Performance Model Extraction Using Kernel Event Tracing

by

Shieryn Tjandra

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Department of Systems and Computer Engineering

Faculty of Engineering

Carleton University

Ottawa, Ontario, Canada, K1S 5B6

© Copyright 2019, Shieryn Tjandra

Abstract

Models are used in performance analysis when the analyst needs to be able to predict the effect of system changes that go beyond what can be measured. The model can be obtained from a combination of system knowledge and experimentation. This thesis addresses an experimental approach to obtaining layered queueing network (LQN) models of distributed systems. It applies and extends an approach called SAME (Software Architecture and Model Extraction) which was developed to interpret application-level traces, to interpreting Kernel-level traces. Kernel-level traces have the benefit that application instrumentation is not required, and communication with attached devices can be modeled, but they lack application context information. The research shows that modeling from Kernel traces is feasible in systems which communicate via TCP messages, including Java remote procedure calls. This covers most web-based systems. Systems using middleware pose special problems. The combination of Kernel and application-level tracing was included in some experiments. Tools are described that adapt the Kernel traces to SAME, and that extract CPU demand parameter calibration information.

Acknowledgements

Study in Canada was not an easy decision for me. With all supports, I could finish my study.

I would first like to thank God for giving me a chance to fulfil my dream and give me strength whenever I feel exhausted. Also, thank God for listening to the prayers of my grand parents, my mother, the Tjandra family and me.

To my beloved mother who always trusts my decisions, has my back all the time, sacrifices yourself to be the best single-mother in the world. You made me what I became today. I know my love to you never been enough for what you have given to me but thank you.

To my mentor, Dr. C. Murray Woodside, thank you for helping me to solve my problems, for trusting me when I doubted myself. If it has not been you, I would never finish my study. You have been more than a great teacher for me. Thank you.

To my cousins, my best friends thank you for supporting me and standing up for me when people doubted me.

To Geneviève Bastien and Housseem Daoud from École Polytechnique de Montréal, thank you for the LTTng and Trace Compass one-week training, for helping me to figure out the concept of TCP algorithm.

And never forget to say thank you to colleagues in the RADS lab for all the encouragements, and to my Company colleagues who supported me while I was doing research. Thank you.

Last but not least, financial support provided by Mitacs Accelerate is greatly appreciated.

Table of Contents

Abstract.....	ii
Acknowledgements	iii
Table of Contents.....	iv
List of Tables	viii
List of Illustrations	ix
Chapter 1: Introduction	1
1.1 Motivation and Objectives.....	1
1.2 Thesis Contributions	2
1.3 Challenges and Solutions.....	2
1.4 Industrial Case Study	3
1.5 Thesis outline.....	3
Chapter 2: Background	4
2.1 Performance Engineering	4
2.2 Layered Queueing Network (LQN)	4
2.2.1 Types of Communication in LQN	7
2.2.2 Making a Model by Hand.....	7
2.3 Inter-Process Communication in Linux.....	7
2.4 Linux Kernel Tracing	8
2.4.1 LTTng Tracing Tools	9
2.4.2 LTTng User-Space Tracepoints.....	9
2.4.3 LTTng and Common Trace Format (CTF).....	10
2.4.4 LTTng Traces Viewer (Trace Compass).....	10
2.5 Perf Tools	10
2.6 Java Remote Method Invocation.....	11

2.7	State of The Art of Model Building	12
Chapter 3: SAME Tools	18	
3.1	SAME phase 1	18
3.2	SAME Phase 2 (ExtractModel).....	22
Chapter 4: LTTng Traces	25	
4.1	LTTng Kernel Tracepoints	25
4.2	Challenges of using LTTng Kernel	27
4.3	Dealing with LTTng Kernel challenges	28
Chapter 5: Filtering Tools.....	29	
5.1	Obtain the Application Name for each Process	29
5.2	Extraction of Messages Using TCP	30
5.3	Conversion of File Operations	33
5.4	Inferring CPU Times: CPUStat Algorithms	35
5.5	Add CPU Times	38
5.6	Combining Parts of a Message	39
5.7	Dealing with Non-Traced Components	40
Chapter 6: The Method.....	42	
6.1	Overall Workflow of the Method.....	42
6.2	Executing the Program	44
6.3	Filtering the Trace Data, and SAME Phase 1	49
6.4	Extracting the Model with SAME.....	52
Chapter 7: Client-Server Case Study.....	57	
7.1	Description.....	57
7.2	Model Expectation	57
7.3	The Collected Data.....	58

7.4	SAME Result	58
7.5	Lessons Learned.....	59
Chapter 8: Java File Transfer Case Study		61
8.1	Description.....	61
8.2	Model Expectation	62
8.3	Data cleaning	63
8.4	JFT experiments on transferring 10 KB file	63
8.4.1	LQN model using Kernel events.....	63
8.4.1.1	Concurrency Question	65
8.4.2	LQN model using Kernel events and Activities (UST).....	68
8.4.3	Comparison between LQN model with and without activities	71
8.5	Performance Prediction on Larger File Size.....	71
8.5.1	The Predicted LQN model.....	77
8.6	Lessons Learned.....	79
Chapter 9: Industrial Case Study		81
9.1	The Expected Model of the Telecommunication System.....	81
9.2	The Limitation of Kernel Tracing of the Upgrade Software	82
9.3	The Collected Data.....	83
9.4	Processing Steps to Obtain the LQN Model Structure	84
9.5	Analyzing Finite State Machine Changes.....	85
9.6	CPUStat traces	86
9.7	Model Structure Found for Upgrade	88
9.8	Lessons Learned.....	91
Chapter 10: Conclusions		93
10.1	Contributions	93

10.2	Summary.....	95
10.3	Future Works	95
Appendices.....		96
	Appendix A JFT Experiments on transferring 10 KB File.....	96
A.1	LTTng Kernel Traces	96
A.2	LTTng UST Traces	97
A.3	Perf result	98
A.4	CPUStat result	99
A.5	LQN model file using ‘mergeByCallSet’ and Kernel.....	101
A.6	LQN model file using ‘mergeByTask’ and Kernel	102
A.7	LQN model file using ‘mergeByCallSet’ with Kernel and Activities.....	103
	Appendix B LQN solved model on JFT transferring 100 KB File.....	105
	Appendix C JFT 10 KB experiment for concurrency question	117
	Appendix D Layered system LQN file	121
References		122

List of Tables

Table 1 Survey journal approaches list.....	15
Table 2 JFT – Service demands prediction (Time in seconds)	72
Table 3 JFT – Call frequencies prediction.....	75
Table 4 Prediction errors on 100 KB file size.....	79
Table 5 Industrial Case Study : State changes data and prediction.....	87

List of Illustrations

Illustration 1 LQN for a client-server system	6
Illustration 2 Client-Server Java RMI	12
Illustration 3 SAME input format	18
Illustration 4 SAME1 Synchronous Communication Pattern	19
Illustration 5 SAME1 Synchronous and Asynchronous Communication Pattern	20
Illustration 6 SAME1 View of Synchronous and Asynchronous Interactions	21
Illustration 7 ExtractModel Extraction based on Interaction Pattern following from Illustration 5	23
Illustration 8 SAME1 result – Activities and ExtractModel result	24
Illustration 9 LTTng traces – read()	26
Illustration 10 LTTng traces – write()	26
Illustration 11 LTTng traces – net_dev_queue()	27
Illustration 12 Perf tools-execsnoop command-line	30
Illustration 13 TCP Message Extraction Algorithm pseudo code	31
Illustration 14 TCP algorithm result	32
Illustration 15 FilteringTools – NTTrace command-line	32
Illustration 16 File operations algorithm flowchart	34
Illustration 17 File operations algorithm result	35
Illustration 18 FilteringTools – FSTrace command-line	35
Illustration 19 CPUStat tools - diagram	36
Illustration 20 CPUStat tools - example Output of CPUStat	37
Illustration 21 MsgSimplifier Pattern	39

Illustration 22 SAME phase 1 before MsgSimplifier applied.....	40
Illustration 23 SAME phase 1 after MsgSimplifier applied	40
Illustration 24 MsgSimplfier command-line for execution	40
Illustration 25 Expected model structure of Java tutorial case	42
Illustration 26 Model Building work flow with SAME	43
Illustration 27 LTTng tools configuration	44
Illustration 28 Kernel tracepoints configuration	45
Illustration 29 LTTng UST instrumentation and Java files compilation.....	47
Illustration 30 LTTng UST module installation and configuration	47
Illustration 31 UST(s) configuration	48
Illustration 32 LTTng add-context configuration.....	48
Illustration 33 CPUStat tools - execution	48
Illustration 34 LTTng command-line : start tracing	49
Illustration 35 LTTng command-line : stop tracing	49
Illustration 36 LTTng application level (UST) raw traces fragment.....	49
Illustration 37 LTTng traces fragment.....	51
Illustration 38 Perf tools result fragment for HostA.....	52
Illustration 39 Perf tools result fragment for HostB.....	52
Illustration 40 Filtering tools result of example program.....	52
Illustration 41 Interaction Trace from SAME Phase 1 for the tutorial case	53
Illustration 42 SAME Phase 1 result with CPU time added	54
Illustration 43 SAME result for the tutorial case	55
Illustration 44 Resource modification on LQN model for the tutorial case	56

Illustration 45 LCS model based on the software structure	57
Illustration 46 SAME result on LCS (Time in seconds) – Appendix D	59
Illustration 47 JFT Scenario	61
Illustration 48 JFT expected model	62
Illustration 49 JFT LQN model using Kernel events with ‘mergebyCallSet’ (Time in seconds) – Appendix A5	64
Illustration 50 JFT LQN model using Kernel events with ‘mergebyTask’ rule (Time in seconds) – Appendix A6	65
Illustration 51 Modeling based on thread id (Time in seconds) – Appendix C	66
Illustration 52 JFT LQN model with Activities (Time in seconds) – Appendix A7	69
Illustration 53 JFT – Service demands predictions on FileService entries	73
Illustration 54 JFT – Service demands predictions on ClientA and ClientB entries	73
Illustration 55 JFT – Call frequencies prediction towards FileService entries	76
Illustration 56 JFT predicted LQN model – service times (Time unit in seconds)	78
Illustration 57 JFT predicted LQN model – Throughputs and Utilizations	78
Illustration 58 Industrial Case Study: Expected Model	82
Illustration 59 Industrial Case Study : Activities	83
Illustration 60 Industrial Case Study : SAME phase 1 result first modification	84
Illustration 61 Industrial Case Study : SAME phase 1 result second modification	85
Illustration 62 Industrial Case Study : FSM events	85
Illustration 63 Industrial Case Study : CPUStat Traces	86
Illustration 64 Industrial Case Study : CPU time increments (left) and values (right)	87
Illustration 65 Industrial Case Study : FSM activities service demands	88

Illustration 66 Industrial Case Study: ExtractModel result.....	89
Illustration 67 Industrial Case Study : LQN solved model – elapsed times (Time in seconds)	90
Illustration 68 Industrial Case Study : LQN solved model – throughputs and utilization	91

Chapter 1: Introduction

1.1 Motivation and Objectives

Nowadays, organizations use distributed systems to manage their activities, and require adequate performance at reasonable cost. Distributed systems provide scalability and reliability for operations and services, but they raise complex performance engineering issues. Performance can be addressed by a combination of measurement and modeling, and this thesis considers the use of a performance model, and creation of the model from operating data. Performance models can identify the components that need to be improved.

LQN (Layered Queueing Network) [28] is an extension of queueing networks, developed for modeling distributed software, and SAME [13] (Software Architecture and Model Extraction) is a technique, with software tools, that extracts an LQN model from execution traces which show the behavior of a distributed system.

This research examines the feasibility of applying the SAME approach to Kernel traces. Experiments are conducted using Linux and execution traces collected by the LTTng (Linux Trace Toolkit: next generation) tracing tools [19], which are distributed with Linux. The purpose of this research is to apply the existing SAME application to practical model-building, with goal to discover and fill up any gaps that are found in process. It is applied to two kinds of software, communicating Java processes and an industrial real-time software system.

Filtering tools and CPUStat tools were developed as part of this research. The tools deal with any gaps to extract an LQN model using SAME and LTTng execution traces. The Filtering tools bridge a LTTng execution trace into SAME, and SAME returns the LQN model based on the traces. The CPUStat tools record the CPU time utilized by the processes.

The work of the research is motivated by the following concerns:

- Utilize the LTTng Linux Kernel tracing
- Integrate the existing SAME tools with LTTng traces
- Extract LQN models from an industrial distributed system
- Collect the CPU time and use it to calibrate the LQN model

1.2 Thesis Contributions

The contributions that were achieved in this research are as follows:

- An exploration of the suitability of Kernel traces in particular of LTTng for performance model building using SAME, which found some serious limitations.
- A workable approach to modeling a distributed system that communicates with TCP; including message identification, and aggregating multi-event messages.
- An approach to including interactions with system components that are not traced, such as the disks, or an external task.
- An approach to adding CPU demand parameter calibration to SAME, using a separate logger.
- Test of effectiveness case studies on Java systems in the lab, and an industrial telecommunication system.

1.3 Challenges and Solutions

Some of the challenges in this work were:

- Dealing with Java process names encapsulation by Java Virtual Machine (JVM). As the execution of Java applications is done by the JVM, the LTTng identifies the Java processes as 'java' instead of by their program names. To distinguish the Java processes, a tool named Perf tools is used to record the Java program names that are executed by the Java Virtual Machine.
- Dealing with features of the industrial system. For interaction between processes, for instance, there may be middleware. To overcome barrier of middleware, the LTTng user-space instrumentation was introduced.
- Gather the CPU time data for relevant processes. In Linux systems, the proc filesystem records the CPU time for active processes. However, the relevant processes are not always there as they might vanish if they are inactive. Without an appropriate tool, it is hard to capture the CPU time for relevant processes before it vanished. Thus, the CPUSat tool is introduced to solve the problem. As it can record the CPU time increments every one second.
- Adapt the data to SAME using LTTng traces. Using SAME requires a specific format. LTTng traces need to be filtered and converted to the SAME required format, in order to get the LQN model extracted from the LTTng traces. Therefore,

the Filtering tools are developed to bridge the LTTng traces into SAME to get an LQN model out of it. This required a deep understanding of the LTTng trace information.

1.4 Industrial Case Study

Telecommunications equipment should never face any downtime. However, the software must be upgraded regularly to provide new features, or product enhancements. A complete software upgrade normally requires the system to be rebooted to apply the changes. To avoid any system downtime, a ‘rolling upgrade’ is applied to the primary and a secondary resource that also provides for reliability. In this research, SAME is applied to the software upgrade process in order to show that it is feasible to extract an LQN architecture from LTTng traces that are collected from industrial real-time software.

1.5 Thesis outline

This thesis is composed of nine chapters and appendices. The brief description of the content each chapter is as follows:

Chapter 2, Background, discusses the background concepts and some previous studies.

Chapter 3, The pre-existing SAME tools.

Chapter 4, LTTng traces, introduces the LTTng tools along with the relevant Kernel tracepoints; the challenges of using Kernel traces and how to deal with it.

Chapter 5, Filtering tools, includes the algorithms of identifying TCP interactions, converting file operations, combining parts of a message and how to break the JVM encapsulation of Java process name.

Chapter 6, The overall method for extracting a model from a Kernel trace.

Chapter 7, Client-Server Case Study.

Chapter 8, Java File Transfer (JFT) Case Study, creates an LQN model and predicts performance for different file sizes.

Chapter 9, Industrial Case Study.

Chapter 10, Conclusions.

Chapter 2: Background

2.1 Performance Engineering

Performance Engineering is a collection of activities concerned with ensuring the performance requirements (such as throughput, latency or memory usage) of a software/hardware system are met. Within performance engineering, models are used to relate performance to the execution resources. With the performance model, the analyst can predict how good the system performance will be with particular software and resources. The book by Bondi [2] gives an overview of performance engineering.

Performance is described in terms of *metrics*. A metric is defined as a standard of measurement. The following are examples of metrics that are commonly used in evaluating the performance of a system, from [2] :

- The *response time* is defined as the difference between the time at which an activity is completed and the time at which it was initiated. The average response time is the arithmetic mean of the response times that have been observed.
- The *average device utilization* is the proportion of time that a device, such as a CPU, disk, or other I/O device, is busy. For this metric to be valid, the length of time during which the average was taken should be stated, because averages taken over long intervals tend to hide fluctuations, while those taken over short intervals tend to reveal them.
- The *average throughput* of a system is the rate at which the system completes units of a particular activity.

2.2 Layered Queueing Network (LQN)

Queueing Network performance models analyze how contention for resources (such as the CPU) affects system delays. LQN is an extension of Queueing Networks that compactly represents simultaneous resource possession and the tasks in layers that is a common structure of a distributed system. It has been applied to distributed service systems (e.g [21] [8]) and to real-time systems, especially for communications (e.g. the example in [8]).

The model concepts are defined in the paper [9] and the User Guide [7] and explained in the Tutorial [28] An LQN model consists of components called tasks (representing any software component), which have operations (or service classes) called

entries. Interactions among different entries are represented by “calls”, or requests for service. Finally, there are the host processors on which the tasks are deployed. Tasks and processors are modeled as servers with queues and a multiplicity. Notice that a device like a disk is modeled as having a task with entries that define the operations such as open, read, write, (and local caching) and a processor that represents the hardware that executes the operations. Network delay on a call may be modeled by a separate delay task/host for the network.

The parameters that define the model behaviour are the processor demands of the entries, and the number of calls between entries. Task priorities and processor speed ratios also affect the model behavior.

The system drivers are modeled as special “reference tasks” which make requests but do not receive any, and cycle endlessly. A reference task may represent a set of users, or a timer that triggers a response, or any workload source. A single response is modeled as if it is repeated endlessly. Separate classes of workload are modeled by separate reference tasks.

A task has a host processor, which models the physical entity which carries out the operations. This separates the logic of an operation from its physical execution. Thus, a disk is modeled by two entities, a disk device, and a disk task which represents the disk operations logic which includes the disk device – level caching logic and the difference between say a read and a write, including the logic of disk device-level caching. A processor has a queue and a discipline for executing its tasks, and a task may have a priority relative to other tasks on its host. Each host processor has a speed ratio relative to a standard processor. The execution demand of an operation is the time demand on the standard processor, and on other processors it is divided by the speed ratio.

A task has one or more entries which represent different operations it may perform. If the task is seen as a concurrent object, then the entries are considered as its methods (the name entry comes from the Ada language). Entries with different workload parameters are equivalent to separate classes of service in queueing. The host execution demand is the workload parameters of an entry, think time and its calls to other entries. The model does

not support the priorities between entries and in the same task, an entry may not call another entry. The basic model of entry has a single activity called phase 1, and some systems have a second activity after sending a reply. Entries with phase 1 and phase 2 can be represented by two activities, one performed before the reply and one after.

Interactions between entries (*Calls*) represent requests for service from an entry to an entry of another task. A call may be synchronous, asynchronous, or forwarding. This describes parallel and concurrent calls, expressed in a manner very close to that of a software architecture description language.

The demands of an operation are defined by the total average number of calls for service operations and average amounts of host processing which is required to complete an entry [7] [9] .

Illustration 1 LQN for a client-server system

The figure above illustrates the LQN structure for a client-server system. In this case, the tasks are the software servers which are Client, HTTPServ, and Application. The entries are their operations which are ClientEntry, webService, Browse and Buy. The processors are drawn as circles. The 'Client' task is the reference task since it does not receive any calls from other tasks. It runs on single processor named 'ClientHost' with the pool of 150 clients (m), and the gap between requests is 2 seconds which called as think time (Z). In addition, the annotation y stands for the average number of calls. The request

into ‘application’ task is divided between 90% of requests going to the ‘Browse’ entry, and 10% going to ‘Buy’ entry. The ‘application’ task has multiplicity 10 which represent the number of threads, and it runs on a two-core processor named ‘AppHost’.

2.2.1 Types of Communication in LQN

LQN describes the communication between tasks or entries. In this research, there are three types of communication which are asynchronous, synchronous and forwarding. The communication represents the calls between processes [7] [9] :

- Asynchronous is a request that sent to another task or process, and it does not wait for a reply to send the second request.
- Synchronous is a request that sent to another task or process, and it waits for a reply before it sends another request.
- Forwarding is an extension of asynchronous calls. It happens when a process sends an asynchronous request to a task, and that task forwards the request to another task then continue the operation of the request in the second phase. Another task received the request from the first task replies directly to the first task after some time.

2.2.2 Making a Model by Hand

To make a model by hand, first a set of scenarios is defined, which are translated into an LQN diagram, and based on the diagram, the construction of an LQN file is started. There are three ways to define the model by hand: an LQN language [7] which is rather compact, an XML language [7] which is verbose, and a menu-driven editor that guides the user and creates a graphical display, and outputs either of the language forms to save a model file. The file is input to the solver (lqns) or to a transformation tool (lqn2xml) which can switch between languages or output a graphic figure for publication. The solver outputs a readable result file and also a model file with embedded results, which can be displayed in the editor or used for further automated processing.

2.3 Inter-Process Communication in Linux

Linux applies different Inter-Process Communication (IPC) concepts: signals,

pipes, sockets, semaphores, and shared memory [14] .

- Signals are a simple type of IPC that can be used by processes to interact. There are several signals that have roles in Linux, such as SIGINT, SIGKILL, SIGSTOP, etc. The SIGSTOP signal is used to terminate the execution of a process, and SIGKILL is used to force-close a process while it is running.
- Pipes are a common interaction in the Linux Kernel between processes on the same host. The example of Linux process that uses pipes as its communication is the 'ls' command. The 'ls' command pipes its result into an input of the 'pr' command, then the result of the 'pr' command is piped into the 'lpr' command which prints the result on the Linux terminal. The shell commands set up temporary pipes between them.
- Sockets allow customizable interactions between two different processes on the same machine or different machines. Stream sockets and datagram sockets are the common types of sockets.
- Semaphores implement mutual exclusion and are used to block a process execution that interferes with other processes. It may for example be used when many processes read and write into one single data file. The semaphore can help with the coordination of file access.
- Shared memory allows processes to interact through a memory that appears in their virtual address space. Any process can write to the memory, and any process can read from it.

In this research, the experiment is focussed on stream sockets IPC which implements the TCP/IP protocol in Linux, to capture programmed messaging.

2.4 Linux Kernel Tracing

The Linux kernel is the core of Linux, that has responsibilities to manage the memory allocation, the resources, the scheduling, and the attached external devices such as keyboard, monitor, etc.

As Linux developed, the tracepoints have been instrumented in the Linux code from the beginning, using the trace recording tool that has evolved into the LTTng. The tracepoints are used to measure the performance of a system, and also capture the behavior of a system. Whenever processes are executed by Linux, the tracepoints correspond to the

processes can potentially be recorded by LTTng. A tracepoint is recorded when it is turned on at the LTTng configuration phase, however, when it is turned off, the tracepoint will not be recorded [3] .

To record the execution result of each tracepoint, tracing recorder tools are needed to record all the execution result [3] . For example, if the ‘read’ tracepoint is turned on, then it records the execution of the read() system call on Linux. The tracing recorder tools also record the parameters that the read() function returns. The LTTng tracing tools used in this work are explained in the next section.

2.4.1 LTTng Tracing Tools

LTTng (Linux Trace Toolkit: next generation) is a software to record actions by the Linux Kernel, Linux user applications, and user libraries. LTTng has Kernel modules to trace the Linux Kernel, shared libraries to trace user-space tracepoints instrumented in C/C++ applications, Java packages to trace Java user-space tracepoints instrumented in Java applications, daemons and a command-line tool. Tracing control is one of the main feature of LTTng which selects the specific tracepoints for the user to trace instead of all tracepoints being traced. LTTng defines a session for each tracing activity to do the configuration of memory allocation to save the traces, the number of buffers intended to be used, and the session name [19] .

Another main feature of LTTng is user-space tracing. This feature allows users to instrument LTTng tracing functions in their source code to return the desired data. When the user-space tracepoints are enabled by the tracing control, and the instrumented program runs, the LTTng records the user-space tracepoints being executed, and it returns the parameters that have been set in the LTTng user-space functions [19] . This makes it easy to combine Kernel-space and user-space tracing information.

2.4.2 LTTng User-Space Tracepoints

In terms of C environment on Linux, a user-space tracepoint’s name is defined in a header file. Users program the parameters that they want to record, using calls to the LTTng library. Tracepoints are defined similarly in the Java environment [3] . Chapter 5 describes

user-space tracing in more detail.

2.4.3 LTTng and Common Trace Format (CTF)

The LTTng tracing tools return the tracing result in the Common Trace Format (CTF) described in [26] . CTF is a standard designed to deal with the flexibility problem. It is a binary trace format which is fast to write and read, which describes events in form of streams. It allows users to customize how many events in each stream. In LTTng terms, these streams are known as channels. CTF's streams require a metadata stream which contains the description of the other channels. The metadata stream is written in Trace Stream Description Language (TSDL) [26] .

2.4.4 LTTng Traces Viewer (Trace Compass)

There are specific applications to extract the data from CTF such as 'lttng view' command, Babeltrace, and Trace Compass.

The 'lttng view' command shows the LTTng result in a human readable format. It launches external viewer. However, this command is only available before destroying the LTTng session. In the other words, it cannot be used for converting the archived CTF to human readable format [18] .

Babeltrace is an external application in Linux that converts the CTF format result into human readable format. It can also store the conversion into a text file. Babeltrace was developed specifically by the LTTng community to display the human-readable trace format [17] .

Trace Compass is a Java written tool to extract and analyze any kind of trace. It can visualize traces as well as shows the useful information out of traces. It supports all size of large traces file, and it synchronizes multiple traces into one based on the timestamp on the traces. Do this, Trace Compass modifies the timestamp values for all but one node, based on internode message events. Besides visualization, Trace Compass helps users in filtering the specific events that user cares about and export it to separate .csv file [4] .

2.5 Perf Tools

“Perf tools” or Performance Counters for Linux (PCL) are an open source tracing tool for profiling on a Linux system. The execution of “Perf tools” use the command-line

in a Linux terminal. Moreover, both Kernel and user-space information are covered by “Perf tools”. It also can be used to do the analysis on hardware that is attached on the system.

Besides the performance data, the activities of processes are recorded by “Perf tools”. It extracts the information from the execution of the `exec()` function on Linux. In this research, “Perf tools” are used to record the list of processes that executed by Java Virtual Machine with the process Id and the Java process name [10] .

2.6 Java Remote Method Invocation

Remote Method Invocation (RMI) is a Java mechanism that creates objects in Java Virtual Machines and allows the objects to interact with each other from different Java Virtual Machines [11] . The mechanism adapts the concept of distributed system, and it can be implemented on different hosts. In RMI, Java programs interact using a remote communication. To use the feature, a Java package called ‘`java.rmi.*`’ need to be imported into the Java programs [22] .

RMI focuses on remote object serialization which allows a Java program send the remote objects and their references to another Java program. It provides reliable communication between two Java programs using the TCP protocol. The remote interface specifies the object’s method that can be invoked remotely. However, as there are remote objects and local objects, they have to be distinguished by adding the `RemoteException` into the interface file. The interface file needed to be part of the implementation file, and makes the remote objects on the interface file available to be registered [22] .

Prior to executing the Java RMI client-server programs, remote objects should have been registered. The RMI registry opens the way between the available remote objects to be accesses. It is a lookup mechanism for remote objects based on the Java Naming and Directory Interface (JNDI). The RMI registry generates stub and skeleton classes to be a

bridge between client and server [22] .

Illustration 2 Client-Server Java RMI

2.7 State of The Art of Model Building

Software Performance Engineering (SPE) is an approach to model-based software performance analysis developed by Smith and Williams [24] using a quantitative model of software based on its proposed architecture. The performance model requires a model calibration. Expanding on Smith's idea, Woodside et al [27] advocated a deeper integration between measurement and modeling approaches to analysis (this thesis may be seen as a step towards such integration). This makes the quantitative model more practical, more automated and more easily applied to different cases. Therefore, the authors suggested the needs of CPU demand measurement tools along with the system operations tracing tools.

Model Calibration

With regards to determining the CPU demand and call rate parameters, some approaches in predicting parametric dependencies were described by Ackermann and Spinner [1] . They applied a range of machine learning methods to datasets collected from the execution of several algorithms. However, none of the machine learning methods works well in predicting parametric dependencies. On the other hand, something they call a

“meta-selector technique” works better than machine learning methods. It has error rate 30% less than Support Vector Machine’s prediction which is claimed as the best predictor on machine learning methods.

The survey paper [25] describes a range of approaches to estimate resource demands, listed and described in Table 1.

Approximation with the Response Times	Resource demands can be approximated by measured response time if the queueing delay is smaller than the resource demands
Service Demand Law	<p>Using the Service Demand Law and the Utilization Law, the resource demands can be estimated by utilization and the throughput data based on the following relationship.</p> <p>(Utilization Law) $U_k = X_k S_k$</p> <p>(Service Demand Law) $U_k = X D_k$</p> <p>D_k The service demand at resource k</p> <p>U_k The resource k utilization rate</p> <p>X_k The throughput at resource k</p> <p>X The system throughput</p> <p>S_k The service requirement per visit</p> <p>The utilization is derived from the Utilization Law [15]</p>
Linear Regression	Utilization Law is a linear model. Least Squares (LSQ) regression focuses only on the mean of resource demands and may cause significant estimation errors. Least Trimmed Squares (LTS) regression is used to increase the robustness of the regression to deal with multicollinearity and outliers. The response times and queue length on arrival can be modeled by regression analysis to estimate the resource demands.
Kalman Filter	Creating a system state model that deals with the system changes over time. An extended filter must be used to estimate a nonlinear model. The model is based on Utilization Law which used to estimate resource demands for multi-user applications.
Optimization	Mathematical programming methods can be used to estimate the resource demands based on objective functions. The objective functions focus on minimizing the estimation error of response time and utilizations.

Machine Learning	<p>Using cluster-wise regression to deal with the discontinuities in the resource demands due to system configuration changes. Observation clustering is performed into groups that have stable resource demands. The resource demands are estimated per cluster.</p> <p>Independent Component Analysis (ICA) estimates the resource demands using the Utilization Law linear model. It also groups requests into workload classes.</p> <p>Support Vector Machine (SVM) is the recommended method to estimate the resource demands which is considering the workload classes.</p>
Maximum Likelihood Estimation (MLE)	<p>Uses a probability model to estimate the resource demands with measured response times and queue lengths on request arrivals. The likelihood function can be based on phase-type distributions.</p>
Gibbs Sampling	<p>This approach is used to construct Markov-Chain Monte Carlo experiment that estimates the density function of the resource demands. The function can be used on single class, open queueing networks and multi-class, closed queueing networks.</p>

Table 1 Survey journal approaches list

The authors evaluated all these approaches by an experiment using a dataset that is obtained from a queueing simulator based on an M/M/1 queue with FCFS (First Come First Served) scheduling. The result shows that using the measured response times can improve the resource demands estimation accuracy significantly. There are many references in the paper [25] that provide details on the different approaches to model calibration.

Deriving Model Structure

For deriving the structure of a performance model from execution data, Hrischuk [12] pioneered the creation of layered models, using special traces which he called “angio traces”; an identifier called an “angio dye” was injected for each response and passed from one component to another. Israr [13] developed a simpler method that did not need the angio dye, so that ordinary traces could be used. He created a tool, called Software

Architecture and Model Extraction (SAME), that automatically creates the LQN model using the communication patterns. SAME analyzes the trace and identifies the pattern of the communications between the components. As a conclusion, the SAME tools are feasible for the industrial-size traces, and it has been verified to produce the correct LQN models with different scenarios. The SAME analysis has since then been extended to handle phases of service and simple sequences of activities. SAME was used in this project, to analyze the LTTng traces and calibrate CPU time of phases and activities, which is described in detail in Chapter 3.

Another approach called ‘WebTime’ was developed by Mizanojehdehi [22] . WebTime also extracts LQN performance models based on the execution traces of a distributed application. The execution traces were logged by WebTime, when application level events are executed. The application level events need to be instrumented in the source code. The traces were attached by timestamp that was customized using a virtual clock concept, to specify the sequence of events. Using the traces, WebTime identifies the connection between messages in the traces, and the connected messages are converted into an execution graph. The next step, WebTime analyzes the CPU time spent each entry, the tasks and the interactions of the distributed application from the execution traces and constructs the LQN sub-models. In terms of phases and activities mechanism, WebTime constructs the LQN sub-model for each of them as ‘phases’ model and ‘activity’ model. The sub models are united into a complete LQN model. Moreover, WebTime covers the three types of interaction patterns which are similar with SAME [13] . In this work, the execution traces have been found adequate to be a basis for model building. WebTime is more complete than the earlier methods, as it can model phases and arbitrary activity sequences including forking and joining.

Application Modeling

A wide variety of projects to build LQN models of particular applications have been carried out, not necessarily using tracing. Many have been related to telecommunications software. A research, titled “A Performance Model for a Network of Prototype Software Routers”, was studied by Wu [30] . The research focused on the approach to building the LQN model for the packet transmissions system adapted by internet routers. Almost similar with Israr’s study [13] , this research identifies the packet transmissions between processes,

instead of the messages. Different from the previous study, to avoid the complexity, the author used the assembling sub-models for different operations as an approach. Besides different operations, the sub-models also cover many scenarios.

Maly and Woodside [20] modeled hardware and software of an application to a LAN extension router, and El-Sayed, Cameron, and Woodside [5] modeled a bidirectional optical ring and its failure-handling mechanism. In these works, the LQN approach has been found adequate to model a wide variety of telecommunication systems.

Chapter 3: SAME Tools

Chapter 3 explains about the details of the Software Architecture and Model Extraction (SAME) software mentioned in Section 2.7 above [13] for extracting LQN model structures based on traces. The SAME has two phases in extracting an LQN model structure. The SAME Phase 1 analyzes event trace and produces an interaction trace. The interactions are synchronous, asynchronous and forwarding interactions, and the SAME Phase 2 extracts an LQN model structure based on interaction trace and creates tasks, entries, activities and calls.

3.1 SAME phase 1

SAME phase 1 uses message traces as the source data, with records formatted as:

< Timestamp: Long >< ProcessName: String >< eventType >< MessageID: String >

in which timestamp is an integer, ProcessName is a string identifying the deployed software process that sends or receives a message, eventType is a string “send”, “receive” or “exec”, and messageID is a string which must be generated by Filtering Tools after identifying each particular message from other messages, so the sending and receiving events for that message can be correlated together.

LTTng traces have their own format, as described in Chapter 4, with many more fields, so the LTTng trace data must be rearranged for input to SAME Phase 1. The LTTng timestamp is a real value in seconds, which was multiplied to by 10^9 to give an integer in nanoseconds, and a filter (described Section 5.2) was applied to detect the send and receive events, and to extract the messageID field. A sample of the filtered LTTng trace is shown.

```
2104139142313470 Process_A send NT02123
2104155999921860 Process_B receive NT0106107
2104165948627170 Process_B send NT0148149
2104165951997810 Process_A receive NT0148149
2104165941646900 Process_C send NT0146147
2104165944971070 Process_D receive NT0146147
```

Illustration 3 SAME input format

The communication patterns illustrate the interaction between processes. SAME Phase 1 deduces the relationships between messages, based on assumptions that only the

user task spontaneously sends messages, everything else only reacts to messages, and reactions are part of the execution of a model entry. A technique called Interaction Tree Transformation is applied by SAME Phase 1 to detect the three kinds of interactions between tasks from the message sequence: asynchronous, synchronous forwarding. It outputs an interaction trace with the format:

$$< CommPattern = S|A|F > < Sender > < Receiver > < t1 > \{ < t2 > \}$$

where CommPattern is a one-letter code for the interaction pattern, the Sender and Receiver are strings giving the process names, t1 is the time of the first message, and for a synchronous message, t2 is the time of the reply. This work does not consider forwarding messages, for which a more complex record structure is used to define a sequence of messages along a forwarding path.

For synchronous messages, it connects two communication events linking two different processes. t1 stands for the timestamp when the receiver receives a message from sender, and t2 stands for the timestamp when the sender receives a reply message from the receiver. The sender is assumed to not perform any activities between t1 and t2.

SAME Phase 1 Transformation for a Synchronous Interaction

Input trace records

<timestamp><process name><event type><message id>

50 Process_A send fs019362

100 Process_B receive fs019362

150 Process_B send fs019363

200 Process_A receive fs019363

Interaction trace records (S: Synchronous; A: Asynchronous)

<CommPattern><Process1><Process2><ReceivedTime_1><ReceivedTime_2>

S Process_A Process_B 100 200

Illustration 4 SAME1 Synchronous Communication Pattern

$$< CommPattern \{ Async \}; A > < Sender > < Receiver > < t1 >$$

For an asynchronous interaction, it relates to only one message, because there is no reply.

In the sequence, two processes do the send-receive interaction with the identical message id. Therefore, SAME phase 1 returns only one timestamp which is t1, when the receiver receives a message from the sender.

SAME Phase 1 Transformation Including an Asynchronous Interaction

<timestamp><process name><event type><message id>

50 Process_A send fs019362

100 Process_B receive fs019362

150 Process_C send nt0116117

200 Process_D receive nt0116117

300 Process_B send fs019363

400 Process_A receive fs019363

Interaction Pattern (S: Synchronous; A: Asynchronous)

<CommPattern><Process1><Process2><ReceivedTime_1><ReceivedTime_2>

S Process_A Process_B 100 400

A Process_C Process_D 200

Illustration 5 SAME1 Synchronous and Asynchronous Communication Pattern

Illustration 5 indicates a part of the SAME phase 1 result. Synchronous and asynchronous messages are included in the illustration.

Illustration 6 SAME1 View of Synchronous and Asynchronous Interactions

The last type of interaction pattern is the forwarding type. However, in this research, the forwarding type of interaction is not covered.

The trace input to SAME Phase 1 can also include activity records in which the event type is “exec” and the message-id field conveys an activity identifier. These mark the start of some step of execution, and normally have to be obtained from application-level traces. LTTng can be arranged to capture application level events, and this was done in the communication-software experiments. SAME Phase 1 passes each such record through to its output in the form:

E <process-name> <timestamp> <message-id>

3.2 SAME Phase 2 (ExtractModel)

The second phase of SAME (ExtractModel) is in charge of the model building. It builds an LQN model in XML format based on the interaction trace. In the second phase, SAME creates sets of tasks, entries, activities and calls based on the behavior of send-receive events from the phase 1 interaction trace. ExtractModel first identifies a separate entry to handle every call in the trace, and then merges similar entries so as to produce a model of reasonable complexity. It has two rules for entries simplification, called ‘mergeByCallSet’ and ‘mergeByTask’. The ‘mergeByTask’ rule generates an LQN model with single entry for each type of call (synchronous or asynchronous) to each task, which is the simplest model possible. The ‘mergeByCallSet’ rule merges entries that have the same type and task and the same set of destinations of calls.

Using the example of Illustration 7, ExtractModel generates an LQN model structure for Process A, Process B, Process C and Process D.

SAME phase 1 result

Communication Pattern (S: Synchronous; A: Asynchronous)

<CommPattern><Process1><Process2><ReceivedTime_1><ReceivedTime_2>

S Process_A Process_B 100 400

A Process_C Process_D 200

S Process_B Process_C 400 600

ExtractModel result

Illustration 7 ExtractModel Extraction based on Interaction Pattern following from Illustration 5

If application-level activities are also traced, the activities information may be prepended to the SAME phase 1 result file to get analyzed by ExtractModel in SAME2. The previous example will be extended by adding one activity executed by entry e2 in task Process_B.

Interaction and Activity Trace input to ExtractModel (SAME Phase 2)

E Process_B 191033504016360 activity1_start

E Process_B 191033504016361 activity1_end

S Process_A Process_B 100 400

A Process_C Process_D 200

S Process_B Process_C 400 600

ExtractModel result

Illustration 8 SAME1 result – Activities and ExtractModel result

Illustration 8 shows the example of activities information format and how ExtractModel extracts the activities information with the corresponding process. As explained above, ‘activity1’ happened in Process_A (task) when ‘t’ equals to 191033504016360, and ‘activity1’ finished when ‘t’ equals to 191033504016361. The dashed line from the second activity back to the entry e2 indicates that the reply from the entry is generated when the activity is finished.

Chapter 4: LTTng Traces

Chapter 4 contains a brief explanation of LTTng, its limitation and its traces. LTTng stands for Linux Trace Toolkit: next generation which is used to trace the Linux Kernel, and user application. LTTng includes

- Kernel modules to trace the Linux Kernel
- shared libraries to trace user application which is written in C or C++
- Java packages to trace Java applications
- daemons and command-line tools to control the LTTng tracers.

To trace a system, user is required to create event rules which enable tracepoints to be traced. The LTTng Kernel tracepoints that are used in this research will be explain on the following section.

4.1 LTTng Kernel Tracepoints

Linux Kernel tracepoints, delivered with the standard Linux distributions, provide hooks to call trace functions that are executed at runtime. The Kernel tracepoints can be enabled using LTTng by inserting the name of desired Kernel tracepoints in the event rules for a tracing session. If the predefined tracepoints are unsuitable a user may define new ones, but the process of inserting a new Kernel tracepoint is arduous. It requires modification of the Linux code and Kernel compilation. LTTng also reports events at Linux system calls. Events must be enabled to be reported.

With the enabled events LTTng records the execution and stores it as an LTTng Kernel trace. This research used certain events to capture the process interactions by messaging, and the file I/O operations. For process interactions, ‘net_dev_queue’ is a Kernel tracepoint that was enabled, and for file operations, tracing ‘read’ and ‘write’ system calls captures the behavior involving the file service. The events are described, with examples of each one, as follows.

1. read() System Call.

Enabling this event causes two event records to be created in the trace, on entry to the read() routine, and on exit, as shown in the example. The event arguments give the host, the time down to nanoseconds, the event name such as syscall_entry_read, the file descriptor, the byte count, the process ids, the name of executable of the

process (in this case, Java), and some other fields.

```
HostA-20181210-210017/kernel 21:04:14.406 406 273 more-subbuf_0 0
syscall_entry_read fd=3, count=4096, context._tid=5906, context._pid=5905,
context._procname=java 5906 20 5905

HostA-20181210-210017/kernel 21:04:14.406 415 199 more-subbuf_0 0
syscall_exit_read ret=4096, buf=140284104814592, context._tid=5906,
context._pid=5905, context._procname=java 5906 20 5905
```

Illustration 9 LTTng traces – read()

2. write() System Call

Similarly this event produces a trace record for entry and exit for a write.

```
HostB-20181210-210259/kernel__ 21:04:38.219 226 828 more-subbuf_2 2
syscall_entry_write fd=2, buf=139758118289824, count=89, context._tid=5421,
context._pid=5401, context._procname=java 5421 20 5401

HostB-20181210-210259/kernel__ 21:04:38.219 231 517 more-subbuf_2 2
syscall_exit_write ret=89, context._tid=5421, context._pid=5401,
context._procname=java 5421 20 5401
```

Illustration 10 LTTng traces – write()

3. net_dev_queue() Tracepoint

This event records a variety of network interface events identified by their arguments. There is a buffer address, a data length, a network name, a lot of protocol related arguments including the transport header type (here, tcp) with a sequence number and an acknowledgment sequence number.

```

HostA-20181210-210017/kernel 21:04:27.104 671 165 more-subbuf_2 2
net_dev_queue skbaddr=0xffff9602c30ef600, len=66, name=eth0,
network_header_type=_ipv4, network_header=ipv4=[version=4, ihl=5, tos=0, tot_len=52,
id=0x62aa, frag_off=16384, ttl=64, protocol=_tcp, checksum=0x60f9, saddr_padding=[],
saddr=[192, 168, 122, 213], daddr_padding=[], daddr=[192, 168, 122, 250],
transport_header_type=_tcp, transport_header=tcp=[source_port=54182, dest_port=5555,
seq=1286023967, ack_seq=1833918913, data_offset=8, reserved=0, flags=0x10,
window_size=229, checksum=0x7747, urg_ptr=0]], context._tid=5914,
context._pid=5905, context._procname=java 5914 20 5905

HostB-20181210-210259/kernel 21:04:27.109 384 564 more-subbuf_3 3
net_dev_queue skbaddr=0xffff9415759180f8, len=88, name=eth0,
network_header_type=_ipv4, network_header=ipv4=[version=4, ihl=5, tos=0, tot_len=74,
id=0x2792, frag_off=16384, ttl=64, protocol=_tcp, checksum=0x9bfb, saddr_padding=[],
saddr=[192, 168, 122, 250], daddr_padding=[], daddr=[192, 168, 122, 213],
transport_header_type=_tcp, transport_header=tcp=[source_port=5555, dest_port=54182,
seq=1833918913, ack_seq=1286023974, data_offset=8, reserved=0, flags=0x18,
window_size=227, checksum=0x775d, urg_ptr=0]], context._tid=5454,
context._pid=5401, context._procname=java 5454 20 5401

```

Illustration 11 LTTng traces – net_dev_queue()

4.2 Challenges of using LTTng Kernel

Many challenges surfaced while using the LTTng Kernel to extract information from it.

- Every LTTng Kernel tracepoint has different information. For instance, Section 4.1 lists the relevant LTTng Kernel tracepoints along with its trace examples. ‘read()’ and ‘write()’ have different contents between them. That information needs to be converted to the SAME required format.
- LTTng Kernel tracepoints record everything. That extra information might be irrelevant for model building.
- For Java programs, LTTng tools cannot report the Java program that is being executed by the JVM
- LTTng tools only record the CPU time for processes with process scheduling events. However, there are so many of these events that they could not be used in the industrial setting (due to pressure on space for the trace) and an alternative approach was sought for CPU time.
- The Linux Kernel does not report on the complete transfer of UDP messages, so it is

not possible to tell which message reception corresponds to which message sending. Indeed this limitation originates in UDP itself, which does not use sequence numbers in messages, and is not a limitation so much of LTTng, but of the whole Kernel-level approach.

4.3 Dealing with LTTng Kernel challenges

In order to use LTTng traces as source data in model building, these approaches were taken to overcome the challenges:

- Trace Compass was used to merge and synchronize traces from different sources.
- Filtering tools were developed to deal with the various information and formats of LTTng Kernel tracepoints mentioned in Section 4.1 that are recorded by LTTng tools. They also convert the formats into SAME required format.
- Filtering algorithms were created to identify some communications events: file operation algorithm and TCP communication algorithm are developed along with Filtering tools.
- To determine the program that is executed by a JVM, Perf tools includes a tool named ‘execsnoop’ that records all the command line and thus the names of Java programs running in a JVM.
- CPUSat is a tool that was developed to overcome a problem related to CPU time. It captures the cumulative CPU time of each process from the `/proc/{PID}/stat` file.

Chapter 5: Filtering Tools

Chapter 5 explains about the extension of SAME by Filtering Tools that interpret the trace into a form for model extraction. There are five concerns addressed by the filters:

- obtaining the application process name for each process, from PerfTools. In many cases LTTng records only the process id and the name of the executable, which for Java programs is always just Java.
- identification of the sending and receiving of a message between Linux processes, from LTTng events.
- conversion of file access events into events describing interactions that SAME can interpret. In these interactions the disk controller is treated as a task, the beginning of an operation is treated as a request message, and the completion as a reply message.
- recording of the progress of CPU time consumption by the processes on each host, using periodic sampling of system counters.
- combining messages that are part of the same application level message between processes.

This section describes the filters that do these four steps. For a distributed system, the trace synchronization across the separate traces for each node, described in Chapter 4, is performed first.

5.1 Obtain the Application Name for each Process

The Java Virtual Machine (JVM) conceals the Java processes name. In LTTng traces, all Java processes are considered as a JVM process, and they have similar name which is ‘java’ with different process ID (PID). LQN requires a task name for doing a further analysis, and the well-known Java Perf tools have a tool that can solve this.

The Perf tools feature called ‘execsnoop’ can report the Java process names along with the process ID. However, root access is required in order to use execsnoop. The command below needs to be executed in the Terminal before executing the Java processes. It automatically writes a file written in ‘[Destination_File_Path]’ so the further processing is automated.

```
[root@pc ~] # ./execsnoop >> [Destination_File_Path]
```

Illustration 12 Perf tools-execsnoop command-line

The result contains the active process name, its process ID (PID) and its parent process ID (PPID). Filtering tools match the PID in Perf result with the PID of Java processes in the SAME phase 1 input file. If they are matched, then, the Filtering tools will replace the 'java' and its PID with the process name of the matched PID.

5.2 Extraction of Messages Using TCP

LTtng records TCP communication events by the `net_dev_queue()` tracepoint. It returns parameters that include:

- `saddr` (source IP address),
- `daddr` (destination IP address),
- `sport` (source port),
- `dport`(destination port),
- transport header type,
- `seq` (packet sequence number), and
- `ack_seq` (packet acknowledge sequence number).

Illustration 14 shows the data that is the basis for matching two `net_dev_queue` events to identify a single message. The event matching has one condition with regard to timestamps, the sender should have an earlier timestamp than the receiver. There are three matching parameters such as IP address, port and packet sequence number, which are highlighted in red.

The first matching parameter is the destination IP address (`daddr`) on the first '`net_dev_queue`' event matched to the source IP address (`saddr`) on the second '`net_dev_queue`' event. Consider the first event executed by ProcessA in machine A. It has `daddr` which is 192.168.122.1 that matches with the `saddr` on the second event which is located in machine B and executed by ProcessB.

The second matching parameter is the port of each event. The first event has destination port (dport) 38668 and it matches with second event's source port (sport). The third matching parameter is the packet sequence number. The first event has 25956237 as packet sequence number that was sent (ack_seq) and the second event has the identical packet sequence number that was received. Therefore, using the three matching parameters, Process A is considered as the sender with earlier timestamp, whereas Process B is the receiver with a later timestamp.

```
for each event  $E_i$  in the trace
    find the first later event  $E_j$  in the trace which satisfies:
        daddr[ $i$ ] = saddr[ $j$ ] and
        dport[ $i$ ] = sport[ $j$ ] and
        ackseq[ $i$ ] = seq[ $j$ ]
    create two records as follows:
        timestamp[ $i$ ]  procname[ $i$ ] 'send' 'msg $ij$ '
        timestamp[ $j$ ]  procname[ $j$ ] 'receive' 'msg $ij$ '

where the field msg $ij$  is a string 'msg' followed by two integers packed together
```

Illustration 13 TCP Message Extraction Algorithm pseudo code

LTtng result : net_dev_queue() events

Record #20

```
Mcha-20181210-210017/kernel 21:04:12.989 419 058 more-subbuf_0 0 net_dev_queue
skbaddr=0xffff9602fba0d500, len=74, name=lo, network_header_type=_ipv4,
network_header=ipv4=[version=4, ihl=5, tos=0, tot_len=60, id=0x0, frag_off=16384, ttl=64,
protocol=_tcp, checksum=0xc3c0, saddr_padding=[], saddr=[192, 168, 122, 213],
daddr_padding=[], daddr=[192, 168, 122, 1], transport_header_type=_tcp,
transport_header=tcp=[source_port=1099, dest_port=38668, seq=3799395175,
ack_seq=259562370, data_offset=10, reserved=0, flags=0x12, window_size=43690,
checksum=0x772a, urg_ptr=0]], context._tid=5883, context._pid=5882,
context._procname=ProcessA ----- 1
```

Record #35

```
Mchb-20181210-210017/kernel 21:04:12.989 439 366 more-subbuf_0 0 net_dev_queue
skbaddr=0xffff9602fba0d400, len=66, name=lo, network_header_type=_ipv4,
network_header=ipv4=[version=4, ihl=5, tos=0, tot_len=52, id=0x8915, frag_off=16384,
ttl=64, protocol=_tcp, checksum=0x3ab3, saddr_padding=[], saddr=[192, 168, 122, 1],
daddr_padding=[], daddr=[192, 168, 122, 213], transport_header_type=_tcp,
transport_header=tcp=[source_port=38668, dest_port=1099, seq=259562370,
ack_seq=3799395176, data_offset=8, reserved=0, flags=0x10, window_size=342,
checksum=0x7722, urg_ptr=0]], context._tid=5862, context._pid=5861,
context._procname=ProcessB ----- 2
```

Message Extraction in SAME format

```
21:04:12.989419058 ProcessA send msg2035
21:04:12.989439366 ProcessB receive msg2035
```

Illustration 14 TCP algorithm result

The TCP algorithm is implemented in the NTTrace module in FilteringTools. NTTrace, first, filters the irrelevant events from traces, and keeps the ‘net_dev_queue’ events that are executed by Java processes. It also transforms the ‘net_dev_queue’ events into SAME input format with the implemented TCP algorithm. The command below is executed in the Terminal to run the Filtering Tools, then choose the option number 0 for the Networking analysis which is on ‘NTTrace’ module.

```
[user@pc FT] # java filteringtools.FilterTool 0 [Traces_File_Path]
```

Illustration 15 FilteringTools – NTTrace command-line

5.3 Conversion of File Operations

LQN models file operations as messages to a “task” representing the disk driver or file system library, running on a “processor” which is the disk hardware. To construct a model, the calls to the file system library are interpreted as pseudo-messages to a pseudo-task, and the trace events are transformed to correspond to this pattern.

The file system events that are considered here are read and write operations. On LTTng, those events are covered in Systemcall events to `read( )` and `write( )` functions (Section 4.1 Illustration 9 and Illustration 10). Every `read( )` function has `entry_read( )` and `exit_read( )` events that return the parameters when it starts and it ends. For writing it is similar.

The `entry_read()` event reports these arguments:

- timestamp
- `fd` = the file descriptor that will be used to execute the read operation.
- a ‘count’ parameter implies the amount of data that will be read every single read operation. It could be broken down into chunks.
- process id. Call the process `P` for this discussion.

The transformation interprets this pair of events for SAME as four events. It invents a task with the name “file_service” and creates these four events for the input to SAME:

1. sending a message from `P` to file_service
2. receiving a message from `P` at file_service
3. sending a message to `P` from file_service
4. receiving a message at `P` from file_service

The first two events are given the timestamp of the `entry_read( )` event, and the second two are given the timestamp of the `exit_read( )` event.

Illustration 16 File operations algorithm flowchart

Illustration 16 indicates the steps covered by a filtering tool that was developed for analyzing file operations. In addition, the Illustration 17 shows an example of the data transformations. The file system algorithm is applied on the cleaned read event data lists which is illustrated by step number 5. For example, the LTTng result on 'ProcessA' process returns two read events which are entry_read and exit read. As the SAME only deals with the send-receive events. Using the Filtering tools, the read events are converted to send-receive events. When a process accesses file system for read or write operations, the process is considered to send a request to the "file service", and the "file service" replies to the process using the read operation result. Therefore, an assumption has been set. While the process requests the "file service" to do read operation. It means process is the sender and the "file service" is the receiver. As every read request, must have it replies, the replies make the "file service" as the sender and the process as the receiver.

LTTng result : Read events

Mch02/kernel 19:09:57.969 021 374 more-subbuf_1 1 syscall entry_read fd=33,
count=**16**, context._tid=8524, context._pid=8555, context._procname=ProcessA

Mch02/kernel 19:09:57.969 023 603 more-subbuf_1 1 syscall exit_read
ret=**16**, buf=3070432016, context._tid=8524, context._pid=8555, context._procname=ProcessA

File System Pseudo-Messages

19:09:57.9690213740 ProcessA send msg01

entry_read

19:09:57.9690213741 Read_FileService receive msg01

19:09:57.9690236030 Read_FileService send msg02

exit_read

19:09:57.9690236031 ProcessA receive msg02

Illustration 17 File operations algorithm result

FileSystem algorithm specifically is implemented on FSTrace module in the FilteringTools project. FSTrace covers step number 1 to step number 5 in Illustration 16. The command-line below is executed to run the FSTrace module in the Filtering tools.

```
[user@pc FT] # java filteringtools.FilterTool 1 [Traces_File_Path]
```

Illustration 18 FilteringTools – FSTrace command-line

5.4 Inferring CPU Times: CPUStat Algorithms

To calibrate CPU demand parameters we require the elapsed CPU time of a process for all its messaging event. This is not available. Another approach would be to determine the process CPU times from scheduling events and use the most recent one at each messaging event. Since this was not feasible in the industrial case study (it required too much trace storage space) an approximate work-around was created, which logs the CPU times separately and then interpolates values using the message event timestamps. The recording was done by a logging program CPUStat that was written for the purpose.

CPUStat is a tool that records the cumulative CPU time for every process, from the information in every /proc/{PID}/stat directory (where PID stands for the id of the process). The CPUStat tool consists of C program and the shell script that helps record the

output of C program into a file. Time is recorded in seconds and milliseconds, which are system dependent but a tick is usually 10 millisecc. The overhead of tracing and CPUStat do not impact the CPU time measurements for the application processes.

Illustration 19 CPUStat tools - diagram

CPUStat tools contain of a C program that record the `/proc/{PID}/stat` for every process that run in the Linux, and also a shell script that executes the C program every 1 second period and stores the CPUStat result in a directory. The one-sec interval is a compromise between accuracy and overhead, and can be changed.

The C program opens the `/proc/` directory and lists the PID(s) that currently are run in Linux. For every PID, it will access the stat file, and stores every content in stat file into variables. The contents that are related to the system time e.g. `utime`, and `stime`, are converted into seconds measurement unit using `sysconf(_SC_CLK_TCK)`. Then, the

system times put into CPUTime accounting function which use the following formula,

$$CPUTime = utime + stime$$

The “utime” stands for user time, and the “stime” stands for system time.

The shell script maintains the execution of the C program. It executes the C program every 1 second period before executing the main program (Java processes).

Then, the shell script defines the name of file that contains the CPUSat result and where it is stored. The script will tell User the directory and the file name printed in the Terminal once it is stopped using a keyboard interrupt (CTRL+C) when User decides to stop the CPUTime recording or when the main program finished. The example of recording result is shown below. The main programs are Java processes with pid 5861 and 5905 with their CPU time increments in one second period (21:04:02 – 21:04:03).

The CPUSat samples the CPU time in one-second interval. Shorter intervals such as half-second, one-millisecond, have been evaluated. It causes disk overhead, and does not record more significant changes on the CPU time. Therefore, interpolation is a key to obtain the CPU time in a fraction of one second.

```
Local Time: 12-10-2018_21:04:02
pid 5822, (cpustat),cpu 0.000000
pid 5646, (ltnng-runas),cpu 0.000000
pid 5861, (java),cpu 0.670000
pid 5905, (java),cpu 0.010000
pid 1, (systemd),cpu 25.170000
Local Time: 12-10-2018_21:04:03
pid 5822, (cpustat),cpu 0.02000
pid 5646, (ltnng-runas),cpu 0.010000
pid 5861, (java),cpu 0.770000
pid 5905, (java),cpu 0.020000
pid 1, (systemd),cpu 25.200000
Local Time: 12-10-2018_21:04:09
pid 5822, (cpustat),cpu 0.08000
pid 5646, (ltnng-runas),cpu 0.070000
pid 5861, (java),cpu 0.840000
pid 5905, (java),cpu 0.110000
pid 1, (systemd),cpu 25.380000
```

Illustration 20 CPUSat tools - example Output of CPUSat

5.5 Add CPU Times

To solve LQN model, model parameters are required such as interaction frequency and CPU time as the service demands. ExtractModel deals with the aggregation of model parameters. However, the CPU time needs to be calibrated into the interaction trace, and the calibrated file is fed into ExtractModel to produce an LQN model structure. This requires an extended format for the interaction trace as illustrated below.

$\langle S \rangle \langle \text{Sender} \rangle \langle \text{Receiver} \rangle \langle t1 \rangle \langle t2 \rangle \langle \text{NULL} \rangle \langle \text{CPUtime}_{t1} \rangle \langle \text{CPUtime}_{t2} \rangle$ For synchronous interactions, CPUtime_{t1} is a result of a linear interpolation at $t1$, and CPUtime_{t2} is a linear interpolation at $t2$. The “NULL” is a string value demanded by the SAME Phase 1 if process names and activity names are not provided. This work did not use process names but it did use activity names.

$$\langle A \rangle \langle \text{Sender} \rangle \langle \text{Receiver} \rangle \langle t1 \rangle \langle \text{NULL} \rangle \langle \text{CPUtime}_{t1} \rangle$$
$$\langle E \rangle \langle \text{TaskName} \rangle \langle t1 \rangle \langle \text{ActivityName} \rangle \langle \text{CPUtime}_{t1} \rangle$$

For an asynchronous interaction $\langle A \rangle$ and activity pattern $\langle E \rangle$, CPUtime_{t1} is a result of a linear interpolation at $t1$.

With regards to CPUStat result, as it was gathered from `/proc/{PID}/stat` file, Linux captures the CPU time with the resolution of 10 ms by default. It becomes a resolution problem for analysis.

The interpolation uses a linear interpolation package [16] that is imported in Filtering tools. LinearInterpolation takes input from the CPUStat output for every process e.g. ClientA, ClientB. Then, it requires the CPU time increments as Y and the timestamp as X to form a linear interpolation function for each receiver process.

$$\text{LinearInterpolation}(\text{Double}[] X, \text{Double}[] Y)$$

Next, using the times of messaging events in the interaction trace, Filtering tools calls the Interpolate method to do the interpolation for the specific timestamp.

$$\text{Interpolate}(\text{Double } t)$$

The interpolate function returns the interpolated $CPUtime_{t_1}$ and $CPUtime_{t_2}$ for each interaction. However, the interpolation is not useful for the file service interactions. Linux does not capture the processing time of the disk in `/proc/` file. Thus, file service interactions use the elapsed time as their CPU time. ExtractModel requires that the Kernel trace timestamps and the CPUStat time values be brought to common units, which here were in nanoseconds.

5.6 Combining Parts of a Message

The TCP protocol software divides a message into chunks before sending through TCP protocol, so one message may give rise to many send/receive pairs. Thus one message may have many `net_dev_queue` events and will produce many send events and receive events for SAME Phase 1. If the chunks are all part of an asynchronous interaction, they can be modeled by as many messages as they appear in the trace. However, for a synchronous interaction LQN requires that the request and reply each be a single message, and processing by SAME requires this to correctly detect synchronous interactions. If the chunks are left separate only the last chunk of the request and the first chunk of the reply will be detected as a synchronous interaction, and the other chunks will be represented as separate asynchronous interactions, cluttering the model.

To adapt the trace to SAME, a program MsgSimplifier was developed to combine the parts of a message. MsgSimplifier analyzes the SAME Phase 1 output. These assumptions are made to combine many parts of a message.

1. Before a synchronous interaction from A to B, a sequence of asynchronous messages from A to B is assumed to be part of the “request” part of the interaction.
2. After a synchronous interaction from A to B, a sequence of asynchronous messages from B to A is assumed to be part of the “reply” part of the interaction.

Illustration 21 MsgSimplifier Pattern

The extra asynchronous interactions are simply removed from the output of SAME Phase 1.

```
A ClientA Client 2148115439781720 --- Pattern number 1
S ClientA ClientB 2148115439791720 2148115439791750
A ClientB ClientA 2148115439791751 --- Pattern number 2
S ClientB ClientA 2148115443363081 2148115443499661
```

Illustration 22 SAME phase 1 before MsgSimplifier applied

```
S ClientA ClientB 2148115439791720 2148115439791750
S ClientB ClientA 2148115443363081 2148115443499661
```

Illustration 23 SAME phase 1 after MsgSimplifier applied

However, applying MsgSimplifier on the data could lead into a possibility of losing meaningful asynchronous interactions which immediately precede or follow asynchronous ones.

MsgSimplifier is part of the Filtering Tools. The command-line below is needed to execute the MsgSimplifier with the SAME phase 1 result as the input file.

```
[user@pc FT] # java filteringtools.FilterTool 5 [SAME1result_File_Path]
```

Illustration 24 MsgSimplifier command-line for execution

5.7 Dealing with Non-Traced Components

In many systems there may be components that cannot be traced, but for which evidence of interactions is present in the trace. An example is the disk system, but it could equally be a remote server or client running in a host that cannot be traced because it is inaccessible, or because it does not run Linux (remember that LTTng tools are specific to Linux).

For a remote server or client, there may be evidence of a message, Its IP address is a key information to know which incomplete events are associated to the nontraceable components. In such a case, the Filtering Tools creates a task to represent the nontraceable component along with messages to represent its interactions based on the incomplete events. An example is found in the industrial case study, where an untraced controller task sends command messages that trigger the system activity,

For local devices which should be modelled, such as a disk, system calls provide the evidence of interactions. In such a case the Filtering Tools creates a model task to represent the device and messages to represent its interactions. Therefore, Filtering tools deal with the nontraceable components of a system and help SAME to extract a more complete LQN model from a whole system.

Chapter 6: The Method

Chapter 6 describes the entire process to build a model end to end, illustrated by an example called the “Tutorial Case”. In the example, Java process ClientA sends a message to a second process ClientB, which gives a reply to ClientA. The example is run in two different machines, which is ClientA executed in HostA, and ClientB executed in HostB. The Java code was written to implement the architecture shown in Illustration 25.

Illustration 25 Expected model structure of Java tutorial case

6.1 Overall Workflow of the Method

Overall there are three stages to the method,

1. Data collection: run the program with LTTng tools, Perf-execsnoop and CPUStat
2. Data cleaning and SAME Phase 1
3. Intermediate transformations and SAME Phase 2 (ExtractModel)
4. Create a deployment for the model

This workflow is illustrated with details in Illustration 26.

Illustration 26 Model Building work flow with SAME

Illustration 26 shows the overall work flow in extracting LQN model. First, in data collection stage, LTTng tools, “execsnoop”, CPUStat are the data collection tools that are

required to be started before the program is run. After running the software, the collected data are LTTng traces, execsnoop result and CPUStat result.

The second stage is SAME Phase 1 which extracts the interaction trace. In this stage, the Filtering Tools are used to bridge the collected data into SAME. For the starting point, as LTTng traces are collected from two different machines, the traces need to be synchronized using TraceCompass and exported into a CSV-type file. Then, using Filtering Tools, the synchronized data is analyzed for networking events and file system events, and it converted into SAME input file. Using SAME Phase 1, the input data is processed.

The third stage modifies the SAME Phase 1 result by combining parts of messages, and determining the CPU time at events and adding it to the interaction trace. Then SAME phase 2 is used to extract the LQN model structure.

The model that is output by SAME has all the tasks deployed on a single artificial processor. In the fourth stage this is replaced by a more realistic deployment to give a useful model. Then, the final result is a calibrated LQN model.

6.2 Executing the Program

Configure LTTng Tools

LTTng tools start-up and configuration is the first step. This includes the execution of LTTng daemon, the creation of LTTng session and the buffer size configuration for the session. The command-line is shown below.

```
[root@pc ~] # lttng-sessiond --daemonize
[user@pc ~] # lttng create session_name
[user@pc ~] # lttng enable-channel -kernel -subbuf-size 16777216 channel_name_k
[user@pc ~] # lttng enable-channel -userspace -subbuf-size 1777216 channel_name_us
```

Illustration 27 LTTng tools configuration

The ‘-- daemonize’ option is to make sure the LTTng daemon is started in background. A tracing session must be created after the daemon is started, it gives the user control of the LTTng daemon and starts the tracing. In a tracing session, there are channels for user-space tracepoints and channels for Kernel tracepoints. As many channels as desired, of either type, can be created. Every channel has its sub-buffer size to store the

traces from specific tracepoints. The bigger the sub-buffer, the lower the risk of losing event records. LTTng tools tend to overwrite the stored traces if the sub-buffer is limited. Especially when tracing Kernel tracepoints, it requires a big sub-buffer. In this case, a Kernel channel has been configured with 16 MB sub-buffer size, and there is no overwriting event record happened in minutes. For a user-space channel, it may not require such a big sub-buffer as it has a less event records, but for a safety reason, it is configured with 1 MB sub-buffer-size.

Configure Kernel Tracepoints

LTTng must be configured to capture the tracepoints needed for modeling. The present experiments use tracepoints for `read()`, and `write()` system calls, and `net_dev_queue()` Kernel events. The traces can be very large, so the number of tracepoints should be minimized to minimize the required data storage space. The tracepoints will be configured in Kernel channel ‘channel_name_k’ (Illustration 28). The command-line below shows how to configure the Kernel tracepoints.

```
[user@pc ~] # lttng enable-event -k --channel channel_name_k --syscall
                read,write
[user@pc ~] # lttng enable-event -k --channel channel_name_k
                net_dev_queue
```

Illustration 28 Kernel tracepoints configuration

In the Kernel channel configuration, ‘read’ and ‘write’ events have to be defined under system call option (‘-- syscall’) and the networking tracepoint ‘net_dev_queue’ is defined under the Kernel option. After executing the commands above, the Kernel tracepoints have been enabled and LTTng only records the specified tracepoints and stores them into the Kernel channels.

Configure Application Tracepoints

The purpose of creating LTTng user-space tracepoints (UST) is to obtain the information about the application context, such as the context or content of a message. The Kernel is unaware of this context information.

LTTng has a feature to create and trace user-space tracepoints which need to be

instrumented in the source code. LTTng deals with UST written in C/C++ and in Java. Here the additional configuration steps below are explained for a Java program.

1. Instrument user-space tracepoints in the Java source code, as in the example in Illustration 29.
2. Configure LTTng tools to recognize the user-space tracepoints
3. Introduce LTTng-compatible probes into the source code with the ‘java.util.logging’ (JUL) libraries.

To trace the application using the JUL logging facility, some codes are required to be instrumented in the Java source file. The ‘java.util.logging.*’ (JUL) libraries provided by LTTng for instrumenting user-space tracepoints are imported, and using the library, the source code must define Logger and Handler objects. Of these, the Logger object must be called at log points and the Handler object processes the log events to generate and save them.

Illustration 29 shows an example of user-space tracepoints instrumentation on the Example.java source file. Importing JUL libraries to the source file is the first step. By calling the `Logger.getLogger()` method, the Java agent initializes the ‘TracepointNameString’ as the user-space tracepoint identifier, and its communication with the LTTng daemon. The ‘`LttngLogHandler()`’ method is called to create an LTTng user-space tracepoint log handler. The handler is added to the ‘tracepointNameString’ logger, and the `logger.info()` method is called to log events using the ‘TracepointNameString’ logger created before. The LTTng user-space traces return the timestamp, the channel name, the string which is an argument of the ‘`logger.info`’ method (in this case “this is log1”), along with the ‘TracepointNameString’ identifier, class name and the instrumented method name as shown in Illustration 36 [19] .

Example.java

```
import java.util.logging.Handler;
import java.util.logging.Logger;
import org.lttng.ust.agent.jul.LttngLogHandler;
public class Example{
    private static final Logger logger =
        Logger.getLogger("TracepointNameString");
    public static void main(String[] args)
    {
        Handler lttngHandler = new LttngLogHandler();
        logger.addHandler(lttngHandler);
        logger.info("this is log1");
        String msg ="msg"
        logger.info(msg);
    }
}
[user@pc ~] # javac -cp /usr/local/share/java/lttng-ust-agent-
common.jar:/usr/local/share/java/lttng-ust-agent-jul.jar
example.java
```

Illustration 29 LTTng UST instrumentation and Java files compilation

The ‘TracepointNameString’ logger must be enabled in the LTTng configuration or all instrumented user-space loggers can be enabled in user-space channel as shown in Illustration 31.

To configure LTTng for user-space tracing, the LTTng UST module must be installed and configured. The following command-line does this for Red Hat or Centos Linux. The command-line may be different for different Linux distribution [19] .

```
[user@pc ~] # cd $(mktemp -d) &&
> wget http://lttng.org/files/lttng-ust/lttng-ust-latest-2.10.tar.bz2 &&
> tar -xf lttng-ust-latest-2.10.tar.bz2 &&
> cd lttng-ust-2.10.* &&
> ./configure --enable-java-agent-jul &&
> make &&
> sudo make install &&
> sudo ldconfig
```

Illustration 30 LTTng UST module installation and configuration

Once LTTng is instrumented as in Illustration 29 and configured for user-space tracing as in Illustrations 30 and 31, the user-space tracepoints are enabled on their own channel.

```
[user@pc ~] # lttng enable-event --jul --channel channel_name_us
--all
```

Illustration 31 UST(s) configuration

Here, the user-space tracepoints are enabled in the channel ‘channel_name_us’, after the command-line in Illustration 31 is executed. The ‘--all’ option stands for activation of all user-space tracepoints that are being instrumented in the source file. It is equally possible to define separate channels for different groups of events [19] .

Configure LTTng context

By default, LTTng only returns a few parameters defined for each tracepoint. These do not include process Id and the process name. Additional parameters can be recorded by adding some additional configuration data, which the LTTng documentation calls the “context”.

```
[user@pc ~] # lttng add-context --jul -t procname
[user@pc ~] # lttng add-context --kernel -t tid -t pid -t procname
```

Illustration 32 LTTng add-context configuration

The command-lines in Illustration 32 need to be the last configuration commands before LTTng starts tracing.

Execution of the Run

CPUStat must be started before initiating the main program, and periodically records the total cumulative CPU time (user + system) for each process from the Linux “proc” file system; here a period of 1 second was used. To execute CPUStat, the command-line below is executed.

```
[user@pc ~] # sh cpustat.sh
```

Illustration 33 CPUStat tools - execution

LTTng was started in background before the main program is executed. The command-line below will start the LTTng recording.

```
[user@pc ~] # lttng start
```

Illustration 34 LTTng command-line : start tracing

Perf-tools (as described in Section 2.5) was also initiated before starting the main program. The command-line is shown in Section 5.1 Illustration 12.

Then, the application was executed and the tracing and CPUStat results were stored on the file system.

When sufficient data has been recorded, the data-gathering tools are stopped. If you go too long, LTTng will start over-writing the buffer data. CPUStat and Perf-tools are stopped by using ‘CTRL+C’ interruption, and LTTng is stopped using the command-line below.

```
[user@pc ~] # lttng stop
```

Illustration 35 LTTng command-line : stop tracing

Illustration 36 shows the fragment of LTTng application level raw traces based on the ‘TracepointNameString’ instrumented tracepoint in Illustration 29.

LTTng Application Level Trace Fragment

```
vm01-20190130-141050/ust/uid/1000/64-bit 14:12:12.511 033 572 lttng_jul_channel_2
2      lttng_jul:event msg= this is log1, logger_name= TracepointNameString,
class_name=Example, method_name=main, long_millis=1548875532500,
int_loglevel=800, int_threadid=1, context._procname=java
```

Illustration 36 LTTng application level (UST) raw traces fragment

6.3 Filtering the Trace Data, and SAME Phase 1

The collected LTTng trace files from the various distributed compute nodes are next imported into Trace Compass, and TraceCompass is used to synchronize them (as described in Section 6.1) and export the combined trace to a CSV-type file. A filter was developed to convert the CSV file into the required input file format for SAME1. Then, the filtered data becomes the SAME phase 1 input file.

For the tutorial case, Illustration 37 shows a fragment of the LTTng trace including file operation events e.g. `syscall_entry_read`, `syscall_exit_read`, and TCP networking events e.g. `net_dev_queue`. Perf-tools result fragments are shown in Illustration 38 and Illustration 39. In Illustration 40, it shows the Filtering tools result on file operations and TCP messages.

With regards to the file operations the filtering tools also re-interpret the trace events into a form suitable for the model-building step in SAME, as explained in Section 5.3. In the tutorial case the `'syscall_entry_read'` event is recorded when `'ClientA'` starts accessing the disk to do read operation, which is interpreted as two messages events, first that `'ClientA'` sends a read-access request to the disk (Illustration 40 row no.1), and second that the disk receives the request (Illustration 40 row no.2). A dummy task is created called a “file service” that represents the disk, which in the example is `'read_FileService_vm01'`, and a “message identifier” is attached to both message records. Similarly, a `'syscall_exit_read'` event occurs when `'ClientA'` finishes the read operation and exits the reading state. Filtering tools considers it as a reply from the disk that granted the read-request access and converts it into two events, which are: `'read_FileService_vm01'` sends a reply (Illustration 40 row no.3), and `'ClientA'` receives the reply (Illustration 40 row no.4).

Three `'net_dev_queue'` events are shown in Illustration 36. Using the matching parameters explained in Section 5.2, Filtering tools converts the first event into `'ClientA'` sending a message (see Illustration 40 TCP part, row no.1) and received by `'ClientB'` (see Illustration 40 TCP part, row no.2) that converted from the second event. Whereas, the second and third events are converted into `'ClientB'` sends a reply-message to (see Illustration 40 TCP part, row no.3) and is received by `'ClientA'` (see Illustration 40 TCP part, row no.4).

```

HostA-20181210-214817/kernel 21:48:11.543 979 175    channel_0    0
        syscall_entry_read      fd=5, count=4096, context._tid=5988, context._pid=5989,
context._procname=java

HostA-20181210-214817/kernel 21:48:11.544 124 930    channel_0    0
        syscall_exit_read       ret=4096, buf=140284204814592, context._tid=5988,
context._pid=5989, context._procname=java

HostA-20181210-214817/kernel 21:48:19.020 095 202 channel_0 0 net_dev_queue
skbaddr=0xffff9602fba0d500, len=74, name=lo, network_header_type=_ipv4,
network_header=ipv4=[version=4, ihl=5, tos=0, tot_len=60, id=0x0, frag_off=16384, ttl=64,
protocol=_tcp, checksum=0xc3c0, saddr_padding=[], saddr=[192, 168, 122, 213], daddr_padding=[],
daddr=[192, 168, 122, 1], transport_header_type=_tcp, transport_header=tcp=[source_port=1099,
dest_port=38668, seq=3799395175, ack_seq=259561370, data_offset=10, reserved=0, flags=0x12,
window_size=43690, checksum=0x772a, urg_ptr=0]], context._tid=5988, context._pid=5989,
context._procname=java

HostB-20181210-214817/kernel 21:48:19.020 540 095 channel_0 0 net_dev_queue
skbaddr=0xffff9602fba0d400, len=66, name=lo, network_header_type=_ipv4,
network_header=ipv4=[version=4, ihl=5, tos=0, tot_len=52, id=0x8915, frag_off=16384, ttl=64,
protocol=_tcp, checksum=0x3ab3, saddr_padding=[], saddr=[192, 168, 122, 1], daddr_padding=[],
daddr=[192, 168, 122, 213], transport_header_type=_tcp, transport_header=tcp=[source_port=38668,
dest_port=1099, seq=259561370, ack_seq=3211395176, data_offset=8, reserved=0, flags=0x10,
window_size=342, checksum=0x7722, urg_ptr=0]], context._tid=3588, context._pid=3589,
context._procname=java

HostA-20181210-214817/kernel 21:48:21.878 138 667 channel_0 0 net_dev_queue
skbaddr=0xffff9602fcd0d500, len=74, name=lo, network_header_type=_ipv4,
network_header=ipv4=[version=4, ihl=5, tos=0, tot_len=60, id=0x0, frag_off=16384, ttl=64,
protocol=_tcp, checksum=0xc3c0, saddr_padding=[], saddr=[192, 168, 122, 213], daddr_padding=[],
daddr=[192, 168, 122, 1], transport_header_type=_tcp, transport_header=tcp=[source_port=1099,
dest_port=38668, seq=3211395176, ack_seq=223261340, data_offset=10, reserved=0, flags=0x12,
window_size=43690, checksum=0x772a, urg_ptr=0]], context._tid=5988, context._pid=5989,
context._procname=java

HostB-20181210-214817/kernel 21:48:19.023 069 126    channel_0    0
        syscall_entry_read      fd=4, count=30, context._tid=3588, context._pid=3589,
context._procname=java

HostB-20181210-214817/kernel 21:48:19.023 075 749    channel_0    0
        syscall_exit_read       ret=30, buf=135274204812462, context._tid=3588,
context._pid=3589, context._procname=java

```

Illustration 37 LTTng traces fragment

```

Tracing exec()s. Ctrl-C to end.
Instrumenting sys_execve
  PID  PPID ARGS
  5222  5223 date +%y%m%d_%H
  5122  5123 sleep 1
  5989  3233 java ClientA —
Ending tracing...

```

These identifies Client A are executed in Host A with process id 5989 Client A

Illustration 38 Perf tools result fragment for HostA

```

Tracing exec()s. Ctrl-C to end.
Instrumenting sys_execve
  PID  PPID ARGS
  5362  5361 date +%y%m%d_%H
  5363  5359 sleep 1
  3589  3464 java ClientB —
Ending tracing...

```

This identifies that the ClientB are executed in Host B with given Process ID 3589

Illustration 39 Perf tools result fragment for HostB

Filtering result – File operations

```

2148115439791750 ClientA send FS01852
2148115439791751 read_FileService_vm01 receive FS01852
2148115441249300 read_FileService_vm01 send FS01852
2148115441249301 ClientA receive FS01852
2148190230691260 ClientB send FS03842
2148190230691261 read_FileService_vm02 receive FS03842
2148190230757490 read_FileService_vm02 send FS03842
2148190230757491 ClientB receive FS03842

```

Filtering result – TCP

```

2148190200952020 ClientA send NT023
2148190205400950 ClientB receive NT023
2148190205400950 ClientB send NT034
2148218781386670 ClientA receive NT034

```

Illustration 40 Filtering tools result of example program

6.4 Extracting the Model with SAME

To extract an LQN model structure from the filtered traces, the process using SAME has three steps as follows,

1. extract the synchronous and asynchronous interactions (using the program SAME Phase 1)

2. eliminate extra asynchronous message by MsgSimplifier and add the CPU time used by each process up to the interaction moment (found by a separate step involving the CPUstat program)
3. from the components and interactions, build the model including its CPU demand and interaction frequency (call rate) parameters (using the program ExtractModel, evolved from the earlier program SAME2).

SAME Phase 1 Processing

The filtered trace is input to SAME. Phase 1 identifies matching calls and replies for synchronous messages and produces a derived trace of synchronous and asynchronous interactions.

For the tutorial case a fragment of the interaction trace is shown in Illustration 41. This trace corresponds to the Filtering Tools result in Illustration 40.

SAME Phase 1 result

```
S ClientA ClientB 214811543979172 214811543979175
S ClientA read_ FileService_vm01 214811543979175 214811544124930
S ClientA read_ FileService_vm01 214811544336308 214811544349966
S ClientA read_ FileService_vm01 214812736194746 214812736198654
S ClientB read_ FileService_vm02 214812736314398 214812736319152
S ClientA ClientB 214819020095202 214819020540095
S ClientA read_ FileService_vm01 214814885559903 214814885566835
S ClientB read_ FileService_vm02 214819022966325 214819022974535
S ClientB read_ FileService_vm02 214819023018574 214819023034369
S ClientB read_ FileService_vm02 214819023069126 214819023075749
```

Illustration 41 Interaction Trace from SAME Phase 1 for the tutorial case

After SAME Phase 1, the CPU times at each event were found as described in Section 5.5 and inserted into the interaction trace, and message-combining was applied to multipart messages as described in Section 5.6.

```

S ClientA ClientB 214811543979172 214811543979175 NULL 240000000 240000000
S ClientA read_ FileService_vm01 214811543979175 214811544124930 NULL 543979175 544124930
S ClientA read_ FileService_vm01 214811544336308 214811544349966 NULL 544336308 544349966
S ClientA read_ FileService_vm01 214812736194746 214812736198654 NULL 736194746 736198654
S ClientB read_ FileService_vm02 214812736314398 214812736319152 NULL 736314398 736319152
S ClientA ClientB 214819020095202 214819020540095 NULL 260000000 260000000
S ClientA read_ FileService_vm01 214814885559903 214814885566835 NULL 885559903 885566835
S ClientB read_ FileService_vm02 214819022966325 214819022974535 NULL 022966325 022974535
S ClientB read_ FileService_vm02 214819023018574 214819023034369 NULL 023018574 023034369
S ClientB read_ FileService_vm02 214819023069126 214819023075749 NULL 023069126 023075749

```

Illustration 42 SAME Phase 1 result with CPU time added

SAME Phase 2 Processing

The interaction trace with or without CPU times is input to Phase 2, which is a program called ExtractModel. If CPU times are absent, the CPU demands of entries will be set to 10^{-6} . Phase 2 has two processing options, settable in the source code at this point:

- a direction of interaction can be imposed on messages between any pair of tasks. If the architecture imposes a client-server relationship, for instance, the interactions can be constrained to only detect interactions initiated by the client. Messages from the client to the server will be interpreted as replies to earlier requests if possible, or ignored. This feature needs to be used with care. It is useful to suppress initialization interactions which may be structured differently, which clutter the model and which often are of less interest in a performance model.
- operation-executions during an interaction are interpreted as entries in the model. The gathering of similar entries into one model entry is important, as there may be thousands of repetitions of one operation. It can be controlled to either gather all the entries of each task into one entry, or to gather only entries that initiate the same set of interactions (by destination and type). The second option may create a model with many entries in some tasks.

The output of Phase 2 is an LQN model. For the example, with one entry per task, this was the result.

Illustration 43 SAME result for the tutorial case

The output LQN model has all tasks deployed on one processor, since SAME does not use the host information (this could be improved). To make a model of a deployed application, suitable processors should be added. Also, the CPU demand parameter calibration made using the CPUstat data gives a demand value for the processor that the task is running on. If some processor that was used in the experiment is faster than another (and if we need a portable model that is not processor-specific), a speed factor should be derived for it and the CPU demand values for entries that run on that processor should be multiplied by the speed factor (the same time spent executing on a faster processor implies a larger demand for execution cycles).

Illustration 44 Resource modification on LQN model for the tutorial case

Illustration 44 shows the modified LQN model, and it matches with the expected LQN model in Illustration 25.

Chapter 7: Client-Server Case Study

Chapter 7 introduces a Layered Client-Server (LCS) sample program and using SAME extracts its LQN model structure based on LTTng traces.

7.1 Description

The LCS implements a layered system that is a common architectural pattern in distributed systems. Java socket programming was used for communications in the LCS. There are 6 tasks including a User and five Servers. The LCS is run in two hosts, HostA has the User, Server01, Server03 running, and Server02, Server04 and Server05 are run in HostB. During initialization the User invokes the Servers in both hosts, to set up the connection. Server01 sends random requests to Server02 and Server03. When it gets a request, Server02 sends a random number of requests to Server04 and Server05.

7.2 Model Expectation

The software design is interpreted to give the expected model. The high-level architecture of LCS is illustrated by an LQN model structure in Illustration 45. It will later help us to validate to what degree the extracted model corresponds to the expected model.

Illustration 45 LCS model based on the software structure

7.3 The Collected Data

The data that were collected are as follows

- LTTng Kernel traces for networking events
- CPUStat data at one-second intervals
- Perf-tool data for Java processes name

Using the tools explained in Chapter 5, the LTTng trace data were collected with focusing on the networking events. The collected trace size was approximately 15 megabytes, plus 2 megabytes collected data for CPUStat data and Perf-tool data.

In the model extraction steps (explained in Chapter 6), some data cleaning was used including:

- removing interactions between a process and itself, these must be messages between process threads, which we are not modeling.
- removing requests directed towards the User task by introducing a constraint in SAME phase 1 that the User task cannot receive requests.

Next, using ExtractModel, the cleaned data were processed and returned an LQN model structure.

7.4 SAME Result

The SAME tools generate the model shown in Illustration 46. With CPU demand parameter calibration, ExtractModel also generates the service demands of the entries. A task that is considered as source task, never received calls, and is located in the top of the diagram. As designed, ‘user_source’ makes calls into all Servers tasks for invocation.

Illustration 46 SAME result on LCS (Time in seconds) – Appendix D

As a result, the LCS model extracted by SAME and shown in Illustration 46 has some differences compared to the LCS software structure model in Illustration 45. The extracted model has additional entries, which appear to represent initialization operations. However, the extracted model shows what system really doing. Also the extracted model has some extra messages, and the calling pattern is a little different from the LCS software structure model.

7.5 Lessons Learned

A great deal was learned in modeling the layered Client-Server system using TCP Kernel events and LTTng. The main findings are,

- Layered system modeling: the trace synchronization feature on TraceCompass was essential to make use of multiple traces from different computers.

- The capability to trace inter-process messages using TCP was completely sufficient for socket-based communication using the TCP algorithm (Section 5.2).
- Extra messages in Kernel traces: the initialization messages for establishing connection between a server and User are also recorded on the traces. Server sends a signal to User before User is connected to Server. Those signals may lead to misidentification of communication pattern. Removing the initialization messages could be helped by application level traces. However, on this layered system, a constraint feature on SAME tools helps to remove some of initialization messages.
- Fragmentation of messages: the TCP protocol divides a message into chunks before sending. The example required the use of the MsgSimplifier algorithm on Filtering Tools which combines parts of a message. (Section 5.6).
- Incapability to obtain application context on LTTng traces. JVM encapsulates the Java processes name and LTTng traces record 'Java' as the process name for all Java processes along with the process id. In this experiment, the Perf tools was necessary to get additional information about application context associated with process id (Section 5.1).

Chapter 8: Java File Transfer Case Study

Chapter 8 explains the result of model building on a program which transfers files and sorts the data, and then applies the model to predict the time to transfer files and sort of different sizes.

8.1 Description

A Java File Transfer program (JFT) was developed that focuses on file transfer feature between two different machines. JFT uses Remote Invocation Method (RMI) that designed for high level Java to Java distributed computing. RMI adapts parallel computing that allows servers threads to be utilized concurrently in serving client requests, and it also uses built-in Java security mechanisms that protect the system from hostile applets while downloading the implementations.

Illustration 47 JFT Scenario

As shown in the Illustration 47 above, JFT is designed with a controller and two clients. The Controller sets up the system and triggers the file transfer by ClientA. Once Client A receives the file-transfer command, Client A generates a file that filled with random integer numbers, and then Client A sends the file to Client B which is located in another machine. ClientB receives the file, stores the file, and then, do the sorting operation on the file and stores the sorted version into the disk. Basically, the experiments were performed in two different virtual machines running Linux. Controller and Client A are

executed in host A, and the Client B is executed in host B. However, as it is RMI implementation, RMI registry should be executed in two hosts in order the machines interact with each other using Java RMI application. With regards to acknowledgement towards Controller, RMI mechanism has a request-reply protocol which generates the acknowledgements for every operation done by Client A and ClientB.

8.2 Model Expectation

The high-level architecture of JFT is described by an LQN model in Illustration 48. It helps to check to what degree the extracted model corresponds to the model expectation.

Illustration 48 JFT expected model

8.3 Data cleaning

Data cleaning is a mandatory step to overcome the LTTng Kernel tracing limitation. It includes,

- removing interactions towards the process itself on SAME phase 1 input and ExtractModel input)
- removing calls toward a source task by introducing SAME phase 1 with a constraint term for the task (e.g. {controller,*}) on SAME phase 1 input.
- If user-space tracepoints are used, removing initialization interactions between a range of time on SAME phase 1 input.
- Inserting dummy invocation interactions from Controller to Clients on ExtractModel input.

8.4 JFT experiments on transferring 10 KB file

Some experiments were conducted on JFT using Kernel events and application events. The obtained data are,

1. LTTng traces for messaging events and file service events.
2. Perf tools result for Java processes name.
3. LTTng application level traces for activities.
4. CPUSat data at one second intervals.

8.4.1 LQN model using Kernel events

The JFT was traced using LTTng with Kernel events enabled, and the glimpse of the collected data can be found in Appendix A.1 – Appendix A.4. The traces have gone through model extraction stages (explained in Chapter 6) along with the CPU demand parameter calibration stage. Two types of LQN model have been generated using the rules in ExtractModel (Chapter 3 Section 3.2). The LQN performance models in text form are in Appendix A.5, and Appendix A.6, and the graphical versions are displayed in Illustrations 49 and 50 below.

Illustration 49 JFT LQN model using Kernel events with 'mergebyCallSet' (Time in seconds) – Appendix A5

Illustration 50 JFT LQN model using Kernel events with ‘mergebyTask’ rule (Time in seconds) – Appendix A6

Two rules affect the generated results. The ‘mergeByCallSet’ rule tends to generate more entries, whereas the ‘mergebyTask’ generates one entry per task. The difference between the models is on the complexity of model. The first model (Illustration 49) has more entries, and it makes easier to identify the operations if the entries are separate since every entry represents different operations. The multiple entries version also has more information than the second model (Illustration 50).

The activities in JFT are expected, and it requires application level traces that will be explained in Section 8.4.2.

8.4.1.1 Concurrency Question

SAME assumes all processes are single threaded, to identify what messages are sent by an entry. This assumption permits the inference of causality between higher-level and lower level requests. Creating an LQN model that distinguishes task based on thread id would satisfy this assumption.

Illustration 51 Modeling based on thread id (Time in seconds) – Appendix C

Illustration 51 shows the extracted LQN model made from traces with thread id identified. The threads are created by the JVM, and mostly do the file operations. Some of threads seem to be killed after doing operations (e.g. clienta_4145), and those killed threads are followed by the new threads (e.g. clienta_4156) that are designed to do other operations.

The order of execution of the threads can be determined from the entry numbers in the model. These numbers come from the initial ordering of messages in the input file. The 'E_1S' entry is the entry triggered by the first message in the input file. Where an entry has been merged from a number of instances in the trace the first message number is the one that is on the entry.

The controller, apparently, spawns threads to do stages of processing. The 'controller_4163' task has 'E_1S' that makes a number of calls to Client A, and one to 'read01' entry on 'FileService_VM01' task. The same behavior was also detected on the 'E_2S' entry on the 'controller_4168' task. In addition, ClientA also spawns different threads to receive these messages. Some threads of ClientA such as 4145, 4146, 4156, 4158, 4166, receive messages from 'E_1S' entry of 'controller_4168' task.

With regards to file transfer, ClientB spawns threads to handle the upcoming file sent by ClientA. The 'E_24S' entry represents the first file chunk sent by ClientA to ClientB. In Illustration 51, the 'E_24S' entry on 'clientb_4039' receives one call from the 'E_5S' entry on the 'clienta_4146' which also makes 3913 calls to do read operations on FileService_VM01.

After receiving the file, ClientB spawns some threads to tackle the sorting operation. The thread id number 4007, for example, has three entries such as 'E_19854S', 'E_19799S', 'E19860S'. Those entries make a number of calls to do read and write operations on 'FileService_VM02'. The 'E_19799S' entry makes 150.7 calls to 'read02' entry which assumed to do read operations before doing the sorting. Then, the 'E19854S' thread makes 5 calls to 'write02' entry, followed by 15.5 calls sent from 'E_19860S' entry. The calls sent to 'write02' entry are assumed to be the writing operations of the sorted file content.

However, adding thread id to the LQN model leads to considerable complexity. There is clearly an advantage to cluster the tasks by process id for simplicity. There may be a loss of causal connection when there are overlapping operations by concurrent threads

of the same process.

8.4.2 LQN model using Kernel events and Activities (UST)

JFT was designed with activities in the ClientA and ClientB processes. The activities are as follows,

- ClientA : generatefile_start – generatefile finish

The pair of activities record the start point and the end point of the random file generation before sending it to ClientB. The length of random integer numbers are generated and written in a file.

- ClientA : send_start – send-finish

After the file is generated, the send activities imply the start point and end point of the file transferring operation on the ClientA side. Within the send activities time frame, the generated file is opened, read and sent through TCP/IP socket associated with ClientBd.

- ClientB : receive_start – receive_finish

The receive activities annotate the start and the end point of the file receiving operation on the ClientB side. Within the time frame of the receive activities, ClientB receives the file and stores the file into a local disk.

- ClientB : sort_start – sort_finish

After file received, the ClientB starts to move forward with the received file to perform sorting operation. The pair of sort activities annotate the start and the end point of sorting operation. Within the sort activities time frame, the received file is opened, read and sorted, then a new file with sorted integer number is created.

However, Kernel events do not record them because they are application level events. Therefore, JFT source code instrumentation was done to record the activities in the processes. These application level events were then recorded by LTTng.

Illustration 52 JFT LQN model with Activities (Time in seconds) – Appendix A7

Illustration 52 shows the extracted LQN model with activities identified. ClientA has 2 operations which are the operation of file generation and file transfer, and ClientB has the receiving operation and the sorting operation. ExtractModel takes an activity event as the beginning of an activity, which ends at the next activity. The read operation, for example, is started in the activity `read_start`, and the read operation is finished at the activity `read_finish`. Therefore, every operation is specified into two different activities which are `operation_start` and `operation_finish`. As a result, both ClientA and ClientB have 4 activities in total.

In terms of invocation, a new entry on both ClientA (E_10S) and ClientB (E_102S) are designed for receiving an invocation command from Controller. It is supported by the entry name of the invocation entry which annotates the initial ordering of messages of in the input file. The 'E_10S' entry is triggered by the 10th message in the input file, and it was when the Controller sent the first chunk of invocation command to ClientA, same applied to the 'E_102S'.

With regards to the file generation and the file transfer activities of ClientA, two entries (E_18S and E_94A) trigger the activities. Based on the entry names, the activities are started and finished on the 18th and the 94th messages on the input file, after ClientA receives an invocation command from Controller tackled by the 'E_10S'. The 'generate_start' activity is bound to the 'E_18S' entry, and it makes 30 calls to the read operation and 15 calls to the write operation. On 94th message on the input file, the file generation is finished, and it makes 3 calls to the read operation and one call to the write operation. File transfer is started after the file generation is done, and 44 calls towards the 'read01' entry are made by the 'send_start' activity. Besides file operations, the 'send_finish' activity also makes 2 calls to ClientB entries (E_106S and E_133S). One of these apparently actually sends the file to ClientB.

In ClientB, the 'E_106S' and 'E_133S' entries are bound to activity executions. The 'receive_start' activity on the 'E_106S' entry makes 22 calls and 19 calls to read and write operations, respectively. Whereas, the receiving operation is ended on the 133th message on the input file, and significant calls are made to do the reading operation along with 12 calls towards the 'write02' entry. The sorting operation, however, makes 228 calls and 10 calls to do read operation and write operation, respectively. So the sorting operation of ClientB (which does buffered file I/O by record) does much finer-grained file access, than the bulk read and write operations of both client tasks. Interestingly, while the generate_file and send activities of ClientA are dealing with the same file, they do a different number of operations. These are not features that would be detected by only using Kernel tracing.

The differences between the model with and without with activities are considered next.

8.4.3 Comparison between LQN model with and without activities

The model made only with Kernel tracing will be called the “Kernel model”, and with activities the “UST model”. Comparing the Kernel model shown in Illustration 49 and the UST model shown in Illustration 52, the differences are:.

- Kernel version includes more entries and messages, which from the message numbering are associated with initialization of the system (e.g. interactions happened while establishing connection) rather than with the operation performance. In the UST model analysis the events before the first activity were removed. Thus, application level events may help to remove unwanted details.
- UST version has additional user level information. It breaks the sequence of operations down according to the sequence of activities, and associates them with known parts of the operation. Thus we see that the receive operation of ClientB makes almost as many calls to read a file as to write it.

8.5 Performance Prediction on Larger File Size

The performance of JFT depends on the size of file being transferred and sorted. To be able to predict the performance for different file sizes the model parameters need to be understood as functions of the file size. For this purpose three cases were modeled, with file sizes 10 KB, 30KB, and 50KB, the variation of model parameters across the cases was examined, and the parameters were expressed as functions of file size. The resulting model was tested on a 100 KB file.

For fitting the parameter functions, the Least Squares approach was used. Microsoft Excel implements the approach in ‘TREND()’ and ‘LINEST()’ functions that requires data

that follows,

- Known dependent variable values (e.g. Call Frequencies or Service Demands)
- Known independent variable values (e.g. File sizes)
- New independent variable values (e.g. larger file size)

The call frequencies variable is extrapolated using the ‘TREND()’ function that performs linear regression. Whereas, the service demands variable is extrapolated using the ‘LINEST()’ function which implements a polynomial curve fitting method.

Service demands data were collected only on those entries which have service demands that change with file sizes.

Task	Entries	File Sizes (Unit in KB)			Predicted
		10	30	50	100
ClientA	E_10S	9.28E-03	1.33E-02	3.63E-01	2.75
	E_18S	8.73E-03	7.62E-02	6.08E-01	3.97
ClientB	E_102S	3.81E-03	5.96E-02	3.25E-01	1.91
	E_112S	2.88E-03	4.52E-02	3.00E-01	1.86
	E_125S	2.80E-03	4.52E-02	1.75E-01	8.85E-01
	E_106S	3.31E-03	9.49E-02	2.04E-01	5.51E-01
FS1	read1	3.45E-04	8.92E-04	3.14E-03	1.62E-02
	write1	8.43E-05	7.92E-05	5.61E-04	3.90E-03
FS2	read2	1.73E-04	7.71E-04	4.42E-03	2.69E-02
	write2	2.53E-05	6.91E-05	4.32E-04	2.74E-03

Table 2 JFT – Service demands prediction (Time in seconds)

Table 2 shows the collected service demands and the prediction result for the service demands variable of 100 KB experiment. The JFT 10 KB experiment is a basis of the entries name on Table 2. ClientA has two entries (E_10S and E_18S) that have service demands increases while transferring larger file sizes; whereas, ClientB has four entries. Supporting Table 2, Illustration 53 and 54 show the data trend which is quadratic for the FileService, ClientA and ClientB tasks, along with the extrapolated points in the red rectangle.

Illustration 53 JFT – Service demands predictions on FileService entries

Illustration 54 JFT – Service demands predictions on ClientA and ClientB entries

Illustration 53 shows the data trend of the service demands on the FileService entries, and it shows that the experiment data has a non-linear trend. Therefore, a quadratic function was fitted extrapolate the service demands for 100 KB file size.

For each entry listed on Table 2, the CPU demand parameter was derived as a function of file size, as the best quadratic fit line to three points, using the Micorosoft Excel

‘LINEST()’ function. The fitted functions for the parameters were:

1. ClientA – E_10S entry

$$SD_{E10S}(FS_{Kb}) = 0.00043(FS_{Kb})^2 - 0.017(FS_{Kb}) + 0.137$$

2. ClientA – E_18S entry

$$SD_{E18S}(FS_{Kb}) = 0.00058(FS_{Kb})^2 - 0.02(FS_{Kb}) + 0.149$$

3. ClientB – E_102S entry

$$SD_{E102S}(FS_{Kb}) = 0.00026(FS_{Kb})^2 - 0.008(FS_{Kb}) + 0.054$$

4. ClientB – E_112S entry

$$SD_{E112S}(FS_{Kb}) = 0.00027(FS_{Kb})^2 - 0.0085(FS_{Kb}) + 0.061$$

5. ClientB – E_125S entry

$$SD_{E125S}(FS_{Kb}) = 0.00011(FS_{Kb})^2 - 0.0022(FS_{Kb}) + 0.015$$

6. ClientB – E_106S entry

$$SD_{E106S}(FS_{Kb}) = 2.143 \times 10^{-5}(FS_{Kb})^2 + 0.0037(FS_{Kb}) - 0.036$$

7. FileServiceVM1 – Read01 entry

$$SD_{Read01}(FS_{Kb}) = 2.12 \times 10^{-6}(FS_{Kb})^2 - 5.8 \times 10^{-5}(FS_{Kb}) + 0.00071$$

8. FileServiceVM1 – Write01 entry

$$SD_{Write01}(FS_{Kb}) = 6.08 \times 10^{-7}(FS_{Kb})^2 - 2.5 \times 10^{-5}(FS_{Kb}) + 0.00027$$

9. FileServiceVM2 – Read02 entry

$$SD_{Read02}(FS_{Kb}) = 3.81 \times 10^{-6}(FS_{Kb})^2 - 0.00012(FS_{Kb}) + 0.0010$$

10. FileServiceVM2 – Write02 entry

$$SD_{Write02}(FS_{Kb}) = 4 \times 10^{-7}(FS_{Kb})^2 - 1.4 \times 10^{-5}(FS_{Kb}) + 0.00012$$

In terms of call frequencies, the data were collected for all interactions in the models Table 3 below shows the data and the prediction of call frequencies for every call in the model.

Calls Source-Target	FileSizes (Unit in KB)			Predicted
	10	30	50	
Controller-Read01	1	1	1	1
Controller-E_10S_Clienta	8	8	8	8
Controller-E_18S_Clienta	1	1	1	1
Controller-E_94A_Clienta	1	1	1	1
Controller-E_102S_Clientb	4	4	4	4
Controller-E_112S_Clb	1	1	1	1
Generate_start-Write01	15	20	26	40
Generate_start-Read01	30	34	40	52
Generate_finish-ClientB	1	2	3	6
Generate_finish-Read01	3	4	5	8
Generate_finish-Write01	1	1	1	1
Send_start-Read01	44	51	59	78
Send_finish-E_106S_Clb	1	1	1	1
Send_finish-E_133S_Clb	1	1	1	1
Send_finish-Read01	10	10	10	10
Receive_start-Read02	22	33	41	65
Receive_start-Write02	19	21	27	36
Receive_finish-Read02	228	231	243	260
Receive_finish-Write02	10	14	19	30
Sort_start-Read02	228	231	243	260
Sort_start-Write02	10	14	19	30

Table 3 JFT – Call frequencies prediction

Table 3 shows the data collected on the number of calls made on every interaction along with the predicted call frequencies for JFT 100 KB. The number of calls on interactions between Controller and the entries on ClientA and ClientB are consistent, and they are unaffected by the file sizes. However, the call frequencies on interactions between the ClientA activities (e.g. Generate_start, Generate_finish, Send_start and Send_finish), and the ClientB entries have a slight increase affected by file sizes. Whereas, the significant number of calls increases are detected on the FileService entries by the ClientA and ClientB

activities (e.g Receive_start, Receive_Finish, Sort_start and Sort_finish). Illustration 55 shows the linear data trend along with the predicted call frequencies on 100 KB file.

Illustration 55 JFT – Call frequencies prediction towards FileService entries

Relationships for predicting the call frequencies for different file sizes are defined as follows,

1. ClientA : Generate_start-Write01

$$CF_{GenerateStart,Write01}(FileSize_{Kb}) = 0.275(FileSize_{Kb}) + 12.08$$
2. ClientA : Generate_start-Read01

$$CF_{GenerateStart,Read01}(FileSize_{Kb}) = 0.25(FileSize_{Kb}) + 27.17$$
3. ClientA : Generate_finish-ClientB

$$CF_{GenerateStart,ClientB}(FileSize_{Kb}) = 0.05(FileSize_{Kb}) + 0.5$$
4. ClientA : Generate_finish-Read01

$$CF_{GenerateFinish,Read01}(FileSize_{Kb}) = 0.05(FileSize_{Kb}) + 2.5$$
5. ClientA : Send_start-Read01

$$CF_{SendStart,Read01}(FileSize_{Kb}) = 0.375(FileSize_{Kb}) + 40.1$$
6. ClientB : Receive_start-Read02

$$CF_{ReceiveStart,Read02}(FileSize_{Kb}) = 0.475(FileSize_{Kb}) + 17.75$$
7. ClientB : Receive_start-Write02

$$CF_{ReceiveStart,Write02}(FileSize_{Kb}) = 0.2(FileSize_{Kb}) + 16.34$$
8. ClientB : Receive_finish-Read02

$$CF_{\text{ReceiveFinish,Read02}}(FileSize_{kb}) = 0.375(FileSize_{kb}) + 222.75$$

9. ClientB : Receive_finish-Write02

$$CF_{\text{ReceiveFinish,Write02}}(FileSize_{kb}) = 0.225(FileSize_{kb}) + 7.58$$

10. ClientB : Sort_start-Read02

$$CF_{\text{SortStart,Read02}}(FileSize_{kb}) = 0.375(FileSize_{kb}) + 222.75$$

11. ClientB : Sort_start-Write02

$$CF_{\text{SortStart,Write02}}(FileSize_{kb}) = 0.225(FileSize_{kb}) + 7.58$$

Based on the parameters data on Illustration 53 and 55, file operations in Linux clearly adapt to the file size. It appears that the number of operations to read or write a file is not proportional to the file size but increases more slowly, with larger service demands for each operation. Therefore, to describe operations on different file sizes requires file sizes function on both the call rates and the service demands.

Using the LQN model structure of JFT 10 KB experiment as a basis, a model for 100 KB file size was created. The next section will explain about the predicted LQN model.

8.5.1 The Predicted LQN model

The model for 100 KB file was based on the LQN model structure of JFT 10 KB run (Illustration 52). In the calibrated LQN model structure, the predicted service demands and call frequencies were applied. With the LQN simulator tools (lqsim), the predicted performance of JFT transferring 100 KB file was obtained (Appendix B). Illustration 56 shows the service times of solved JFT predicted LQN model, and the throughputs and utilizations are shown in Illustration 57 below.

Service times:

Task Name	Entry Name	Phase 1	CI_Phase1	Phase 2	CI_Phase2
fileservice_vm01	write01	0.00390044	+/-99% 2.21013e-005	0	
	read01	0.0162007	+/-99% 5.02382e-005	0	
fileservice_vm02	write02	0.0027389	+/-99% 1.16266e-005	0	
	read02	0.0268999	+/-99% 4.67895e-005	0	
controller_Source	sourcecontroller	57.7971	+/-99% 1.98852		
clienta	E_10S	2.74758	+/-99% 0.0260741	0	
	E_18S	3.96558	+/-99% 0.165879	0.998204	+/-99% 0.0396842
	E_94A	21.6683	+/-99% 0.945862	0	
clientb	Activity Name				
	E_18S_lreply	3.96558	+/-99% 0.165879		
	generatefile_finish_A_2	6.52629	+/-99% 0.216722		
	generatefile_start_A_1	0.998204	+/-99% 0.0396842		
	send_finish_A_4	13.8774	+/-99% 0.981006		
	send_start_A_3	1.26453	+/-99% 0.0604975		
	E_106S	0.550234	+/-99% 0.0221629	1.85404	+/-99% 0.0945425
	E_125S	0.884258	+/-99% 0.0105547	0	
	E_102S	1.90547	+/-99% 0.0440736	0	
	E_112S	1.86545	+/-99% 0.0580362	0	
	E_133S	7.08261	+/-99% 0.257176	7.0902	+/-99% 0.257487
	Activity Name				
	E_106S_lreply	0.550234	+/-99% 0.0221629		
	receive_finish_A_6	7.08261	+/-99% 0.257176		
	receive_start_A_5	1.85404	+/-99% 0.0945425		
	sort_finish_A_8	0	+/-99% 0		
	sort_start_A_7	7.0902	+/-99% 0.257487		

Illustration 56 JFT predicted LQN model – service times (Time unit in seconds)

Throughputs and utilizations per phase:

Task Name	Entry Name	Throughput	CI_Throughput	Phase 1	CI_Phase1	Phase 2	CI_Phase2	Total	CI_Total
fileservice_vm01	write01	0.706368	+/-99% 0.0406554	0.00275513	+/-99% 0.000157317	0		0.00275513	+/-99% 0.000157317
	read01	2.57017	+/-99% 0.106222	0.0416385	+/-99% 0.00169587	0		0.0416385	+/-99% 0.00169587
Total:		3.27654	+/-99% 0.129648	0.0443936	+/-99% 0.00170315			0.0443936	+/-99% 0.00177096
fileservice_vm02	write02	1.6658	+/-99% 0.0793148	0.0045625	+/-99% 0.00022317	0		0.0045625	+/-99% 0.00022317
	read02	10.1558	+/-99% 0.526142	0.273191	+/-99% 0.0141782	0		0.273191	+/-99% 0.0141782
Total:		11.8216	+/-99% 0.60005	0.277753	+/-99% 0.01418			0.277753	+/-99% 0.0143831
controller_Source	sourcecontroller	0.0173044	+/-99% 0.000594681	1				1	
clienta	E_10S	0.138135	+/-99% 0.00357494	0.379536	+/-99% 0.00967714	0		0.379536	+/-99% 0.00967714
	E_18S	0.0172496	+/-99% 0.000787968	1.60274e-06	+/-99% 0.00348907	1.3987e-07	+/-99% 0.00103072	0.0856175	+/-99% 0.00163814
	E_94A	0.0172611	+/-99% 0.000567015	2.08774e-05	+/-99% 0.0125927	0		0.373974	+/-99% 0.0125927
	Activity Name								
	E_18S_lreply	0.0172496	+/-99% 0.43141	0.068399	+/-99% 0.00348907				
	generatefile_finish_A_2	0.0172611	+/-99% 0.311235	0.112648	+/-99% 0.00479018				
	generatefile_start_A_1	0.0172496	+/-99% 0.431358	0.0172185	+/-99% 0.00103072				
	send_finish_A_4	0.0172611	+/-99% 0.310567	0.239497	+/-99% 0.0137142				
	send_start_A_3	0.0172611	+/-99% 0.311235	0.0218285	+/-99% 0.00143156				
	Total:	0.172646	+/-99% 0.00379895	0.821909	+/-99% 0.0162603	0.0172185	+/-99% 0.00103072	0.839127	+/-99% 0.00464328
clientb	E_106S	0.017279	+/-99% 0.00081526	4.5825e-08	+/-99% 0.000589971	6.84149e-07	+/-99% 0.00227958	0.0415438	+/-99% 0.00235468
	E_125S	0.102736	+/-99% 0.00505744	0.0908444	+/-99% 0.00451736	0		0.0908444	+/-99% 0.00451736
	E_102S	0.0688671	+/-99% 0.00201624	0.131223	+/-99% 0.00458858	0		0.131223	+/-99% 0.00458858
	E_112S	0.0171971	+/-99% 0.00082191	0.0320807	+/-99% 0.00185829	0		0.0320807	+/-99% 0.00185829
	E_133S	0.0173367	+/-99% 0.000801859	8.44284e-06	+/-99% 0.00800799	7.32702e-06	+/-99% 0.00746007	0.245717	+/-99% 0.0109444
	Activity Name								
	E_106S_lreply	0.017279	+/-99% 0.446732	0.00950748	+/-99% 0.000589971				
	receive_finish_A_6	0.0173367	+/-99% 0.439139	0.122795	+/-99% 0.00800799				
	receive_start_A_5	0.017279	+/-99% 0.446536	0.0320363	+/-99% 0.00227958				
	sort_finish_A_8	0		0					
Total:		0.0173367	+/-99% 0.439196	0.122922	+/-99% 0.00746007	0.154958	+/-99% 0.00780058	0.541408	+/-99% 0.0105994
		0.223416	+/-99% 0.00461226	0.38645	+/-99% 0.010459				

Illustration 57 JFT predicted LQN model – Throughputs and Utilizations

Prediction errors

The prediction performance of 100 KB LQN model were measured by comparing the measured response time of the entire JFT's operation and the predicted response time of the entire JFT's operation. The measured response time is identified from the start time and end time of the traces. The predicted response time is the service time of the controller

task (Illustration 56) which has a 99% confidence interval of 57.79 ± 1.98 seconds.

ResponseTime (seconds)	
Measured	Predicted
57.6711	57.7971 ± 1.98
Error : 0.22%	

Table 4 Prediction errors on 100 KB file size

The error, shown in Table 4, is about 0.22%. The measured time falls within the predicted confidence interval. This is excellent agreement.

8.6 Lessons Learned

In addition to the lessons in Chapter 7, more findings here include:

- The capability to trace file operations using Systemcall events described in Section 5.3 was sufficient for programs communicating by sockets.
- There is more detail in Kernel traces than we need: the initialization messages for establishing connection between ClientA/B and Controller are recorded in the traces. ClientA sends a signal to Controller before Controller is connected to ClientA. Those signals complicate the model and lead to misidentification of communication patterns. Removing the initialization messages is helped but required including user-space events in the traces.
- On the same point an architectural constraint feature of ExtractModel (imposing a direction of calling between tasks) was useful to remove some initialization messages.
- SAME tools consider the processes are single-threaded. To extract an LQN model which identifies concurrent threads, the thread identity must be taken from the trace. However Illustration 51 shows that this leads to considerable complexity. A solution might be to exploit information about thread creation and termination, to eliminate threads which do not provide actual concurrency. (This is a limitation of SAME).
- File operations in Linux clearly adapt to the file size, so the number of operations to read or write a file is not proportional to the file size, but increases more slowly (with a longer time for each operation). Therefore to describe operations on different file sizes requires making both of these

parameters a function of file size. This emphasizes the value of model calibration based on tracing.

Chapter 9: Industrial Case Study

Chapter 9 explains the LQN model extraction from execution of an real industrial telecommunication software system product. The main aim of the case study is to show that it is feasible to extract an LQN architecture from LTTng traces that are collected from the real industrial real-time software. A second aim, which was not realized, was to completely model the system and use the model to study it, however this was frustrated by events. The first aim was achieved.

9.1 The Expected Model of the Telecommunication System

Because of the importance of continuous connectivity, telecommunication equipment should never face any downtime. However, the software must be upgraded regularly to provide new features, or product enhancements. A complete software upgrade generally requires the system to be rebooted to apply the changes. To avoid any system downtime, a ‘rolling upgrade’ is applied to the primary and a secondary resource that provided for reliability. The secondary resources (e.g., a processor card02) are upgraded first and tested. If everything goes as expected, then the primary resource (e.g., card01) is upgraded while the secondary acts as primary. In this research, the upgrade software was traced for LQN model extraction.

Using the knowledge of the system gained from documentation and conversations with developers, the expected model was developed as in Illustration 58. This can serve as a reference to critique the model based on tracing.

Illustration 58 Industrial Case Study: Expected Model

‘Upgrd_sw’ tasks represent the upgrade software in the host processors called Card01 and the Card02. The ‘Upgrd_sw’ on Card01 has 20 states while doing Card01 upgrade, then the processing switches to Card02 which has 7 processing states in order to be upgraded. The Card01 FileSystem pseudotask labelled FileSystem_{Primary} in the Figure will be called “FileService_Card01” in the traces. The system also had a controller process which initiated execution (a message reception was traced) and “Card02” has the FileSystem “FileService_Card02”.

9.2 The Limitation of Kernel Tracing of the Upgrade Software

Due to security reasons, company built its own communication middleware which concealed from LTTng the identity of the communicating processes. This revealed a critical limitation of Kernel tracing, which is insufficient to model process interactions over most if not all middleware. This point is discussed in the Conclusions.

To model software with middleware it is essential to instrument the middleware so as to identify the source and destination of interactions. Steps were attempted to do this within the case study but they could not be completed in the time frame.

Thus the modeling was mostly limited to the behaviour of a single process, UpgradeSW1. However, this was useful because this process has a complex activity structure which could be traced, and rich I/O behaviour. Therefore the case study focusses on these aspects.

There was also a storage limitation in the tracing environment that limited the number of events that could be enabled for tracing. Because CPU times are only reported by standard LTTng probes for Kernel scheduling events, and there are very many such events, this meant that CPU times of the processes could not be obtained from the Kernel traces. For this reason, CPUSat was created and used to approximate the CPU times at events.

9.3 The Collected Data

The data that were collected

- LTTng Kernel traces for file service events
- LTTng application traces for state-change events
- CPUSat data at one-second intervals

The traces data were collected using LTTng tools with file system and application events enabled. The collected trace size was approximately 32 megabytes. However, the obtained data does not cover the whole process of upgrade. With regards to CPUSat data, it had a size of 650 kilobytes.

Here, the LQN model extraction is limited to the first phase which organized into 5 stages (download, load, primary upgrade, secondary upgrade and final) for each card.

Illustration 59 Industrial Case Study : Activities

The image above shows the four states of the first phase (initialization, validation, disk

checking, and load new software). If the new software loading succeeds, the program will pass into the primary phase. However, if it reaches the error_state (State_5), the system retries the load phase, up to four times.

In terms of data processing, the LTTng traces and CPUStat data were extracted by Filtering tools, however, a special filter has been developed to deal with the LTTng application traces due to different format from the Kernel traces. The process will be explained in Section 9.4.

9.4 Processing Steps to Obtain the LQN Model Structure

The application events were instrumented into the source code, and the “process name” was inserted by the developers. The arbitrary UST naming is unrelated to Linux process ids and may even be inconsistent from one tracepoint to the next. Therefore, the Filtering tools had to map these names to the Linux process id..

After modifying the Filtering tools, it converts the traces into SAME 1 results. Some modifications on the SAME phase 1 result were performed to deal with multiple process names and the non-traced components (e.g. Controller).

The following is an example of modification results on SAME phase 1 result

- Process name decoding due to similar PID between ‘upgrd-sw’ and ‘load-sw’

S upgrd-sw6094 read_FileService_Card2 1909579690213741 1909579690236031
S upgrd-sw6094 read_FileService_Card2 1909579731021871 1909579731026001
S upgrd-sw5937 read_FileService_Card1 1909591955509251 1909591955530451
S upgrd-sw5937 read_FileService_Card1 1909591997935041 1909591997942971
S upgrd-sw5937 read_FileService_Card1 1910336115256681 1910336115281361
S upgrd-sw5937 read_FileService_Card1 1910490168016191 1910490168047061

Illustration 60 Industrial Case Study : SAME phase 1 result first modification

- Modification due to non-traced component by adding ‘Controller Dummy’ task.

```
S ControllerDummy upgrd-sw5937 191033504014360 1909591955509250
S ControllerDummy upgrd-sw6094 1909579690213739 1909579690213740
S upgrd-sw6094 read_FileService_Card2 1909579690213741 1909579690236031
S upgrd-sw6094 read_FileService_Card2 1909579731021871 1909579731026001
S upgrd-sw5937 read_FileService_Card1 1909591955509251 1909591955530451
S upgrd-sw5937 read_FileService_Card1 1909591997935041 1909591997942971
S upgrd-sw5937 read_FileService_Card1 1910336115256681 1910336115281361
S upgrd-sw5937 read_FileService_Card1 1910490168016191 1910490168047061
```

Illustration 61 Industrial Case Study : SAME phase 1 result second modification

The dummy controller plays a role to send a command for invoking the ‘upgrd-sw5937’ task on the Card01, and the ‘upgrd-sw6094’ task on the Card02.

The next steps are analyzing the FSM activities (Section 9.5), and the CPU demand parameter calibration (Section 9.6)

9.5 Analyzing Finite State Machine Changes

There were application events to record the changes of the Finite State Machine (FSM), which were also captured by LTTng. The FSM is constructed as an activity graph. The examples show the raw trace events and the same events after being transformed for input to SAME (except that the timestamp still needs to be converted to an integer in nanoseconds).

LTTng result : FSM events

```
card01/ust/uid/0/32-bit 19:10:33.504 016 361 more-subbuf_1 1 FSM:update
prev_state=0, next_state=1, prev_state_name=idle, next_state_name=initialization ,
context._procname=load-sw
```

FSM Extraction

```
Code Task      Timestamp      Identity
E load-sw5934 19:10:33.504016360 state_0
E load-sw5934 19:10:33.504016361 state_1
```

Illustration 62 Industrial Case Study : FSM events

9.6 CPUStat traces

```
Local Time: 12-10-2018_19:10:33 (First Sample)
Process ID, Process Name, Cumulative CPU Time (sec)
pid 5812, (cpustat),cpu 0.000000
pid 5646, (lttng-runas),cpu 0.000000
pid 5861, (upgrade-sw),cpu 0.410000
pid 10, (lru-add-drain),cpu 0.000000
pid 9, (rcu_sched),cpu 4.060000
pid 8, (rcu_bh),cpu 0.100000
pid 7, (migration/0),cpu 1.820000
pid 3, (ksoftirqd/0),cpu 0.170000
pid 2, (kthreadd),cpu 0.010000
pid 1, (systemd),cpu 25.170000
Local Time: 12-10-2018_19:10:34 (Second Sample)
pid 5812, (cpustat),cpu 0.02000
pid 5646, (lttng-runas),cpu 0.010000
pid 5861, (upgrade-sw),cpu 0.480000
pid 10, (lru-add-drain),cpu 0.100000
pid 9, (rcu_sched),cpu 4.080000
pid 8, (rcu_bh),cpu 0.150000
pid 7, (migration/0),cpu 1.830000
pid 3, (ksoftirqd/0),cpu 0.180000
pid 2, (kthreadd),cpu 0.010000
pid 1, (systemd),cpu 25.200000
```

Illustration 63 Industrial Case Study : CPUStat Traces

To filter the useful information from the CPUStat output, the relevant process Id(s) and the process name are used. CPUStat result returns the PID, process name and the aggregated CPU time measured seconds, the resolution of this time is one second.

Illustration 64 Industrial Case Study : CPU time increments (left) and values (right)

The Illustration 64 shows the CPU time increments recorded by CPUSat tools. It shows the changes of CPU time that utilized by load-sw process correspond to the state changes. Within each second the CPU time recorded does not change. For instance, the timestamp range of 19:10:34 – 19:10:38 on the CPUSat data below, it has a static CPU time at 0.48 seconds. Therefore, the interpolation function is applied between points that have an increase in the CPU time.

State	Transition	CPU time	State	Transition	CPU time	State	Transition	CPU time	State	Transition	CPU time
State_1	idle	0.462641	State_1	idle	0.549944	State_1	idle	0.690061	State_1	idle	0.802778
State_1	initialization	0.462641	State_1	initialization	0.551870	State_1	initialization	0.692076	State_1	initialization	0.803879
State_2	initialization	0.463198	State_2	initialization	0.551870	State_2	initialization	0.692076	State_2	initialization	0.803879
State_2	validation	0.463198	State_2	validation	0.553852	State_2	validation	0.694395	State_2	validation	0.805237
State_3	validation	0.463734	State_3	validation	0.553852	State_3	validation	0.694395	State_3	validation	0.805237
State_3	check_disk	0.463734	State_3	check_disk	0.555308	State_3	check_disk	0.696600	State_3	check_disk	0.806248
State_4	check_disk	0.464149	State_4	check_disk	0.555308	State_4	check_disk	0.696600	State_4	check_disk	0.806248
State_4	get_new_sw	0.464149	State_4	get_new_sw	0.560269	State_4	get_new_sw	0.704892	State_4	get_new_sw	0.810000
State_5	get_new_sw	0.465999	State_5	get_new_sw	0.560269	State_5	get_new_sw	0.704892	State_5	get_new_sw	0.810000
State_5	error_state	0.547290	State_5	error_state	0.686602	State_5	error_state	0.800778	State_5	error_state	0.840342
State_0	error_state	0.549944	State_0	error_state	0.686602	State_0	error_state	0.800778			
State_0	idle	0.549944	State_0	idle	0.690061	State_0	idle	0.802778			

Table 5 Industrial Case Study : State changes data and prediction

The Illustration 64 and Table 5 are the CPUStat data and the data of state changes that obtained on the LTTng trace file. The picture shows there are 4 attempts by the ‘load-sw’ process to pass the load phase. It ends on the state 25 which is error_state.

Illustration 65 Industrial Case Study : FSM activities service demands

Illustration 65 shows the service demands of FSM activities. The ‘load-sw’ requires more time on the second attempt of State 5(error_state) and less time on each following successive attempt to get passed to the next phase.

9.7 Model Structure Found for Upgrade

The picture below shows the model structure that was found for the upgrade software which failed on the load phase. Following the steps above, the model structure is extracted from LTTng traces with the helps of filtering tools and SAME. The entry names and the host deployment are not found by SAME and were inserted for readability. When the system enters the failure state (State 5), it could not go into the next phase, therefore, no writing operations were detected on this scenario.

Illustration 66 Industrial Case Study: ExtractModel result

LTTng records the upgrade processes in Card01 and Card02. The ‘Upgrd_sw1’ task implies the upgrade process in Card01, and the ‘Upgrd_sw2’ task implies the upgrade process in Card02. As in the load phase, the Card01 is prepared to do an upgrade on the next phase, therefore on this load phase, the Card01 starts first in loading a new load for upgrade, and the states are detected. On the picture, the states are illustrated by the small rectangles inside the ‘Upgrd_sw1’ task. There are 4 attempts of ‘Upgrd_sw1’ after hitting the error state (5) to go back again to idle state (0), and try again into the initiation state (1). Thus the sequence 1 – 4 is repeated four times.

To obtain the CPU time of the operations the CPUSat tool was run to record the

increment in CPU time for each operation, and this is explained next.

The LQN file is solved with the calibration of FSM changes data and its interpolation. There is no saturated task found in the solved LQN model. However, we can analyze the service time required for each task, entry and activity.

Service time stands for the time when the system is busy for server a request. It concludes the execution time and the waiting time.

The naming on activities, the activity with the last letter ‘b’ stands for 2nd attempt, ‘c’ means 3rd attempt, and d means 4th attempt on the FSM changes.

Service times:

Task Name	Entry Name	Phase 1	
Controller	e1	2.80036	
Card01	upgrd-sw01	0.630342	0
	Activity Name		
	state5	0.13729	
	state5b	0.126333	
	state5c	0.0958857	
	state5d	0.230342	
	state4	0	
	state4b	0.00496156	
	state4c	0.00829196	
	state4d	0.00375246	
	state0	0.00265414	
	state0b	0.00345853	
	state0c	0.00200028	
	state1	0	
	state1b	0.00192639	
	state1c	0.00201519	
	state1d	0.00110109	
	state2	0	
	state2b	0.00198209	
	state2c	0.00231856	
	state2d	0.00135767	
	state3	0	
	state3b	0.0014555	
	state3c	0.00220547	
	state3d	0.00101059	
FileService_Card01	eRead	0.2	
Card02	upgrd-sw02	0.340045	
FileService_Card02	eRead2	0.2	

Illustration 67 Industrial Case Study : LQN solved model – elapsed times (Time in seconds)

As the output above, it shows that the ‘Card01’ task requires 0.630342 for its execution and its waiting time on changing the states and accessing the disk. Supporting the FSM analysis earlier that state 5 on every attempt require time before going back to the first state. Moreover, the last error state (error 5d) likely to take more time than the other attempts before the upgrade goes to failed.

The ‘Card02’ task requires shorter time than the ‘Card01’ since the upgrade were failed on the load phase which does not involve the execution of ‘Card02’. The time that spent by ‘Card02’ is for the initiation and the wait time for commands.

In terms of file service on Card01 and Card02, they have similar capabilities as their required time are similar.

Throughputs and utilizations per phase:

Task Name	Entry Name	Throughput	Phase 1	Total	
Controller	e1	0.208318	0.583365	0.583365	
Card01	upgrd-sw01	0.208318	0	0	0.131311
	Activity Name				
	state5	0.208318	0.0285999		
	state5b	0.208318	0.0263174		
	state5c	0.208318	0.0199747		
	state5d	0.208318	0.0479842		
	state4	0.208318	0		
	state4b	0.208318	0.00103358		
	state4c	0.208318	0.00172736		
	state4d	0.208318	0.000781703		
	state0	0.208318	0.000552903		
	state0b	0.208318	0.000720472		
	state0c	0.208318	0.000416693		
	state1	0.208318	0		
	state1b	0.208318	0.000401301		
	state1c	0.208318	0.000419799		
	state1d	0.208318	0.000229377		
	state2	0.208318	0		
	state2b	0.208318	0.000412905		
	state2c	0.208318	0.000482997		
	state2d	0.208318	0.000282827		
	state3	0.208318	0		
	state3b	0.208318	0.000303207		
	state3c	0.208318	0.000459439		
	state3d	0.208318	0.000210523		
FileService_Card01	eRead	0.208318	0.0416635	0.0416635	
Card02	upgrd-sw02	0.104159	0.0354187	0.0354187	
FileService_Card02	eRead2	2.36725e-005	4.73449e-006	4.73449e-006	

Illustration 68 Industrial Case Study : LQN solved model – throughputs and utilization

As we do not obtain the full trace of upgrade, the throughput and utilization do not give a lot of information. In a more complete trace it would show where the time is being spent and predict the overall impact of local changes to processing in any one state.

9.8 Lessons Learned

The main findings relate to:

- distributed system modeling: the trace merge capability of TraceCompass makes it possible to model distributed system if all nodes are running Linux.

- middleware in the application creates a barrier to tracing messages that seems to be insuperable. Since so many systems rely on middleware, this is a major concern. The solution attempted here was to instrument the application. A better solution might be to instrument the interfaces of the middleware; a new priority is to find efficient ways to do this across multiple middlewares.
- capability to trace inter-process messages: TCP messages can be reconstructed from the trace, but UDP messages cannot. The tracepoint parameter data for UDP is insufficient to correlate the sending event with the corresponding receive event. Java programs communicating via RPCs use TCP for remote calls and can be traced and modelled. A new research priority is to evaluate this issue for container-based distributed software.
- In this system UDP was used by a middleware layer, creating two barriers to the analysis. User-space events were implemented to overcome this problem.
- Kernel tracing of non-Linux hosts is a significant barrier, because adding tracing to other Kernels is a major undertaking and adapting what tracing might be there already will probably give a patchwork.
- Due to limitations, it became impossible to record both ends of some middleware layer conversations, where the component did not run on Linux.
- Arbitrary process names will be different from Linux process ids and may even be inconsistent from one tracepoint to the next. So, the Filtering Tools had to map these names to the Linux process id.

Chapter 10: Conclusions

This chapter draws the summary of the thesis work, and also lists out some possible future works.

10.1 Contributions

Contribution 1: Limitations were identified while exploring the suitability of LTTng Kernel traces.

LTTng tools cannot trace communication via a middleware layer because the Kernel only sees messages between middleware components.

LTTng tools supports TCP communications only.

The LTTng traces lack of application context (e.g Java processes name executed by JVM), and used Perf-tools as a solution to get the information of application context.

It is difficult to reconstruct messages by UDP from Kernel traces (with any confidence), because of limitations in the protocol itself. Information about the identity of each message is absent. For these messages application or middleware information is required.

Also LTTng is purely for Linux and does not trace non-Linux based components. Efforts to create compatible traces for other operating systems, that can be viewed and integrated for analysis by TraceCompass, may make other operating systems accessible for analysis by the methods described here.

Initialization messages on system startup sometimes use callbacks, which are visible to the Kernel traces and confusing for SAME and LQN modeling. This is not a Kernel tracing limitation, but the model derivation process might benefit if some of the early part of the trace could be eliminated.

Contribution 2: Filtering tools and an approach to deal with non-traceable components

Filtering tools written by the author, bridge LTTng Kernel traces into SAME and

deal with the non-traceable components.

For non-traceable software components, the tools use IP address information, a psuedo-task representing the nontraceable component is created by Filtering Tools. This was used in the industrial study in Chapter 9.

For peripheral devices system calls to the device are interpreted as messages to and from a similar psuedo-task. This approach was used for capturing disk operations.

Contribution 3: Efficient Capture of CPU time of operations, for SAME.

This research created a (conventional) logging tool CPUStat that records the CPU time increments associated with particular process id. It logs CPU times separately and then interpolates values using the timestamps to calibrate it with SAME. The interpolated values were used to calibrate demands for entries with and without activities. The algorithms are explained in Section 5.4 with the interpolation steps in Section 5.5.

Contribution 4: Two studies of lab-scale Java programs that communicated between two machines

The experiments demonstrated the use of the contributions listed above and show the result of tracing with and without thread ids, and with and without application activity events.

Contribution 5: Industrial Case Study

This case study is to show the feasibility of extracting an LQN architecture from LTTng traces on industrial real-time software using SAME. The experiment system is a software that implement the ‘rolling upgrade’ concept. The system uses a customized communication middleware which concealed from LTTng the identity of communication process. This revealed the critical limitation of Kernel tracing, which is insufficient to model process interaction over most if not all middleware. The contribution result is written in Chapter 9.

10.2 Summary

To build more significant LQN model, Kernel traces cannot provide all the required information. The application level traces complete the required information. Therefore, there is always some system knowledge that needed, in terms of instrumenting the application level event on the source code.

10.3 Future Works

This thesis has been mainly focused on the use of Kernel traces in analyzing TCP messages and file operations. Future work concerns deeper analysis of particular mechanisms. In this case, the other Linux inter-process communications (e.g. signals, file locks, etc), systems using forwarding, and some kind of approach for analyzing UDP messages are recommended for future experiments. Broader analysis may need the further research on the relevant Kernel events for data collection, and the Filtering Tools could be improved to cope with new format of the new Kernel events. In terms of LQN performance model parameters, another idea might be to improve the CPUStat tools accuracy by using a smaller tick in the Linux Kernel.

Appendices

Appendix A JFT Experiments on transferring 10 KB File

A.1 LTTng Kernel Traces

Trace	Timestamp	Channel CPU	Event type	Contents
jft10kb	/vm01-20190130-141050/kernel_	14:12:11.795 258 007	more-subbuf_2 2	
	syscall_entry_read	fd=3, count=832, context_tid=4114, context_pid=4114,		
	context_procname=java	4114 20 4114		
jft10kb	/vm01-20190130-141050/kernel_	14:12:11.796 483 421	more-subbuf_2 2	
	syscall_exit_read	ret=832, buf=140728515921552, context_tid=4114, context_pid=4114,		
	context_procname=java	4114 20 4114		
jft10kb	/vm01-20190130-141050/kernel_	14:12:16.153 464 990	more-subbuf_1 1	
	syscall_entry_read	fd=3, count=4096, context_tid=4146, context_pid=4145,		
	context_procname=java	4146 20 4145		
jft10kb	/vm01-20190130-141050/kernel_	14:12:16.153 872 910	more-subbuf_1 1	
	syscall_exit_read	ret=2295, buf=140156303601664, context_tid=4146, context_pid=4145,		
	context_procname=java	4146 20 4145		
jft10kb	/vm01-20190130-141050/kernel_	14:12:16.154 900 442	more-subbuf_1 1	
	syscall_entry_write	fd=5, buf=140156304661184, count=1, context_tid=4146,		
	context_pid=4145, context_procname=java	4146 20 4145		
jft10kb	/vm01-20190130-141050/kernel_	14:12:16.154 922 213	more-subbuf_1 1	
	syscall_exit_write	ret=1, context_tid=4146, context_pid=4145, context_procname=java		
	4146 20 4145			
jft10kb	/vm02-20190130-141105/kernel_	14:12:22.824 120 983	more-subbuf_3 3	
	syscall_entry_read	fd=3, count=832, context_tid=4006, context_pid=4006,		
	context_procname=java	4006 20 4006		
jft10kb	/vm02-20190130-141105/kernel_	14:12:22.824 125 078	more-subbuf_3 3	
	syscall_exit_read	ret=832, buf=140726112136800, context_tid=4006, context_pid=4006,		
	context_procname=java	4006 20 4006		
jft10kb	/vm02-20190130-141105/kernel_	14:12:23.076 846 697	more-subbuf_3 3	
	syscall_entry_write	fd=5, buf=139926804040384, count=1, context_tid=4007,		
	context_pid=4006, context_procname=java	4007 20 4006		
jft10kb	/vm02-20190130-141105/kernel_	14:12:23.076 872 249	more-subbuf_3 3	
	syscall_exit_write	ret=1, context_tid=4007, context_pid=4006, context_procname=java		
	4007 20 4006			
jft10kb	/vm01-20190130-141050/kernel_	14:12:31.062 410 238	more-subbuf_2 2	
	net_dev_queue	skbaddr=0xffff91975a709cf8, len=67, name=eth0, network_header_type=ipv4,		
	network_header=ipv4=[version=4, ihl=5, tos=0, tot_len=53, id=0x8830, frag_off=16384, ttl=64,			
	protocol=_tcp, checksum=0x3b72, saddr_padding=[], saddr=[192, 168, 122, 213],			
	daddr_padding=[], daddr=[192, 168, 122, 250], transport_header_type=_tcp,			
	transport_header=tcp=[source_port=33064, dest_port=44508, seq=603151431,			
	ack_seq=2164662313, data_offset=8, reserved=0, flags=0x18, window_size=237,			
	checksum=0x7748, urg_ptr=0]], context_tid=4146, context_pid=4145, context_procname=java			
	4146 20 4145			
jft10kb	/vm02-20190130-141105/kernel_	14:12:31.063 133 924	more-subbuf_3 3	
	net_dev_queue	skbaddr=0xffff891edcf1baf8, len=67, name=eth0, network_header_type=ipv4,		
	network_header=ipv4=[version=4, ihl=5, tos=0, tot_len=53, id=0xd095, frag_off=16384, ttl=64,			
	protocol=_tcp, checksum=0xf30c, saddr_padding=[], saddr=[192, 168, 122, 250],			
	daddr_padding=[], daddr=[192, 168, 122, 213], transport_header_type=_tcp,			
	transport_header=tcp=[source_port=44508, dest_port=33064, seq=2164662313,			
	ack_seq=603151432, data_offset=8, reserved=0, flags=0x18, window_size=235,			
	checksum=0x7748, urg_ptr=0]], context_tid=4061, context_pid=4006, context_procname=java			
	4061 20 4006			

A.2 LTTng UST Traces

```
jft10kb/vm01-20190130-141050/ust/uid/1000/64-bit_14:12:13.112 838 137 lttng_jul_channel_2
2 lttng_jul:event msg=generatefile_start, logger_name=clienta, class_name= clienta,
method_name=main, long_millis=1548875532500, int_loglevel=800, int_threadid=1,
context._procname=java
jft10kb/vm01-20190130-141050/ust/uid/1000/64-bit_14:12:14.122 821 126 lttng_jul_channel_3
3 lttng_jul:event msg= generatefile_finish, logger_name= clienta, class_name= clienta,
method_name=main, long_millis=1548875533108, int_loglevel=800, int_threadid=1,
context._procname=java
jft10kb/vm01-20190130-141050/ust/uid/1000/64-bit_14:12:14.122 821 126 lttng_jul_channel_3
2 lttng_jul:event msg=send_start, logger_name= clienta, class_name= clienta,
method_name=main, long_millis=1548875533126, int_loglevel=800, int_threadid=1,
context._procname=java
jft10kb/vm01-20190130-141050/ust/uid/1000/64-bit_14:12:15.231 811 228 lttng_jul_channel_3
3 lttng_jul:event msg=send_finish, logger_name= clienta, class_name= clienta,
method_name=main, long_millis= 1548875537075, int_loglevel=800, int_threadid=1,
context._procname=java
jft10kb/vm01-20190130-141050/ust/uid/1000/64-bit_14:12:15.231 811 229 lttng_jul_channel_3
3 lttng_jul:event msg=receive_start, logger_name= clientb, class_name= clientb,
method_name=main, long_millis= 1548875537862, int_loglevel=800, int_threadid=1,
context._procname=java
jft10kb/vm02-20190130-141105/ust/uid/1000/64-bit_14:12:16.911 241 921 lttng_jul_channel_1
1 lttng_jul:event msg=receive_finish, logger_name=clientb, class_name=clientb,
method_name=main, long_millis=1548875555208, int_loglevel=800, int_threadid=1,
context._procname=java
jft10kb/vm02-20190130-141105/ust/uid/1000/64-bit_14:12:16.911 241 921 lttng_jul_channel_1
1 lttng_jul:event msg=sort_start, logger_name=clientb, class_name=clientb,
method_name=main, long_millis=1548875555222, int_loglevel=800, int_threadid=1,
context._procname=java
jft10kb/vm02-20190130-141105/ust/uid/1000/64-bit_14:12:17.812 545 261 lttng_jul_channel_1
1 lttng_jul:event msg=sort_finish, logger_name=clientb, class_name=clientb,
method_name=main, long_millis=1548875555247, int_loglevel=800, int_threadid=1,
context._procname=java
```

A.3 Perf result

Tracing exec()s. Ctrl-C to end.

Instrumenting sys_execve

	PID	PPID	ARGS
4114	3251		java -cp /usr/local/share/java/ltng-ust-agent-common.jar:/usr/local/share/java/ltng-ust-agent-jul.jar:. Controller
4145	3323		java -cp /usr/local/share/java/ltng-ust-agent-common.jar:/usr/local/share/java/ltng-ust-agent-jul.jar:. ClientA
4006	3372		java -cp /usr/local/share/java/ltng-ust-agent-common.jar:/usr/local/share/java/ltng-ust-agent-jul.jar:. ClientB

A.4 CPUStat result

vm01,141211794113018,controller,cpu 0.000000
vm01,141212630758275,controller,cpu 0.490000
vm01,141213706039994,controller,cpu 0.890000
vm01,141214750124105,controller,cpu 0.890000
vm01,141215790401334,controller,cpu 0.900000
vm01,141211794113018,clienta,cpu 0.000000
vm01,141216924919554,clienta,cpu 0.510000
vm01,141216924919554,controller,cpu 0.910000
vm01,141218223682700,clienta,cpu 1.230000
vm01,141218223682700,controller,cpu 1.020000
vm01,141219713395110,clienta,cpu 1.240000
vm01,141219713395110,controller,cpu 1.020000
vm01,141220106635568,clienta,cpu 1.240000
vm01,141220106635568,controller,cpu 1.020000
vm01,141221156900662,clienta,cpu 1.250000
vm01,141221156900662,controller,cpu 1.020000
vm01,141222194369982,clienta,cpu 1.250000
vm01,141222194369982,controller,cpu 1.030000
vm01,141223278571531,clienta,cpu 1.270000
vm01,141223278571531,controller,cpu 1.040000
vm01,141224437013931,clienta,cpu 1.270000
vm01,141224437013931,controller,cpu 1.040000
vm01,141225489019856,clienta,cpu 1.270000
vm01,141225489019856,controller,cpu 1.060000
vm01,141226525629432,clienta,cpu 1.280000
vm01,141226525629432,controller,cpu 1.060000
vm01,141227552034542,clienta,cpu 1.290000
vm01,141227552034542,controller,cpu 1.070000
vm01,141228578650317,clienta,cpu 1.290000
vm01,141228578650317,controller,cpu 1.070000
vm01,141229615168266,clienta,cpu 1.290000
vm01,141229615168266,controller,cpu 1.070000
vm01,141230654859448,clienta,cpu 1.290000
vm01,141230654859448,controller,cpu 1.070000
vm01,141231696504561,clienta,cpu 1.350000
vm01,141231696504561,controller,cpu 1.070000
vm01,141232726478081,clienta,cpu 1.350000
vm01,141232726478081,controller,cpu 1.070000
vm01,141233770482967,clienta,cpu 1.350000
vm01,141233770482967,controller,cpu 1.070000
vm01,141234813164798,controller,cpu 1.080000
vm01,141235851009850,controller,cpu 1.080000
vm01,141236885149858,controller,cpu 1.090000
vm01,141237927221634,controller,cpu 1.090000
vm01,141238975455980,controller,cpu 1.100000
vm01,141240452219190,controller,cpu 1.100000
vm01,141241868283280,controller,cpu 1.100000
vm01,141242120333520,controller,cpu 1.100000
vm01,141243174694572,controller,cpu 1.100000
vm01,141244214565413,controller,cpu 1.110000
vm01,141245255476836,controller,cpu 1.110000
vm01,141246285260737,controller,cpu 1.110000
vm01,141247323777692,controller,cpu 1.120000

vm01,141248361192502,controller,cpu 1.120000
vm01,141249390018448,controller,cpu 1.120000
vm01,141250437026615,controller,cpu 1.120000
vm01,141251524900624,controller,cpu 1.120000
vm01,141252569832863,controller,cpu 1.120000
vm01,141253611092199,controller,cpu 1.140000
vm01,141254653813023,controller,cpu 1.140000
vm01,141255692209703,controller,cpu 1.140000
vm01,141256730038516,controller,cpu 1.140000
vm01,141257768658629,controller,cpu 1.140000
vm01,141258821379046,controller,cpu 1.150000
vm01,141259846685303,controller,cpu 1.150000
vm02,141211794113018,clientb,cpu 0.000000
vm02,141224500964516,clientb,cpu 0.820000
vm02,141225559112035,clientb,cpu 1.350000
vm02,141226595915581,clientb,cpu 1.360000
vm02,141227643609813,clientb,cpu 1.360000
vm02,141228682646863,clientb,cpu 1.370000
vm02,141229716537212,clientb,cpu 1.370000
vm02,141230740660613,clientb,cpu 1.370000
vm02,141231783700326,clientb,cpu 1.420000
vm02,141232811599111,clientb,cpu 1.420000
vm02,141233844185486,clientb,cpu 1.420000
vm02,141234876379107,clientb,cpu 1.430000
vm02,141235918567790,clientb,cpu 1.610000
vm02,141236972979074,clientb,cpu 1.610000
vm02,141238751488600,clientb,cpu 1.610000

A.5 LQN model file using 'mergeByCallSet' and Kernel

G " " 1e-06 50 1 0.9 -1

P 4

p VM01 f
p VM02 f
p HDiskVM01 f
p HDiskVM02 f
-1

T 5

t controller_Source r sourcecontroller -1 VM01
t clienta f E_13248A E_6330S E_12523S -1 VM01
t clientb f E_13256S E_19383S -1 VM02
t fileservice_vm01 f write01 read01 -1 HDiskVM01
t fileservice_vm02 f write02 read02 -1 HDiskVM02
-1

E 10

```
# ----- controller_Source -----
s sourcecontroller 0 -1
z sourcecontroller E_13248A 1 -1
y sourcecontroller read01 6547 -1
y sourcecontroller E_6330S 1 -1
y sourcecontroller E_13256S 1 -1
y sourcecontroller E_19383S 5 -1
y sourcecontroller E_12523S 8 -1
# ----- clienta -----
s E_13248A 0.00346 0 -1
y E_13248A write01 8 0 -1
y E_13248A read01 14 0 -1
y E_13248A E_19383S 3 0 -1
s E_6330S 0.0692 0 -1
y E_6330S write01 33 0 -1
y E_6330S read01 6666 0 -1
s E_12523S 0.117 0 -1
# ----- clientb -----
s E_13256S 0.0891 0 -1
y E_13256S write02 39 0 -1
y E_13256S read02 6855 0 -1
s E_19383S 0.135 0 -1
# ----- fileservice_vm01 -----
s write01 4.13e-05 0 -1
s read01 7.99e-05 0 -1
# ----- fileservice_vm02 -----
s write02 1.83e-05 0 -1
s read02 0.000103 0 -1
-1
```

A.6 LQN model file using ‘mergeByTask’ and Kernel

```
G " " 1e-06 50 1 0.9 -1
P 4
  p VM01 f
  p VM02 f
  p HDiskVM01 f
  p HDiskVM02 f
-1
T 5
  t controller_Source r sourcecontroller -1 VM01
  t clienta f E_6330S -1 VM01
  t clientb f E_13256S -1 VM02
  t fileservice_vm01 f write01 read01 -1 HDiskVM01
  t fileservice_vm02 f write02 read02 -1 HDiskVM02
-1
E 7
# ----- controller_Source -----
s sourcecontroller 0 -1
y sourcecontroller read01 6547 -1
y sourcecontroller E_6330S 10 -1
y sourcecontroller E_13256S 6 -1
# ----- clienta -----
s E_6330S 0.101 0 -1
y E_6330S write01 4.1 0 -1
y E_6330S read01 668 0 -1
y E_6330S E_13256S 0.3 0 -1
# ----- clientb -----
s E_13256S 0.13 0 -1
y E_13256S write02 4.33 0 -1
y E_13256S read02 761.67 0 -1
# ----- fileservice_vm01 -----
s write01 4.13e-05 0 -1
s read01 7.99e-05 0 -1
# ----- fileservice_vm02 -----
s write02 1.83e-05 0 -1
s read02 0.000103 0 -1
-1
```

A.7 LQN model file using ‘mergeByCallSet’ with Kernel and Activities

G " " 1e-006 50 1 0.9 -1

P 4

p VM01 f
p HDiskVM01 f
p VM02 f
p HDiskVM02 f
-1

T 5

t controller_Source r sourcecontroller -1 VM01
t clienta n E_10S E_18S E_94A -1 VM01
t fileservice_vm01 n write01 read01 -1 HDiskVM01
t clientb n E_106S E_125S E_102S E_112S E_133S -1 VM02
t fileservice_vm02 n write02 read02 -1 HDiskVM02
-1

E 13

----- controller_Source -----
s sourcecontroller 0 -1
Z sourcecontroller 0 -1
y sourcecontroller E_10S 8 -1
y sourcecontroller read01 1 -1
z sourcecontroller E_94A 1 -1
y sourcecontroller E_112S 1 -1
y sourcecontroller E_18S 1 -1
y sourcecontroller E_102S 4 -1
----- clienta -----
s E_10S 0.00928 0 -1
Z E_10S 0 0 -1
A E_18S E_18S_1reply
A E_94A generatefile_finish_A_2
----- fileservice_vm01 -----
s write01 8.43e-005 0 -1
Z write01 0 0 -1
s read01 0.000345 0 -1
Z read01 0 0 -1
----- clientb -----
A E_106S E_106S_1reply
s E_125S 0.0028 0 -1
Z E_125S 0 0 -1
s E_102S 0.00381 0 -1
Z E_102S 0 0 -1
s E_112S 0.00288 0 -1
Z E_112S 0 0 -1
A E_133S receive_finish_A_6
----- fileservice_vm02 -----
s write02 2.53e-005 0 -1
Z write02 0 0 -1
s read02 0.000173 0 -1
Z read02 0 0 -1
-1

```

A clienta
s E_18S_1reply 0.00873
Z E_18S_1reply 0
s generatefile_finish_A_2 0
Z generatefile_finish_A_2 0
y generatefile_finish_A_2 read01 3
y generatefile_finish_A_2 write01 1
y generatefile_finish_A_2 E_125S 1
s generatefile_start_A_1 0
Z generatefile_start_A_1 0
y generatefile_start_A_1 write01 15
y generatefile_start_A_1 read01 30
s send_finish_A_4 0
Z send_finish_A_4 0
y send_finish_A_4 read01 10
y send_finish_A_4 E_106S 1
y send_finish_A_4 E_133S 1
s send_start_A_3 0
Z send_start_A_3 0
y send_start_A_3 read01 44
:
E_18S_1reply[E_18S] -> generatefile_start_A_1;
generatefile_finish_A_2 -> send_start_A_3;
send_start_A_3 -> send_finish_A_4
-1

```

```

A clientb
s E_106S_1reply 0.00331
Z E_106S_1reply 0
s receive_finish_A_6 0
Z receive_finish_A_6 0
y receive_finish_A_6 read02 228
y receive_finish_A_6 write02 10
s receive_start_A_5 0
Z receive_start_A_5 0
y receive_start_A_5 read02 22
y receive_start_A_5 write02 19
s sort_finish_A_8 0
Z sort_finish_A_8 0
s sort_start_A_7 0
Z sort_start_A_7 0
y sort_start_A_7 read02 228
y sort_start_A_7 write02 10
:
E_106S_1reply[E_106S] -> receive_start_A_5;
receive_finish_A_6[E_133S] -> sort_start_A_7;
sort_start_A_7 -> sort_finish_A_8
-1

```

Appendix B LQN solved model on JFT transferring 100 KB File

Generated by: lqsim, version 5.10
Copyright the Real-Time and Distributed Systems Group,
Department of Systems and Computer Engineering
Carleton University, Ottawa, Ontario, Canada. K1S 5B6

Invoked as: lqsim -B30,300000 PredictedModel100Kb_QuadV2.lqnx
Input: PredictedModel100Kb_QuadV2.lqnx
Comment:

Convergence test value: 1.86162
Number of iterations: 30
Seed Value: 1552500787

Solver:
User: 0:00:00.00
System: 0:00:00.00
Elapsed: 0:00:00.32

Processor identifiers and scheduling algorithms:

Processor Name	Type	Copies	Scheduling Rate
HDiskVM01	Uni	1	FCFS
HDiskVM02	Uni	1	FCFS
VM01	Uni	1	FCFS
VM02	Uni	1	FCFS

Task information:

Task Name	Type	Copies	Processor Name	Pri	Entry List
fileservice_vm01	Uni	1	HDiskVM01		write01, read01
fileservice_vm02	Uni	1	HDiskVM02		write02, read02
controller_Source	Ref(1)	1	VM01		sourcecontroller
clienta	Uni	1	VM01		E_10S, E_18S, E_94A : E_18S_1reply, generatefile_finish_A_2, generatefile_start_A_1, send_finish_A_4, send_start_A_3
clientb	Uni	1	VM02		E_106S, E_125S, E_102S, E_112S, E_133S : E_106S_1reply, receive_finish_A_6, receive_start_A_5, sort_finish_A_8, sort_start_A_7

Entry execution demands:

Task Name	Entry Name	Phase 1	Phase 2
fileservice_vm01	write01	0.0039	0
	read01	0.0162	0
fileservice_vm02	write02	0.00274	0
	read02	0.0269	0
controller_Source	sourcecontroller	0	
clienta	E_10S	2.74893	0
	Activity Name		
	E_18S_1reply	3.97028	
	generatefile_finish_A_20		
	generatefile_start_A_10		

	send_finish_A_4	0	
	send_start_A_3	0	
clientb	E_125S	0.884879	0
	E_102S	1.90776	0
	E_112S	1.86362	0
	Activity Name		
	E_106S_1reply	0.550679	
	receive_finish_A_60		
	receive_start_A_50		
	sort_finish_A_8	0	
	sort_start_A_7	0	

Mean number of rendezvous from entry to entry:

Task Name	Source Entry	Target Entry	Phase 1	Phase 2
controller_Source	sourcecontroller	read01	1	
	sourcecontroller	E_10S	8	
	sourcecontroller	E_112S	1	
	sourcecontroller	E_18S	1	
	sourcecontroller	E_102S	4	
clienta		Activity Name		
	generatefile_finish_A_2	read01	8	
	generatefile_finish_A_2	write01	1	
	generatefile_finish_A_2	E_125S	6	
	generatefile_start_A_1	write01	40	
	generatefile_start_A_1	read01	52	
	send_finish_A_4	read01	10	
	send_finish_A_4	E_106S	1	
	send_finish_A_4	E_133S	1	
	send_start_A_3	read01	78	
clientb		Activity Name		
	receive_finish_A_6	read02	260	
	receive_finish_A_6	write02	30	
	receive_start_A_5	read02	65	
	receive_start_A_5	write02	36	
	sort_start_A_7	read02	260	
	sort_start_A_7	write02	30	

Mean number of non-blocking sends from entry to entry:

Task Name	Source Entry	Target Entry	Phase 1	Phase 2
controller_Source	sourcecontroller	E_94A	1	

Phase type flags:

All phases are stochastic.

Squared coefficient of variation of execution segments:

All executable segments are exponential.

Open arrival rates per entry:

All open arrival rates are 0.

Mean delay for a rendezvous:

Task Name	Source Entry	Target Entry	Phase 1	Phase 2
controller_Source	sourcecontroller	read01		0.000657676
			+/-95%	0.000124054
			+/-99%	0.000167185
	sourcecontroller E_10S			2.35951
			+/-95%	0.11531
			+/-99%	0.1554
	sourcecontroller E_112S			0.254603
			+/-95%	0.0391286
			+/-99%	0.0527327
	sourcecontroller E_18S			2.35247
			+/-95%	0.309709
			+/-99%	0.417388
	sourcecontroller E_102S			0.256107
			+/-95%	0.0246876
			+/-99%	0.033271
clienta	Activity Name			
	generatefile_finish_A_2	read01		6.00232e-005
			+/-95%	1.47705e-005
			+/-99%	1.99058e-005
	generatefile_finish_A_2	write01		5.39646e-005
			+/-95%	3.75513e-005
			+/-99%	5.0607e-005
	generatefile_finish_A_2	E_125S		0.189934
			+/-95%	0.0153786
			+/-99%	0.0207254
	generatefile_start_A_1	write01		1.39854e-005
			+/-95%	3.21484e-006
			+/-99%	4.33257e-006
	generatefile_start_A_1	read01		1.42093e-005
			+/-95%	2.89699e-006
			+/-99%	3.90421e-006
	send_finish_A_4	read01		4.58257e-007
			+/-95%	8.89919e-007
			+/-99%	1.19932e-006
	send_finish_A_4	E_106S		3.01615
			+/-95%	0.107025
			+/-99%	0.144235
	send_finish_A_4	E_133S		3.01464
			+/-95%	0.18832
			+/-99%	0.253794
	send_start_A_3	read01		1.24774e-006
			+/-95%	7.22552e-007
			+/-99%	9.73766e-007
clientb	Activity Name			
	receive_finish_A_6	read02		0
			+/-95%	0
			+/-99%	0
	receive_finish_A_6	write02		0
			+/-95%	0
			+/-99%	0
	receive_start_A_5	read02		0
			+/-95%	0
+/-99%			0	

receive_start_A_5	write02	0
	+/-95%	0
	+/-99%	0
sort_start_A_7	read02	0
	+/-95%	0
	+/-99%	0
sort_start_A_7	write02	0
	+/-95%	0
	+/-99%	0

Variance of delay for a rendezvous:

Task Name	Source Entry	Target Entry	Phase 1	Phase 2
controller_Source	sourcecontroller	read01	1.9212e-005	
		+/-95%	6.19226e-006	
		+/-99%	8.34516e-006	
	sourcecontroller	E_10S	115.434	
		+/-95%	10.95	
		+/-99%	14.757	
	sourcecontroller	E_112S	3.08511	
		+/-95%	1.10186	
		+/-99%	1.48495	
	sourcecontroller	E_18S	115.078	
		+/-95%	33.9409	
		+/-99%	45.7414	
	sourcecontroller	E_102S	3.16431	
		+/-95%	0.656154	
		+/-99%	0.884283	
clienta		Activity Name		
	generatefile_finish_A_2	read01	1.99302e-006	
		+/-95%	8.71333e-007	
		+/-99%	1.17428e-006	
	generatefile_finish_A_2	write01	1.72337e-006	
		+/-95%	2.04092e-006	
		+/-99%	2.7505e-006	
	generatefile_finish_A_2	E_125S	1.27464	
		+/-95%	0.232557	
		+/-99%	0.313412	
	generatefile_start_A_1	write01	4.45964e-007	
		+/-95%	1.60824e-007	
		+/-99%	2.16739e-007	
	generatefile_start_A_1	read01	4.56458e-007	
		+/-95%	1.58128e-007	
		+/-99%	2.13105e-007	
	send_finish_A_4	read01	1.75745e-008	
		+/-95%	5.65796e-008	
		+/-99%	7.62511e-008	
	send_finish_A_4	E_106S	27.2435	
		+/-95%	2.19562	
		+/-99%	2.95899	
	send_finish_A_4	E_133S	27.3973	
		+/-95%	4.32955	
		+/-99%	5.83483	
	send_start_A_3	read01	3.85869e-008	
		+/-95%	3.59064e-008	
		+/-99%	4.83902e-008	

clientb	Activity Name	
receive_finish_A_6	read02	0
	+/-95%	0
	+/-99%	0
receive_finish_A_6	write02	0
	+/-95%	0
	+/-99%	0
receive_start_A_5	read02	0
	+/-95%	0
	+/-99%	0
receive_start_A_5	write02	0
	+/-95%	0
	+/-99%	0
sort_start_A_7	read02	0
	+/-95%	0
	+/-99%	0
sort_start_A_7	write02	0
	+/-95%	0
	+/-99%	0

Mean delay for a send-no-reply request:

Task Name	Source Entry	Target Entry	Phase 1	Phase 2
controller_Source	source	controller E_94A	24.091	
		+/-95%	0.828833	
		+/-99%	1.117	

Variance of delay for a send-no-reply request:

Task Name	Source Entry	Target Entry	Phase 1	Phase 2
controller_Source	source	controller E_94A	695.49	
		+/-95%	72.5659	
		+/-99%	97.7954	

Arrival Loss Probabilities:

Task Name	Source Entry	Target Entry	Phase 1	Phase 2
controller_Source	source	controller E_94A	0	
		+/-95%	0	
		+/-99%	0	

Service times:

Task Name	Entry Name	Phase 1	Phase 2
fileservice_vm01	write01	0.00390044	0
	+/-95%	1.63996e-005	0
	+/-99%	2.21013e-005	0
	read01	0.0162007	0
	+/-95%	3.72776e-005	0
	+/-99%	5.02382e-005	0
fileservice_vm02	write02	0.0027389	0
	+/-95%	8.62715e-006	0
	+/-99%	1.16266e-005	0

```

read02          0.0268999    0
               +/-95% 3.47186e-005 0
               +/-99% 4.67895e-005 0
controller_Source sourcecontroller 57.7971
               +/-95% 1.47551
               +/-99% 1.98852
clienta      E_10S          2.74758    0
               +/-95% 0.0193475    0
               +/-99% 0.0260741    0
               E_18S          3.96558    0.998204
               +/-95% 0.123085    0.0294464
               +/-99% 0.165879    0.0396842
               E_94A          21.6683    0
               +/-95% 0.701846    0
               +/-99% 0.945862    0
Activity Name
E_18S_1reply    3.96558
               +/-95% 0.123085
               +/-99% 0.165879
generatefile_finish_A_2 6.52629
               +/-95% 0.160812
               +/-99% 0.216722
generatefile_start_A_1 0.998204
               +/-95% 0.0294464
               +/-99% 0.0396842
send_finish_A_4 13.8774
               +/-95% 0.727924
               +/-99% 0.981006
send_start_A_3  1.26453
               +/-95% 0.0448902
               +/-99% 0.0604975
clientb      E_106S          0.550234    1.85404
               +/-95% 0.0164453    0.0701522
               +/-99% 0.0221629    0.0945425
               E_125S          0.884258    0
               +/-95% 0.00783179    0
               +/-99% 0.0105547    0
               E_102S          1.90547    0
               +/-95% 0.0327033    0
               +/-99% 0.0440736    0
               E_112S          1.86545    0
               +/-95% 0.0430639    0
               +/-99% 0.0580362    0
               E_133S          7.08261    7.0902
               +/-95% 0.190829    0.19106
               +/-99% 0.257176    0.257487
Activity Name
E_106S_1reply    0.550234
               +/-95% 0.0164453
               +/-99% 0.0221629
receive_finish_A_6 7.08261
               +/-95% 0.190829
               +/-99% 0.257176
receive_start_A_5 1.85404
               +/-95% 0.0701522
               +/-99% 0.0945425
sort_finish_A_8  0

```

```

+/-95% 0
+/-99% 0
sort_start_A_7 7.0902
+/-95% 0.19106
+/-99% 0.257487

```

Service time variance (per phase)
and squared coefficient of variation (over all phases):

Task Name	Entry Name	Phase 1	Phase 2	coeff of var **2
fileservice_vm01	write01	1.52035e-005 0		0.999342
		+/-95% 2.04193e-007 0		
		+/-99% 2.75186e-007 0		
	read01	0.000262323 0		0.999466
		+/-95% 1.7863e-006 0		
		+/-99% 2.40735e-006 0		
fileservice_vm02	write02	7.49758e-006 0		0.99947
		+/-95% 5.71918e-008 0		
		+/-99% 7.70761e-008 0		
	read02	0.000723754 0		1.00021
		+/-95% 2.84468e-006 0		
		+/-99% 3.83371e-006 0		
controller_Source	sourcecontroller	4638.08		1.38844
		+/-95% 395.78		
		+/-99% 533.383		
clienta	E_10S	7.52104	0	0.996269
		+/-95% 0.22632	0	
		+/-99% 0.305007	0	
	E_18S	15.7036	1.01698	0.678621
		+/-95% 1.17247	0.0874198	
		+/-99% 1.58011	0.117814	
	E_94A	565.966	0	1.20543
		+/-95% 61.7577	0	
		+/-99% 83.2294	0	
	Activity Name			
	E_18S_1reply	15.7036		
		+/-95% 1.17247		
		+/-99% 1.58011		
	generatefile_finish_A_2	48.66		
		+/-95% 3.04071		
		+/-99% 4.09789		
	generatefile_start_A_1	1.01698		
		+/-95% 0.0874198		
		+/-99% 0.117814		
	send_finish_A_4	517.431		
		+/-95% 65.9024		
		+/-99% 88.8152		
	send_start_A_3	1.64633		
		+/-95% 0.146705		
		+/-99% 0.197711		
clientb	E_106S	0.303933	3.55075	0.66684
		+/-95% 0.0267204	0.318134	
		+/-99% 0.0360104	0.428742	
	E_125S	0.782518	0	1.00077
		+/-95% 0.0198772	0	

	+/-99%	0.0267881	0	
E_102S		3.627	0	0.998952
	+/-95%	0.137438	0	
	+/-99%	0.185222	0	
E_112S		3.47979	0	0.999961
	+/-95%	0.230871	0	
	+/-99%	0.31114	0	
E_133S		50.5127	50.6022	0.503389
	+/-95%	4.3432	4.57683	
	+/-99%	5.85323	6.16809	
Activity Name				
E_106S_1reply		0.303933		
	+/-95%	0.0267204		
	+/-99%	0.0360104		
receive_finish_A_6		50.5127		
	+/-95%	4.3432		
	+/-99%	5.85323		
receive_start_A_5		3.55075		
	+/-95%	0.318134		
	+/-99%	0.428742		
sort_finish_A_8		0		
	+/-95%	0		
	+/-99%	0		
sort_start_A_7		50.6022		
	+/-95%	4.57683		
	+/-99%	6.16809		

Throughputs and utilizations per phase:

Task Name	Entry Name	Throughput	Phase 1	Phase 2
Total				
fileservice_vm01	write01	0.706368	0.00275513	0
0.00275513				
	+/-95%	0.030167	0.000116732	0
0.000116732				
	+/-99%	0.0406554	0.000157317	0
0.000157317				
	read01	2.57017	0.0416385	0
0.0416385				
	+/-95%	0.0788188	0.00125836	0
0.00125836				
	+/-99%	0.106222	0.00169587	0
0.00169587				
Total:		3.27654	0.0443936	
0.0443936				
	+/-95%	0.0962014	0.00126376	
0.00126376				
	+/-99%	0.129648	0.00170315	
0.00170315				
fileservice_vm02	write02	1.6658	0.0045625	0
0.0045625				
	+/-95%	0.058853	0.000165596	0
0.000165596				
	+/-99%	0.0793148	0.00022317	0
0.00022317				

0.273191	read02	10.1558	0.273191	0
		+/-95%	0.390406	0.0105205
0.0105205		+/-99%	0.526142	0.0141782
0.0141782				0
Total:			11.8216	0.277753
0.277753		+/-95%	0.445251	0.0105218
0.0106725		+/-99%	0.600055	0.01418
0.0143831				
controller_Source	sourcecontroller	0.0173044	1	
1		+/-95%	0.000441264	0
		+/-99%	0.000594681	0
clienta	E_10S	0.138135	0.379536	0
0.379536		+/-95%	0.00265267	0.0071806
0.0071806		+/-99%	0.00357494	0.00967714
0.00967714				0
0.0856175	E_18S	0.0172496	1.60274e-006	1.3987e-007
0.00269956		+/-95%	0.000584686	0.00258895
0.00363814		+/-99%	0.000787968	0.00348907
0.373974	E_94A	0.0172611	2.08774e-005	0
0.00934398		+/-95%	0.000420735	0.00934398
0.0125927		+/-99%	0.000567015	0.0125927
	Activity Name			
	E_18S_1reply	0.0172496	0.068399	
		+/-95%	0.320114	0.00258895
		+/-99%	0.43141	0.00348907
	generatefile_finish_A_2	0.0172611	0.112648	
		+/-95%	0.230942	0.0035544
		+/-99%	0.311235	0.00479018
	generatefile_start_A_1	0.0172496	0.0172185	
		+/-95%	0.320246	0.000764813
		+/-99%	0.431588	0.00103072
	send_finish_A_4	0.0172611	0.239497	
		+/-95%	0.230446	0.0101761
		+/-99%	0.310567	0.0137142
	send_start_A_3	0.0172611	0.0218285	
		+/-95%	0.230942	0.00106224
		+/-99%	0.311235	0.00143156
Total:			0.172646	0.821909
0.839127				0.0172185
		+/-95%	0.00281889	0.0120654
0.00344539		+/-99%	0.00379895	0.0162603
0.00464328				0.00103072

clientb	E_106S	0.017279	4.5825e-008	6.84149e-007
0.0415438				
		+/-95%	0.000604937	0.000437768
0.00174722				0.00169149
		+/-99%	0.00081526	0.000589971
0.00235468				0.00227958
	E_125S	0.102736	0.0908444	0
0.0908444				
		+/-95%	0.00375271	0.00335196
0.00335196				0
		+/-99%	0.00505744	0.00451736
0.00451736				0
	E_102S	0.0688671	0.131223	0
0.131223				
		+/-95%	0.00149608	0.0034048
0.0034048				0
		+/-99%	0.00201624	0.00458858
0.00458858				0
	E_112S	0.0171971	0.0320807	0
0.0320807				
		+/-95%	0.000609871	0.00137888
0.00137888				0
		+/-99%	0.00082191	0.00185829
0.00185829				0
	E_133S	0.0173367	8.44284e-006	7.32702e-006
0.245717				
		+/-95%	0.000594993	0.00594207
0.00812096				0.0055355
		+/-99%	0.000801859	0.00800799
0.0109444				0.00746007
	Activity Name			
	E_106S_1reply	0.017279	0.00950748	
		+/-95%	0.331483	0.000437768
		+/-99%	0.446732	0.000589971
	receive_finish_A_6	0.0173367	0.122795	
		+/-95%	0.325849	0.00594207
		+/-99%	0.439139	0.00800799
	receive_start_A_5	0.017279	0.0320363	
		+/-95%	0.331338	0.00169149
		+/-99%	0.446536	0.00227958
	sort_finish_A_8	0	0	
		+/-95%	0	0
		+/-99%	0	0
	sort_start_A_7	0.0173367	0.122922	
		+/-95%	0.325891	0.0055355
		+/-99%	0.439196	0.00746007
Total:		0.223416	0.38645	0.154958
0.541408				
		+/-95%	0.00342238	0.00776076
0.00786493				0.00578817
		+/-99%	0.00461226	0.010459
0.0105994				0.00780058

Utilization and waiting per phase for processor: HDiskVM01

Task Name	Pri n	Entry Name	Utilization Phase 1	Phase 2
-----------	-------	------------	---------------------	---------

fileservice_vm01	0	1	write01	0.00275513	0	0
			+/-95%	0.000116732	0	0
			+/-99%	0.000157317	0	0
			read01	0.0416385	0	0
			+/-95%	0.00125836	0	0
			+/-99%	0.00169587	0	0
Task Total:				0.0443936		
			+/-95%	0.00126376		
			+/-99%	0.00170315		

Utilization and waiting per phase for processor: HDiskVM02

Task Name	Pri	n	Entry Name	Utilization	Phase 1	Phase 2
fileservice_vm02	0	1	write02	0.0045625	0	0
			+/-95%	0.000165596	0	0
			+/-99%	0.00022317	0	0
			read02	0.273191	0	0
			+/-95%	0.0105205	0	0
			+/-99%	0.0141782	0	0
Task Total:				0.277753		
			+/-95%	0.0105218		
			+/-99%	0.01418		

Utilization and waiting per phase for processor: VM01

Task Name	Pri	n	Entry Name	Utilization	Phase 1	Phase 2
controller_Source	0	1	sourcecontroller	0	0	
			+/-95%	0	0	
			+/-99%	0	0	
clienta	0	1	E_10S	0.379536	0	0
			+/-95%	0.0071806	0	0
			+/-99%	0.00967714	0	0
			E_18S	0.068399	0	0
			+/-95%	0.00258895	0	0
			+/-99%	0.00348907	0	0
			E_94A	0	0	0
			+/-95%	0	0	0
			+/-99%	0	0	0
			Activity Name			
			E_18S_1reply	1	0	
			+/-95%	0	0	
			+/-99%	0	0	
			generatefile_finish_A_2	0	0	
			+/-95%	0	0	
			+/-99%	0	0	
			generatefile_start_A_1	0	0	
			+/-95%	0	0	
			+/-99%	0	0	
			send_finish_A_4	0	0	
			+/-95%	0	0	
			+/-99%	0	0	
			send_start_A_3	0	0	
			+/-95%	0	0	
			+/-99%	0	0	
Task Total:				0.447935		

	+/-95%	0.00763307
	+/-99%	0.0102869
Processor Total:		0.447935
	+/-95%	0.00763307
	+/-99%	0.0102869

Utilization and waiting per phase for processor: VM02

Task Name	Pri n	Entry Name	Utilization	Phase 1	Phase 2
clientb	0 1	E_106S	0.00950748	0	0
		+/-95%	0.000437768	0	0
		+/-99%	0.000589971	0	0
		E_125S	0.0908444	0	0
		+/-95%	0.00335196	0	0
		+/-99%	0.00451736	0	0
		E_102S	0.131223	0	0
		+/-95%	0.0034048	0	0
		+/-99%	0.00458858	0	0
		E_112S	0.0320807	0	0
		+/-95%	0.00137888	0	0
		+/-99%	0.00185829	0	0
		E_133S	0	0	0
		+/-95%	0	0	0
		+/-99%	0	0	0
		Activity Name			
		E_106S_1reply	0.99999	0	
		+/-95%	0.000106998	0	
		+/-99%	0.000144198	0	
		receive_finish_A_6	0	0	
		+/-95%	0	0	
		+/-99%	0	0	
		receive_start_A_5	0	0	
		+/-95%	0	0	
		+/-99%	0	0	
		sort_finish_A_8	0	0	
		+/-95%	0	0	
		+/-99%	0	0	
		sort_start_A_7	0	0	
		+/-95%	0	0	
		+/-99%	0	0	
Task Total:			0.263655		
		+/-95%	0.00499212		
		+/-99%	0.00672777		

Appendix C JFT 10 KB experiment for concurrency question

G " " 1e-06 50 1 0.9 -1

P 1

p proc1 f
-1

T 31

t controller_Source r sourcecontroller -1 proc1
t clienta_4157 f E_6S -1 proc1
t controller_4114 f E_7S -1 proc1
t controller_4115 f E_8S -1 proc1
t controller_4125 f E_10S -1 proc1
t controller_4126 f E_9S -1 proc1
t controller_4127 f E_11S -1 proc1
t controller_4132 f E_15S -1 proc1
t controller_4163 f E_1S -1 proc1
t controller_4168 f E_2S -1 proc1
t controller_4201 f E_3S -1 proc1
t controller_4202 f E_4S -1 proc1
t clienta_4145 f E_12S -1 proc1
t clienta_4146 f E_13182S E_5S E_13179S E_13167S E_13080S -1 proc1
t clienta_4156 f E_13S -1 proc1
t clienta_4158 f E_14S -1 proc1
t clienta_4165 f E_27S -1 proc1
t clienta_4166 f E_16S -1 proc1
t clientb_4006 f E_17S -1 proc1
t clientb_4007 f E_19854S E_19799S E_19860S -1 proc1
t clientb_4011 f E_28S -1 proc1
t clientb_4020 f E_20S -1 proc1
t clientb_4021 f E_19S -1 proc1
t clientb_4031 f E_21S -1 proc1
t clientb_4037 f E_22S -1 proc1
t clientb_4085 f E_29S -1 proc1
t clientb_4039 f E_23S E_24S -1 proc1
t clientb_4060 f E_19895S E_19914S -1 proc1
t clientb_4061 f E_19919S E_19907S E_26S -1 proc1
t fileservice_vm01 f read01 write01 -1 proc1
t fileservice_vm02 f write02 read02 -1 proc1
-1

E 43

----- controller_Source -----
s sourcecontroller 0 -1
y sourcecontroller E_11S 1 -1
y sourcecontroller E_10S 1 -1
y sourcecontroller E_9S 1 -1
y sourcecontroller E_8S 1 -1
y sourcecontroller E_7S 1 -1
y sourcecontroller E_6S 1 -1
y sourcecontroller E_4S 1 -1
y sourcecontroller E_3S 1 -1
y sourcecontroller E_2S 1 -1
y sourcecontroller E_1S 1 -1

```

y sourcecontroller E_15S      1    -1
# ----- clienta_4157 -----
s E_6S      2.6e-09 0    -1
y E_6S      read01      1    0    -1
# ----- controller_4114 -----
s E_7S      1.19e-08 0    -1
y E_7S      read01      9    0    -1
# ----- controller_4115 -----
s E_8S      1.14e-08 0    -1
y E_8S      read01     4017    0    -1
# ----- controller_4125 -----
s E_10S     1.03e-08 0    -1
y E_10S     read01     2070    0    -1
# ----- controller_4126 -----
s E_9S      1.09e-08 0    -1
y E_9S      read01      1    0    -1
# ----- controller_4127 -----
s E_11S     9.8e-09 0    -1
y E_11S     read01     232    0    -1
# ----- controller_4132 -----
s E_15S     7.5e-09 0    -1
y E_15S     read01      4    0    -1
# ----- controller_4163 -----
s E_1S      1.52e-08 0    -1
y E_1S      read01     179    0    -1
y E_1S      E_13167S     2    0    -1
y E_1S      E_13080S     1    0    -1
y E_1S      E_5S         1    0    -1
y E_1S      E_13179S     2    0    -1
y E_1S      E_16S        1    0    -1
y E_1S      E_12S        1    0    -1
y E_1S      E_14S        1    0    -1
y E_1S      E_13S        1    0    -1
# ----- controller_4168 -----
s E_2S      1.46e-08 0    -1
y E_2S      read01     35    0    -1
y E_2S      E_13182S     1    0    -1
y E_2S      E_13167S     3    0    -1
y E_2S      E_13080S     1    0    -1
# ----- controller_4201 -----
s E_3S      1.4e-08 0    -1
y E_3S      E_23S        1    0    -1
y E_3S      E_22S        1    0    -1
y E_3S      E_20S        1    0    -1
y E_3S      E_28S        1    0    -1
y E_3S      E_19854S     1    0    -1
y E_3S      E_19799S     2    0    -1
y E_3S      E_21S        1    0    -1
y E_3S      E_27S        1    0    -1
y E_3S      E_19860S     1    0    -1
y E_3S      E_17S        1    0    -1
y E_3S      E_19S        1    0    -1
y E_3S      E_29S        1    0    -1
# ----- controller_4202 -----
s E_4S      1.36e-08 0    -1
y E_4S      E_19799S     1    0    -1

```

```

y E_4S      E_19860S      1      0      -1
# ----- clienta_4145 -----
s E_12S      1.9e-09 0      -1
y E_12S      read01      9      0      -1
# ----- clienta_4146 -----
s E_13182S    0.00142 0      -1
y E_13182S    read01      84     0      -1
y E_13182S    E_19914S     1      0      -1
y E_13182S    E_19919S     1      0      -1
y E_13182S    write01     22     0      -1
y E_13182S    E_19907S     1      0      -1
y E_13182S    E_19895S     1      0      -1
s E_5S        2.7e-09 0      -1
y E_5S        E_24S        1      0      -1
y E_5S        read01     3913    0      -1
y E_5S        E_26S        1      0      -1
y E_5S        E_19895S     1      0      -1
s E_13179S    9.02e-05 0      -1
y E_13179S    write01     2.5    0      -1
s E_13167S    0.0236 0      -1
s E_13080S    0.00324 0      -1
y E_13080S    read01     208.5  0      -1
# ----- clienta_4156 -----
s E_13S       1.8e-09 0      -1
y E_13S       read01     2070    0      -1
# ----- clienta_4158 -----
s E_14S       1.7e-09 0      -1
y E_14S       read01     182     0      -1
# ----- clienta_4165 -----
s E_27S       3e-10 0      -1
y E_27S       read01      2      0      -1
# ----- clienta_4166 -----
s E_16S       1.4e-09 0      -1
y E_16S       read01      2      0      -1
# ----- clientb_4006 -----
s E_17S       9e-10 0      -1
y E_17S       read02      9      0      -1
# ----- clientb_4007 -----
s E_19854S    3.53e-07 0      -1
y E_19854S    write02      5      0      -1
s E_19799S    0.0014 0      -1
y E_19799S    read02     150.667 0      -1
s E_19860S    3.67e-07 0      -1
y E_19860S    read02     2054.5 0      -1
y E_19860S    write02     15.5    0      -1
# ----- clientb_4011 -----
s E_28S       0      0      -1
y E_28S       read02      2      0      -1
# ----- clientb_4020 -----
s E_20S       6e-10 0      -1
y E_20S       read02     2070    0      -1
# ----- clientb_4021 -----
s E_19S       7e-10 0      -1
y E_19S       read02      1      0      -1
# ----- clientb_4031 -----
s E_21S       6e-10 0      -1

```

```

y E_21S      read02      182   0   -1
# ----- clientb_4037 -----
s E_22S      5e-10   0   -1
y E_22S      read02       2   0   -1
# ----- clientb_4085 -----
s E_29S      0   0   -1
y E_29S      read02       6   0   -1
# ----- clientb_4039 -----
s E_23S      0   0   -1
s E_24S      4e-10   0   -1
y E_24S      read02       2   0   -1
# ----- clientb_4060 -----
s E_19895S    1.96e-07 0   -1
y E_19895S    read02      5   0   -1
s E_19914S    5.25e-07 0   -1
# ----- clientb_4061 -----
s E_19919S    6.09e-07 0   -1
y E_19919S    read02      4   0   -1
y E_19919S    write02     3   0   -1
s E_19907S    1.41e-06 0   -1
y E_19907S    read02      6   0   -1
s E_26S      3e-10   0   -1
# ----- fileservice_vm01 -----
s read01      7.99e-05 0   -1
s write01     5.35e-05 0   -1
# ----- fileservice_vm02 -----
s write02     1.83e-05 0   -1
s read02      0.000103 0   -1
-1

```

Appendix D Layered system LQN file

```
G " " 1e-06 50 1 0.9 -1

P 1
  p vm1 f
  p vm2 f
-1

T 6
  t User_Source r sourceUser -1 vm1
  t Server01 f E_1S E_9A E_6A -1 vm1
  t Server02 f E_2S E_8S -1 vm2
  t Server03 f E_3S -1 vm1
  t Server04 f E_4S -1 vm2
  t Server05 f E_5S -1 vm2
-1

E 9
# ----- User_Source -----
s sourceUser 1e-06 -1
y sourceUser E_5S 0.0909091 -1
y sourceUser E_2S 0.181818 -1
y sourceUser E_3S 0.181818 -1
y sourceUser E_4S 0.181818 -1
y sourceUser E_1S 0.181818 -1
z sourceUser E_9A 0.0909091 -1
z sourceUser E_6A 0.0909091 -1
# ----- Server01 -----
s E_1S 90800 0 -1
s E_9A 0.0265 0 -1
y E_9A E_3S 1 0 -1
s E_6A 0.0241 0 -1
y E_6A E_8S 1 0 -1
# ----- Server02 -----
s E_2S 0.0223 0 -1
s E_8S 0.0169 0 -1
y E_8S E_5S 0 1 -1
y E_8S E_4S 0 1 -1
# ----- Server03 -----
s E_3S 60500 0 -1
# ----- Server04 -----
s E_4S 0.0793 0 -1
# ----- Server05 -----
s E_5S 0.0129 0 -1
-1
```

References

- [1] Ackermann, V., Grohmann, J., Eismann, S., & Kounev, S., "Black-box Learning of Parametric Dependencies for Performance Models", *CEUR Workshop Proceedings*, vol. 2245, pp. 78-86, USA, 2017.
- [2] Bondi, A.B., *Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice*, Addison-Wesley, 2014.
- [3] Desnoyers, M., *Using the Linux Kernel Tracepoints*. Retrieved from Linux Kernel Documentation:
<https://www.kernel.org/doc/Documentation/trace/tracepoints.txt>, Nov 29 2018.
- [4] Eclipse Foundation, *Trace Compass User Guide*. Retrieved from Eclipse Archives:
<http://archive.eclipse.org/tracecompass/doc/stable/org.eclipse.tracecompass.doc.user/User-Guide.html>, Nov 29 2018.
- [5] El-Sayed, H., Cameron, D., & Woodside, C., "Automation Support for Software Performance Engineering", *Proc Joint Int. Conf on Measurement and Modeling of Computer Systems (Sigmetrics 2001/ Performance 2001)*. Cambridge, 2001.
- [6] Ezzati, N., Nemati, H., & Tétreault, F., "Execution Flow Analysis Across Virtualized Environment for Performance Understanding and Optimization", *Tracing Summit 2018*. Retrieved from
<https://tracingsummit.org/wiki/TracingSummit2018>, Dec 01, 2018.
- [7] Franks, G., Maly, P., Woodside, M., Petriu, D., Hubbard, A., & Mroz, M., *LQN User Manual*, Retrieved from www.layeredqueues.org/:
www.sce.carleton.ca/rads/lqns/LQNSUserMan-jan13.pdf, Jan 01, 2019.
- [8] G. Franks, D. Petriu, M. Woodside, J. Xu, P. Tregunno, "Layered bottlenecks and their mitigation", *Proc of 3rd Int. Conference on Quantitative Evaluation of Systems QEST'2006*, Riverside, CA, USA, pp. 103-114, Sept. 2006
- [9] Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, Salem Derisavi, "Enhanced Modeling and Solution of Layered Queueing Networks", *IEEE*

Trans. on Software Eng, vol. 35, no.2, pp. 148-161, Aug, 2008

- [10] Gregg, B., *Linux Performance*, Retrieved from Perf Linux Profiler:
<http://www.brendangregg.com/perf.html>, Nov 29,2018
- [11] Grosso, W., *Java RMI*, Cambridge: O'Reilly Media, Inc, 2011.
- [12] Hrischuk, C., Rolia, J., & Woodside, M., "Automatic Generation of a Software Performance Model Using an Object-Oriented Prototype". *Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pg 399-409, 1995.
- [13] Israr, T. A., *A Lightweight Technique for Extracting Software Architecture and Performance Models from Traces*, Master Thesis, Carleton University, Ottawa, Canada, 2001.
- [14] Kerrisk, M., *The Linux Programming Interface*, pp.388-882, San Francisco, 2010.
- [15] Lazowska, E., Zahorjan, J., Graham, G., & Sevcik, K., *Quantitative System Performance*. New Jersey: Prentice-Hall, 1984.
- [16] Li, H., *Linear Interpolation*. Retrieved from Smile - Statistical Machine Intelligence and Learning Engine: <http://haifengl.github.io/smile/>, Feb 01, 2019.
- [17] Linux Foundation, *Babeltrace*. Retrieved from Diamon Project:
<https://diamon.org/babeltrace/>, Nov 29,2018.
- [18] LTTng Community, *lttng-view*. Retrieved from LTTng Manual Page:
<https://lttng.org/man/1/lttng-view/v2.8/>, Nov 29,2018.
- [19] LTTng Community, *The LTTng Documentation*. Retrieved from LTTng.org:
<https://lttng.org/docs/v2.10/>, Nov 29,2018.
- [20] Maly, P., & Woodside, C., "Layered Modeling of Hardware and Software, with Application to A LAN Extension Router", *International Conference on Modelling Techniques and Tools fore Computer Performance Evaluation TOOLS 2000*, Springer, Chicago, pp. 10-24, 2001.
- [21] Mani, N, Petriu, D.C., Woodside, C.M., "Towards studying the performance effects of design patterns for service oriented architecture", *Proc. Int. Conf. on*

- Performance Engineering (ICPE 2011)*, Karlsruhe, pp 499-504, May 2011.
- [22] Mizannojehdehi, A., *Automatic Trace-Based Performance Evaluation Model Building for Concurrent Distributed Systems*, PhD Thesis, Carleton University, Ottawa, Canada, 2013.
 - [23] Niemeyer, P., & Knudsen, J., *Learning Java, 3rd Edition*. O'Reilly Media, Inc, 2005.
 - [24] Smith, C., & Williams, L., *Performance Solutions : A Practical Guide to Creating Responsive, Scalable Software*. United States : Pearson Education, 2001.
 - [25] Spinner, S., Casale, G., Brosig, F., & Kounev, S., "Evaluating Approaches to Resource Demand Estimation", *Performance Evaluation*, Vol 92, October 2015.
 - [26] The Linux Foundation, *The Common Trace Format*. Retrieved from Diamon.org: <https://diamon.org/ctf/>, Nov 29, 2018.
 - [27] Woodside, M., Franks, G., & Petriu, D., "The Future of Software Performance Engineering", *International Conference on Software Engineering, ICSE 2007*, USA, pg 171-187, May 2007.
 - [28] Woodside, M., *Tutorial Introduction to Layered Modeling of Software Performance*. Ottawa: Carleton University, 2013.
 - [29] Woodside, M., *Papers, Reports and Presentations on Layered Queueing*. Retrieved from www.layeredqueues.org: www.layeredqueues.org/bibliography.html, Jan, 2019.
 - [30] Wu, P., *A Performance Model for a Network of Prototype Software Routers*, Master Thesis, Carleton University, Ottawa, Canada, 2003.