also has big effect, when the subscribers' number increases above 300, the mean response time is over 100ms. The increase of getList operation doesn't have much effect on the mean response time. The reason for the different impact on the response time is that the PUA processes the PI first and doesn't block for waiting the reply from the PA. PUA sends reply to the Client through Communication and makes asynchronous call to PA.

Figure 5.5 shows the Presence Processor Utilization results for four cases. The test results show that the bottleneck component of the system is Presence Agent. When the notify demand increases to 100ms, the presence processor utilization reaches 90% when the N is above 200 and can't increase further. When the getList and communication demands are increased to 100ms, the presence processor utilization reaches about 50% when N is above 500 and can't increase further. The reason is that PA task's utilization is 100%. This means that PA is fully loaded and is the software bottleneck in the system even if the Presence processor is not fully loaded to 100%.

Figure 5.6 is the tests results for Communication Processor Utilization for four cases. The communication Processor Utilization is increased only when communication demand is increased. For other three cases, the utilization is almost the same.

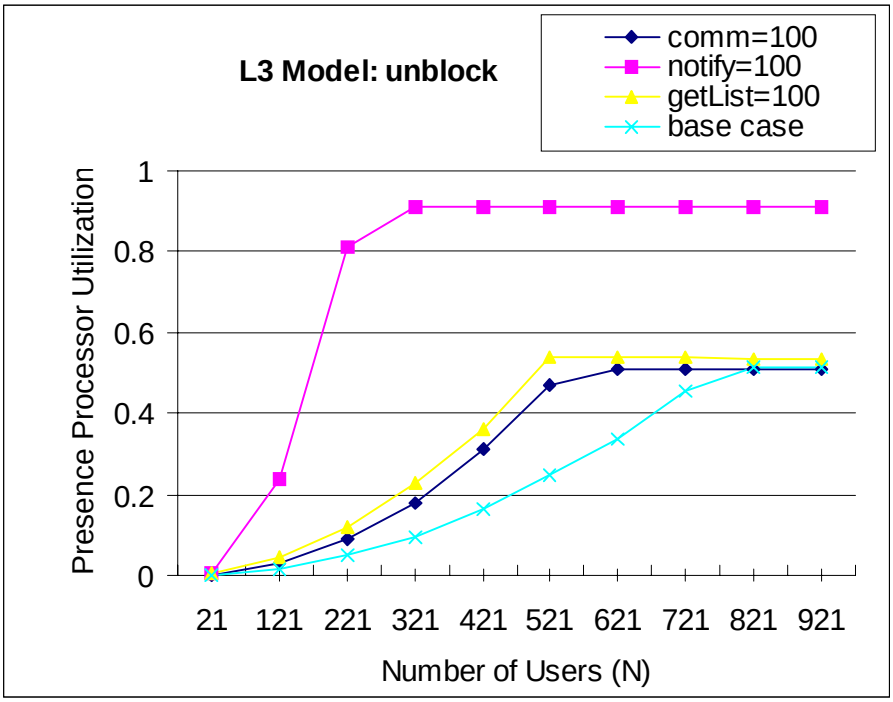

Figure 5.5 Comparison of Presence Processor Utilization for Model L3 Unblock Cases

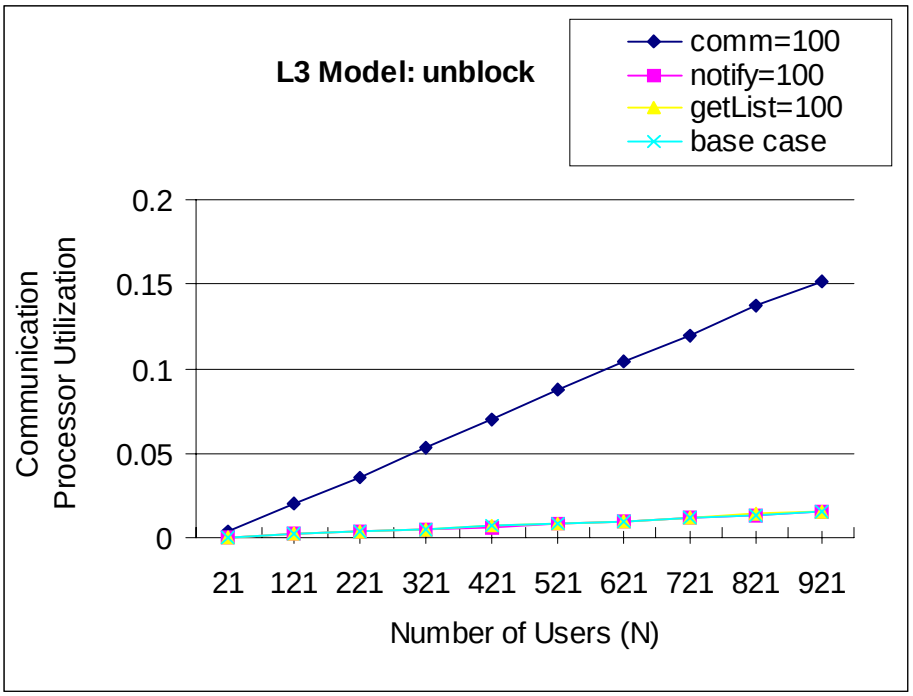

Figure 5.6 Comparison of Communication Processor Utilization for Model L3 Unblock Cases

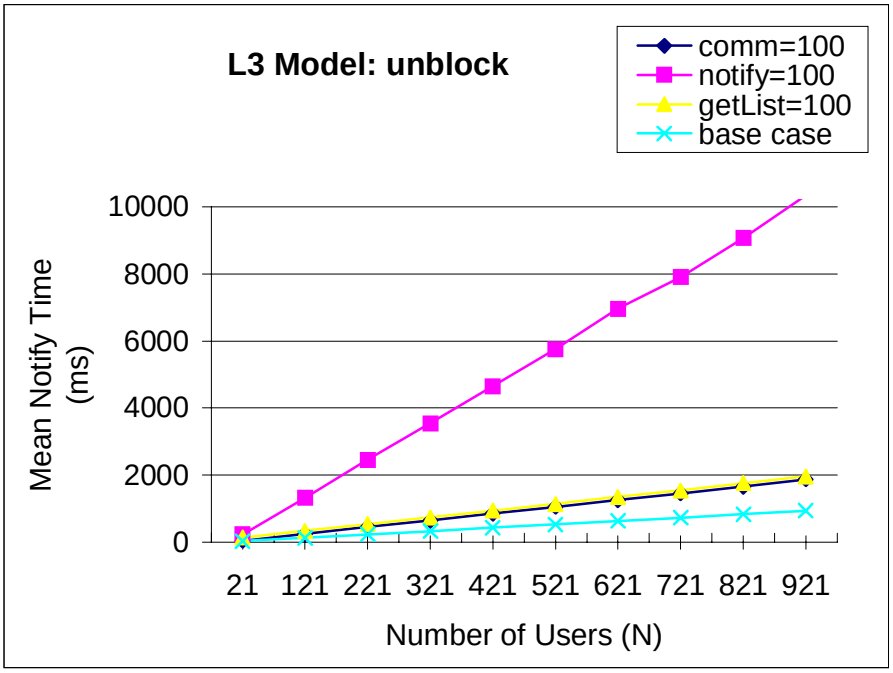

Figure 5.7 Comparison of Mean Notify Time for Model L3 Unblock Cases

Figure 5.7 shows the mean notify time for four cases. The mean notify time is the time between the beginning of the first notify operation to the end of the last notify operation to all the watchers. The test results show that the increase of the communication or getList demand has some effect on the mean notify time, which doubles the mean notify time compared with base case for all N. The increased notify demand has very big effect on the mean notify time. When N increases, the mean notify time increases sharply. When N is 500, the mean notify time is over 5000ms and when N is over 900, the mean notify time is over 10000ms.

From the above four cases results, we know that the presence system saturates when the notify demand increases from 10 ms to 100ms and the user number is over 200 even when the presence processor doesn't saturate. In this case, the system mean response time is about 100ms but the mean notify time is increased drastically because of the software bottleneck exists.

5.4.2 Performance Analysis for Block Cases

The above unblock deployment of the system does not inform the end users when the system saturates. More users may still send requests to the system and worsen the saturation problem further.

Another deployment choice is to let the users know the saturation situation and do not make more requests to the system. Figure 5.8 illustrates the modified UCM model M3block. In this model, PUA sends a synchronous call to PA and waits for the reply from the PA and then sends reply to the Client through Communication. This case is called block case.

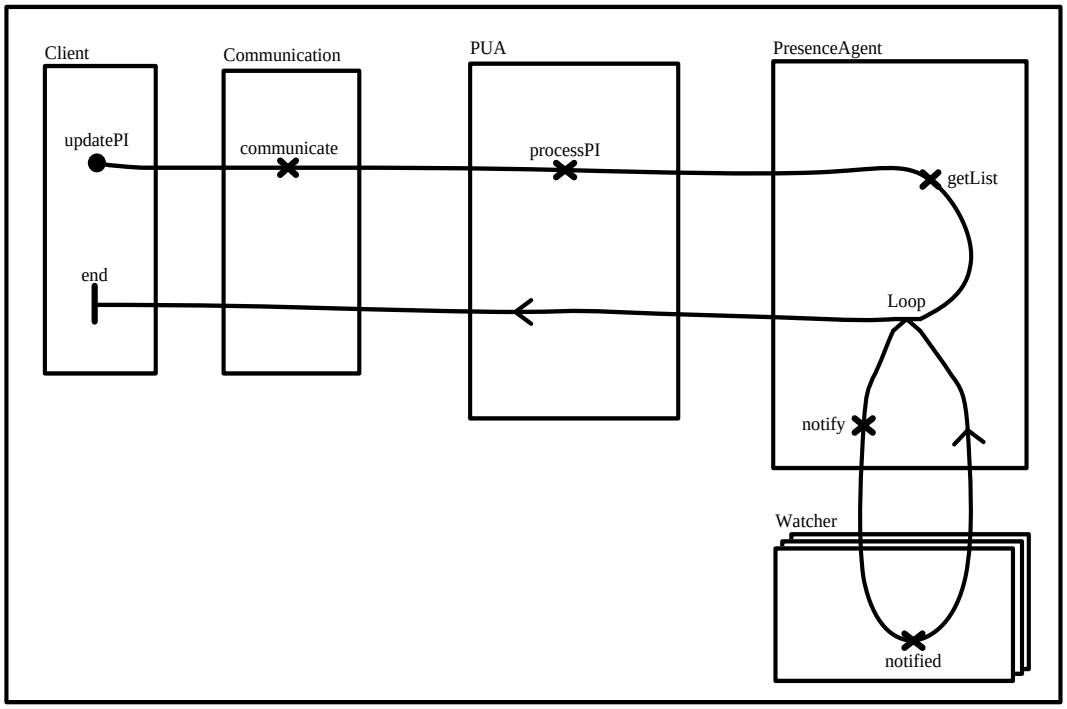

Figure 5.8 Presence System UCM Model M3block

The correspondent LQN Model L3block is shown in Figure 5.9.

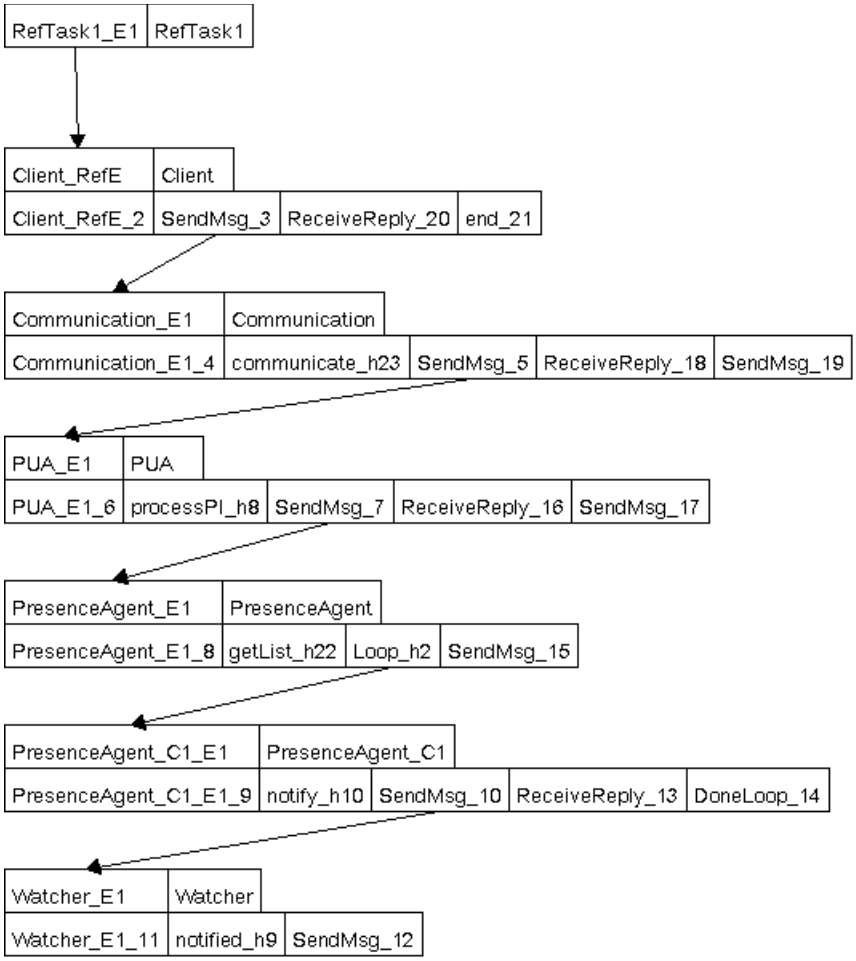

Figure 5.9 Presence System LQN Model L3: Block case

Figure 5.10 to Figure 5.12 show the outputs of the comparison for block and unblock cases.

Figure 5.10 shows the output of throughputs against the number of users N. When the users number increase above 700, the throughputs decrease for block case while the throughput increases linearly for unblock case.

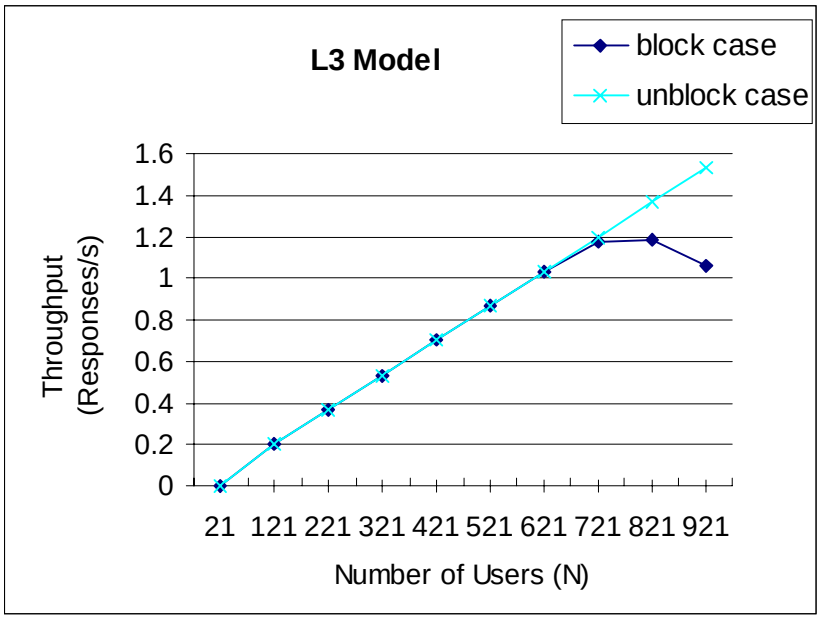

Figure 5.10 Comparison Throughput for Block and Unblock Cases

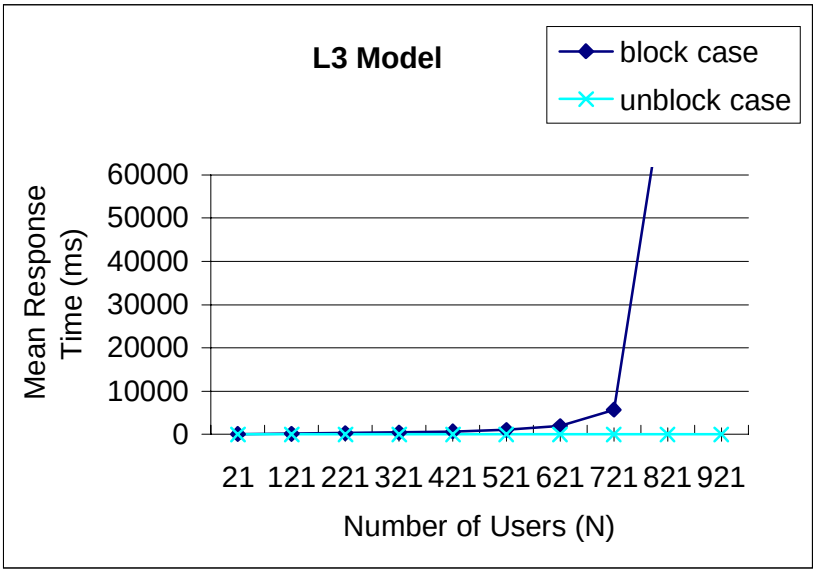

Figure 5.11 Comparison Mean Response Time for Block and Unblock Cases

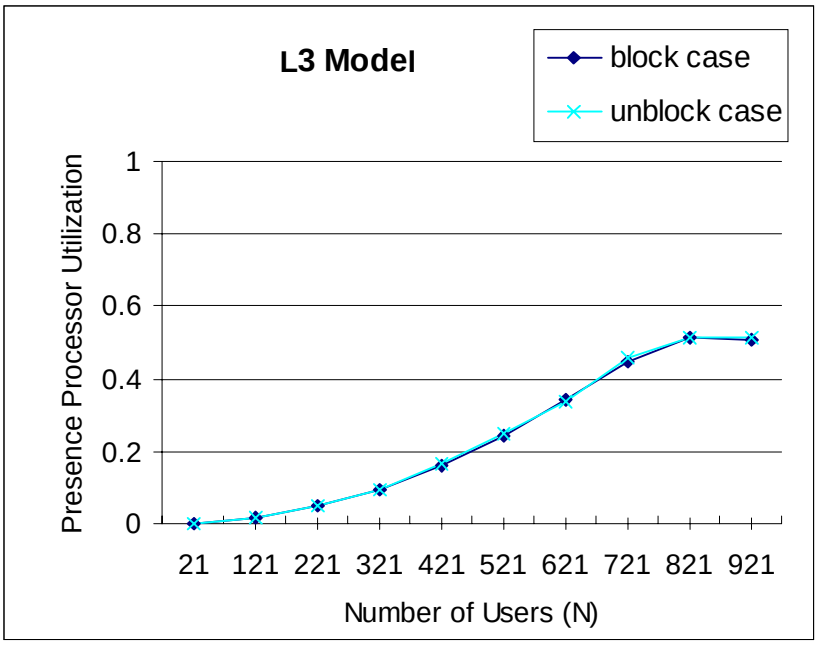

Figure 5.12 Comparison Presence Processor Utilization for Block and Unblock Cases

Figure 5.11 shows the Mean Response Time results for two cases. When the total number of users N increases over 700, the mean response time increases greatly. This indicates that the system's saturation happens for block case. For unblock case, the mean response time does not change with the N.

Figure 5.12 shows the Presence Processor Utilization results for block and unblock cases. The test results show that the bottleneck component of the system is PA task when N is greater than 700 and the presence processor utilization reaches 50%. This is because the software bottleneck happens and the processor is not fully loaded to 100%.

5.5 Discussion of Model L3

The comparison of the results of L3 model with the previous model L2 is done. To make the comparison, we should chose two tests that are compatible.

For L2 model, one processor for communication and one processor for presence server test case is chosen. The demands for all operations are budgeted to 10ms. For L3

model, the block case is chosen with PUA and PA working on presence processor, and communication component working on communication processor. The demands for all operations are budgeted as 10 ms, except notify and notified are 5ms.

Figure 5.13 and Figure 5.14 illustrate the throughput and presence processor utilization of two different level models L3 and L3. The curves show same trends in the change of the outputs with N. The L2 and L3 models give same users number 700 when the system saturation happens. The difference is that the L3 model shows that the presence processor's utilization reaches 50% and can't increase further because the software task PA saturates and L2 model shows that the presence processor saturates first. This is because the details about notify operations are added to the model L3.

Mean response time curves are same for L2 and L3 cases, which do not shown here.

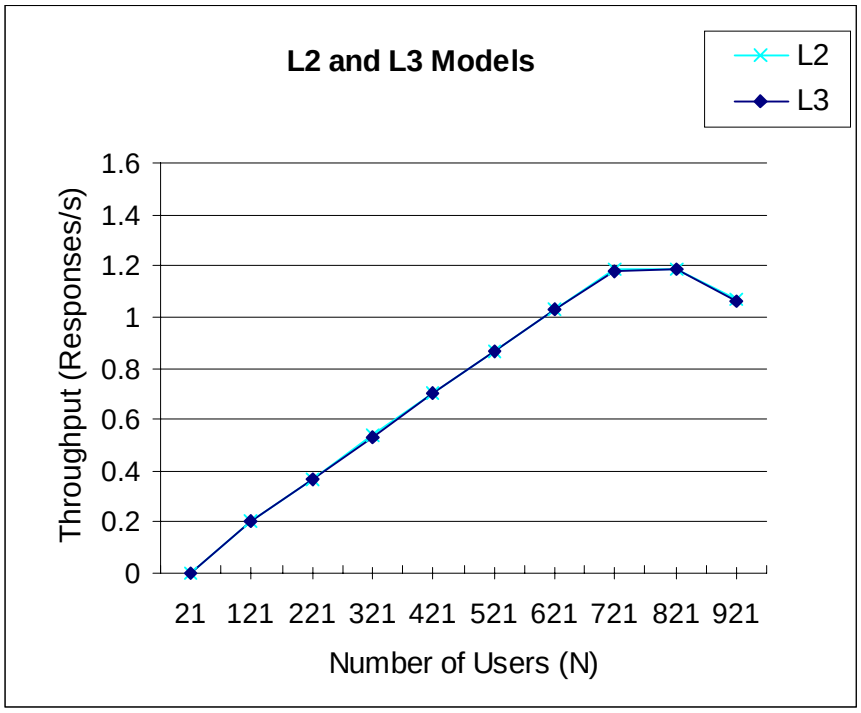

Figure 5.13 Comparison of Throughput for L2 and L3 Models

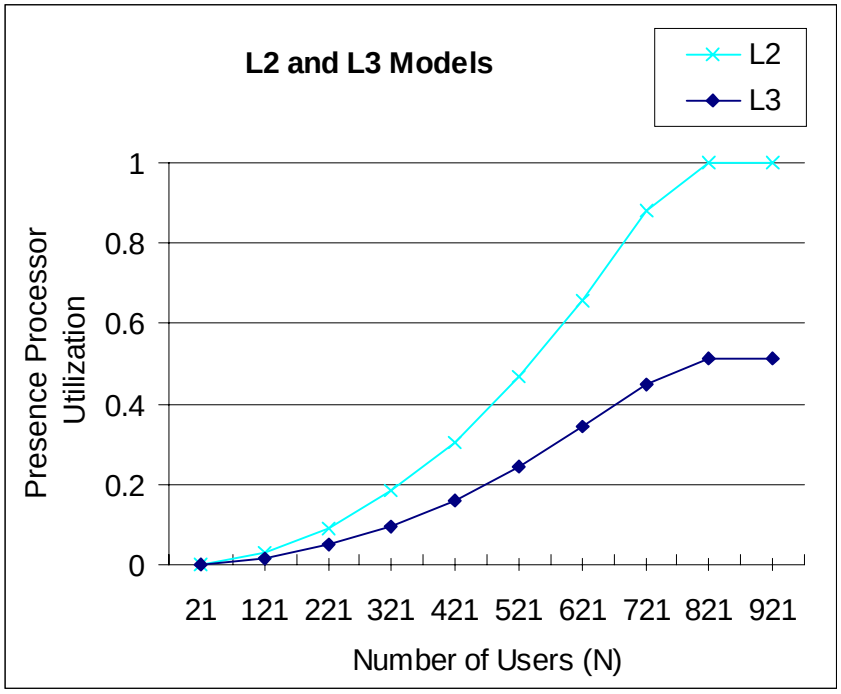

Figure 5.14 Comparison of Presence Processor Utilization for L2 and L3

Chapter 6 Apply MSPA to SIP Implementation Model

In this chapter, a SIP notification system is added to the level 3 abstract model of the presence system. The SIP notification system's architecture and the components' functions are described in the previous section 3.2. The specification UCM model M4-SIP is described in section 6.1. In section 6.2, the performance annotated UCM model P4-SIP is created. In section 6.3, some experiments were performed and the results are presented. Finally, the discussion of the SIP implementation model is presented.

6.1 SIP Implementation Model (M4-SIP)

In this section, the SIP implementation of the presence system model M4-SIP is described. The model adds two new components to the level 3 abstract model L3, one is a Proxy and another is Redirect/Registrar. The Proxy is used to forward notification message to the target subscriber through the Redirect server. The Redirect is used for searching the current subscriber's address. The Redirect server receives SIP request from the Proxy and then accesses a database or a location server, which may be a Registrar, and tells the Proxy the address to contact. In this model, we assume the Registrar server is collocated with the Redirect server.

Figure 6.1 shows the scenario UpdatePI for the M4-SIP model. The user starts UpdatePI scenario by entering the current presence information to the Client. The Client sends the updated PI to the PUA through Communication component. The PUA processes the PI, and makes a synchronous call to the PA. The PA receives the call and gets the subscribers' list according to the stored subscription information and access policy. Then the PA notifies all the subscribers through the SIP notification system. For each

subscriber, the PA makes a synchronous call to the Proxy and the Proxy makes a synchronous call to the Redirect server. The Redirect finds the current contact address of the subscriber and sends the address to the Proxy. After that, the Proxy makes a synchronous call to the Watcher using its current address. The Watcher replies to the Proxy and the Proxy sends the reply message to the PA. The Loop has a counter number to record how many subscribers the PA wants to notify.

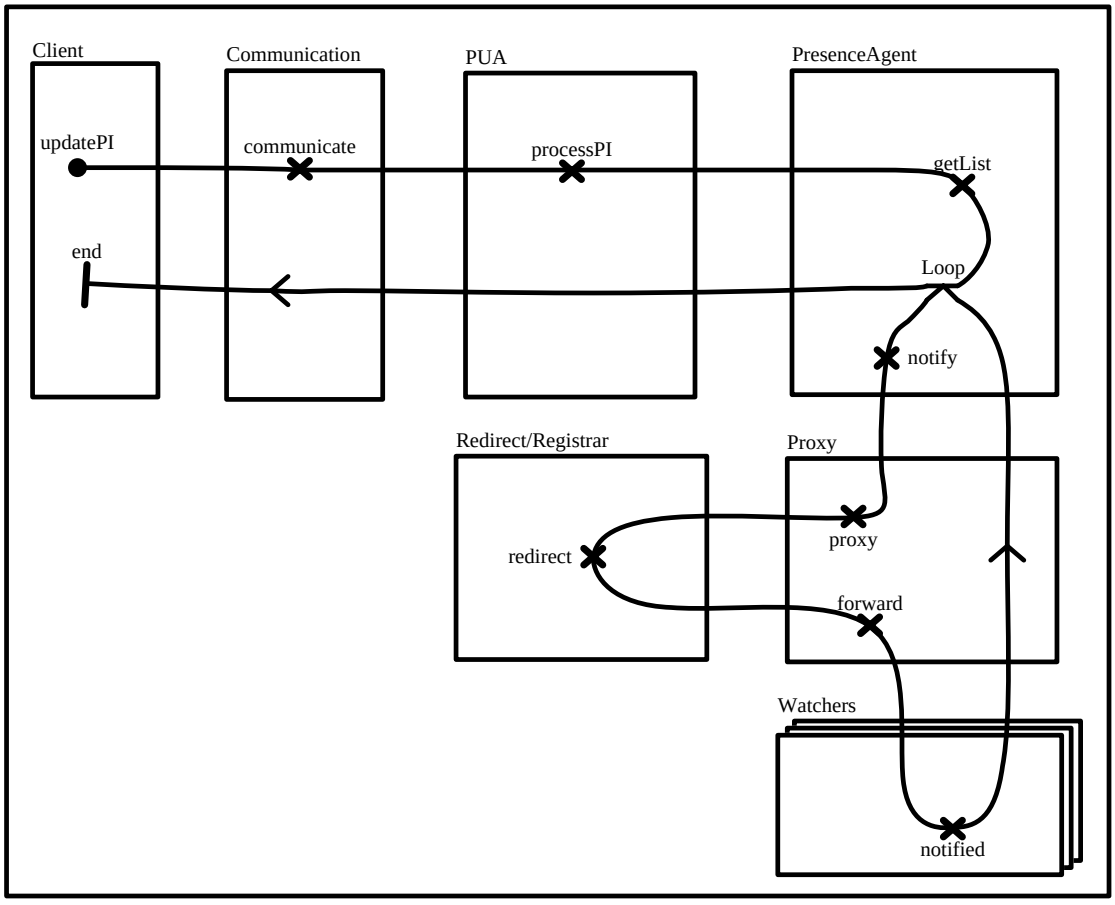

Figure 6.1 Presence System UCM Model M4-SIP

6.2 Create Performance Annotated UCM Model P4-SIP

The performance annotated UCM model P4-SIP can be created with the performance parameters and system deployment choices.

New assumptions are added as follows.

- The responsibilities' CPU demands of the Proxy and the Redirect components do not change when the subscribers' number Nsub and users' number N change. Therefore, the budgeted CPU demand of responsibilities **proxy, redirect,** and **forward** are constant for all N and Nsub.
- The initial budgeted demands are 2ms for **proxy, redirect,** and **forward** and 10ms for the other responsibilities in the model.
- The details of how to get the list of all subscribers are not specified in detail in this level. We assume the subscription information is kept in PA component and no disk or database accesses are needed.

There are two deployment choices. One choice is allocating the Proxy, Redirect/Registrar on a Proxy Processor. Another is allocating the Proxy on Proxy processor and Redirect/Registrar on Redirect processor. The remaining components are allocated on the processors as level 3 model P3.

The LQN model L4-SIP can be created using UCM2LQN from P4-SIP. (please refer Appendix D: L4-SIP.xlqn).

Following figure is the LQN model L4-SIP.

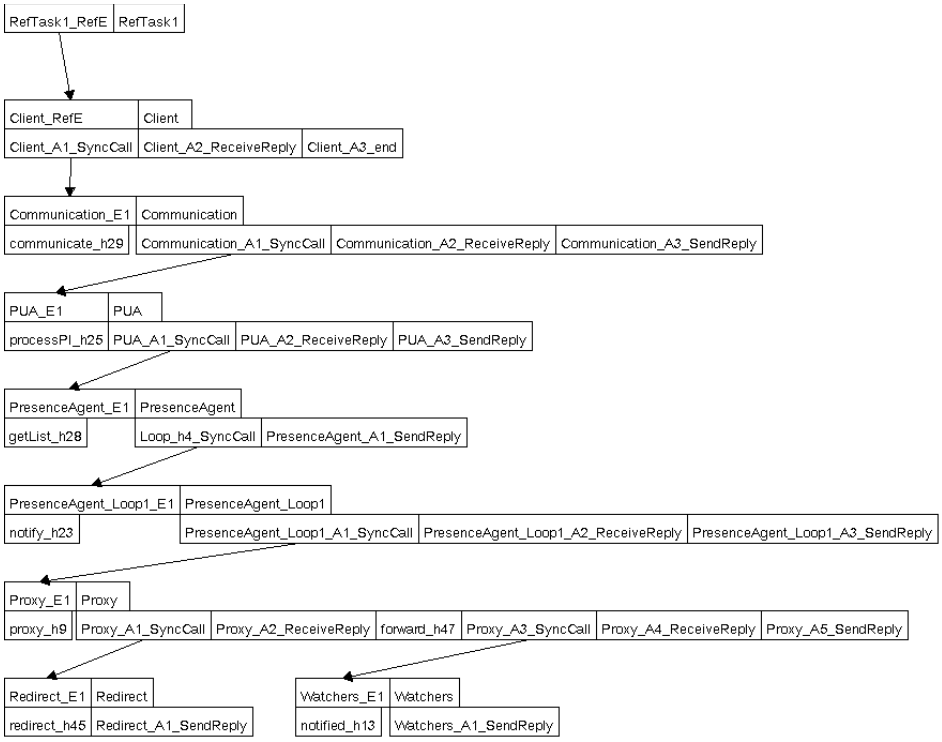

Figure 6.2 Presence System LQN Model L4-SIP

6.3 Performance Analysis Results for LQN Model L4-SIP

Several tests were performed to get the performance analysis results. All tests set the subscribers' number per user Nsub as 0.1*N. The total users number N ranges from 21 to 921 with increment 100.

6.3.1 Performance Analysis for 2P Cases

The following test cases allocate Proxy and Redirect on the same Proxy processor and the PA on the Presence processor. We call the test as 2P cases.

- Test L4sip-1: This is 2P base case. All budgeted demands are 10ms except the following responsibilities: **notify, redirect, proxy, forward a**nd **notified**. The five responsibilities' CPU demands are budgeted as 2ms.
- Test L4sip-2: This case is like the base case except the **proxy** demand is 100ms.
- Test L4sip-3: This case is like the base case except the **notify** demand is 100ms.

Figure 6.3 to Figure 6.6 shows the comparison of tests' results for the 2P cases.

Figure 6.3 shows the three throughputs versus the number of users N. For base case, when N increases till about 800, the throughputs increase linearly. When N is greater than 800, the throughput decreases. This is because the notification operation needs more time when the subscribers' number Nsub increases. For case L4sip-2 and L4sip-3, the **proxy and notify** operations CPU demands increase from 2ms to 100ms, the two throughput curves change with the number of users and the two curves are almost same. For these two cases, when the users' number N increases to about 200, the throughputs reach their highest value and then decrease when more users and more subscribers are using the presence system at the same time.

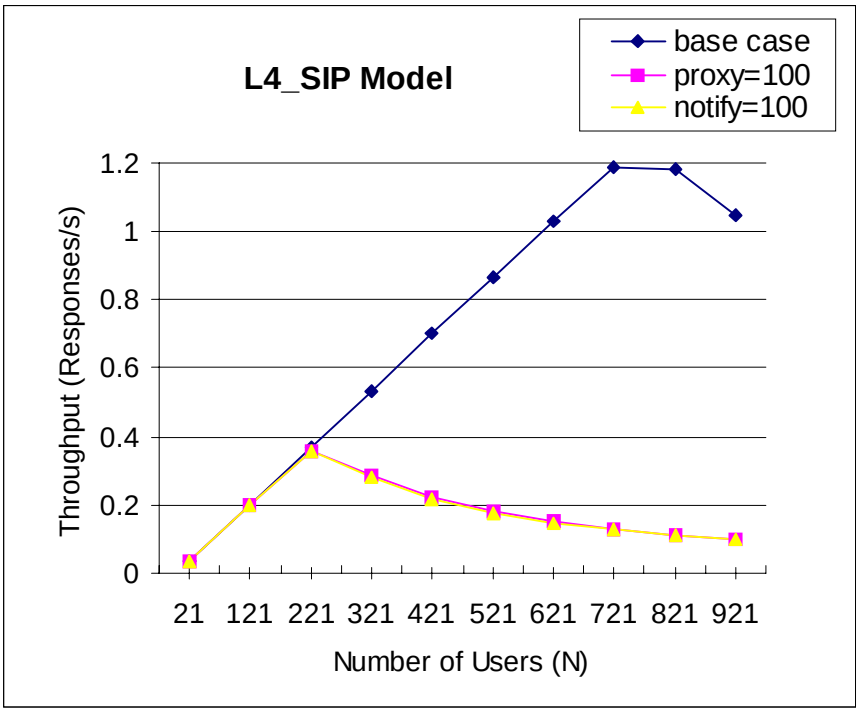

Figure 6.3 Comparison of Throughput for Model L4-SIP 2P Cases

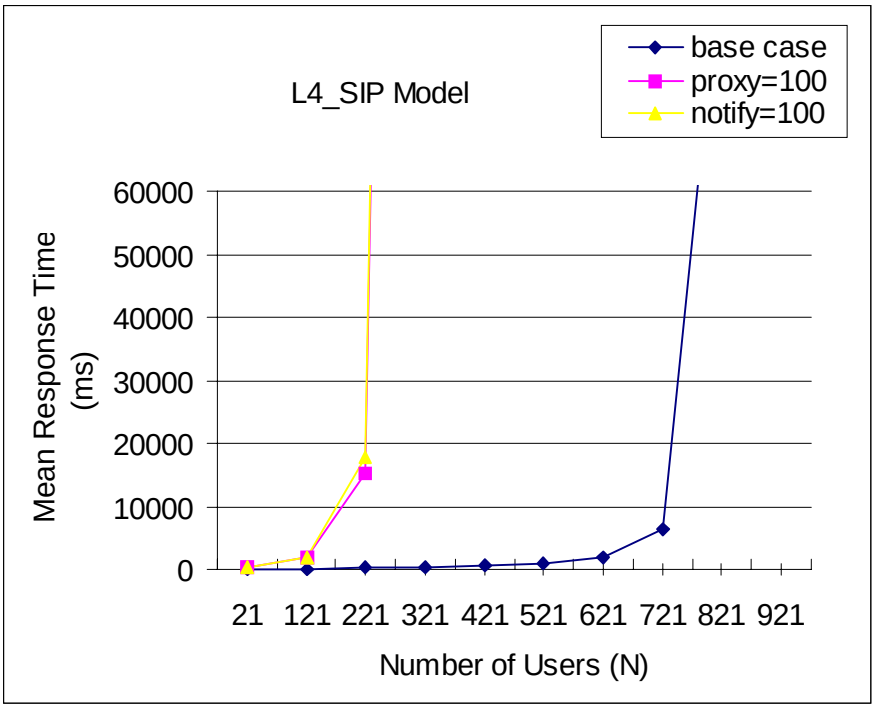

Figure 6.4 Comparison of Mean Response Time for Model L4-SIP 2P Cases

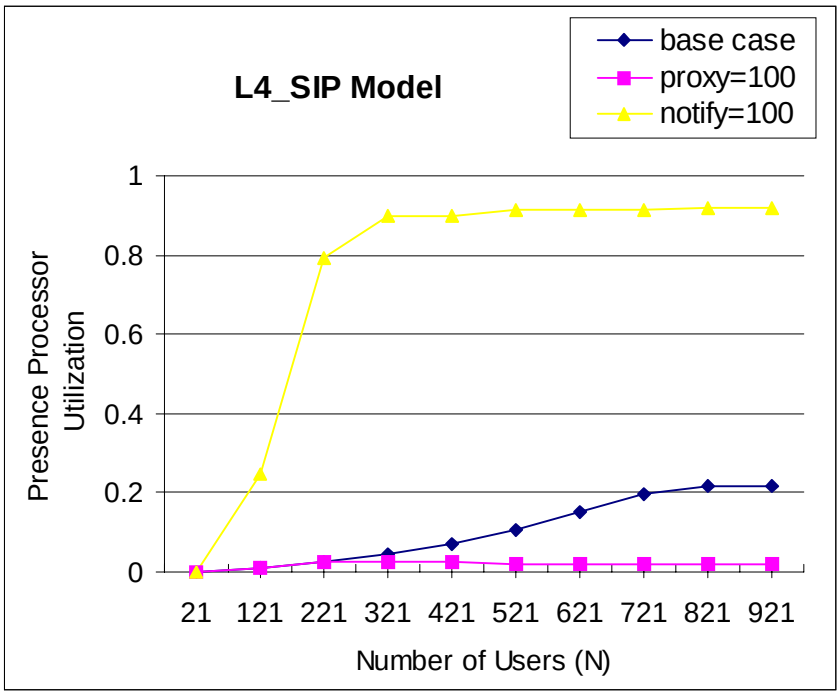

Figure 6.5 Comparison of Presence Processor Utilization for Model L4-SIP 2P Cases

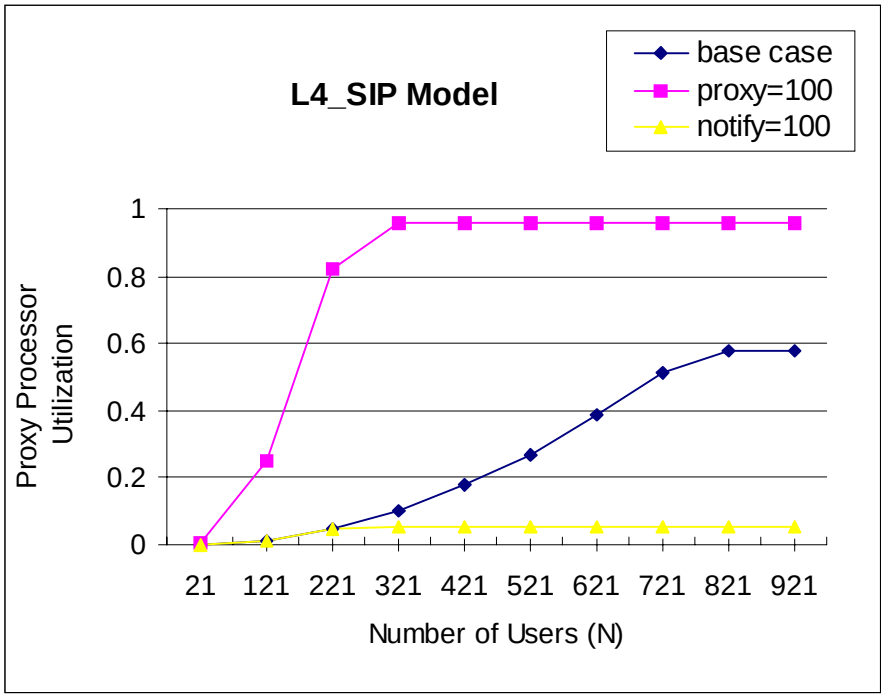

Figure 6.6 Comparison of Proxy Processor Utilization for Model L4-SIP 2P Cases

Figure 6.4 shows the Mean Response Time results for the 2P cases. For 2P base case, the mean response time increases sharply when the users' number N is greater than 800. For other two cases when the demands of **notify** and **proxy** is 100ms, the maximal users' number N is 200. The mean response time becomes unacceptable when the users' number N is greater than 200. The mean response time curves of the two cases are almost same. This shows that whether increase the notify demand or the proxy demand, the presence system's performance worsen.

Figure 6.5 shows the Presence Processor Utilization results for the 2P cases. The test results show that the presence processor reaches 20% when the users' number N is greater than 800 for 2P base case and is very low when the proxy demand is 100ms. For case L4sip-3, where notify is 100ms, when the users' number N is above 200, the processor utilization reaches highest point 90%. At the same time, from the test results, we can find that the PA task utilization is 100% when the users' number N is greater than 200. In this case, the software bottleneck happens at PA task.

In Figure 6.6, the test results show that the proxy processor utilization reaches 60% when the users' number N is greater than 800 for 2P base case and above 95% when users' number N is greater than 200 for case L4sip-2. For case L4sip-3, the proxy processor utilization is very low.

The above figures show that the software bottleneck is PA task and its utilization is 100% even if the processor' utilization does not reaches 100%.

6.3.2 Performance Analysis for 3P Cases

In the following tests, the Proxy is allocated to the Proxy Processor, the Redirect/Registrar is allocated to the Redirect Processor, and the PA is allocated to the Presence Processor. We call these tests as 3P cases.

- Test L4sip-4: this is 3P base case. The CPU demand values are like the 2P base case.
- Test L4sip-5: This case increases the **proxy** demand is to 100ms.
- Test L4sip-6: This case increases the **notify** demand is to 100ms.

Figure 6.7 to Figure 6.8 show the test results of test L4sip-4, L4sip-5 and L4sip-6.

Figure 6.7 shows the throughput of the three cases. For base case L4sip-4, the throughput increases until the users' number N is above 800. The highest throughput is 1.2 responses/second. For case L4sip-6, when the users' number N is above 200, the system saturates and the throughput reaches the maximal value 0.4. The same result for case L4sip-6.

Figure 6.8 shows the mean response time versus the change number of users N. For base case L4sip-4, the mean response time is low when the users' number N is below 800. For other two cases, the mean response time is low when the users' number N is below 200. When the users' number N is greater than the saturate point N=200, the mean response time will increase dramatically and unacceptable for the end users.

Figure 6.9 shows the Presence Processor Utilization results for the 3P cases. The test results show that the presence processor reaches 90% when the users' number N is greater than 200 for notify demand is 100ms and very low for other two test cases. The system's bottleneck is task PA, which reaches the 100% utilization.

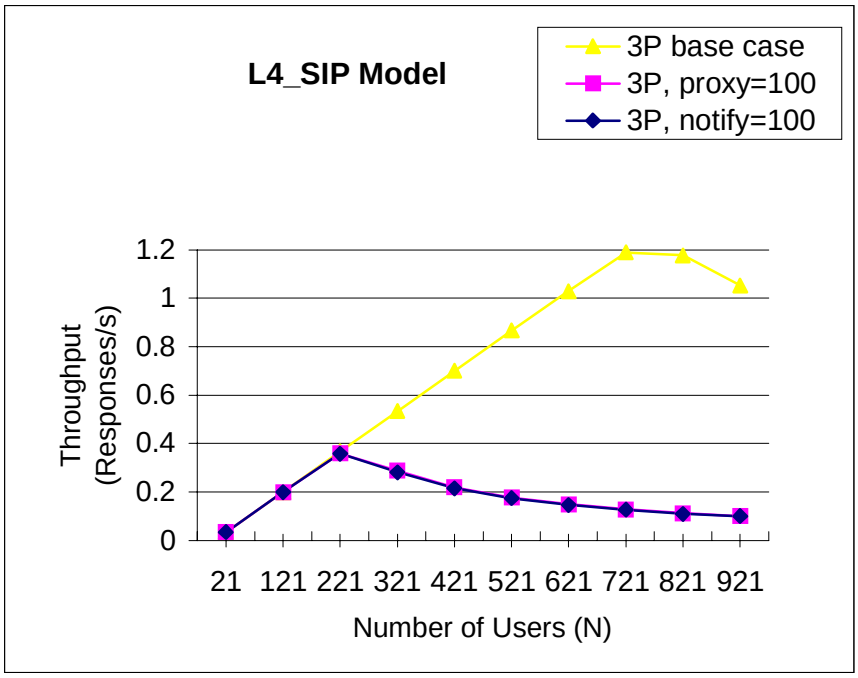

Figure 6.7 Comparison of Throughput for Model L4-SIP 3P Cases

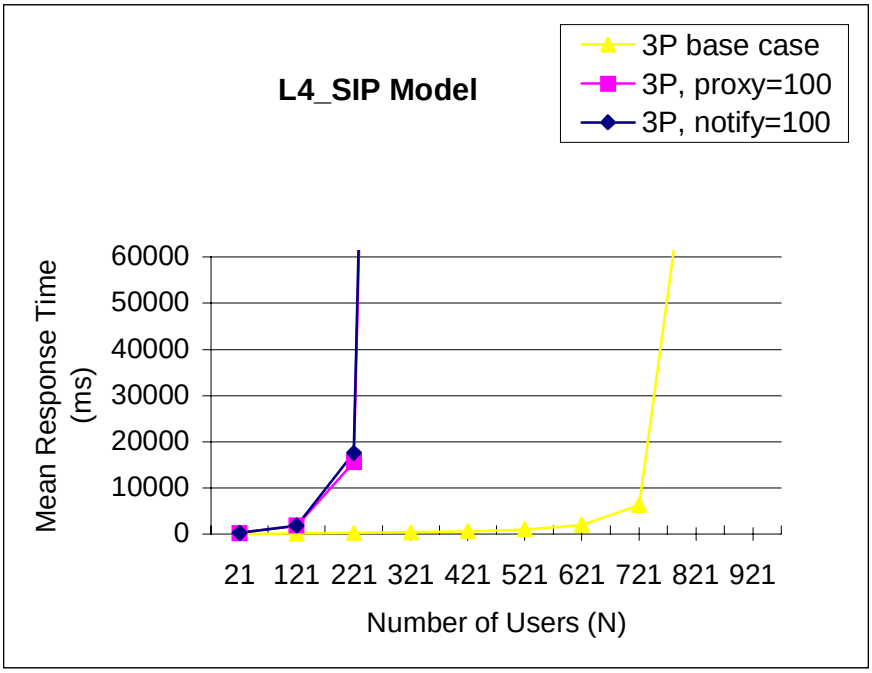

Figure 6.8 Comparison of Mean Response Time for Model L4-SIP 3P Cases

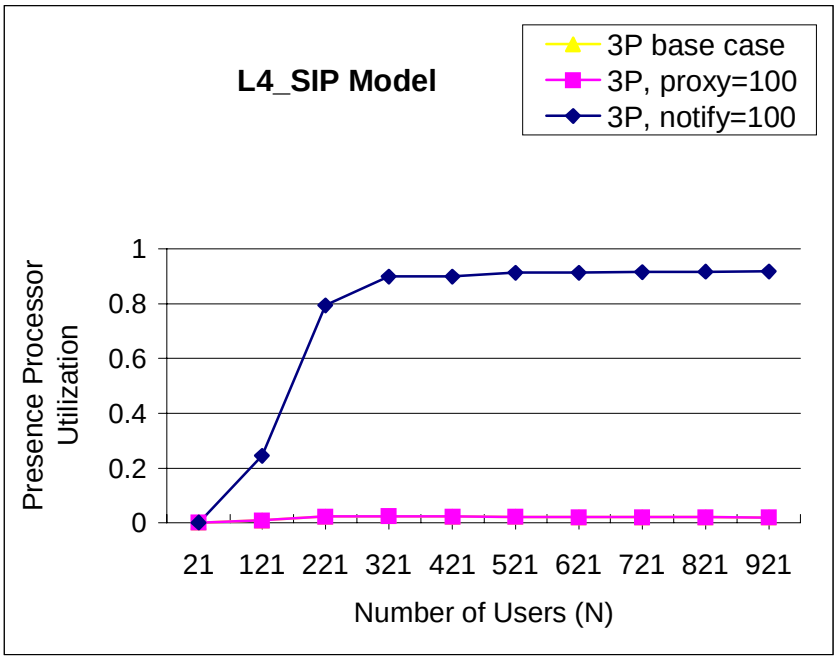

Figure 6.9 Comparison of Presence Processor Utilization for Model L4-SIP 3P Cases

6.3.3 Performance Analysis for 3P2T Cases

To break down the bottleneck of the system, we assume there are two PA tasks in the system, and each PA task manages half of the total users and half of the subscribers. When PUA sends a request to PA, it sends one request to each PA. Then the PA sends request to the Proxy like the test cases in the previous section.

The following tests set two threads of PA and allocate the two PA threads to the Presence Processor. Other tasks are allocated to the same processors like cases in section 6.3.2. We call these tests as 3P2T cases.

- Test L4sip-7: This case sets PA 2 threads on the Presence Processor, and the **notify** demand is 100ms.

Following figures shows the test results compared with test case L4sip-6 where notify

demand is 100ms and PA task has 1 thread on Presence Processor.

In the following Figures 6.10 to 6.12, we call test L4sip-6 as 3P1T case and test case L4sip-7 as 3P2T case.

From Figure 6.10 to Figure 6.12, we can see that two threads of PA improve the performance of the system. For example, when N is about 200, the mean response time decreases by 50% (Figure 6.11). The highest utilization is about 90% for 3P1T case and is about 100% for 3P2T case.

In Figure 6.12, the PA task's utilization is also shown for 3P2T test case. The system's bottleneck is still PA task when the system saturates. The PA task's utilization reaches 200% when N is greater than 200 because PA has 2 threads working on the Presence Processor.

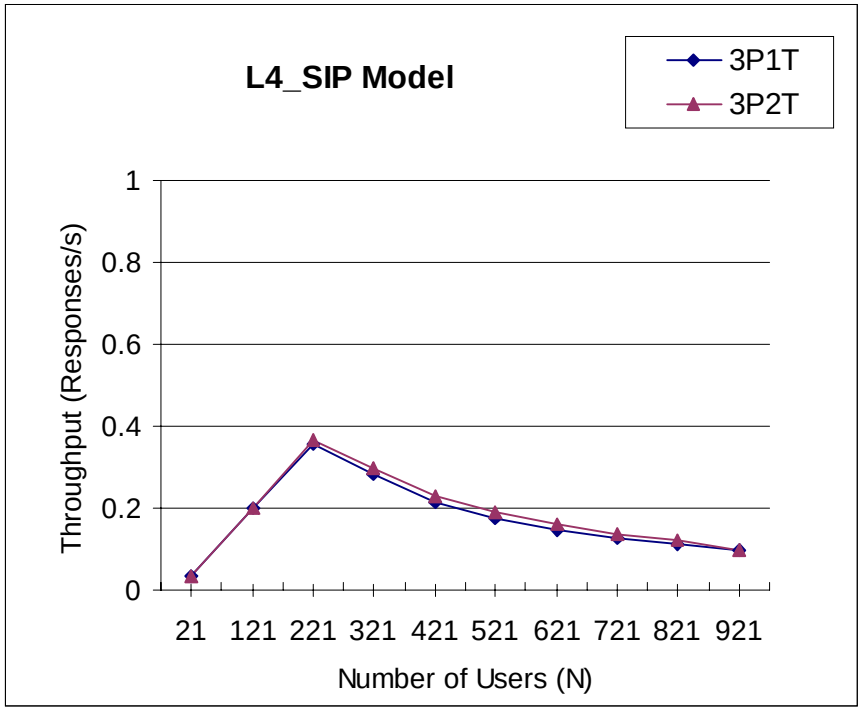

Figure 6.10 Comparison of Throughput for Model L4-SIP 3P2T Case

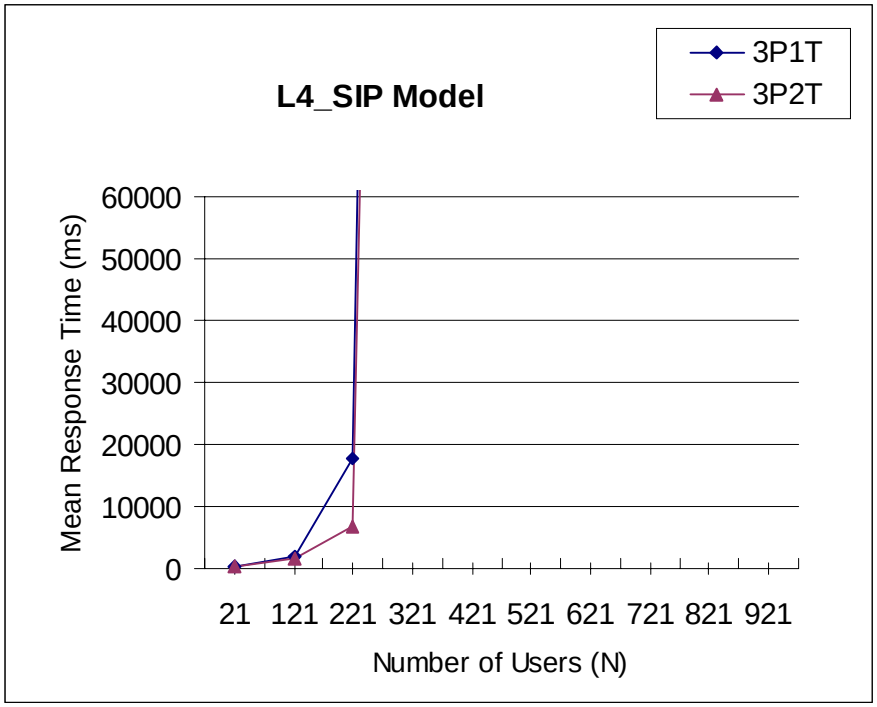

Figure 6.11 Comparison of Mean Response Time for Model L4-SIP 3P2T Case

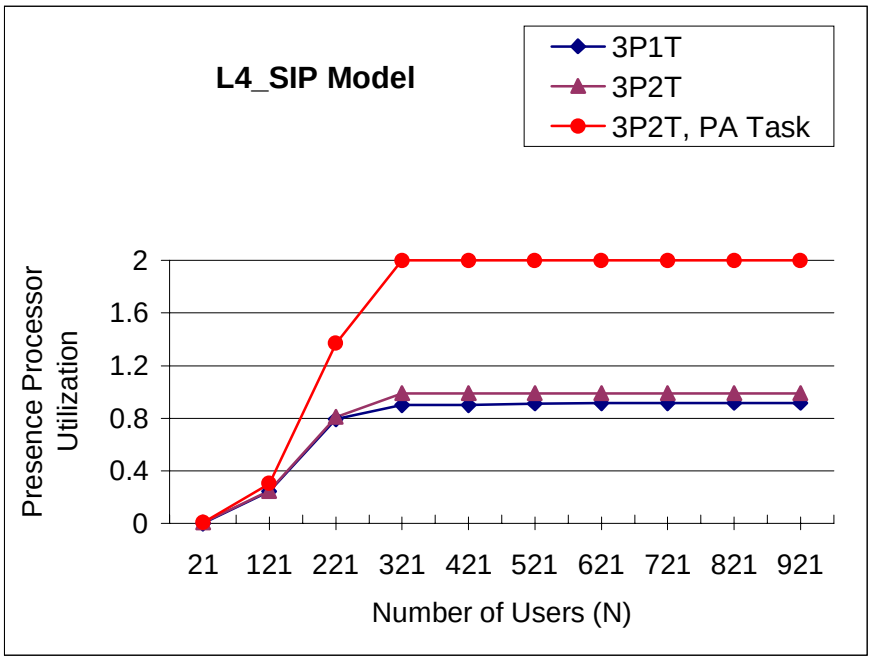

Figure 6.12 Comparison of Processor Utilization for Model L4-SIP 3P2T Case

6.4 Discussion of Model L4-SIP

From above tests, we get the following results.

The listed issues should be considered when designing the presence system with SIP implementation.

- Using separate processor for proxy and Redirect/Registrar will increase the throughput and performance.
- The system is very sensitive to the CPU demand of **notify** operation. When it increases from 10ms to 100ms, the effect is very large. The throughput reduces and the mean response time increases greatly.
- The bottleneck happens at the PA task. The PA task's utilization reaches 100% for 1T case and reaches 200% for 2T case.
- When the system saturates, 2 threads PA in the system reduce the loop number to 50%,

and reduce the mean response time. The throughput increases and presence processor utilization increases also.

Figure 6.13 and Figure 6.14 compare the L4-SIP model and L3 model. In the figures, “L4-SIP 2P” case represents the L4-SIP 2P base case model, and the “L3 block” represents the L3 model block case.

Figure 6.13 and Figure 6.15 illustrate the throughput, mean response time and presence processor utilization of the two different level models L3 and L4-SIP. The curves in the figures show that the throughputs and mean response time are the same for both L3 and L4-SIP models. The processor utilization curves show that the system saturates when N is above 800. The difference is that the L3 model shows that the presence processor’s utilization reaches 50%, while the L4-SIP model’s maximal utilization is about 20%. For both models, the bottleneck is Presence Agent task. The tests show the performance results are consistent. The performance results of the L3block model are still valid for the more complex and more detailed SIP implementation model L4-SIP.

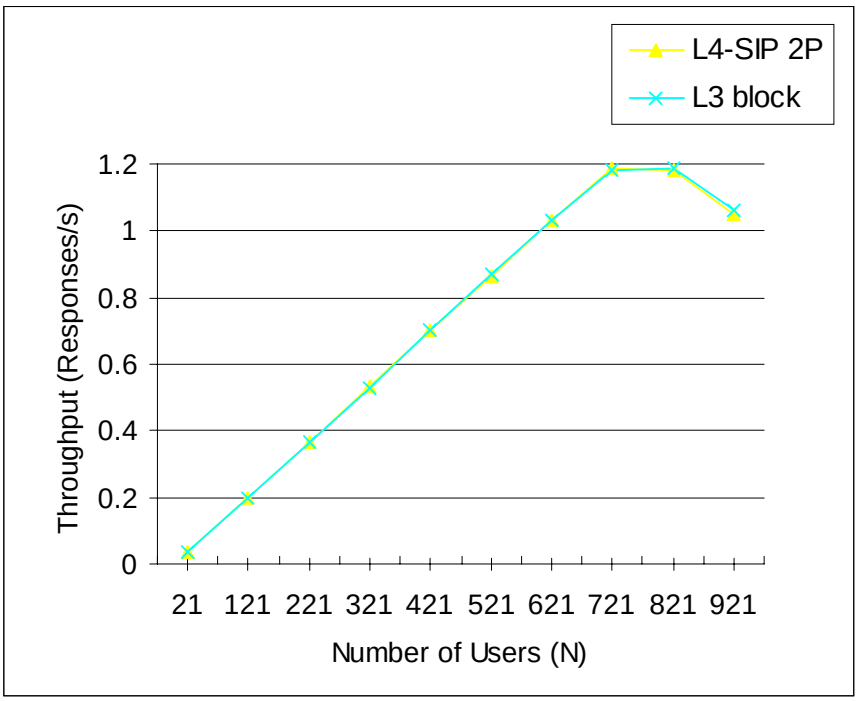

Figure 6.13 Comparison of Throughput for L4-SIP and L3 Models

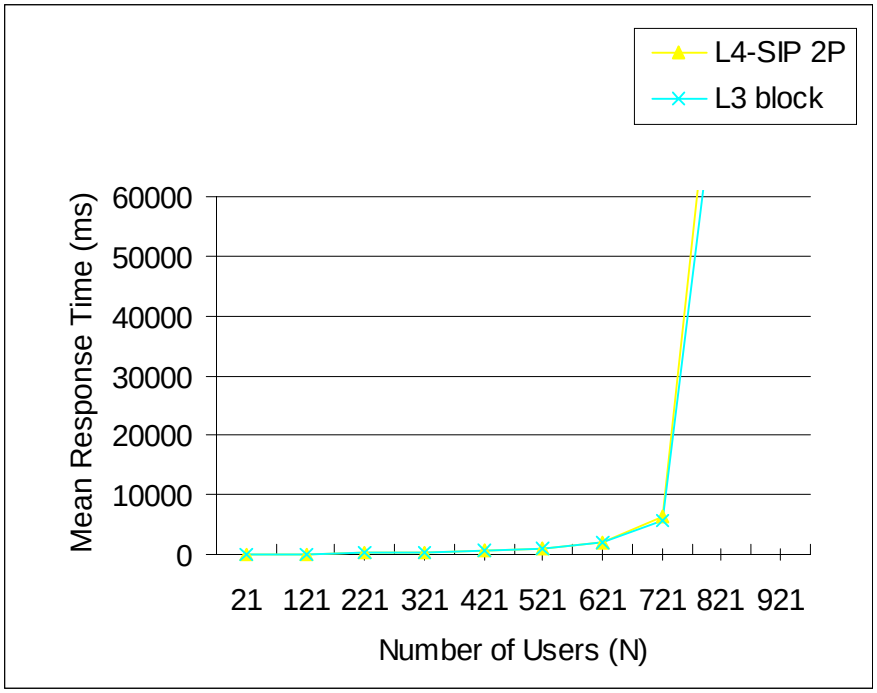

Figure 6.14 Comparison of Mean Response Time for L4-SIP and L3 Models

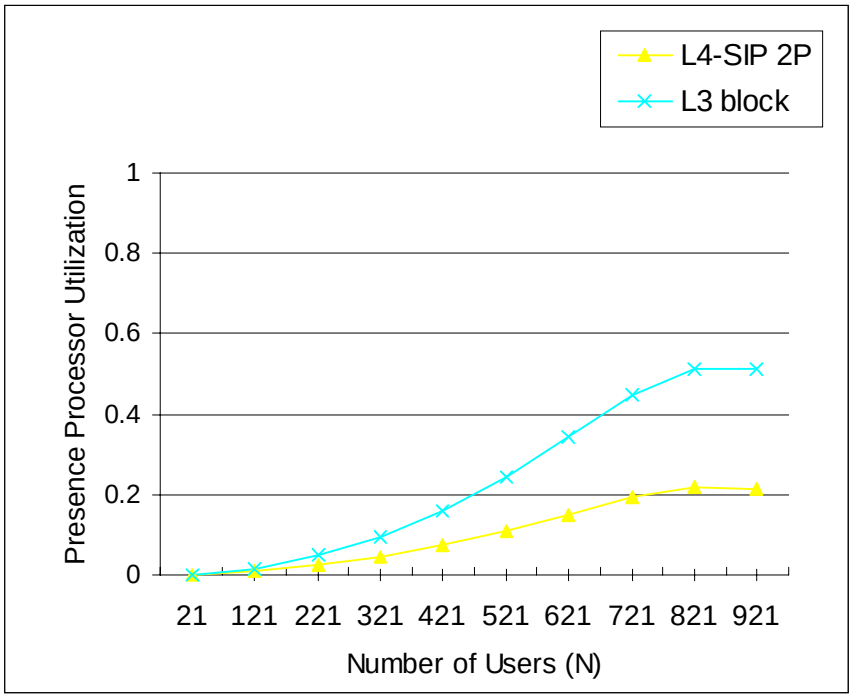

Figure 6.15 Comparison of Processor Utilization for L4-SIP and L3 Models

Chapter 7 Apply MSPA to Tuplespace Implementation Model

In this chapter, the IBM T Spaces middleware system is used to facilitate the communication between the PA and Watcher components. The T Spaces event register and event notification mechanism is described in the previous Section 2.5.1. Because the T Spaces middleware is a COTS system, the T Spaces can be viewed as a black box and the detailed decomposition inside the T Spaces is not further described in this level of abstract analysis.

After the specification model M4-TS is defined in section 7.1, the MSPA methodology is applied to analyze the performance of the model. Section 7.2 describes the performance annotated UCM model P4-TS. In section 7.3, some experiments were performed and the results are presented. Finally, discussion is presented in section 7.4.

7.1 Tuplespace Implementation Model (M4-TS)

Figure 7.1 illustrates the UCM model M4-TS. Three new components are added to the system. They are the Tuplespace, Event Manager (EManager) and Event Listener (EListener). The Tuplespace component works as a Tuplespace server and manages the tuples and calls EListener component whenever the registered tuple is written to the Tuplespace. The component EManager works as an EventNotifier, which is described in section 2.5.1, and writes the event tuple into the Tuplespace. EListener works as a CallbackApp component and implements callback interface. It registers with Tuplespace for certain events and waits for the notification from the Tuplespace.

The PA has the information about all the subscribers. PA notifies all the subscribers by a synchronous call to EManager. For each notification, EManager construct an event

tuple, for example, an UpdatePI event to the specific subscriber, to the Tuplespace. EListener of the subscriber will be notified by Tuplespace because EListener has already register the target event with the Tuplespace before the writing operation. The EListener then notifies the Watcher about the updated PI information.

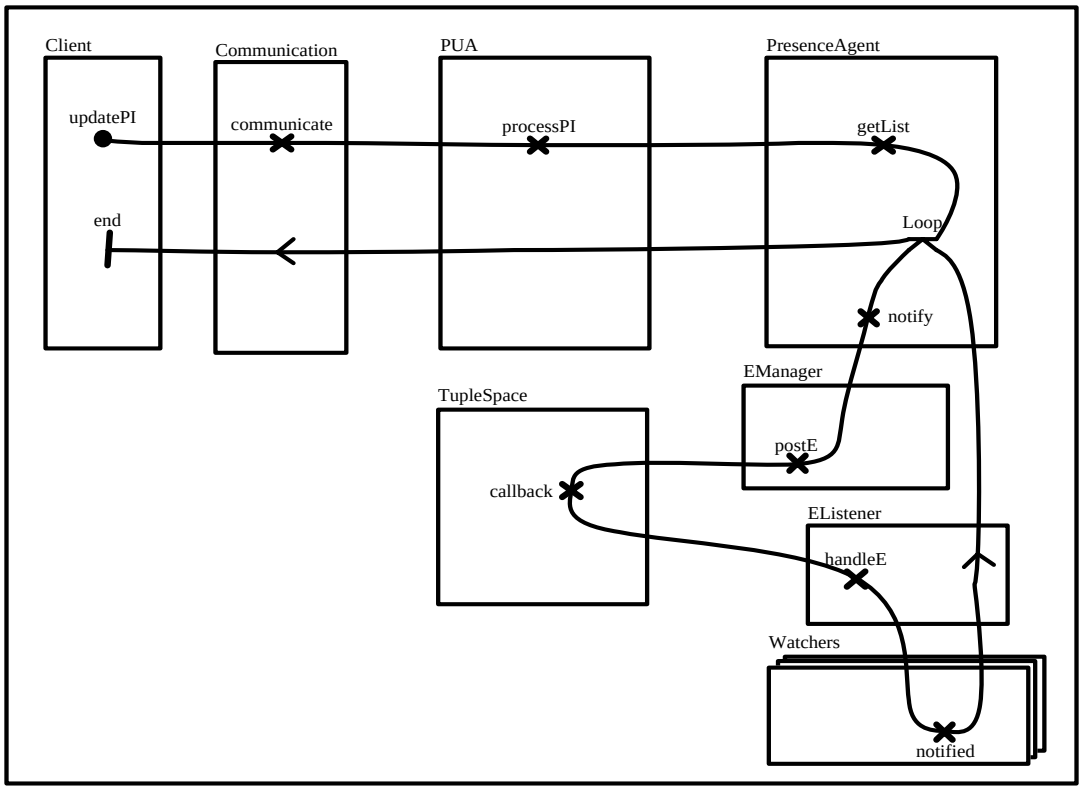

Figure 7.1 Presence System UCM Model M4-TS

In Figure 7.1, the scenario UpdatePI is started by the end user entering the new presence information to the Client. The Client sends the updated PI to the PUA through the Communication component. The PUA processes the PI and makes a synchronous call to the PA. The PA receives the call and gets the subscribers' list according to the stored subscription information and access policy. Then the PA notifies all the subscribers through the Tuplespace middleware system. The PA calls EManager and EManager writes an event tuple to the Tuplespace. The Tuplespace calls back the registered EListener about the event. This is represented as a forward call in the UCM model. The EListener handles

the notification and makes a synchronous call to the Watcher. As in level 3 abstract model, a Loop is used to represent the counter of notification to all watchers. The Watchers receive the notification and send reply message to the EListener.

With the help of the Tuplespace event notification mechanism, the PA does not keep all the current addresses of Watchers and does not need to know the addresses to send the notification. The current address of each Watcher can be kept in Tuplespace or its EListener and the Watcher can update its current address very easily whenever the Watcher moves to another place. This adds to the flexibility to the presence system.

In model M4-TS, we let EListener return the reply to the PA to record the notification time from the start of the first notification till the last subscriber getting the notification message.

7.2 Create Performance Annotated UCM Model P4-TS

Following MSPA methodology, the performance annotated UCM model P4 -TS can be created by entering performance parameters and system deployment data into the UCM model M4-TS.

When budgeting the CPU demands for all the responsibilities, there are some new assumptions that are considered besides the assumptions described in section 4.3.1.

The callback CPU demand of Tuplespace is assumed not changing with the users' number N and subscribers' number per user Nsub. Actually the more users or subscribers, the more information is stored in the Tuplespace server and the processing time may change with different N and Nsub. And the processing time may not increase linearly with N and Nsub because the users' and subscribers' information may be cached in the server's memory or be stored in a database. We assume the budgeted CPU demand of callback is a constant for all N and Nsub.

The details of how to get the list of all subscribers are not considered in this level of abstraction. We assume the subscription information is kept in PA.

There are two deployment choices. One choice is allocating the Tuplespace, EManager and EListener components to the Presence Processor (called 1P case). Another is allocating the Tuplespace on a separate Tuplespace processor (TuplespaceP) and allocating EManager and EListener to PresenceSP (called 2P case). Other components are allocated to the processors as in the level 3 model P3.

LQN Model L4-TS can be created by UCM2LQN from P4-TS. Some execution controlling values are added to get series of test results by using SPEX controller (See Appendix E: L4-TS.xlqn).

Following figure is the LQN model L4-TS.

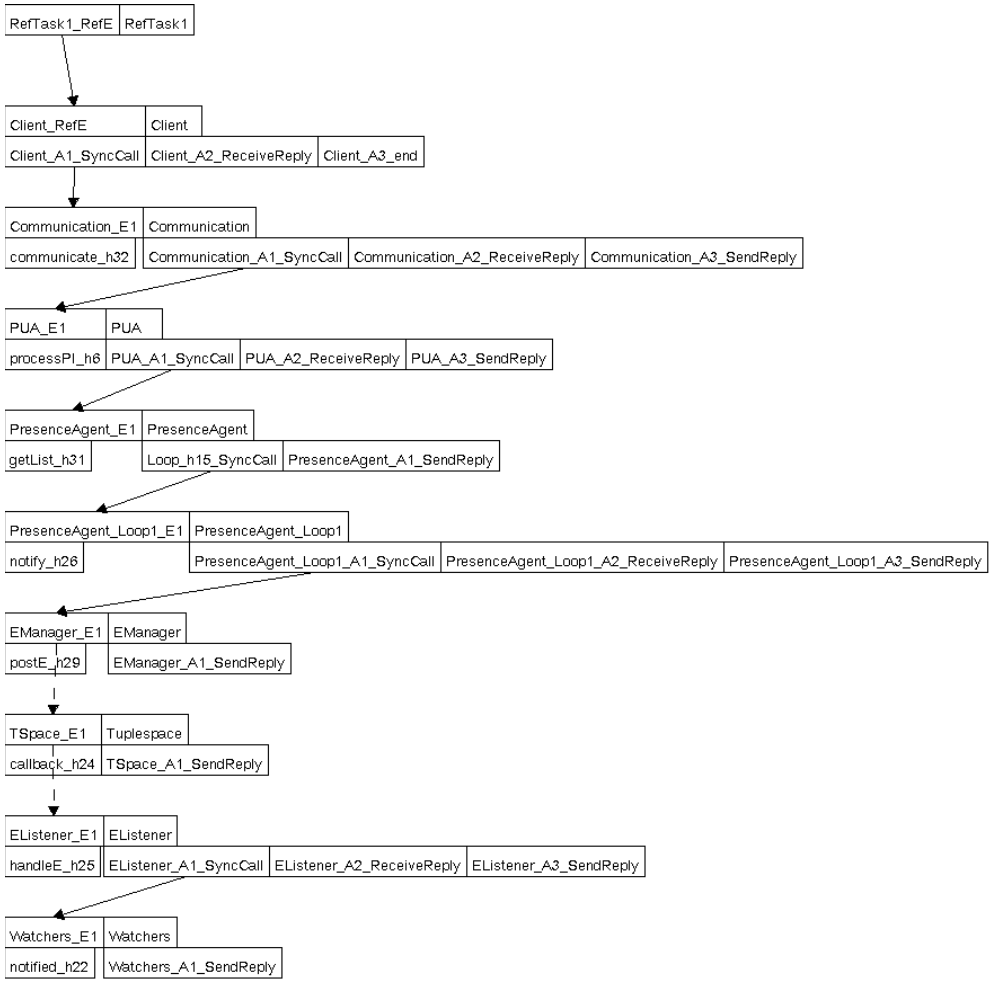

Figure 7.2 Presence System LQN Model L4-TS.

7.3 Performance Analysis Results for LQN Model L4-TS

Several tests were performed to get the performance analysis results. All tests set the subscribers' number Nsub be 0.1*N. The total users number N ranges from 21 to 921 with increment 100.

7.3.1 Performance Analysis for 1P Cases

The following test cases allocate Tuplespace on Presence processor. We call the tests as 1P cases.

- Test L4ts-1: This is 1P base case. All budgeted demands are 10ms except the **notify,**

postE, callback, handleE and **notified** responsibilities. The five notification responsibilities are budgeted as 2ms.

- Test L4ts-2: This case is the same as the base case except that the **notify** demand is increased to 100ms.
- Test L4ts-3: This case is the same as the base case except that the **callback** demand is increased to 100ms.

Figure 7.3 to Figure 7.5 show the outputs of three cases comparison.

Figure 7.3 shows the three throughputs against the number of users N. For the base case, when the users number N increase till about 600, the throughputs increase linearly. When the users' number N is greater than 600, the throughput decreases because the notification processing needs more time when the subscribers' number Nsub increases. For case L4ts-2 and L4ts-3, where the **notify** and **callback** demands increase from 2ms to 100ms, the throughput curves are almost same because the Tuplespace works on the same presence processor. For these two cases, when the users' number N increases to above 200, the throughputs reach their highest value and then decrease when more users and more subscribers using the presence system.

Figure 7.4 shows the Mean Response Time results for above three cases. For base case, the mean response time increases sharply when users' number N is greater than 600. For L4ts-2 and L4ts-3 cases, when the users' number N is greater than 200, the mean response time becomes unacceptable. The two curves are almost same.

Figure 7.5 shows the Presence Processor Utilization results for three cases. The test results show that the presence processor utilization are about 50% when th`e users' number N is greater than 600 for base case and 45% for other two cases when the users'

number N is greater than 200. This shows that the saturation happens somewhere else in the system. From the test results, we find that the PA task's utilization is 100% when the users' number N is greater than 200. This means that the PA task is the bottleneck of the system. The reason is that the PA makes a synchronous call to the EManager and waits the reply come back from the EManager task. The EManager task forwards the call to the Tuplespace server and both callback and notify operations execute inside synchronous call, so no matter which demand is increased, the performance effects are the same.

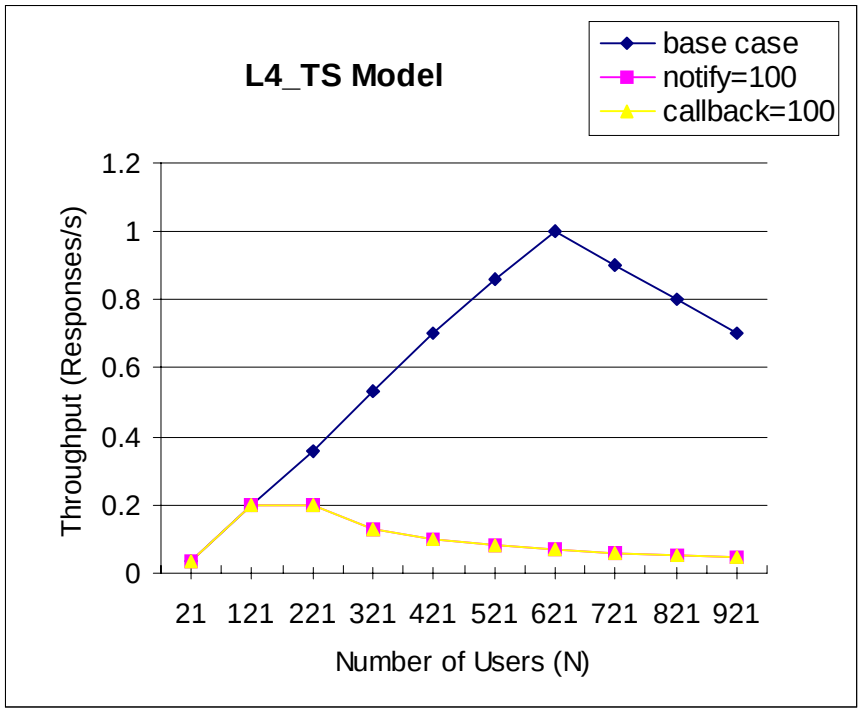

Figure 7.3 Comparison of Throughput for 1P Cases

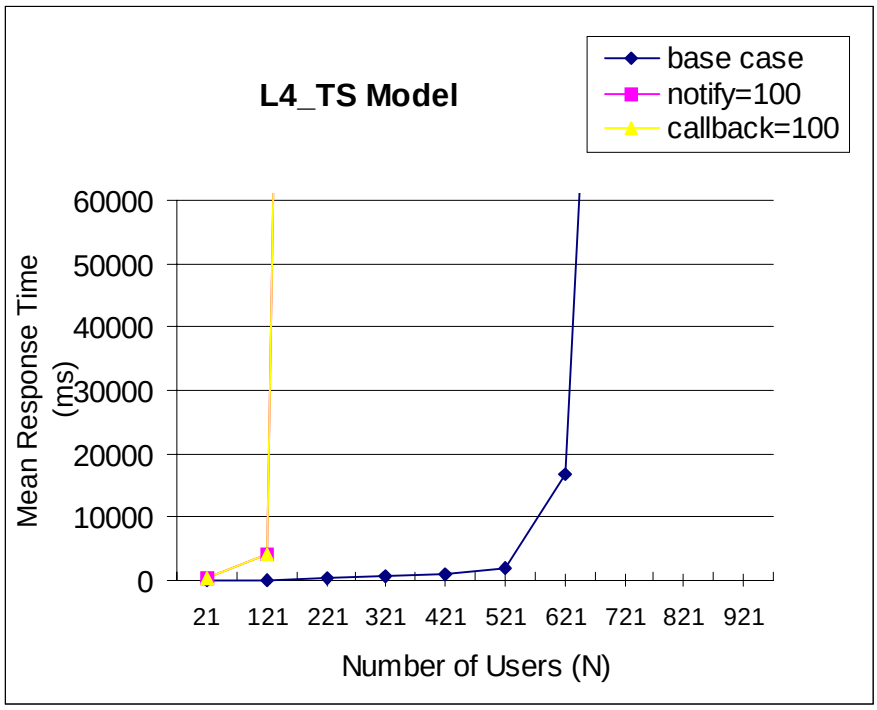

Figure 7.4 Comparison of Mean Response Time for 1P Cases

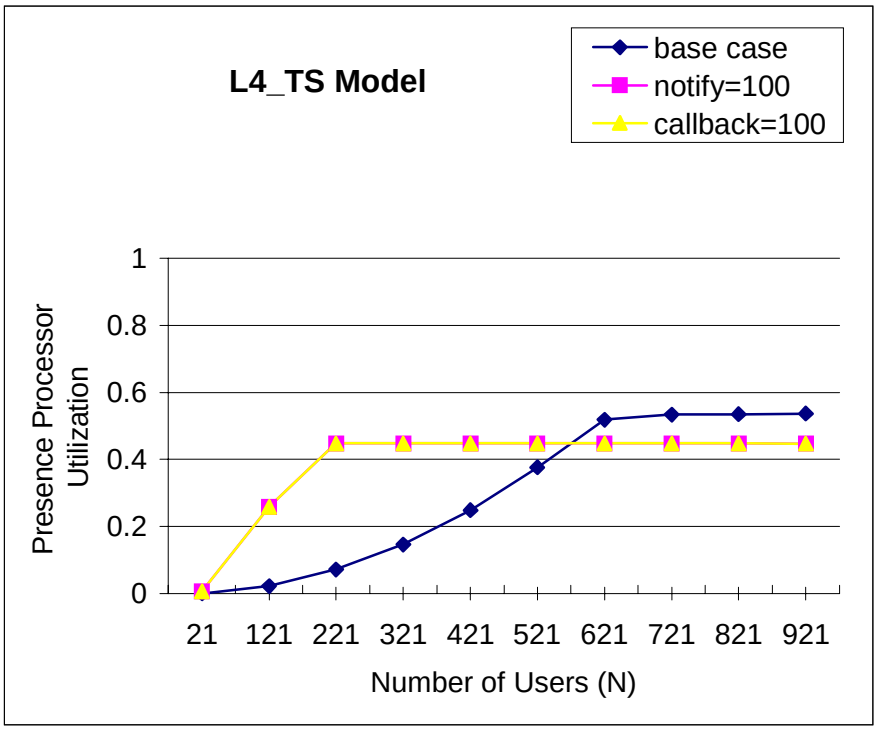

Figure 7.5 Comparison of Presence Processor Utilization for 1P Cases

7.3.2 Performance Analysis for 2P Cases

Following tests allocate the Tuplespace task works on Tuplespace processor. All other tasks' allocations are the same like the previous tests. We call these tests as 2P cases.

- Test L4ts-4: This is 2P base case. All tasks allocation is the same as 1P base case except the Tuplespace task works on TuplespaceP.
- Test L4ts-5: This case is like the base case except that the **callback** demand is increased to 100ms and the Tuplespace works on Tuplespace processor.

From Figure 7.6 to Figure 7.8, we can see that when a Tuplespace processor is introduced for the Tuplespace task, the performance is improved for both 2P cases. For 2P base case, the saturation users' number N is increased from 600 to 700 compared with 1P base case. For L4ts-5 2P case, the saturation users' number N is increased from 200 to 300 compared with 1P L4ts-3 case. The maximal Presence processor utilization is decreased from 50% to 47% for base cases, and from 45% to 6%. The big decrease is because the callback operation executes on the Tuplespace processor for 2P case. The test results show that when a separate processor for Tuplespace task improves the system's performance. From Figure 7.7 and Figure 7.8, we can also know that the higher the callback operation demand is, the more benefits will gain for the system's performance.

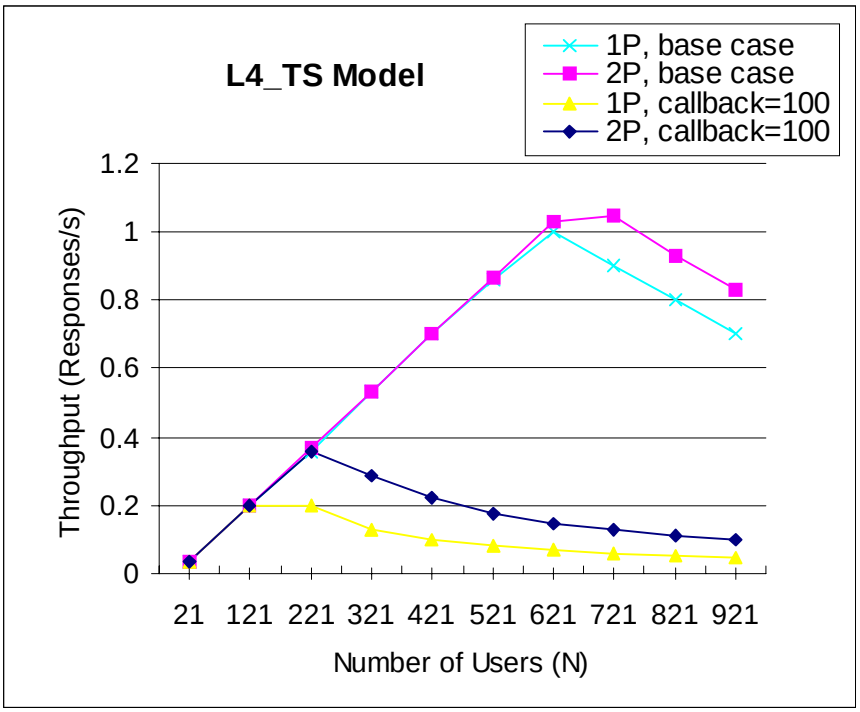

Figure 7.6 Comparison of Throughput for 1P and 2P Cases

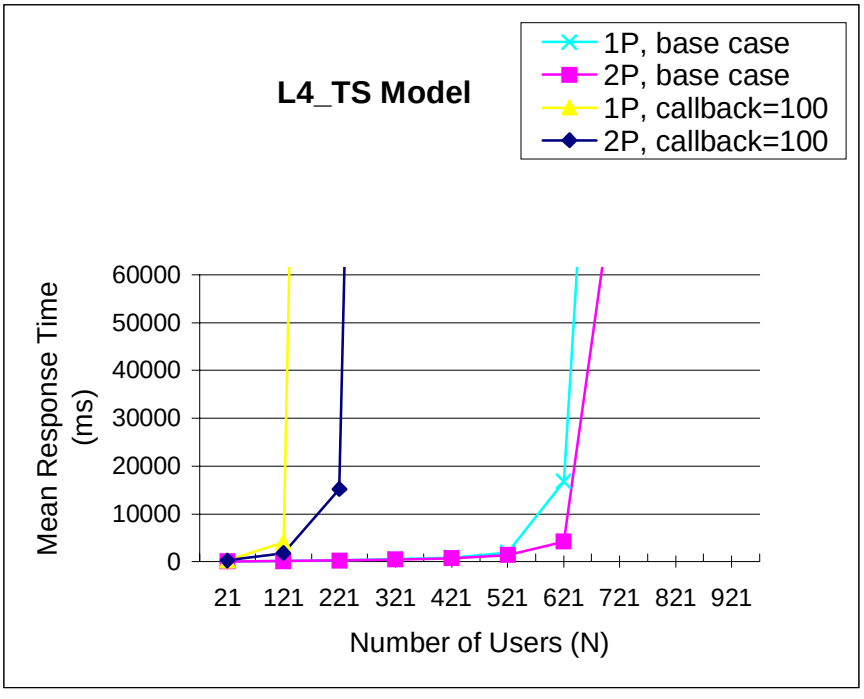

Figure 7.7 Comparison of Mean Response Time for 1P and 2P Cases

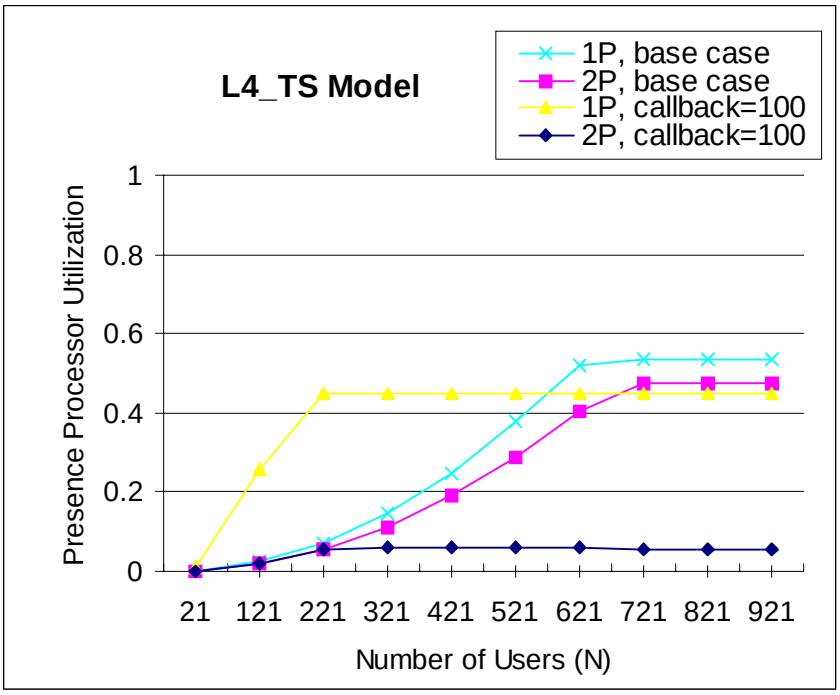

Figure 7.8 Comparison of Presence Processor Utilization for 1P and 2P Cases

7.3.3 Performance Analysis for 2T2P Cases

From the tests in section 7.3.1, we know that the notification is the main workload of the presence system and the PA task is the bottleneck of the system. The reason is that the notification execution is related with the subscribers' number Nsub. When the total users' number N increases, the subscribers' number (Nsub=0.1*N) increases, then the repeat number of notification increases. Therefore, the system saturates when the users' number N is above 200 when notify demand is 100ms.

To break the bottleneck of the system, we need to reduce the workload of the notification operation. The repeat number is the key value of the notification operation. We assume that the subscribers are managed by two PA tasks and each PA task works on one Presence processor. One PA manages the half of the total subscribers. The practical implementation example may let one PA manage the subscribers in the same domain. In

this way, we can reduce the notification repeat number by 50%. Then the notification workload of the presence system will be reduced.

The following deployment is based on the tests in section 7.3.2, 2P cases, where the separate processor is provided for the Tuplespace task.

From the test results in section 6.3.3, we know that the 2 threads for PA on 1 presence processor has some improvement for the performance. In the following test case, we let 2 threads of PA work on 2 separate presence processors. We call this test case as 2T2P case.

- Test L4ts-6: This is 2P case. The PA task has 1 thread and works on one presence processor. The **notify** demand is increased to 100ms.
- Test L4ts-7: This is 2T2P case. The PA has 2 threads and works on two presence processors. The loop number is half of the subscribers' number. The **notify** demand is 100ms.

Following figures 7.9 to 7.11 shows the output of comparison of 2P case with 2T2P case. From the test results, we know that the 2T2P case's performance is improved compared with 2P case.

Figure 7.9 shows that the throughput of 2T2P case increases from 0.2 to 0.4 responses/second. Figure 7.10 shows the mean response time decreases from more than 400s to about 3s for 200 users.

In Figure 7.11, the presence processor utilization can be read as "the mean number of busy processor" and has a maximum value of 2.0. It shows that the presence processor utilization is about 50% for L4ts-6 case and about 130% for L4ts-7 case. The system saturates at bigger number of users N. For L4ts-6 case, N is 200, and for L4ts-7 case, N is 300.

From the tests results, we found that the presence agent task is still the bottleneck for both cases and the utilization is 100% for L4ts-6 case, and 200% for L4ts-7 case.

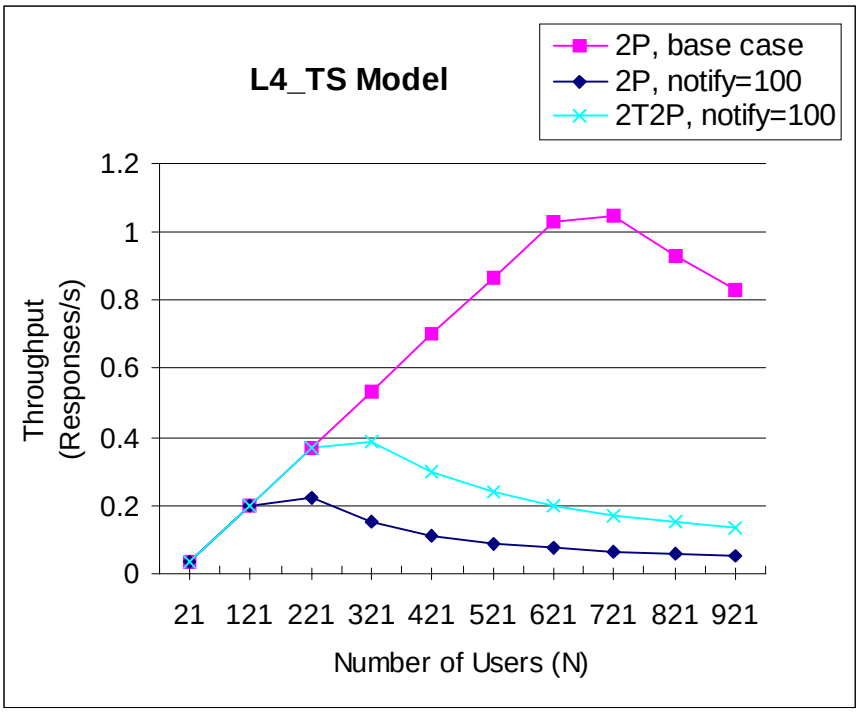

Figure 7.9 Comparison of Throughput for 2T2P Cases

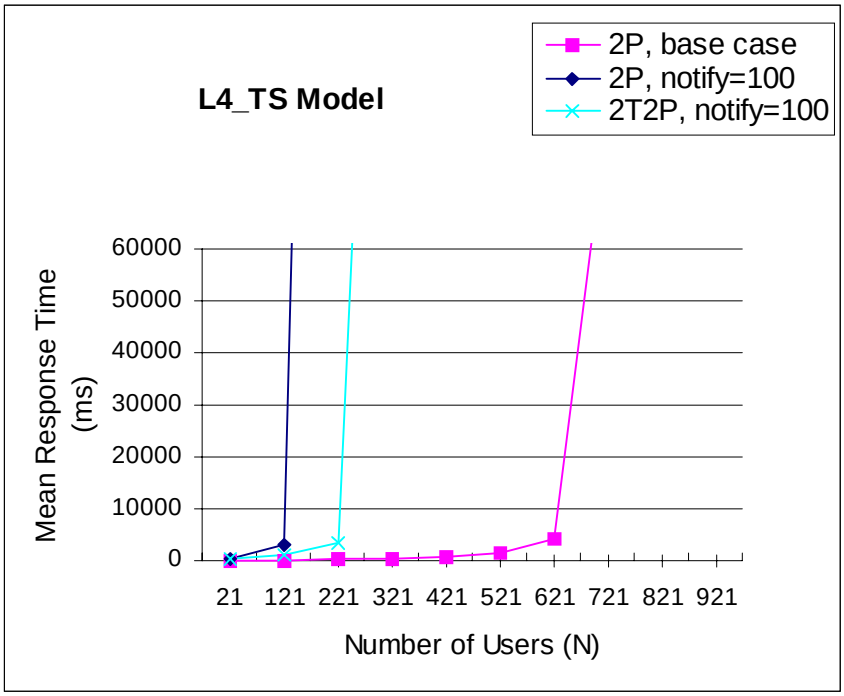

Figure 7.10 Comparison of Mean Response Time for 2T2P Cases

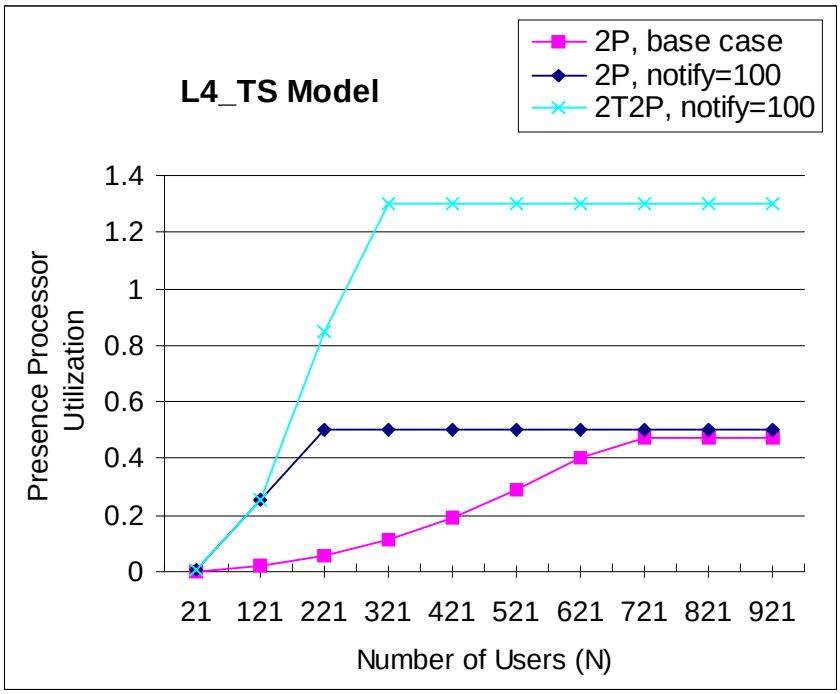

Figure 7.11 Comparison of Processor Utilization for 2P2T Cases

7.3.4 Performance Analysis for Network Delay

Following tests add a Network task between the EListener and the Watchers. The Network task receives the notification from the Elistener task and forwards the notification to the Watcher task. It models the pure communication delay between the presence server and the end users. In the LQN model the Network task is allocated to a Network processor. Let the delay time take the value 10 ms or 50ms to represent the fast network transmission and the slow network transmission.

In the following tests, the Network task works on the Network processor. Other tasks' allocation is the same as the 2P base case. The demands' values are the same as 2P base case also.

- Test L4ts-8: The Network delay time is 10 ms.
- Test L4ts-9: The Network delay time is 50 ms.

Figure 7.12 to Figure 7.14 show the performance results and the comparison with the 2P base case.

From the Figure 7.12 to figure 7.14, we know that the network delay between the presence server and the Watcher has effect on the presence system's performance. When the network delay is 10ms, the throughput decreases from 1 to about 0.8, and the system saturates at above 500 users. The presence processor utilization decreases from 50% to 30%. When the network delay is 50ms, the throughput decreases from 1 to about 0.5, and the system saturates at about 300 users. The presence processor utilization decreases from 50% to 10%. The network delay makes the software bottleneck at PA more severe, because it is included in the blocking time of PA.

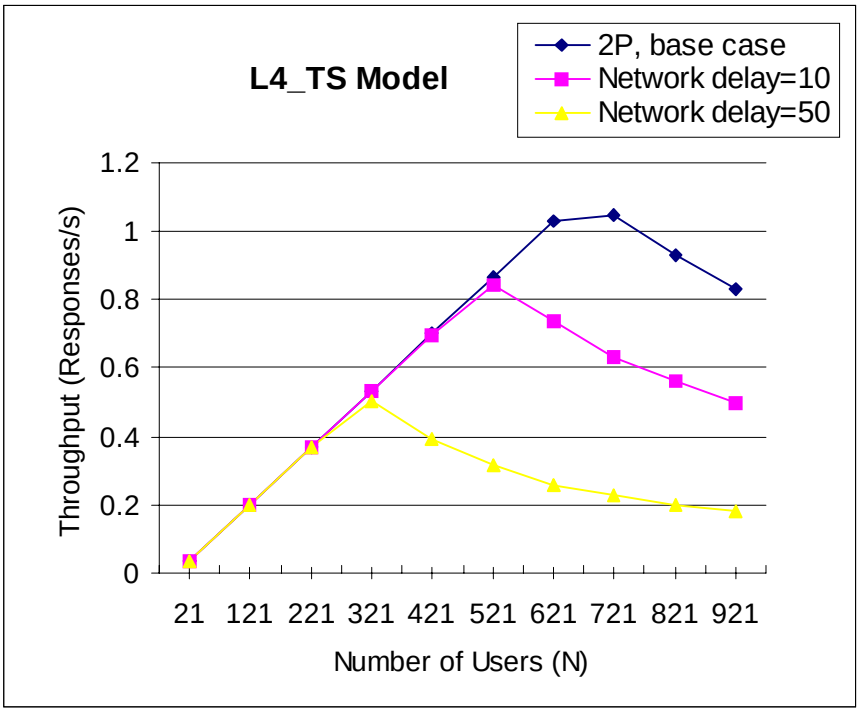

Figure 7.12 Comparison of Throughput for Network Delay Cases

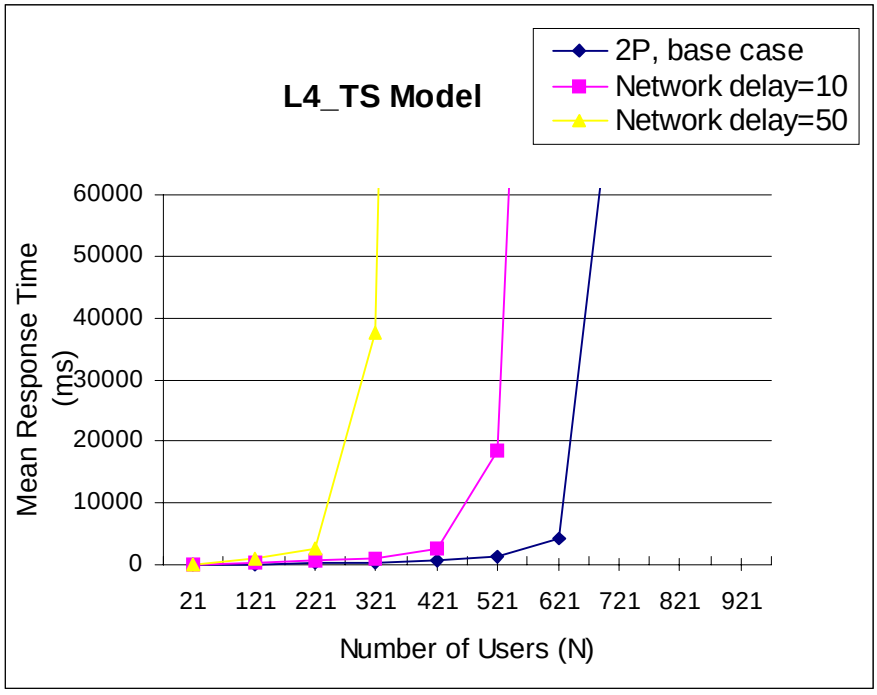

Figure 7.13 Comparison of Mean Response Time for Network Delay Cases

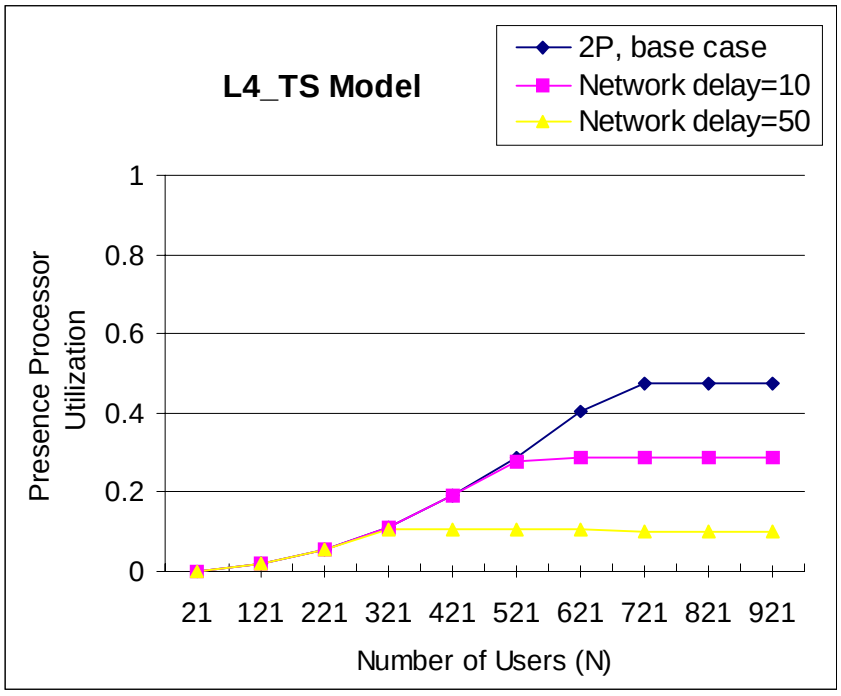

Figure 7.14 Comparison of Processor Utilization for Network Delay Cases

7.4 Discussion of Model L4-TS

From the above tests results, we know the following issues should be considered when design the presence system with Tuplespace implementation.

- Even if the network delay is 10 ms, it still affects the performance of the presence system. It increases the mean response time, decreases the throughput and lowers the presence processor utilization.
- Using a separate Tuplespace processor for Tuplespace task will increase the throughput and performance.
- The system is very sensitive to the notification CPU demand. When **notify** demand increases from 10ms to 100ms, the effect is very large to the performance.
- The bottleneck happens at the PA task. The PA task's utilization reaches 100% before any other tasks or devices. It is a software bottleneck.
- When the system saturates, make the PA 2 threads increase the performance. When 2

PA threads work on 2 separate presence processors, the improvement is significant.

Following figures plot the comparison of the L4-TS model and L3 block model. In the figures, the L4-TS 2P curve represents the L4-TS 2P base case, and the L3 block represents the L3 model block case.

Figure 7.15 and Figure 7.17 illustrate the throughput, mean response time and presence processor utilization of the two different level models L3 block and L4-TS. The L4-TS model shows the consistent trends with the L3 model. The presence processor utilization reaches 50% and the bottleneck is Presence Agent task for both cases. However, the system saturates points are different. For L3 block model, the N is 700. While for L4-TS model, the N is 600. The difference is because more components are added to the L4-TS model.

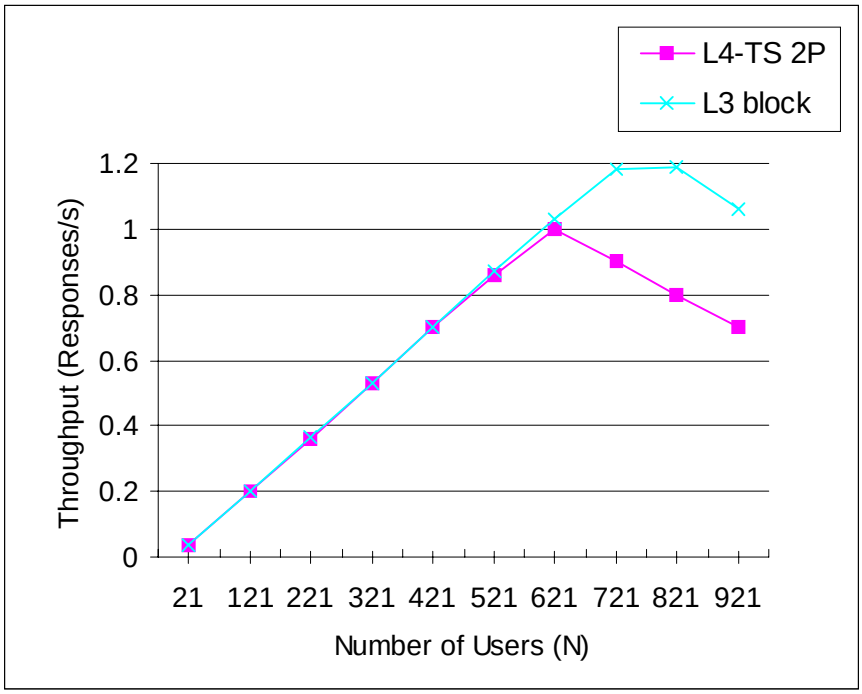

Figure 7.15 Comparison of Throughput for L4-TS and L3 Models

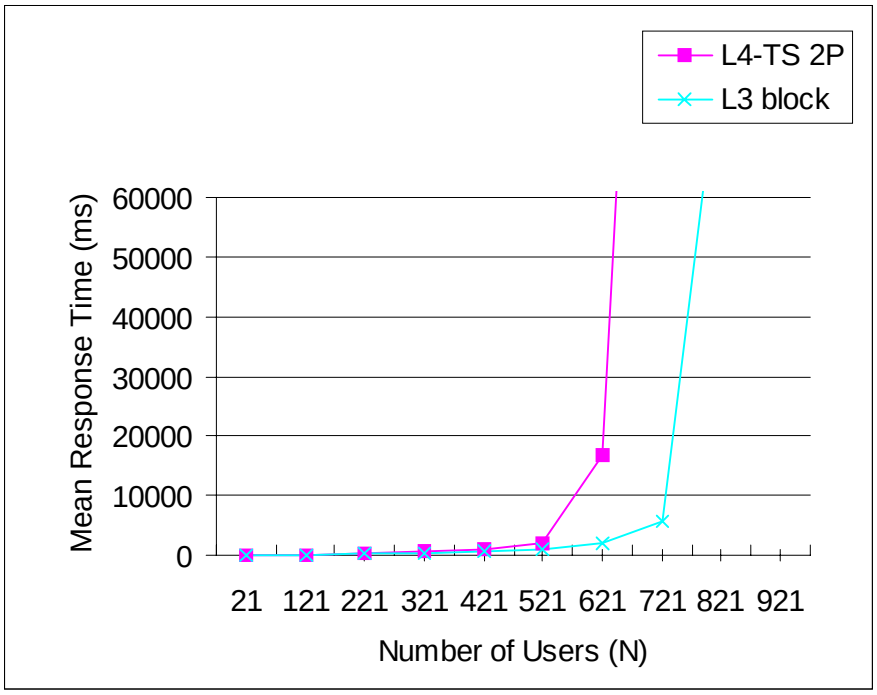

Figure 7.16 Comparison of Mean Response Time for L4-TS and L3 Models

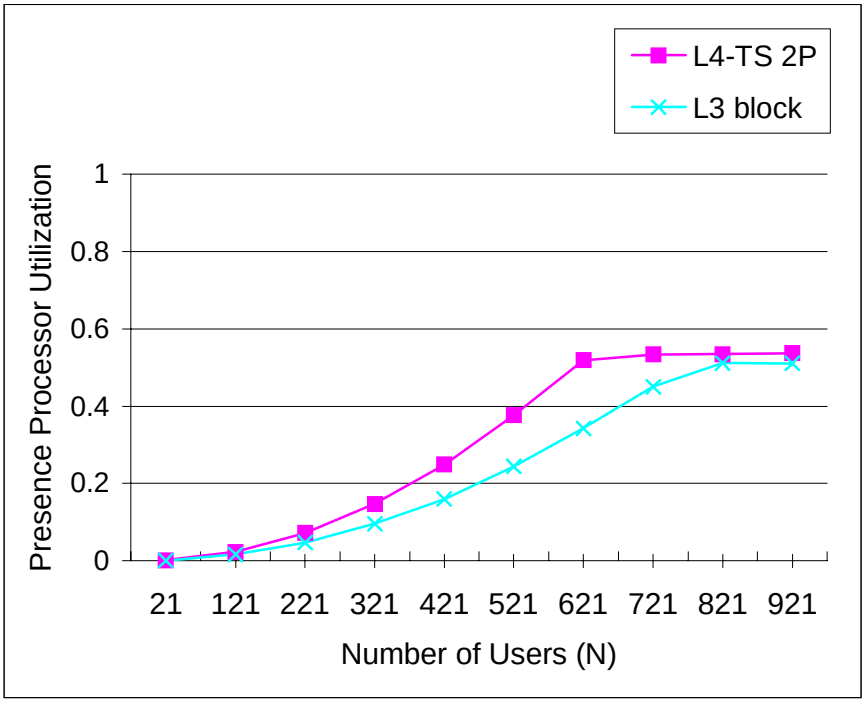

Figure 7.17 Comparison of Processor Utilization L4-TS and L3 Models

Chapter 8 Apply MSPA to A Presence System Prototype

In this chapter, a presence system prototype is described and the MSPA methodology is applied to the prototype. Section 8.1 describes the presence system prototype. Section 8.2 describes the event notification mechanism used in the prototype. Section 8.3 presents the UCM model of the prototype. Section 8.4 shows the measurement of the prototype. Section 8.5 gives the performance model and the analysis results. Finally, in section 8.6 the analysis results are discussed and compared with the higher abstract model L4-TS.

An important difference from the previous levels come from the ability to measure the CPU demands of responsibilities, and use these values in the performance model.

8.1 Description of the Prototype

The presence system prototype provides many functions to the end users. These functions include the administrators' functions and the users' function. Examples of the administrators' functions are management of users and groups and initialization of the system. For instance, an administrator can register a user with user's name and password and can delete the user. The administrator can also create a group or delete a group. Only the registered users can use the presence system. The users' functions include logging in and logging out of the system, subscribing to a Presentity, unsubscribing from a Presentity or canceling a subscription. Users can also update their current presence status such as contact address and devices. Users can modify their subscription rules or policies.

In this thesis, the update presence information (UpdatePI) function was chosen for the full analysis. After the system is initialized and the users log in the system and subscribe to

their interesting Presentity, the UpdatePI scenario may execute whenever the users update their presence status. Therefore, the scenario may occur frequently and has a relatively large influence on the system's performance.

The presence system prototype is implemented with the Java language. Java's portability and platform's independence allow the prototype executing on any environment with a Java Virtual Machine (JVM). The Client component is implanted with the Java applet which supports graphical user interface (GUI) development and let users access the presence system through a web browser. The Java Servlet technology is applied on the server end and the dynamic presence information can be generated and present to the GUI.

8.2 T Spaces Event Notification Mechanism

Like model M4-TS in chapter 7, the prototype uses the IBM T Space notification mechanism described in section 2.5 to facilitate the communication between the distributed components

For different functions, the system uses Tuplespace event notification mechanism to handle the different events. When the system receives a request from the user, the corresponding event management component is responsible for posting the event to the Tuplespace. The corresponding event listener that registers with the Tuplespace will be notified about the event. This architecture gives the flexibility for adding new events to the system. In the UCM model, Events Manager (EsManager) and Events Listener (EsListener) components represent the any pair of the events.

For the notification to the subscribers, T Spaces notification is used to facilitate the communication between the Presence Agent (PA) and the Watcher User Agent (WUA) components. This notification is accomplished by adding two components, one is

Availability Event Manager (AvailEManager) and the other is Availability Event Listener (AvailEListener).

One user can register more than one device such as PC and mobile phone to the system. If the user subscribes these devices/resources to a Presentity at the same time, when the Presentity changes its status, the updated presence information will be notified to all the devices/resources of the user. In the prototype, the device/resource that wants to be notified automatically whenever the presence changes is represented as a Client Proxy. The communication between the user's resource/device manager component and the Client Proxy is also implemented using Tuplespace notification mechanism. The notification pair are Resource Manager (RM) and Client Proxy (ClientProxy) components.

8.3 UCM Model of the Prototype M5

This section represents the UCM model M5 of the prototype. As described before, the UpdatePI scenario is the one to be analysed in detail. The exception path and the time out path are not considered in this model. The model M5 has three UCM maps that are illustrated in Figure 8.1 to figure 8.3. Figure 8.1 is the root map of the UpdatePI scenario. It has two stubs, one is ProcessReq (refer Figure 8.2) and the other is Notify (refer Figure 8.3).

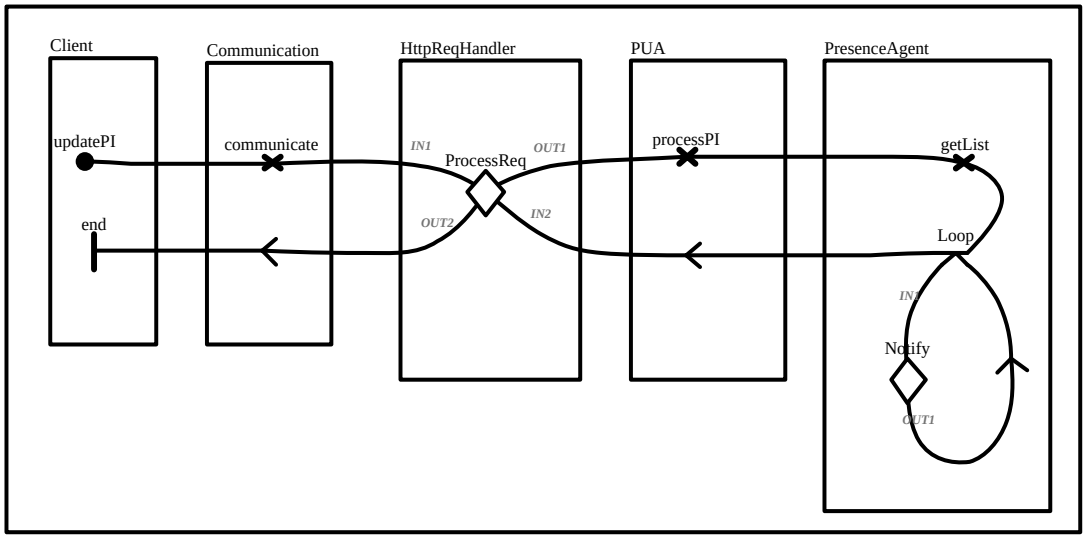

Figure 8.1 Presence System UCM Model M5: Root Map

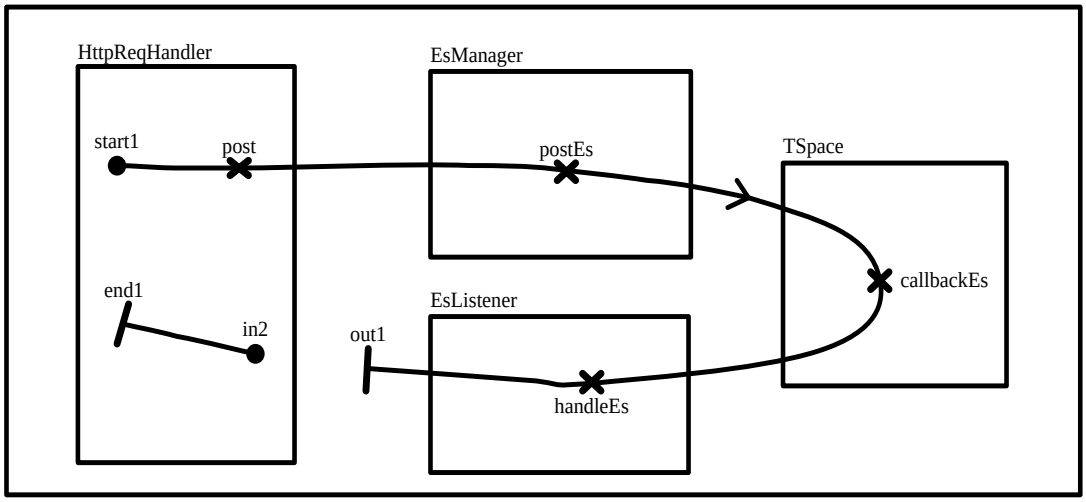

Figure 8.2 Presence System UCM Model M5: Stub ProcessReq

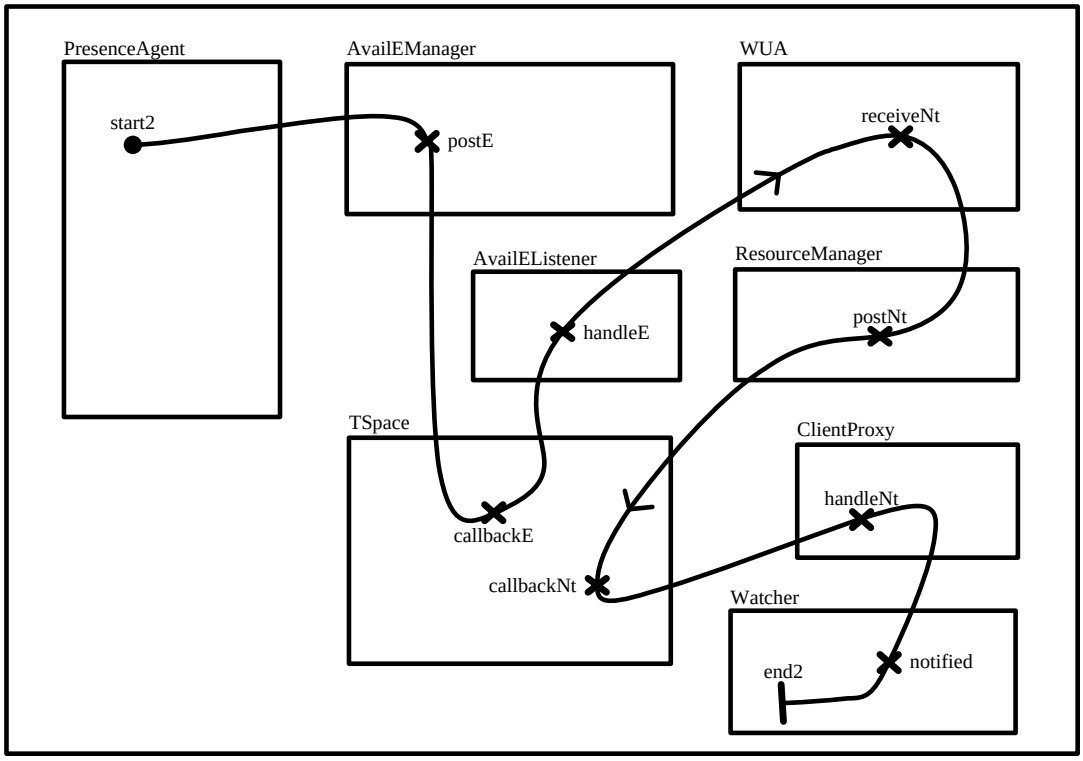

Figure 8.3 Presence System UCM Model M5: Stub Notify

Following is the list of components and their functions (represented with responsibility name) in the model M5.

- Client: is a GUI for the end user to interact with the presence system. It may be an applet and runs inside a browser. The Client accepts the users' requests and makes a synchronous call to the Communication component. Communication.
- Communication: is a component that gets requests from the Client and sends the requests to the Http Request Handler. The responsibility name of the component is **communicate**.
- Http Request Handler (HttpReqHandler): is a COTS component named Apache Http Server. It accepts the requests from the Communication component and processes it, and then creates dynamic responses to the Client. When processing the requests, it differentiates different users' requests and handles as different events. It does so by sending the request to an appropriate event manager component. The responsibility

name is **doPost**.

- Events Manager (EsManager): is one event manager component that may represent any event manager. It gets request from the HttpReqHandler and deposits the event request to the Tuplespace. The responsibility name is **postEs**.
- Events Listener (EsListener): is the corresponding event listener component to one event manager. The EsListener registers with the Tuplespace during the subscription process and may be called by the Tuplespace whenever a matching event is written to the Tuplespace. The responsibility name is **handleEs**.
- Tuplespace: is a COTS system and is integrated with the prototype. It provides an event register and notification functions to the presence system. With the callback mechanism, the event listener can be notified whenever its registered event is written to the Tuplespace.
- Presence User Agent (PUA): is a component that manipulates presence information for a Presentity. It exists in the L4-TS model. The responsibility name is **processPI**.
- Presence Agent (PA): is a component that accepts subscriptions, stores subscription information, and generates notifications when there are changes in the presence status. It exists in the model M4-TS. The responsibility name is **getList.** In the implementation of the prototype, the PA is co-located with PUA. The PA keeps the subscription list in a vector.
- Availability Event Manager (AvailEManager): is a component that is responsible for the notification of change to all subscribers. It deposits the UpdatePI change to the Tuplespace. The responsibility name is **postE.**
- Availability Event Listener(AvailEListener): is a component that registers to the Tuplespace and waits for the notification whenever the AvailEManager posts the UpdatePI to the Tuplespace. The responsibility name is **handleE.**

- Watcher User Agent (WUA): is a User Agent that is responsible for the requests of subscription to other users' Presentity. In the prototype, it is implemented together with the PUA. Here in the UCM model M5, we use WUA to represent the Watcher. The responsibility name is **receiveNt.**
- Resource Manager (RM): is a component that keeps the records of the users' resources/devices. RM keeps active resources/devices in a Hashtable. It deposits the UpdatePI change to the Tuplespace. The responsibility name is **postNt.**
- Client Proxy: is a component that registers with the Tuplespace and waits for the notification. The responsibility name is **handleNt.**
- Watcher: is a component represents the end user. The responsibility name is **notified.**

The scenario UpdatePI is started by a user entering his/her current presence information to the system through the Client component. The Client component sends the request to the Communication component. Then the Communication component forwards the request to the HttpReqHandler. Inside the HttpReqHandler component, there is a stub named ProcessReq that illustrates the scenario in the stub map. The HttpReqHandler processes the request and creates a dynamic response to the Client through the Communication. It executes doPost responsibility and forwards the message to the EsManager. The EsManager deposits the UpdatePI change event to the Tuplespace. Then the Tuplespace executes callbackE responsibility. The EsListener is called by the Tuplespace. The scenario in the stub ProcessReq ends at out1 end point. Then the UCM path continues in the root map.

The EsListener component makes a forwarding call to the PUA component and the PUA executes the responsibility processPI. The PUA component then forwards the call to the PresenceAgent component. The PresenceAgent processes the request by executing

getList responsibility and then makes a number of notification operations according to the subscribers' list one after another. In the M5 root map, the repetition of notify responsibility is represented by a loop. The repetition number is subscribers' number.

One notification detail is represented in the stub map Notify. In the stub (Figure 8.3), the PresenceAgent component sends the UpdatePI change information to the AvailEManager component and the AvailEManager executes postE responsibility to write the event to the Tuplespace. The Tuplespace calls the AvailEListener component about the event deposition. Then AvailEListener executes handleE responsibility and forwards the UpdatePI change to one WUA according the subscriber's identification. The WUA makes a forwarding call to its ResourceManager. The ResouceManger finds out all its active resources/devices from the Hashtable it keeps. Then the ResourceManager component writes an event tuple to the Tuplespace. The Tuplespace calls the ClientProxy that registers the event with the Tuplespace.

After that the ClientProxy executes handleNt responsibility and notifies the Watcher. After finishing the Notify stub scenario, the PUA returns a reply to the HttpReqHandler. Finally the HttpReqHandler constructs the response to the Client component through the Communication component.

8.4 Measurement of the Prototype

The CPU demands values of the responsibilities for all components in the model M5 were measured by running the prototype in its implementation environment.

8.4.1 Working Environment of the prototype

The prototype executes on Windows 2000. The machine's processor is Pentium with 466 MHz and 256M memory. The Sun JDK 1.2 is used to run the prototype. The Http

server is Apache version 1.3.9. The IBM T Space framework is version 2.1.1.

The instrumentation probes are inserted in the source code with system call millisecond( ) in Java language. Two time values are recorded for the measured responsibility, one is the start time and another is the stop time. The difference of the time interval is the measured responsibility's CPU execution time. Because only two lines are inserted in the source code to get the execution time and the performance impact of the additional system calls is neglected when compared with the amount of work done by each responsibility. The resolution of the measured time values is 10 milliseconds under the above working environment.

When a number of measured values are averaged, the average is more accurate, and it can be shown that as the number of measured values approach infinity, the average is consistent. However, because scenarios had to be manually trigged, only about ten measured values were obtained for each activity.

The prototype is initialized and three test cases are measured.

- Case 1: has two users and one user subscribes to another user called user1.
- Case 2: has three users and two users subscribe to one user called user1.
- Case 3: has 10 users and nine users subscribe to one user called user1.

For all three test cases, let the user1 change its status and record the execution time for each responsibility. Each test is measured for 10 times. Table 8.1 gives the measured average values for the three cases.

From the measured execution time values in the table 8.1, we know that the execution time values are dependent on the number of total users and the subscribers. To simplify the performance analysis later in section 8.4, we use the mean values in the last column of

table 8.1 of three cases in the performance analysis. Column Mean Value shows the average time.

Because the Apache Http server is a COTS component, we do not measure the responsibilities' execution time. We assume the estimated execution time is 10 ms for the Communication component.

Component Name	Responsibility Name	Case 1 (ms)	Case 2 (ms)	Case 3 (ms)	Mean Value (ms)
HttpReqHandler	post	48	30	30	36
EsManager	postEs	20	9	31	20
Tuplespace	callbackEs	39	40	50	43
EsListener	handleEs	0	0	0	0
PUA	processPI	0	0	0	0
PresenceAgent	getList	57	51	50	53
AvailEmanager	postE	0	0	0	0
Tuplespace	callbackE	1	0	7	3
AvailEListener	handleE	0	0	0	0
WUA	receiveNt	0	0	0	0
RM	postNt	50	65	209	108
Tuplespace	callbackNt	4	0	7	4
ClientProxy	handleNt	7	0	6	4

Table 8.1 Measured Execution Time for Model M5

8.5 Performance Analysis of the Prototype

8.5.1 Create Performance LQN Model L5

Following the MSPA methodology, the performance annotated UCM model P5 was created by entering the performance parameters to the model M5 from the above measurement and the assumption of the model M5's deployment.

We assume there are four processors used in the M5 model, they are User processor (UserP), Communication processor (CommP), Presence Server processor (PresenceP) and Tuplespace Server processor (TuplespaceP). Components Client and Watcher are allocated to UserP, and each user has its own execution thread. Communication component and HttpReqHandle components are allocated to CommP and has infinite threads working on it. The Tuplespace component is allocated to the TuplespaceP processor. The rest of the components are allocated to the PresenceP processor.

When the performance annotated model P5 is finished, the LQN model L5 is created by using tool UCM2LQN from model P5. Some execution controlling values are added to make the model suitable for the SPEX execution controller (See Appendix F: L5.xlqn).

8.5.2 Performance Analysis of the LQN Model L5

Because of the difficulty of setting up hundreds of users and subscribers at the same time to get the measurement of execution time, we assume that the estimated time values in the table 8.1 do not change with different number of users and subscribers.

Like the tests in the chapter 7, the total users' number N changes from 21 to 921 with increment 100. And the subscribers' number Nsub is 0.1*N.

The following test cases were performed.

- Test L5-1: This is L5 base case. The L5 model uses all estimated values in table 8.1

and deployment described in section 8.4.1.

- Test L5-2: This is called RM=10 case. The parameters are same as L5 base case except that the RM component's responsibility **postNt** demand time is 10ms.
- Test L5-3. This is 2T2P case. The PA has 2 threads and works on two presence processors. The loop count number is half of the subscribers' number Nsub. Other parameters are the same as base case L5-1.

Figure 8.4 to Figure 8.8 illustrate the test results.

From the tests' results, we see that when the users number N is greater than 100, the system saturates and the bottleneck is PA task for base case. At the saturation point N=200, the throughput is about 0.2, the presence processor's utilization is about 60% and the PA task's utilization is 100%.

When reducing the execution demand of RM's responsibility **postNt** from 108 ms to 10 ms, the performance of the model L5 is improved greatly. The saturating point N is 400 instead of 200 and the throughput at this point increases from 0.2 to around 0.7. The mean response time is about 10000 ms (10s). The presence processor's utilization decreases from 60% to 50% and the PA task's utilization is 100%. The bottleneck is still PA task.

For 2T2P case, where 2 threads PA task are allocated on 2 presence processors, the performance of model L5 is improved. The saturating point N is about 300, which is increased from 200. And the throughput increases from 0.2 to around 0.4 and the mean response time is about 6700 ms. The presence processor's utilization is about 100% for 2 processors and the PA task's utilization is 200% for 2 threads.

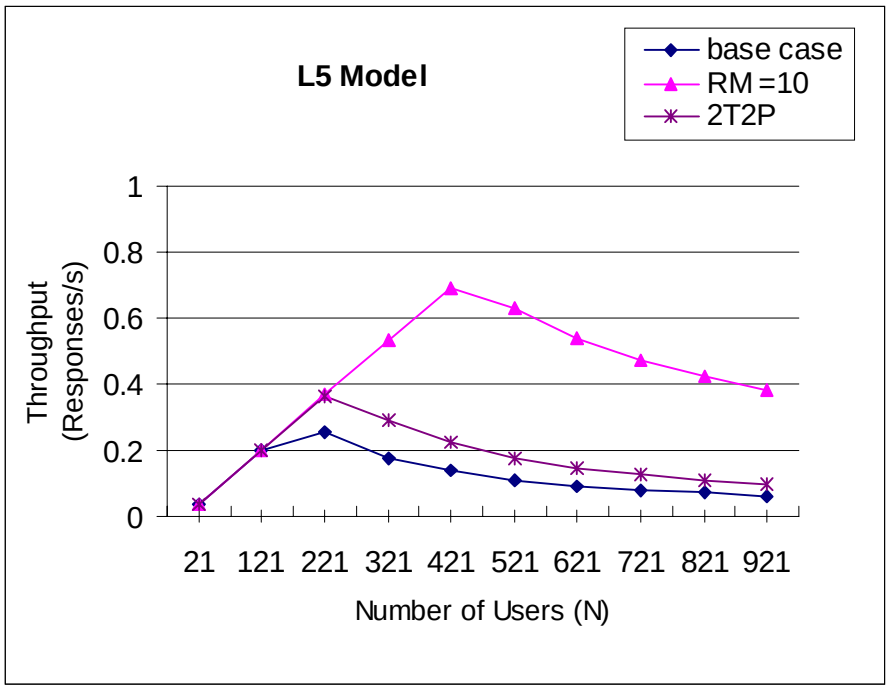

Figure 8.4 Throughput for Model L5

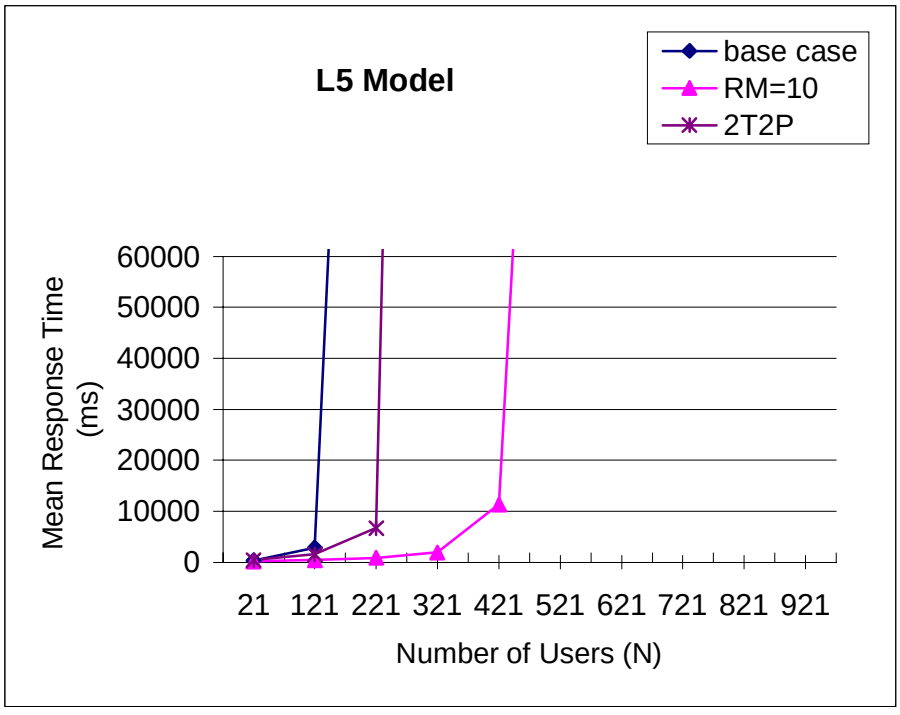

Figure 8.5 Mean Response Time for Model L5

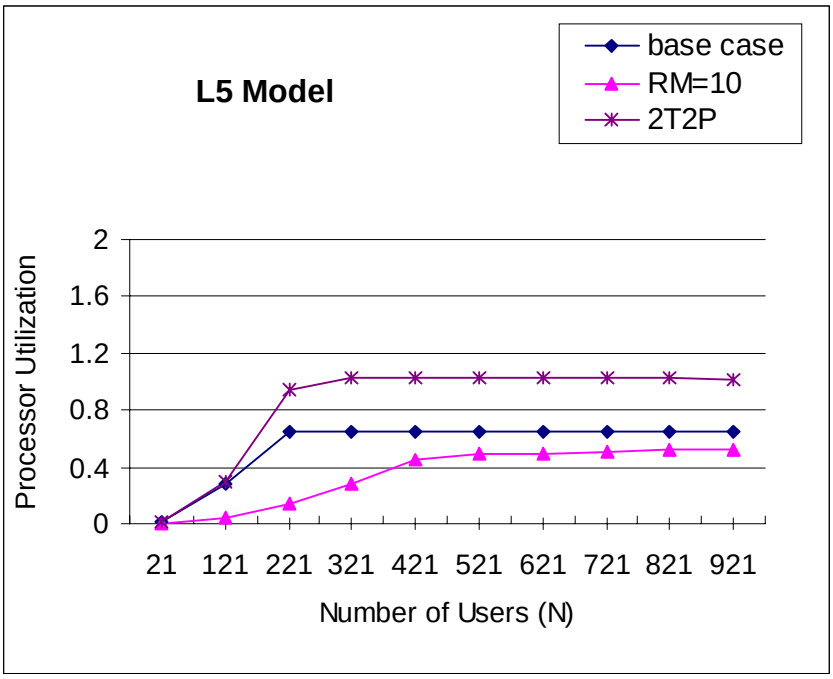

Figure 8.6 Presence Processor Utilization for Model L5

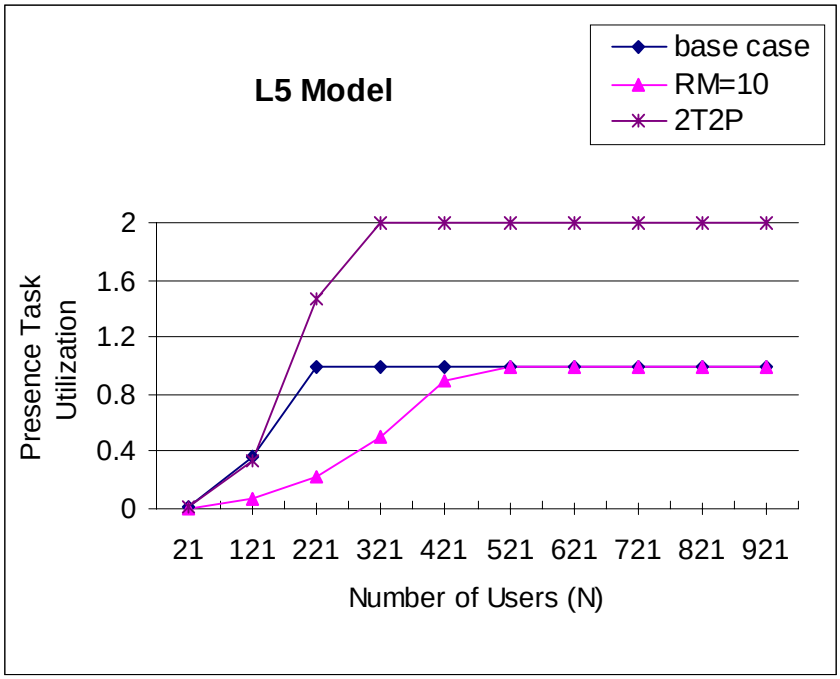

Figure 8.7 Presence Task Utilization for Model L5

8.6 Discussion of Prototype Model L5

From the above tests' results, we know the bottleneck happens at the PA task. The task's utilization reaches 100% before any other tasks or devices in the system and the processor utilization does not reach 100%. For 2 threads PA execute on 2 presence processors case, the PA task's utilization reaches 200% and the presence processor's utilization does not reach 200%.

The RM component's responsibility **postNt** execution demand is large, at 108ms. From the earlier study for L4-TS model in chapter 7, we know that the notification execution time has big effect on the performance of the presence system. The test result shows that when reduce the **postNt** demand from 108ms to 10ms, the system's performance is improved. In the prototype the RM component uses Hashtable to store the information of the notified resources. In the real system, the performance of the access time of the disk or database should be concerned.

2 threads of PA tasks working on 2 processors improve the performance of the system. Because each PA thread manages only half of the total subscribers per user, the notification time is reduced and the mean response time is reduced. The saturating point N increases to 200 from 100 and the throughput increases from 0.2 to around 0.4 responses per second. For 2 threads PA tasks on 2 processors case, the bottleneck is still PA task.

Following figures plot the comparison results for the throughput and response time of L5 model and L4-TS model. In the figure, the L5 model results are the base case test and the L4-TS model results are the 2P case test with **notify** equals 100 ms.

Figure 8.8 show that the two models' throughputs curves are quite close. From the throughputs curves, the system saturating point N is around 200 for both L4-TS and L5 models.

Figure 8.9 show that the two models' mean response time curves are almost same for both cases.

The test results also give the system's bottleneck, which is the Presence Agent task for both cases.

The above result show that the performance prediction from the more abstract model L4-TS can be used for the more detailed and more complex model L5, even though more components are added to the L4-TS model.

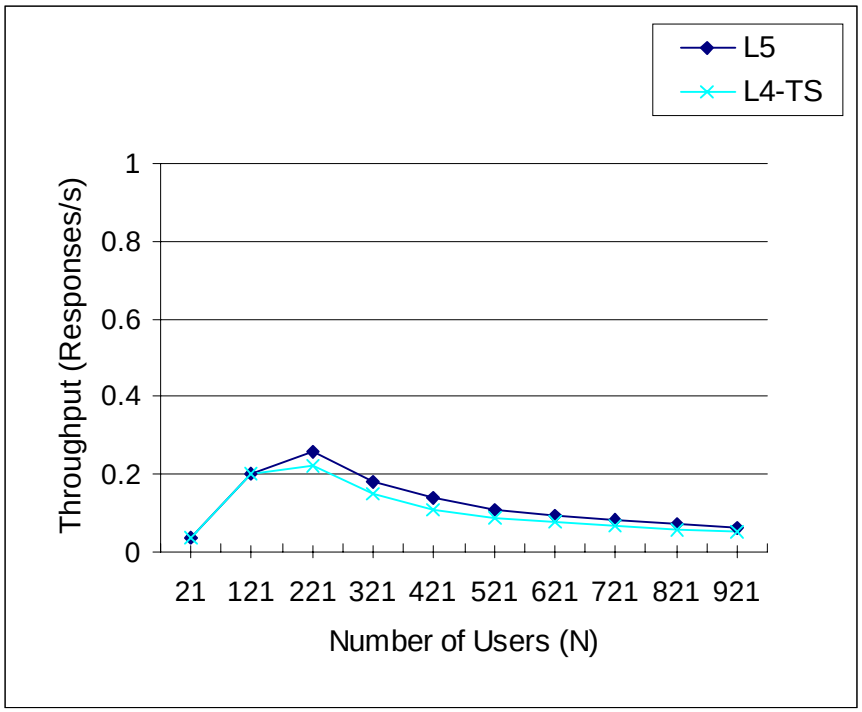

Figure 8.8 Comparison of Throughput for L5 and L4-TS Models

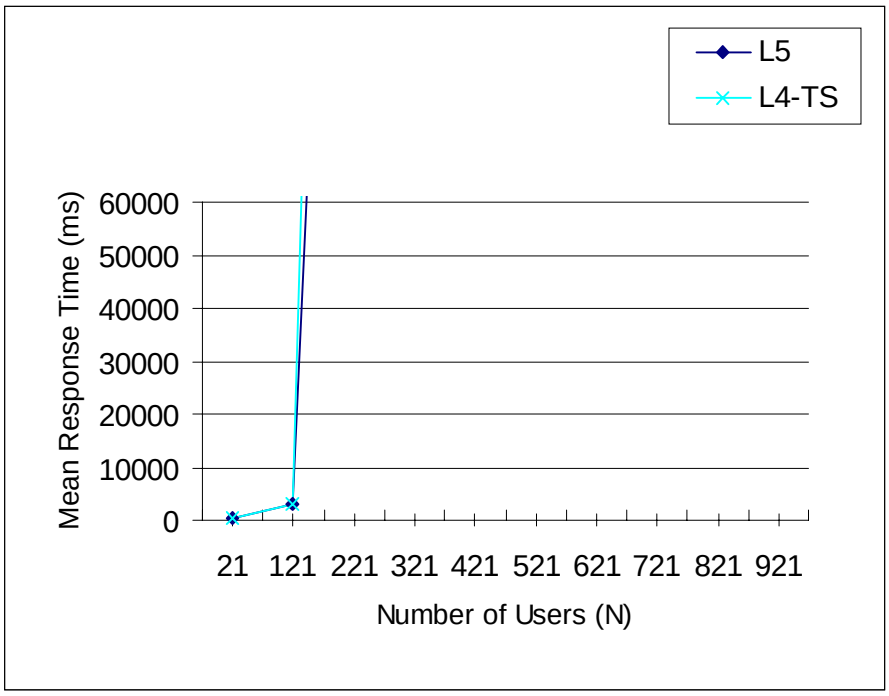

Figure 8.9 Comparison of Response Time for L5 and L4-TS Model

Chapter 9 Conclusions and Future Work

In this thesis, a multilevel specification and performance analysis (MSPA) methodology is proposed to perform performance analysis for the large, complex, distributed and Internet based system during the early software development stage. The presence system, which is integrated with Internet and telephony technologies, is analyzed as a case study. Five levels of abstract specifications in the forms of Use Case Maps (UCM) are developed incrementally. And the correspondent performance models in the forms of Layered Queuing Networks (LQN) are analyzed using LQN model solvers. For each level's LQN model, an automated converter tool UCM2LQN, which was developed by Petriu [Petriu01], is used to transform the UCM model to the LQN model. From the performance analysis results, the scalability of the presence system is studied; the sensitive performance parameters of the system are analysed; and the performance bottleneck of the system was found. The case study shows that the important performance conclusions from the earliest levels are still valid at more detailed levels of specification, even if the structure and the parameters of the system are altered.

This chapter lists the important conclusions that are discussed in this thesis and presents some suggestions for future research.

9.1 Conclusions

The MSPA methodology presented in chapter 4 worked for the presence system case study. The study shows that the MSPA methodology is easily applied to the five

successive specifications of the presence system because of the application of the automatic tools. The specification of the presence system is captured by UCM model and the performance LQN model is generated from the UCM model by UCM2LQN converter. For each level LQN model, the operation demands are budgeted and the performance results are obtained through the LQN solver.

The benefit of applying the MSPA methodology is the obtaining the earlier performance predictions for the more abstract models in the early design phases. The case study shows that the important performance results for the simpler, more abstract models are still valid for the more complex and more detailed models.

From the five level specifications and performance analysis of the presence system, several conclusions can be drawn as follows.

- The subscribers' number per user limits the scalability of the presence system because the subscribers' number determines notification execution time. The more subscribers that need to notify, the more CPU demand the notification operation. The total users' number also limits the scalability of the system.
- The **notification** CPU demand of the Presence Agent is the most sensitive performance parameter to the presence system. The higher the **notification** demand, the lower the throughput, and the longer the mean response time of the system. Reduce the **notification** operation demand improve the performance greatly.
- The presence system's bottleneck is the Presence Agent (PA) task, which is a software bottleneck. PA task utilization reaches 100% fast than any other tasks and devices in the presence system.
- To improve the performance of the system, multithreads and multiprocessors can be used to break the bottleneck of the system. The case study shows that two threads Presence Agent, or two threads Presence Agent work on two presence processors

improve the performance of the system.

- Network delay even if it is small still delay the response time of the system.
- For the implementation with T Spaces framework, a separate processor for Tuplespace task will increase the throughput and performance of the system.
- For the implementation with SIP notification system, separate processors for Proxy and Redirect/Registrar servers improve system's performance.

9.2 Future Work

Several issues need to be addressed in the future work to get more accurate performance model results in the case study.

- In the present study, only one scenario is analyzed. The real system has more than one scenarios executing at the same time. For example, some users may log in and log out the presence system, while other users may subscribe to or unsubscribe from some Presentitities. And at the same time, some users may update their current presence information. With more scenarios are analyzed together, more accurate performance model will be created and more accurate performance results will be obtained.
- The current performance analysis omits the security check and time-out failure scenarios. This simplicity assumption lowers the workload of the system. For example, the security check may access a database or a file system, which needs to access a database server or a disk. Future work may add the exception and check path to the model.
- The current Tuplespace framework has only one Tuplespace server. The deployment of distributed Tuplespace servers can be analyzed to investigate the performance effects with the distributed Tuplespace servers.
- The current SIP deployment model is very simple. Future study can implement a

prototype with more SIP entities. The measurement of the performance parameters can be conducted on the prototype. Then more performance predictions can be made.

References

[Amyot00] D. Amyot, “Use Case Maps as a Feature Description Language”, Language Constructs for Designing Features, Springer-Verlag. pp. 27-44.

[AmyotT99] D. Amyot, “Use Case Maps Quick Tutorial Version 1.0”, 1999, WWW http://www.usecasemaps.org/pub

[Balsamo01] S. Balsamo and M. Simeoni, "Deriving Performance Models from Software Architecture Specifications", European Simulation Multiconference 2001 (ESM 2001), Prague, June 6-9 2001.

[Brown00] A.W. Brown, “Large-Scale Component-Based Development”, Prentice-Hall PTR, 2000.

[Bruin00] H. de Bruin and H van Vliet, “Top-Down Composition of Software Architectures”, Proceedings 9th Annual IEEE International Conference on the Engineering of Computer-Based Systems (ECBS), April 8-11, 2000, IEEE, pp 147-156.

[Buhr96] R.J.A. Buhr and R.S. Casselman, “Use Case Maps for Object-Oriented Systems”, Prentice Hall, 1996.

[CPIM] D. Crocker, A. Diacakis, C. Huitema, G. Klyne, J. Rosenberg, H. Sugano, “Common Presence and Instant Messaging (CPIM)”, Internet draft-ietf-impp-cpim-02, Nov. 2001, work in progress.

[Franks99] R.G. Franks, “Performance Analysis of Distributed Server Systems”, Ph.D Thesis, Carleton University, 1999.

[Gelernter82] D. Gelernter, A.J. Bernstein, “Distributed Communication via Global Buffer”, in Proceedings of ACM Symposium on Principles of Distributed Computing, August 1982, pp.10-18.

[SIP-pres] J. Rosenberg, D. Willis, H. Schulzrinne, C. Huitema, B. Aboba, D. Gurle, D.Oran, “ Session Initiation Protocol (SIP) Extensions for Presence”,

Internet Draft draft-ietf-simple-presence-07.txt, May, 2002, work in progress.

[SIP] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg, “SIP: Session Initiation Protocol,” Request for Comments (Proposed Standard) 2543, March 1999.

[RFC3261] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R.Sparks, M. Handley, E. Schooler, “SIP: Session Initiation Protocol,” Request for Comments: 3261, June 2002.

[SIP-event] A. Roach, “SIP-Specific Event Notification”, Internet Draft draft-ieft-sip-events-05.txt, Feb, 2002, work in progress.

[Miga98] A. Miga, “Application of Use Case Maps to System Design with Tool Support” M. Eng. Thesis, Carleton University, 1998.

[Miga01] A. Miga, D. Amyot, F. Bordeleau, C. Cameron, M. Woodside,“Deriving Message Sequence Charts from Use Case Maps Scenario Specifications”. Tenth SDL Forum (SDL'01), Copenhagen, Denmark, June 2001.

[Petriu01] D.B. Petriu, “Layered Software Performance Models Constructed from Use Case Map Specifications” M Eng. Thesis, Carleton University, 2001.

[Petriu02] D.C. Petriu, H. Shen, "Applying the UML Performance Profile: Graph Grammar based derivation of LQN models from UML specifications", in Computer Performance Evaluation - Modelling Techniques and Tools, Lecture Notes in Computer Science 2324, Springer Verlag, 2002, pp.159-177.

[RFC2778] M. Day, J. Rosenberg, and H. Sugano, “A model for presence and instant messaging,” Request for Comments 2778, Internet Engineering Task Force, February 2000.

[RFC2779] M. Day, S. Aggarwal, J. Vincent, “ Instant Messaging/ Presence Protocol Requirements,” Request for Comments 2779, Internet Engineering Task Force, February 2000.

[Rolia95] J.A. Rolia, K.C. Sevcik,“The Method of Layers”, IEEE Transactions on Software Engineering, Volume: 21 Issue: 8, August 1995, pp. 689 –700.

[Siddiqui01] K.H. Siddiqui, "Time/Performance Budgeting for Software Designs", M.Eng. Thesis, Carleton University, 2001.

[Smith90] C.U. Smith, “Performance Engineering of Software Systems”, Reading, MA, Addison Wesley, 1990.

[Smith00] C.U. Smith, L.G. Willams, “Building Responsive and Scalable Web Applications”, Proceedings Computer Measurement Group Conference, December 2000.

[Smith99] C.U. Smith, M. Woodside, “Performance Validation at Early Stages of Software Development”, in E. Gelenbe ed. System Performance Evaluation: Methodologies and Applications, CRC Press, 1999.

[Spex97] A. Hubbard, “Spex Software Performance Experiment Driver, Version 1”, Real-time and Distributed Systems Group, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada.

[UCMWeb] Use Case Maps Web Page, UCM User Group, and Virtual Library, http://www.UseCaseMaps.org.

[Woodside95] J.E. Neilson, C.M. Woodside, D.C. Petriu and S. Majumdar, "Software Bottlenecking in Client-Server Systems and Rendez-vous Networks", IEEE Trans. On Software Engineering, Vol. 21, No. 9, September 1995, pp. 776-782.

[Woodside02] D. Petriu, M.Woodside, "Software Performance Models from System Scenarios in Use Case Maps", Proceedings 12th International Conference on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation, London, April 2002.

[Wyckoff98] P.Wyckoff, “ T Spaces”, IBM Systems Journal, Volume 37, No. 3, pp. 454-474, 1998.

Appendix A L1.xlqn

This is the L1 LQN model, which is executed by the controller SPEX. The LQN model gives performance results for the test L1-1 described in section 4.3.4.1.

```
# File name: L1.xlqn
# Author: H. Liu

# Add experiment control values
$nusers =21:1000, 100
$loop = 20
$thinktime=600000

#execution control
$solver = parasrvn -B10,10000000

G
""
1e-05
50
5
0.9
-1

P 0
p P_Infinite f i R 1.000000
p UserP f i %u $UserPUtn R 1.000000
p PresenceSP f %u $PServerPUtn R 1.000000
-1

T 0
t RefTask1 r RefTask1_E1 -1  P_Infinite m $nusers %f $Thru
t Client f Client_RefE -1  UserP i
t PresenceServer f PresenceServer_E1 -1  PresenceSP
t PseudoPS f PseudoPS_E1 -1  PresenceSP
t Clients f Clients_E1 -1  UserP i
t PseudoPS_C1 f PseudoPS_C1_E1 -1  PresenceSP
-1

E 0
#RefTask's entry
s RefTask1_E1 0 0.00001 0 -1 %s2 $ResponseTime
Z RefTask1_E1 0 $thinktime 0 -1
y RefTask1_E1 Client_RefE 0 1.000000 0 -1

A Client_RefE Client_RefE_2
A PresenceServer_E1 PresenceServer_E1_4
A PseudoPS_E1 PseudoPS_E1_8
A Clients_E1 Clients_E1_11
A PseudoPS_C1_E1 PseudoPS_C1_E1_9
```

```
-1

A Client
f Client_RefE_2 1
s Client_RefE_2 0.000000
f SendMsg_3 1
s SendMsg_3 0.000000
y SendMsg_3 PresenceServer_E1 1.000000
f ReceiveReply_20 1
s ReceiveReply_20 0.000000
f reply_21 1
s reply_21 0.000000
:
Client_RefE_2 -> SendMsg_3;
SendMsg_3 -> ReceiveReply_20;
ReceiveReply_20 -> reply_21;
reply_21[Client_RefE]
-1

A PresenceServer
f PresenceServer_E1_4 1
s PresenceServer_E1_4 0.000000
f processPI_h13 1
s processPI_h13 10.000000
f AndFork_5 1
s AndFork_5 0.000000
f AndForkBranch_6 1
s AndForkBranch_6 0.000000
f SendMsg_7 1
s SendMsg_7 0.000000
y SendMsg_7 PseudoPS_E1 1.000000
f ReceiveReply_16 1
s ReceiveReply_16 0.000000
f end_17 1
s end_17 0.000000
f AndForkBranch_18 1
s AndForkBranch_18 0.000000
f SendMsg_19 1
s SendMsg_19 0.000000
:
PresenceServer_E1_4 -> processPI_h13;
processPI_h13 -> AndFork_5;
AndFork_5 -> AndForkBranch_6 & AndForkBranch_18;
AndForkBranch_6 -> SendMsg_7;
SendMsg_7 -> ReceiveReply_16;
ReceiveReply_16 -> end_17;
AndForkBranch_18 -> SendMsg_19;
SendMsg_19[ PresenceServer_E1 ]
-1

A PseudoPS
f PseudoPS_E1_8 1
s PseudoPS_E1_8 0.000000
f get_subscribers_h8 1
s get_subscribers_h8 10.000000
f LoopHead_h5 0
s LoopHead_h5 0.000000
y LoopHead_h5 PseudoPS_C1_E1 $loop
```

```
f SendMsg_15 1
s SendMsg_15 0.000000
:
PseudoPS_E1_8 -> get_subscribers_h8;
get_subscribers_h8 -> LoopHead_h5;
LoopHead_h5 -> SendMsg_15;
SendMsg_15[ PseudoPS_E1 ]
-1

A Clients
f Clients_E1_11 1
s Clients_E1_11 0.000000
f notified_h12 1
s notified_h12 10.000000
f SendMsg_12 1
s SendMsg_12 0.000000
:
Clients_E1_11 -> notified_h12;
notified_h12 -> SendMsg_12;
SendMsg_12[ Clients_E1 ]
-1

A PseudoPS_C1
f PseudoPS_C1_E1_9 1
s PseudoPS_C1_E1_9 0.000000
f notify_h22 1
s notify_h22 10.000000
f SendMsg_10 1
s SendMsg_10 0.000000
y SendMsg_10 Clients_E1 1.000000
f ReceiveReply_13 1
s ReceiveReply_13 0.000000
f DoneLoop_14 1
s DoneLoop_14 0.000000
:
PseudoPS_C1_E1_9 -> notify_h22;
notify_h22 -> SendMsg_10;
SendMsg_10 -> ReceiveReply_13;
ReceiveReply_13 -> DoneLoop_14;
DoneLoop_14[ PseudoPS_C1_E1 ]
-1

R 0
$0=$nusers
$Thru
$ResponseTime
$PServerPUtn
$UserPUtn
-1
```

Appendix B L2.xlqn

This is the L2 LQN model, which is executed by the controller SPEX. The LQN model gives performance results for the test L2-5 described in section 4.4.4.2.

```
# File name: L2.xlqn
# Author: H. Liu

# Add experiment control values
# Communication and PresenceServer work separately
# loop = 0.1 *users

$nusers =21:1000, 100
$loop=($nusers-1)*0.1
$thinktime=600000
$solver = parasrvn -B10,10000000

G
""
1e-05
50
5
0.9
-1

P 0
p P_Infinite f i R 1.000000
p PresenceSP f %u $PSPUtn
p UserP f i %u $UserPUtn
p CommP f i %u $CommPUtn
-1

T 0
t RefTask1 r RefTask1_RefE -1 P_Infinite m $nusers %f $Thru
t Client f Client_RefE -1  UserP i
t Communication f Communication_E1 -1  CommP i
t PresenceServer f PresenceServer_E1 -1  PresenceSP
t PresenceServer_Loop1 f PresenceServer_Loop1_E1 -1  PresenceSP
-1

E 0
#RefTask's entry
s RefTask1_RefE 0 0.00001 0 -1 %s2 $ResponseTime
Z RefTask1_RefE 0 $thinktime 0 -1
y RefTask1_RefE Client_RefE 0 1.000000 0 -1


A Client_RefE Client_A1
A Communication_E1 communicate_h15
A PresenceServer_E1 processPI_h14
A PresenceServer_Loop1_E1 notify_h8
```

```
-1

A Client
f Client_A1 1
s Client_A1 0.0
f Client_A1_SyncCall 1
s Client_A1_SyncCall 0.000000
y Client_A1_SyncCall Communication_E1 1.000000
f Client_A2_ReceiveReply 1
s Client_A2_ReceiveReply 0.000000
f Client_A3_end 1
s Client_A3_end 0.000000
:
Client_A1 -> Client_A1_SyncCall;
Client_A1_SyncCall -> Client_A2_ReceiveReply;
Client_A2_ReceiveReply -> Client_A3_end;
Client_A3_end[Client_RefE]
-1

A Communication
f communicate_h15 1
s communicate_h15 100.000000
f Communication_A1_SyncCall 1
s Communication_A1_SyncCall 0.000000
y Communication_A1_SyncCall PresenceServer_E1 1.000000
f Communication_A2_ReceiveReply 1
s Communication_A2_ReceiveReply 0.000000
f Communication_A3_SendReply 1
s Communication_A3_SendReply 0.000000
:
communicate_h15 -> Communication_A1_SyncCall;
Communication_A1_SyncCall -> Communication_A2_ReceiveReply;
Communication_A2_ReceiveReply -> Communication_A3_SendReply;
Communication_A3_SendReply[ Communication_E1 ]
-1

A PresenceServer
f processPI_h14 1
s processPI_h14 10.000000
f getList_h13 1
s getList_h13 10.000000
f Loop_h5_SyncCall 0
s Loop_h5_SyncCall 0.000000
y Loop_h5_SyncCall PresenceServer_Loop1_E1 $loop
f PresenceServer_A1_SendReply 1
s PresenceServer_A1_SendReply 0.000000
:
processPI_h14 -> getList_h13;
getList_h13 -> Loop_h5_SyncCall;
Loop_h5_SyncCall -> PresenceServer_A1_SendReply;
PresenceServer_A1_SendReply[ PresenceServer_E1 ]
-1

A PresenceServer_Loop1
f notify_h8 1
s notify_h8 10.000000
f PresenceServer_Loop1_A1_SendReply 1
s PresenceServer_Loop1_A1_SendReply 0.000000
```

```
:
notify_h8 -> PresenceServer_Loop1_A1_SendReply;
PresenceServer_Loop1_A1_SendReply[ PresenceServer_Loop1_E1 ]
-1


R 0
$0=$nusers
$Thru
$ResponseTime
$PSPUtn
$CommPUtn
$UserPUtn
-1
```

Appendix C L3.xlqn

This is the L3 LQN model, which is executed by the controller SPEX. The LQN model gives performance results for the test L3-1 described in section 5.4.1.

```
# File name: L3.xlqn
# Author: H. Liu

# Add experiment control values
# PUA and PA at PresenceSP
# loop = 0.1 *users

$nusers =21:1000, 100
$loop=($nusers-1)*0.1
$thinktime=600000


G
""
1e-05
50
5
0.9
-1

P 0
p P_Infinite f i R 1.000000
p UserP f i %u $UserPUtn
p PresenceSP f %u $PSPUtn
p CommP f i %u $CommPUtn
-1

T 0
t RefTask1 r RefTask1_E1 -1 UserP m $nusers %f $Thru
t Client f Client_RefE -1  UserP i
t Communication f Communication_E1 -1  CommP i
t Watcher f Watcher_E1 -1  UserP i
t PresenceAgent f PresenceAgent_E1 -1  PresenceSP
t PUA f PUA_E1 -1  PresenceSP i
t PresenceAgent_C1 f PresenceAgent_C1_E1 -1  PresenceSP
t DefTask1 f DefTask1_E1 -1  P_Infinite
-1

E 0
#RefTask's entry
s RefTask1_E1 0 0.00001 0 -1 %s2 $ResponseTime
Z RefTask1_E1 0 $thinktime 0 -1
y RefTask1_E1 Client_RefE 0 1.000000 0 -1


A Client_RefE Client_RefE_2
```

```
A Communication_E1 Communication_E1_4
A Watcher_E1 Watcher_E1_13
A PresenceAgent_E1 PresenceAgent_E1_10 %s $PATime
A PUA_E1 PUA_E1_6
A PresenceAgent_C1_E1 PresenceAgent_C1_E1_11
A DefTask1_E1 DefTask1_E1_18
-1

A Client
f Client_RefE_2 1
s Client_RefE_2 0.000000
f SendMsg_3 1
s SendMsg_3 0.000000
y SendMsg_3 Communication_E1 1.000000
f ReceiveReply_24 1
s ReceiveReply_24 0.000000
f end_25 1
s end_25 0.000000
:
Client_RefE_2 -> SendMsg_3;
SendMsg_3 -> ReceiveReply_24;
ReceiveReply_24 -> end_25;
end_25[Client_RefE]
-1

A Communication
f Communication_E1_4 1
s Communication_E1_4 0.000000
f communicate_h23 1
s communicate_h23 10.000000
f SendMsg_5 1
s SendMsg_5 0.000000
y SendMsg_5 PUA_E1 1.000000
f ReceiveReply_22 1
s ReceiveReply_22 0.000000
f SendMsg_23 1
s SendMsg_23 0.000000
:
Communication_E1_4 -> communicate_h23;
communicate_h23 -> SendMsg_5;
SendMsg_5 -> ReceiveReply_22;
ReceiveReply_22 -> SendMsg_23;
SendMsg_23[ Communication_E1 ]
-1

A Watcher
f Watcher_E1_13 1
s Watcher_E1_13 0.000000
f notified_h9 1
s notified_h9 10.000000
f SendMsg_14 1
s SendMsg_14 0.000000
:
Watcher_E1_13 -> notified_h9;
notified_h9 -> SendMsg_14;
SendMsg_14[ Watcher_E1 ]
-1
```

```
A PresenceAgent
f PresenceAgent_E1_10 1
s PresenceAgent_E1_10 0.000000
f getList_h22 1
s getList_h22 10.000000
f Loop_h2 0
s Loop_h2 0.000000
y Loop_h2 PresenceAgent_C1_E1 $loop
f SendMsg_17 1
s SendMsg_17 0.000000
:
PresenceAgent_E1_10 -> getList_h22;
getList_h22 -> Loop_h2;
Loop_h2 -> SendMsg_17;
SendMsg_17[PresenceAgent_E1]
-1

A PUA
f PUA_E1_6 1
s PUA_E1_6 0.000000
f processPI_h8 1
s processPI_h8 10.000000
f AndFork_7 1
s AndFork_7 0.000000
f AndForkBranch_8 1
s AndForkBranch_8 0.000000
f SendMsg_9 1
s SendMsg_9 0.000000
z SendMsg_9 DefTask1_E1 1.0
f AndForkBranch_20 1
s AndForkBranch_20 0.000000
f SendMsg_21 1
s SendMsg_21 0.000000
:
PUA_E1_6 -> processPI_h8;
processPI_h8 -> AndFork_7;
AndFork_7 -> AndForkBranch_8 & AndForkBranch_20;
AndForkBranch_8 -> SendMsg_9;
AndForkBranch_20 -> SendMsg_21;
SendMsg_21[ PUA_E1 ]
-1

A PresenceAgent_C1
f PresenceAgent_C1_E1_11 1
s PresenceAgent_C1_E1_11 0.000000
f notify_h10 1
s notify_h10 10.000000
f SendMsg_12 1
s SendMsg_12 0.000000
y SendMsg_12 Watcher_E1 1.000000
f ReceiveReply_15 1
s ReceiveReply_15 0.000000
f DoneLoop_16 1
s DoneLoop_16 0.000000
:
PresenceAgent_C1_E1_11 -> notify_h10;
notify_h10 -> SendMsg_12;
SendMsg_12 -> ReceiveReply_15;
```

```
ReceiveReply_15 -> DoneLoop_16;
DoneLoop_16[ PresenceAgent_C1_E1 ]
-1

A DefTask1
f DefTask1_E1_18 1
s DefTask1_E1_18 0.000000
y DefTask1_E1_18 PresenceAgent_E1 1.0
f end1_19 1
s end1_19 0.000000
:
DefTask1_E1_18 -> end1_19
-1

R 0
$0=$nusers
$Thru
$ResponseTime
$PSPUtn
$CommPUtn
$UserPUtn
$PATime
-1
```

Appendix D L4-SIP.xlqn

This is the L4-SIP LQN model, which is executed by the controller SPEX. The LQN model gives performance results for the test L4sip-1 described in section 6.3.1.

```
# File name: L4-SIP.xlqn
# Author: H. Liu

# Add experiment control values

# PUA and PA at PresenceSP
# Proxy and Redirect at Proxy Processor
# loop = 0.1 *users

$nusers =21:1000, 100
$loop=($nusers-1)*0.1
$thinktime=600000

G
""
1e-05
50
5
0.9
-1

P 0
p UserP f i %u $UserPUtn
p PresenceSP f %u $PSPUtn
p CommP f  %u $CommPUtn
p ProxyP f %u $ProxyPUtn
-1

T 0
t RefTask1 r RefTask1_RefE -1  UserP m $nusers %f $Thru
t Client f Client_RefE -1  UserP i
t Communication f Communication_E1 -1  CommP i
t PUA f PUA_E1 -1  PresenceSP i
t PresenceAgent f PresenceAgent_E1 -1  PresenceSP %u $PtaskU
t Proxy f Proxy_E1 -1  ProxyP
t Redirect f Redirect_E1 -1  ProxyP
t Watchers f Watchers_E1 -1  UserP
t PresenceAgent_Loop1 f PresenceAgent_Loop1_E1 -1  PresenceSP
-1

E 0
#RefTask's entry
s RefTask1_RefE 0 0.00001 0 -1 %s2 $ResponseTime
Z RefTask1_RefE 0 $thinktime 0 -1
y RefTask1_RefE Client_RefE 0 1.000000 0 -1
A Client_RefE Client_A1_SyncCall
```

```
A Communication_E1 communicate_h29
A PUA_E1 processPI_h25
A PresenceAgent_E1 getList_h28  %s $PATime
A Proxy_E1 proxy_h9
A Redirect_E1 redirect_h45
A Watchers_E1 notified_h13
A PresenceAgent_Loop1_E1 notify_h23
-1

A Client
f Client_A1_SyncCall 1
s Client_A1_SyncCall 0.000000
y Client_A1_SyncCall Communication_E1 1.000000
f Client_A2_ReceiveReply 1
s Client_A2_ReceiveReply 0.000000
f Client_A3_end 1
s Client_A3_end 0.000000
:
Client_A1_SyncCall -> Client_A2_ReceiveReply;
Client_A2_ReceiveReply -> Client_A3_end;
Client_A3_end[Client_RefE]
-1

A Communication
f communicate_h29 1
s communicate_h29 10.000000
f Communication_A1_SyncCall 1
s Communication_A1_SyncCall 0.000000
y Communication_A1_SyncCall PUA_E1 1.000000
f Communication_A2_ReceiveReply 1
s Communication_A2_ReceiveReply 0.000000
f Communication_A3_SendReply 1
s Communication_A3_SendReply 0.000000
:
communicate_h29 -> Communication_A1_SyncCall;
Communication_A1_SyncCall -> Communication_A2_ReceiveReply;
Communication_A2_ReceiveReply -> Communication_A3_SendReply;
Communication_A3_SendReply[ Communication_E1 ]
-1

A PUA
f processPI_h25 1
s processPI_h25 10.000000
f PUA_A1_SyncCall 1
s PUA_A1_SyncCall 0.000000
y PUA_A1_SyncCall PresenceAgent_E1 1.000000
f PUA_A2_ReceiveReply 1
s PUA_A2_ReceiveReply 0.000000
f PUA_A3_SendReply 1
s PUA_A3_SendReply 0.000000
:
processPI_h25 -> PUA_A1_SyncCall;
PUA_A1_SyncCall -> PUA_A2_ReceiveReply;
PUA_A2_ReceiveReply -> PUA_A3_SendReply;
PUA_A3_SendReply[ PUA_E1 ]
-1

A PresenceAgent
```

```
f getList_h28 1
s getList_h28 10.000000
f Loop_h4_SyncCall 0
s Loop_h4_SyncCall 0.000000
y Loop_h4_SyncCall PresenceAgent_Loop1_E1 $loop
f PresenceAgent_A1_SendReply 1
s PresenceAgent_A1_SendReply 0.000000
:
getList_h28 -> Loop_h4_SyncCall;
Loop_h4_SyncCall -> PresenceAgent_A1_SendReply;
PresenceAgent_A1_SendReply[ PresenceAgent_E1 ]
-1

A Proxy
f proxy_h9 1
s proxy_h9 2.000000
f Proxy_A1_SyncCall 1
s Proxy_A1_SyncCall 0.000000
y Proxy_A1_SyncCall Redirect_E1 1.000000
f Proxy_A2_ReceiveReply 1
s Proxy_A2_ReceiveReply 0.000000
f forward_h47 1
s forward_h47 2.000000
f Proxy_A3_SyncCall 1
s Proxy_A3_SyncCall 0.000000
y Proxy_A3_SyncCall Watchers_E1 1.000000
f Proxy_A4_ReceiveReply 1
s Proxy_A4_ReceiveReply 0.000000
f Proxy_A5_SendReply 1
s Proxy_A5_SendReply 0.000000
:
proxy_h9 -> Proxy_A1_SyncCall;
Proxy_A1_SyncCall -> Proxy_A2_ReceiveReply;
Proxy_A2_ReceiveReply -> forward_h47;
forward_h47 -> Proxy_A3_SyncCall;
Proxy_A3_SyncCall -> Proxy_A4_ReceiveReply;
Proxy_A4_ReceiveReply -> Proxy_A5_SendReply;
Proxy_A5_SendReply[ Proxy_E1 ]
-1

A Redirect
f redirect_h45 1
s redirect_h45 2.000000
f Redirect_A1_SendReply 1
s Redirect_A1_SendReply 0.000000
:
redirect_h45 -> Redirect_A1_SendReply;
Redirect_A1_SendReply[ Redirect_E1 ]
-1

A Watchers
f notified_h13 1
s notified_h13 2.000000
f Watchers_A1_SendReply 1
s Watchers_A1_SendReply 0.000000
:
notified_h13 -> Watchers_A1_SendReply;
Watchers_A1_SendReply[ Watchers_E1 ]
```

```
-1

A PresenceAgent_Loop1
f notify_h23 1
s notify_h23 2.000000
f PresenceAgent_Loop1_A1_SyncCall 1
s PresenceAgent_Loop1_A1_SyncCall 0.000000
y PresenceAgent_Loop1_A1_SyncCall Proxy_E1 1.000000
f PresenceAgent_Loop1_A2_ReceiveReply 1
s PresenceAgent_Loop1_A2_ReceiveReply 0.000000
f PresenceAgent_Loop1_A3_SendReply 1
s PresenceAgent_Loop1_A3_SendReply 0.000000
:
notify_h23 -> PresenceAgent_Loop1_A1_SyncCall;
PresenceAgent_Loop1_A1_SyncCall ->
PresenceAgent_Loop1_A2_ReceiveReply;
PresenceAgent_Loop1_A2_ReceiveReply ->
PresenceAgent_Loop1_A3_SendReply;
PresenceAgent_Loop1_A3_SendReply[ PresenceAgent_Loop1_E1 ]
-1

R 0
$0=$nusers
$Thru
$ResponseTime -600000
$PSPUtn
$PtaskU
$ProxyPUtn
$CommPUtn
$UserPUtn
$PATime
-1
```

Appendix E L4-TS.xlqn

This is the L4-TS LQN model, which is executed by the controller SPEX. The LQN model gives performance results for the test L4ts-4 described in section 7.3.2.

```
# File name: L4-TS.xlqn
# Author: H. Liu

# Add experiment control values
# PUA and PA at PresenceSP
# EManger, EListener at PresenceSP
# Tuplespace at TuplespaceP
# loop = 0.1 *users

$nusers =21:1000, 100
$loop=($nusers-1)*0.1
$thinktime=600000
$solver = parasrvn -B10,10000000

G
""
1e-05
50
5
0.9
-1

P 0

p UserP f i %u $UserPUtn
p TuplespaceP f %u $TSPUtn
p PresenceSP f %u $PSPUtn
p CommP f i %u $CommPUtn
-1

T 0
t RefTask1 r RefTask1_RefE -1  UserP m $nusers %f $Thru
t Client f Client_RefE -1  UserP i
t Communication f Communication_E1 -1  CommP i
t Watchers f Watchers_E1 -1  UserP i
t PUA f PUA_E1 -1  PresenceSP i
t PresenceAgent f PresenceAgent_E1 -1  PresenceSP
t Tuplespace f TSpace_E1 -1  TuplespaceP
t EManager f EManager_E1 -1  PresenceSP
t EListener f EListener_E1 -1  PresenceSP
t PresenceAgent_Loop1 f PresenceAgent_Loop1_E1 -1  PresenceSP
-1

E 0
#RefTask's entry
s RefTask1_RefE 0 0.00001 0 -1 %s2 $ResponseTime
```

```
Z RefTask1_RefE 0 $thinktime 0 -1
y RefTask1_RefE Client_RefE 0 1.000000 0 -1

A Client_RefE Client_A1_SyncCall
A Communication_E1 communicate_h32
A Watchers_E1 notified_h22
A PUA_E1 processPI_h6
A PresenceAgent_E1 getList_h31 %s $PATime
A TSpace_E1 callback_h24
F TSpace_E1 EListener_E1 1.0 -1
A EManager_E1 postE_h29
F EManager_E1 TSpace_E1 1.0 -1
A EListener_E1 handleE_h25
A PresenceAgent_Loop1_E1 notify_h26
-1

A Client
f Client_A1_SyncCall 1
s Client_A1_SyncCall 0.000000
y Client_A1_SyncCall Communication_E1 1.000000
f Client_A2_ReceiveReply 1
s Client_A2_ReceiveReply 0.000000
f Client_A3_end 1
s Client_A3_end 0.000000
:
Client_A1_SyncCall -> Client_A2_ReceiveReply;
Client_A2_ReceiveReply -> Client_A3_end;
Client_A3_end[Client_RefE]
-1

A Communication
f communicate_h32 1
s communicate_h32 10.000000
f Communication_A1_SyncCall 1
s Communication_A1_SyncCall 0.000000
y Communication_A1_SyncCall PUA_E1 1.000000
f Communication_A2_ReceiveReply 1
s Communication_A2_ReceiveReply 0.000000
f Communication_A3_SendReply 1
s Communication_A3_SendReply 0.000000
:
communicate_h32 -> Communication_A1_SyncCall;
Communication_A1_SyncCall -> Communication_A2_ReceiveReply;
Communication_A2_ReceiveReply -> Communication_A3_SendReply;
Communication_A3_SendReply[ Communication_E1 ]
-1

A Watchers
f notified_h22 1
s notified_h22 2.000000
f Watchers_A1_SendReply 1
s Watchers_A1_SendReply 0.000000
:
notified_h22 -> Watchers_A1_SendReply;
Watchers_A1_SendReply[ Watchers_E1 ]
-1

A PUA
```

```
f processPI_h6 1
s processPI_h6 10.000000
f PUA_A1_SyncCall 1
s PUA_A1_SyncCall 0.000000
y PUA_A1_SyncCall PresenceAgent_E1 1.000000
f PUA_A2_ReceiveReply 1
s PUA_A2_ReceiveReply 0.000000
f PUA_A3_SendReply 1
s PUA_A3_SendReply 0.000000
:
processPI_h6 -> PUA_A1_SyncCall;
PUA_A1_SyncCall -> PUA_A2_ReceiveReply;
PUA_A2_ReceiveReply -> PUA_A3_SendReply;
PUA_A3_SendReply[ PUA_E1 ]
-1

A PresenceAgent
f getList_h31 1
s getList_h31 10.000000
f Loop_h15_SyncCall 0
s Loop_h15_SyncCall 0.000000
y Loop_h15_SyncCall PresenceAgent_Loop1_E1 $loop
f PresenceAgent_A1_SendReply 1
s PresenceAgent_A1_SendReply 0.000000
:
getList_h31 -> Loop_h15_SyncCall;
Loop_h15_SyncCall -> PresenceAgent_A1_SendReply;
PresenceAgent_A1_SendReply[ PresenceAgent_E1 ]
-1

A Tuplespace
f callback_h24 1
s callback_h24 2.000000
f TSpace_A1_SendReply 1
s TSpace_A1_SendReply 0.000000
:
callback_h24 -> TSpace_A1_SendReply;
TSpace_A1_SendReply[ TSpace_E1 ]
-1

A EManager
f postE_h29 1
s postE_h29 2.000000
f EManager_A1_SendReply 1
s EManager_A1_SendReply 0.000000
:
postE_h29 -> EManager_A1_SendReply;
EManager_A1_SendReply[ EManager_E1 ]
-1

A EListener
f handleE_h25 1
s handleE_h25 2.000000
f EListener_A1_SyncCall 1
s EListener_A1_SyncCall 0.000000
y EListener_A1_SyncCall Watchers_E1 1.000000
f EListener_A2_ReceiveReply 1
s EListener_A2_ReceiveReply 0.000000
```

```
f EListener_A3_SendReply 1
s EListener_A3_SendReply 0.000000
:
handleE_h25 -> EListener_A1_SyncCall;
EListener_A1_SyncCall -> EListener_A2_ReceiveReply;
EListener_A2_ReceiveReply -> EListener_A3_SendReply;
EListener_A3_SendReply[ EListener_E1 ]
-1

A PresenceAgent_Loop1
f notify_h26 1
s notify_h26 2.000000
f PresenceAgent_Loop1_A1_SyncCall 1
s PresenceAgent_Loop1_A1_SyncCall 0.000000
y PresenceAgent_Loop1_A1_SyncCall EManager_E1 1.000000
f PresenceAgent_Loop1_A2_ReceiveReply 1
s PresenceAgent_Loop1_A2_ReceiveReply 0.000000
f PresenceAgent_Loop1_A3_SendReply 1
s PresenceAgent_Loop1_A3_SendReply 0.000000
:
notify_h26 -> PresenceAgent_Loop1_A1_SyncCall;
PresenceAgent_Loop1_A1_SyncCall ->
PresenceAgent_Loop1_A2_ReceiveReply;
PresenceAgent_Loop1_A2_ReceiveReply ->
PresenceAgent_Loop1_A3_SendReply;
PresenceAgent_Loop1_A3_SendReply[ PresenceAgent_Loop1_E1 ]
-1

R 0
$0=$nusers
$Thru
$ResponseTime
$PSPUtn
$TSPUtn
$CommPUtn
$UserPUtn
$PATime
-1
```

Appendix F L5.xlqn

This is the L5 LQN model, which is executed by the controller SPEX. The LQN model gives performance results for the test L5-1 described in section 8.5.2.

```
# File name: L5.xlqn
# Author: H. Liu

# Add experiment control values
# PUA and PA at PresenceSP
# AvailM, AvailL, ClientProxy, RM, WUA at PresenceSP
# EManger, EListener at PresenceSP
# Tuplespace at TuplespaceP
# Demands using prototype data
# loop = 0.1 *users

$nusers =21:1000, 100
$loop=($nusers-1)*0.1
$thinktime=600000

G
""
1e-05
50
5
0.8
-1

P 0

p UserP f i %u $UserPUtn
p TuplespaceP f %u $TSPUtn
p PresenceSP f %u $PSPUtn
p CommP f i %u $CommPUtn
-1

T 0
t RefTask1 r RefTask1_RefE -1  UserP m $nusers %f $Thru
t ClientProxy f ClientProxy_E1 -1  PresenceSP i
t AvailEManager f AvailEManager_E1 -1  PresenceSP
t WUA f WUA_E1 -1  PresenceSP i
t ResourceManager f ResourceManager_E1 -1  PresenceSP
t PresenceAgent f PresenceAgent_E1 -1  PresenceSP  %u $PSTUtn
t PresenceAgent_Loop1 f PresenceAgent_Loop1_E1 -1 PresenceSP
t AvailEListener f AvailEListener_E1 -1  PresenceSP
t HttpReqHandler f HttpReqHandler_E1 -1  CommP i
t PUA f PUA_E1 -1  PresenceSP i
t EsManager f EsManager_E1 -1  PresenceSP
t EsListener f EsListener_E1 -1  PresenceSP
t Communication f Communication_E1 -1  CommP i
t Tuplespace f callbackEs callbackE callbackNt -1 TuplespaceP
```

```
t Client f Client_RefE -1  UserP i
t Watcher f Watcher_E1 -1  UserP i
-1

E 0
#RefTask's entry
s RefTask1_RefE 0 0.00001 0 -1 %s2 $ResponseTime
Z RefTask1_RefE 0 $thinktime 0 -1
y RefTask1_RefE Client_RefE 0 1.000000 0 -1

# define demand for Tuplespace entries
s callbackEs 43.0 0 0 -1
s callbackNt 3.0 0 0 -1
s callbackE 4.0 0 0 –1

A ClientProxy_E1 handleNt_h49
A AvailEManager_E1 postE_h44
A WUA_E1 receiveNt_h37
A ResourceManager_E1 postNt_h48
A PresenceAgent_E1 getList_h21 %s $PATime
A Watcher_E1 notified_h38
A AvailEListener_E1 handleE_h45
A Client_RefE Client_A1_SyncCall
A Communication_E1 communicate_h22
A HttpReqHandler_E1 post_h64
A PUA_E1 processPI_h12
A PresenceAgent_Loop1_E1 PresenceAgent_Loop1_A1_start2
A EsManager_E1 postEs_h8
A EsListener_E1 handleEs_h9

F AvailEListener_E1 WUA_E1 1.0 -1
F EsManager_E1 callbackEs 1.0 -1
F callbackEs EsListener_E1 1.0 -1
F EsListener_E1 PUA_E1 1.0 -1
F PUA_E1 PresenceAgent_E1 1.0 -1
F AvailEManager_E1 callbackE 1.0 -1
F callbackE AvailEListener_E1 1.0 -1
F ResourceManager_E1 callbackNt 1.0 -1
F callbackNt ClientProxy_E1 1.0 -1
F ClientProxy_E1 Watcher_E1 1.0 -1

-1

A ClientProxy
f handleNt_h49 1
s handleNt_h49 4.000000
:
handleNt_h49[ClientProxy_E1]
-1

A AvailEManager
f postE_h44 1
s postE_h44 0.000000
:
postE_h44 [AvailEManager_E1]
-1

A WUA
```

```
f receiveNt_h37 1
s receiveNt_h37 0.000000
y receiveNt_h37 ResourceManager_E1 1.000000
f WUA_A1_SendReply 1
s WUA_A1_SendReply 0.000000
:
receiveNt_h37 -> WUA_A1_SendReply;
WUA_A1_SendReply[ WUA_E1 ]
-1

A ResourceManager
f postNt_h48 1
s postNt_h48 108.000000
f ResourceManager_A1_SendReply 1
s ResourceManager_A1_SendReply 0.000000
:
postNt_h48 -> ResourceManager_A1_SendReply;
ResourceManager_A1_SendReply[ ResourceManager_E1 ]
-1

A PresenceAgent
f getList_h21 1
s getList_h21 53.000000
f Loop_h4_SyncCall 0
s Loop_h4_SyncCall 0.000000
y Loop_h4_SyncCall PresenceAgent_Loop1_E1 $loop
f PresenceAgent_A1_SendReply 1
s PresenceAgent_A1_SendReply 0.000000
:
getList_h21 -> Loop_h4_SyncCall;
Loop_h4_SyncCall -> PresenceAgent_A1_SendReply;
PresenceAgent_A1_SendReply[ PresenceAgent_E1 ]
-1

A Watcher
f notified_h38 1
s notified_h38 0.000000
f Watcher_A1_end2 1
s Watcher_A1_end2 0.000000
:
notified_h38 -> Watcher_A1_end2;
Watcher_A1_end2[Watcher_E1]
-1

A AvailEListener
f handleE_h45 1
s handleE_h45 0.000000
f AvailEListener_A1_SendReply 1
s AvailEListener_A1_SendReply 0.000000
:
handleE_h45 -> AvailEListener_A1_SendReply;
AvailEListener_A1_SendReply[ AvailEListener_E1 ]
-1

A Client
f Client_A1_SyncCall 1
s Client_A1_SyncCall 0.000000
y Client_A1_SyncCall Communication_E1 1.000000
```

```
f Client_A2_ReceiveReply 1
s Client_A2_ReceiveReply 0.000000
f Client_A3_end 1
s Client_A3_end 0.000000
:
Client_A1_SyncCall -> Client_A2_ReceiveReply;
Client_A2_ReceiveReply -> Client_A3_end;
Client_A3_end[Client_RefE]
-1

A Communication
f communicate_h22 1
s communicate_h22 10.000000
f Communication_A1_SyncCall 1
s Communication_A1_SyncCall 0.000000
y Communication_A1_SyncCall HttpReqHandler_E1 1.000000
f Communication_A2_ReceiveReply 1
s Communication_A2_ReceiveReply 0.000000
f Communication_A3_SendReply 1
s Communication_A3_SendReply 0.000000
:
communicate_h22 -> Communication_A1_SyncCall;
Communication_A1_SyncCall -> Communication_A2_ReceiveReply;
Communication_A2_ReceiveReply -> Communication_A3_SendReply;
Communication_A3_SendReply[ Communication_E1 ]
-1

A HttpReqHandler
f post_h64 1
s post_h64 36.000000
f HttpReqHandler_A1_SyncCall 1
s HttpReqHandler_A1_SyncCall 0.000000
y HttpReqHandler_A1_SyncCall EsManager_E1 1.000000
f HttpReqHandler_A2_ReceiveReply 1
s HttpReqHandler_A2_ReceiveReply 0.000000
f HttpReqHandler_A3_SendReply 1
s HttpReqHandler_A3_SendReply 0.000000
:
post_h64 -> HttpReqHandler_A1_SyncCall;
HttpReqHandler_A1_SyncCall -> HttpReqHandler_A2_ReceiveReply;
HttpReqHandler_A2_ReceiveReply -> HttpReqHandler_A3_SendReply;
HttpReqHandler_A3_SendReply[ HttpReqHandler_E1 ]
-1

A EsManager
f postEs_h8 1
s postEs_h8 20.000000
f EsManager_A1_SendReply 1
s EsManager_A1_SendReply 0.000000
:
postEs_h8 -> EsManager_A1_SendReply;
EsManager_A1_SendReply[ EsManager_E1 ]
-1

A EsListener
f handleEs_h9 1
s handleEs_h9 0.000000
f EsListener_A1_SendReply 1
```

```
s EsListener_A1_SendReply 0.000000
:
handleEs_h9 -> EsListener_A1_SendReply;
EsListener_A1_SendReply[ EsListener_E1 ]
-1

A PUA
f processPI_h12 1
s processPI_h12 0.000000
:
processPI_h12[ PUA_E1 ]
-1


A PresenceAgent_Loop1
f PresenceAgent_Loop1_A1_start2 1
s PresenceAgent_Loop1_A1_start2 0.000000
y PresenceAgent_Loop1_A1_start2 AvailEManager_E1 1.0
:
PresenceAgent_Loop1_A1_start2[ PresenceAgent_Loop1_E1 ]
-1

R 0
$0=$nusers
$Thru
$ResponseTime-600000
$PSPUtn
$PSTUtn
$TSPUtn
$PATime
-1
```