

2006

## A methodology for software performance modeling and its application to a border inspection system

Paola Bracchi  
*West Virginia University*

Follow this and additional works at: <https://researchrepository.wvu.edu/etd>

---

### Recommended Citation

Bracchi, Paola, "A methodology for software performance modeling and its application to a border inspection system" (2006). *Graduate Theses, Dissertations, and Problem Reports*. 2478.  
<https://researchrepository.wvu.edu/etd/2478>

This Thesis is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This Thesis has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact [researchrepository@mail.wvu.edu](mailto:researchrepository@mail.wvu.edu).

**A Methodology for Software Performance Modeling and its  
Application to a Border Inspection System**

Paola Bracchi

Thesis submitted to the  
College of Engineering and Mineral Resources  
at West Virginia University  
in partial fulfillment of the requirements  
for the degree of

Master of Science in Computer Science

Bojan Cukic, Ph.D., Chair  
Elaine Eschen, Ph.D.  
Katerina Goseva-Popstojanova, Ph.D.

Lane Department of  
Computer Science and Electrical Engineering

Morgantown, West Virginia  
2006

*Keywords:* software engineering, software performance, UML, Layered Queuing  
Networks

© Paola Bracchi, 2006

## ABSTRACT

### A Methodology for Software Performance Modeling and its Application to a Border Inspection System

Paola Bracchi

It is essential that software systems meet their performance objectives. Many factors affect software performance and it is fundamental to identify those factors and the magnitude of their effects early in the software lifecycle to avoid costly and extensive changes to software design, implementation, or requirements. In the last decade the development of techniques and methodologies to carry out performance analysis in the early stages of the software lifecycle has gained a lot of attention within the research community. Different approaches to evaluate software performance have been developed. Each of them is characterized by a certain software specification and performance modeling notation.

In this thesis we present a methodology for predictive performance modeling and analysis of software systems. We use the Unified Modeling Language (UML) as a software modeling notation and Layered Queuing Networks (LQN) as a performance modeling notation. Our focus is on the definition of a UML to LQN transformation. We extend existing approaches by applying the transformation to a different set of UML diagrams, and propose a few extensions to the current “UML Profile for Schedulability, Performance, and Time”, which we use to annotate UML diagrams with performance-related information. We test the applicability of our methodology to the performance evaluation of a complex software system used at border entry ports to grant or deny access to incoming travelers.

## ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor, Dr. Bojan Cukic, for giving me the opportunity to work with him, and for his continuous support and supervision in the development of this research and the related publications. I would also like to thank my other committee members, Dr. Elaine Eschen and Dr. Katerina Goseva-Popstojanova for their prompt availability and directions.

I am very grateful to my former advisor, Dr. Vittorio Cortellessa, for introducing me to Dr. Cukic and to West Virginia University. During my studies here, I appreciated his constant encouragement and wise suggestions.

Finally, special thanks to my family for their enduring love and assistance. Many thanks also to my friends and lab mates. They have made my stay in Morgantown an unforgettable experience.

Morgantown, November 1 2006

*Paola Bracchi*

# Table of Contents

<b>List of Figures.....</b>	<b>vii</b>
<b>List of Tables .....</b>	<b>x</b>
<b>List of Abbreviations .....</b>	<b>xi</b>
<b>Chapter 1: Introduction .....</b>	<b>1</b>
1.1 Software Performance .....	2
1.2 Software Performance Evaluation .....	3
1.2.1 Performance Modeling.....	4
1.2.2 Performance Data Collection.....	5
1.2.3 Performance Analysis .....	6
1.3 Thesis Contribution .....	8
1.4 Thesis Outline.....	8
<b>Chapter 2: Literature Review.....</b>	<b>10</b>
2.1 Software Specification Models.....	10
2.2 Performance Models.....	12
2.2.1 Queuing Networks .....	13
2.2.2 Stochastic Timed Petri Nets.....	14
2.2.3 Stochastic Process Algebras .....	16
2.2.4 Simulation Models .....	17
2.3 Evaluation of Performance Models .....	17
2.3.1 Queuing Networks .....	17

2.3.2	Stochastic Timed Petri Nets.....	18
2.3.3	Stochastic Process Algebras .....	19
2.3.4	Simulation Models.....	20
<b>Chapter 3: A Methodology for Early Software Performance Analysis.....</b>		<b>21</b>
3.1	Software Specification Model .....	21
3.1.1	UML.....	22
3.1.2	UML Diagrams .....	22
3.1.3	UML Performance Profile .....	27
3.2	Performance Model .....	30
3.2.1	LQN .....	31
3.2.2	LQN Tools .....	33
3.3	UML to LQN Transformation .....	34
3.3.1	Previous Work .....	34
3.3.2	Our Approach.....	35
<b>Chapter 4: Case Study.....</b>		<b>59</b>
4.1	System Description.....	59
4.1.1	Context.....	60
4.1.2	Structure.....	61
4.1.3	Functions.....	63
4.1.4	Technical and Policy Options .....	72
4.2	Performance Modeling .....	74
4.2.1	Assumptions.....	74
4.2.2	Model Structure .....	75
4.2.3	Model Dynamics.....	78
4.2.4	Model Parameters .....	92
4.3	Performance Experiments.....	92

4.3.1	Technical Options .....	93
4.3.2	Authentication Policies .....	94
4.3.3	Manual Inspection Times.....	95
4.3.4	Biometric Sampling Times .....	96
4.4	Results and Analysis.....	96
4.4.1	Technical Options .....	97
4.4.2	Authentication Policies .....	103
4.4.3	Manual Inspection Times.....	104
4.4.4	Biometric Sampling Times .....	106
4.5	Validation .....	108
<b>Chapter 5: Conclusions .....</b>		<b>110</b>
<b>Appendix A: Parameterization of LQN Model for Options 1 and 2.....</b>		<b>113</b>
A.1	Assumed Execution Environment .....	113
A.2	Expected Size of Data.....	115
A.3	Performance Annotations .....	115
A.4	Model Parameters .....	122
<b>Appendix B: Option 3.....</b>		<b>125</b>
B.1	Description.....	125
B.2	Performance Modeling .....	127
B.2.1	Model Structure .....	128
B.2.2	Model Dynamics.....	128
B.2.3	Model Parameters .....	131
<b>References.....</b>		<b>135</b>

# List of Figures

Figure 1: Example of QN model.....	13
Figure 2: Example of PN model .....	15
Figure 3: Example of Use Case Diagram .....	24
Figure 4: Example of Sequence Diagram .....	26
Figure 5: Example of Deployment Diagram.....	27
Figure 6: Performance analysis domain model.....	28
Figure 7: Example of LQN model .....	33
Figure 8: High-level algorithm for UML to LQN transformation.....	37
Figure 9: Annotated Use Case Diagram .....	39
Figure 10: Annotated Sequence Diagram .....	42
Figure 11: Annotated Deployment Diagram.....	44
Figure 12: Mapping from Deployment Diagram elements to LQN devices .....	45
Figure 13: Mapping from Deployment Diagram elements to LQN tasks .....	46
Figure 14: Mapping between LQN tasks and corresponding devices .....	47
Figure 15: Example of <i>opt</i> fragment.....	50
Figure 16: Translation of <i>opt</i> fragment in LQN notation .....	50
Figure 17: Example of <i>alt</i> fragment.....	51
Figure 18: Translation of <i>alt</i> fragment in LQN notation .....	52
Figure 19: Example of <i>par</i> fragment .....	53



Figure 20: Translation of <i>par</i> fragment in LQN notation .....	53
Figure 21: Example of <i>loop</i> fragment.....	54
Figure 22: Translation of <i>loop</i> fragment in LQN notation .....	54
Figure 23: Sample LQN model at the end of Step 2.....	56
Figure 24: Sample LQN model at the end of Step 3.....	58
Figure 25: Possible Deployment Diagram for the airport inspection system .....	61
Figure 26: Use Case Diagram for the airport inspection system .....	64
Figure 27: Sequence Diagram for the Traveler Inspection use case.....	65
Figure 28: Sequence Diagram for the Traveler Authentication interaction.....	66
Figure 29: Sequence Diagram for the MRTD Authentication interaction.....	68
Figure 30: Sequence Diagram for the TNS Name Check interaction.....	69
Figure 31: Sequence Diagram for the Secondary Inspection interaction .....	70
Figure 32: Sequence Diagram for the Name-based Lookup use case .....	71
Figure 33: Sequence Diagram for the Biometric Verification use case.....	72
Figure 34: Sequence Diagram for the Biometric Identification use case .....	72
Figure 35: LQN devices for the airport inspection system.....	76
Figure 36: LQN tasks for the airport inspection system .....	77
Figure 37: LQN tasks, devices, and their mappings for the airport inspection system	78
Figure 38: High-level framework of the LQN for the airport inspection system .....	82
Figure 39: <i>Traveler Inspection LQN</i> after Traveler Inspection scenario.....	83
Figure 40: <i>Traveler Inspection LQN</i> after Traveler Authentication .....	84
Figure 41: <i>Traveler Inspection LQN</i> after MRTD Authentication .....	85
Figure 42: <i>Traveler Inspection LQN</i> after TNS Name Check .....	86
Figure 43: <i>Primary Inspection LQN</i> after <i>Secondary Inspection</i> .....	87
Figure 44: High-level layout of the <i>Traveler Inspection LQN</i> .....	89
Figure 45: <i>Name-based Lookup LQN</i> .....	90
Figure 46: High-level layout of the LQN for the airport inspection system.....	91

Figure 47: Primary inspection time for different technical options.....	98
Figure 48: Primary inspection throughput for different technical options .....	98
Figure 49: Primary total waiting time for different technical options .....	100
Figure 50: Primary total waiting time vs. airports served by a remote PKD.....	101
Figure 51: Primary total waiting time for different authentication scenarios.....	104
Figure 52: Primary inspection time for different authentication scenarios.....	104
Figure 53: Primary total waiting time for different manual inspection times.....	106
Figure 54: Primary inspection time for different manual inspection times .....	106
Figure 55: Primary total waiting time for different biometric sampling times.....	107
Figure 56: Primary inspection time for different biometric sampling times .....	108
Figure 57: Validation of travelers' total waiting time .....	109
Figure 58: Deployment Diagram for Option 3 .....	126
Figure 59: Use Case Diagram for Option 3 .....	127
Figure 60: Sequence Diagram for the PK Certificate Retrieval use case .....	127
Figure 61: LQN request flow after MRTD Authentication .....	129
Figure 62: High-level layout of the LQN model for the airport inspection system (Option 3).....	130

# List of Tables

Table 1: Types of Sequence Diagram messages.....	26
Table 2: Response time and resource utilization for <i>PKD Shared</i> Option .....	103
Table 3: Execution environment (Options 1 and 2).....	114
Table 4: Expected size of data (Options 1 and 2).....	115
Table 5: Resource demand of scenario steps (Options 1 and 2).....	116
Table 6: LQN parameters for system workloads (Options 1 and 2).....	122
Table 7: LQN parameters for resource demands (Options 1 and 2).....	123
Table 8: Execution environment (Option 3).....	131
Table 9: Resource demand of scenario steps (Option 3).....	132
Table 10: LQN parameters for system workloads (Option 3).....	133
Table 11: LQN parameters for resource demands (Option 3).....	133

# List of Abbreviations

<i>DD</i>	Deployment Diagram
<i>DS</i>	Digital Signature
<i>ICAO</i>	International Civil Aviation Organization
<i>GSPN</i>	Generalized Stochastic Petri Net
<i>LQN</i>	Layered Queuing Network
<i>MC</i>	Markov Chain
<i>MRTD</i>	Machine Readable Travel Document
<i>MRZ</i>	Machine Readable Zone
<i>MSC</i>	Message Sequence Chart
<i>OMG</i>	Object Management Group
<i>PA</i>	Process Algebra
<i>PKD</i>	Public Key Directory
<i>PKI</i>	Public Key Infrastructure
<i>PN</i>	Petri Net
<i>POE</i>	Port Of Entry
<i>QN</i>	Queuing Network
<i>SA</i>	Software Architecture
<i>SD</i>	Sequence Diagram
<i>SPA</i>	Stochastic Process Algebra

<i>SPE</i>	Software Performance Engineering
<i>STPN</i>	Stochastic Timed Petri Net
<i>UC</i>	Use Case
<i>UCD</i>	Use Case Diagram
<i>UCM</i>	Use Case Map
<i>UML</i>	Unified Modeling Language

# Chapter 1: Introduction

Traditional software development process is focused on meeting software functional requirements. Performance issues are usually ignored or considered only towards the end of the software lifecycle. This may cause possible performance problems to require extensive and costly changes at the implementation, design, or, even worse, requirement level.

Over the last decade the research community has been very active in the development of techniques and procedures to avoid these scenarios, proposing different approaches to integrate performance analysis early in the software lifecycle. Although some of them have been successfully applied to case studies both in academic and in industrial environments, a widespread integration of performance assessment in the software development process is not established yet.

This chapter provides an introduction to software performance evaluation. It reviews what software performance is, how it is evaluated, and the benefits of evaluating it. Additionally, it explains how the work and research presented here contributes to the field of software performance engineering. The chapter concludes with a brief outline of the remainder of the thesis.

## 1.1 Software Performance

*Performance* is generally indicative of “[...] how well a system, assumed to perform correctly, works” [16]. Performance represents a fundamental quality attribute of every software system. In particular, according to typical use of this term, it refers to the quality of service provided by the system. Classical performance measures include system-oriented measures such as throughput, resource utilization, and scalability, or user-oriented measures such as waiting time, service time, and queue length. Additional metrics are specific to particular types of software systems, such as power consumption for mobile applications, or bandwidth utilization for networked applications.

It is fundamental to evaluate software performance since the early stages of the software lifecycle to reduce the risk of performance failures. In fact, experience shows that “performance problems are most often due to inappropriate architectural choices, rather than inefficient coding” [52]. The discovery of performance issues in the development, testing, or, even worse, operational phase, requires costly fixes, schedule delays, lost productivity, lost income, damaged organization’s image, etc. In extreme cases problems may be so severe to require considerable redesign and reimplementation, or even project failure [50].

*Software performance evaluation* is the process of predicting (early in the software development process) or assessing (towards the end of the development process) whether a software system is able to meet established performance objectives [5]. In this thesis we focus on early model-based performance evaluation, which relies on two basic steps: the definition of a performance model, according to a suitable description of the software system, and the solution of the performance model to obtain performance results.

## 1.2 Software Performance Evaluation

Software performance evaluation requires a systematic, comprehensive process to characterize the dynamic behavior of a software system in quantitative terms. In this section we outline the main steps of a generic process, based on the Software Performance Engineering (SPE) approach [49] described in [50].

The first step towards software performance analysis should be the assessment of performance risk, so as to understand the level of effort to put into performance evaluation activities. This can be minimal if the system under consideration is not critical to the mission of the organization, or if similar projects have previously been successfully deployed, etc. Otherwise a more significant commitment to performance evaluation is required.

The next step is to understand system functions and design based on appropriate abstractions of the software system. Possible abstractions include software requirements, architectures, specifications, and design documents. In particular, “since performance is a runtime attribute of a software system, performance analysis requires suitable descriptions of the software runtime behavior” [5]. Examples of such descriptions are UML Interaction Diagrams (e.g., Sequence Diagrams, Activity Diagrams), Message Sequence Charts, finite state automata, etc.

Next performance objectives have to be established. They should be expressed in quantitative terms using well defined metrics, usually response time, throughput, and resource utilization. *Response time* is usually intended as the time taken by the system to respond to a request from a user, or from an external system or event. *Throughput* corresponds to the number of requests processed per unit of time. *Resource utilization* is defined as the fraction of a hardware or software resources used by the system to respond to incoming requests.

Performance models are then built and parametrized. More details about these steps are given in the next subsections. Different notations and tools can be used,



depending on the adopted performance evaluation methodology. The next chapter reviews the most relevant options available at this purpose. Analysis of results from the evaluation of performance models indicates if the system is able to meet the established performance goals. If not, system design or performance objectives have to be revised.

Verification and validation of performance models are carried out in parallel with the definition and solution of performance models. Model verification answers the question “Are we building the model right?”. It intends to determine if performance results accurately reflect the actual system performance. On the other hand, model validation answers the question “Are we building the right model?” [9]. It aims at identifying whether the built performance models are accurate descriptions of the structural and behavioral characteristics of the system in exam.

### **1.2.1 Performance Modeling**

A performance model of a software system can be defined using appropriate abstractions of the system structure and functions. The earliest description providing this information is Software Architecture (SA), defined as “the structure or structures of the system, which comprises software components, the externally visible properties of those components, and the relationships among them” [7].

The model should be able to represent factors affecting performance such as:

- system workload;
- hardware service rates;
- software components internal dynamics;
- interactions between software components;
- replicating or multi-threading of software components;
- allocation of software components to hardware platforms;
- software contention, i.e., the time spent to access software resources;
- hardware contention, i.e., the time spent to access a hardware resource;

- demand of software components on hardware devices such as processors, disks, networks, etc.

The level of detail of a performance model should match the degree of abstraction emerging from available system descriptions. The performance model should also be simple with respect to its expression in the adopted modeling notation and to its solution. However, taking into account all the previously listed factors affecting performance could lead to complex models, even for small-sized systems. Therefore, the choice of the most appropriate performance modeling methodology should be driven by an attentive evaluation of the tradeoffs between strengths and weaknesses of candidate modeling notations (e.g., Markov chains, Petri nets, Queuing Networks, Process Algebras, etc.) and the factors affecting performance which are important and/or possible to include in the model.

## **1.2.2 Performance Data Collection**

The hardest part of the performance evaluation process, especially early in the software lifecycle, is the collection of data required to parameterize performance models. Missing data refer to the system execution environment and to software resource requirements. Information about the execution environment includes the system hardware configuration, service rates of computing devices and communication links, number of replicas of processors, disks, etc. Information about software resource requirements consists of the demand of software components on devices in the hardware configuration.

Several options to gather performance data are available. The viability of each option depends on the phase of the development process the software system is in. Early in the system lifecycle precise information is not available. At this stage, the best way to obtain early performance data is through performance walkthroughs, which consist of questioning system experts about system functions and design, the execution environment, expected workload intensity, etc. [50]. If performance

walkthroughs are not possible, approximations, guesses, and estimates of upper and lower bound requirements can be used [50].

Once a system prototype or an implementation are available, parameters for performance models can be obtained through measurements. Tools are available to provide system-level measurements, such as the percentage of time the CPU is busy or code-level measurements, such as the number of times a program executes a particular method [50]. These tools are well defined and widespread; however it is usually difficult to obtain the required information using them. For this reason the best alternative to collect performance data is internal instrumentation through “code (probes) inserted at key points to measure pertinent execution characteristics” [50]. Instrumentation provides a convenient way to obtain data at the desired level of granularity. Another advantage of instrumentation is that it is possible to enable it when it is needed and to disable it otherwise.

Verification of performance data is very important. In fact, the accuracy of performance results depends on the precision of the parameters used to evaluate performance models. Early in the software lifecycle accuracy cannot be high because knowledge of system details is vague and system resource requirements are difficult to estimate [50]. At this stage it is not possible to identify or estimate errors. However, it is possible to evaluate their effect on the performance results conducting a sensitivity analysis [43]. Later, as the development process progresses and more accurate data become available through (partial) software implementations and prototypes, the current estimates can be updated.

### **1.2.3 Performance Analysis**

Performance analysis is the evaluation of the quantitative results obtained from the solution of a performance model. Early performance analysis poses problems due to incompleteness of the software specification, the lack of knowledge about resource requirements, and other issues such as ignorance of the actual workload intensity.

However, sources of deficiencies in the analysis can be identified, and their effects on performance results can be estimated so that high-level performance questions can be addressed at a level of abstraction comparable with that of the software specification [43].

Performance results can report different types of performance indices such as response time, throughput, and resource utilization. The relative importance of each measure depends on the system specifics. For instance, in an interactive web application we may pay more attention to response time, i.e., the total time for a user to complete an interaction with the system. On the other hand, a web service providing commercial services to other applications may give more relevance to throughput, to maximize the number of processed requests, hence profit.

Performance analysis evaluates performance results against the established performance objectives. If these are satisfied nothing needs to be done. Otherwise utilization measures should be explored to identify possible bottlenecks, i.e., overloading of one or more resources. If any bottleneck is found the classic solution to the problem consists of cloning the involved resource, using for instance multi-threading of software processes, multiple processors, or multiple buffers. Repeatedly adjusting the number of instances for different resources in the performance model, and evaluating the performance results, the utilization of the software or hardware components in exam should set to a lower, acceptable level [57].

If performance objectives are not met even after executing the previous step, no standard solution is available. Performance problems have to be addressed using a project-specific strategy. Typical causes for not meeting performance requirements are “execution demand, long scenario paths, or lack of concurrency in the system” [57]. Typical solutions include “changing the scenario design, shortening long scenarios, decomposing large components, using more efficient scheduling strategies, and modifying the deployment” [57].

These solutions can be applied iteratively until performance requirements are finally met (assuming they are reasonable). Afterwards the changes applied to the performance model can be translated into software design model and system configuration description. The obtained information should be reviewed by system designers in the software architecture and software specification phase [57].

### **1.3 Thesis Contribution**

In this thesis we present a methodology to address the problem of early performance analysis of software systems. The methodology uses UML as software modeling notation and LQN as performance modeling notation. We propose a transformation to automatically derive a LQN model from a set of UML diagrams annotated with performance-related information using extensions defined in the “UML Profile for Schedulability, Performance, and Time” [36]. The transformation is largely inspired by previous work presented in [20, 21, 39, 40, 41, 46]; however, our contribution is the adaptation of existing techniques to a different set of UML diagrams, that are more suitable to be used in early stages of the software development lifecycle, compared to those used by existing transformations. Another contribution is the suggestion of extensions to the UML Performance Profile, to allow a more convenient specification of the performance characteristics of the system. Extensions are also proposed to address gaps in the current Profile, which does not cover UML 2.0 diagrams.

### **1.4 Thesis Outline**

The remainder of this thesis is organized as follows: Chapter 2 provides a review of model-based techniques that have been investigated to address the problem of evaluating software performance. Chapter 3 explains the methodology we adopted to develop performance models based on a set of annotated UML diagrams. Chapter 4

describes our case study, reports and analyzes the results we obtained from performance evaluation. Finally, Chapter 5 states our conclusions and directions for further research.

## **Chapter 2: Literature Review**

Many approaches to analyze software performance based on early software descriptions have been proposed in the last ten years. Each approach is characterized by a certain software specification language (e.g., UML, Message Sequence Charts, Petri Nets, etc.) and a certain performance modeling notation (e.g., Queuing Networks and their extensions, Stochastic Process Algebras, simulation models, etc.).

This chapter briefly reviews the most used software and performance modeling notations. For each notation, its strengths and weaknesses are identified, and its suitability to be adopted to conduct a software performance evaluation is discussed. Finally, notations are compared based on factors such as easiness in specifying models starting from early software abstractions, easiness in modifying models as feedback from performance evaluation suggests changes in model structure or parameters, easiness in solving models using analytic or simulation methods, and suitability for use within an automated performance evaluation process.

### **2.1 Software Specification Models**

Software specification models describe static and dynamic aspects of software systems. A static description represents software modules or components and their interconnections. A dynamic description represents software behavior at runtime.

Many options are available to describe software specifications. Possible notations include Petri Nets, Process Algebra, Automata, Message Sequence Charts, Use Case Maps, and the Unified Modeling Language (UML).

Petri Nets [44], Process Algebra [35, 37, 26], and Automata [27] are formal specification languages. They have the advantage of an exact semantics but, on the negative side, they are complex to integrate within common software engineering practice. This limit is overcome by less formal notations such as Message Sequence Charts, Use Case Maps, and the Unified Modeling Language, which are described below.

Message Sequence Charts (MSC) [47] represent a language to describe communication between independent instances of a software system (i.e., modules, components, processes, etc.), or between those instances and the system environment. MSC also allow the expression of restrictions on communicated data values and on the timing of events. MSC are provided with a graphical representation that looks similar to UML Sequence Diagrams.

Use Case Maps (UCM) [11] represent a visual notation to combine the description of system structure and behavior in a single model. The aim of UCM is to help software designers to grasp large grained software behavior patterns. UCM can be used during early stages of the software lifecycle, i.e., at requirement and high-level design level. However, they are not expressive enough to be used in later phases for they are not suited to completely specify software structure and dynamics.

The Unified Modeling Language (UML) [10] is a notation specified by the Object Management Group (OMG), an industry group dedicated to promoting Object-Oriented (OO) technology and its standardization. UML allows to visually represent different views of software systems at different levels of abstraction. At present UML diagrams are widely accepted and adopted within both industry and academic environments because they are flexible and easy to use and maintain. A variety of



diagrams is available to model static and dynamic aspects of software systems (e.g., Use Case Diagrams, Sequence Diagrams, and Deployment Diagrams). To enable users to integrate performance evaluation into early software specifications, OMG defined and adopted the “UML Profile for Schedulability, Performance, and Time” (SPT Profile) [36]. The SPT Profile introduces stereotypes, tagged values, and constraints to formally specify performance annotations (workload information, resource requirements, etc.).

## **2.2 Performance Models**

Different modeling notations can be used to carry out a performance analysis of software systems during early phases of the software life cycle. Three main classes of performance models are available: Queuing Network [32, 34, 51], Stochastic Process Algebra [8, 23, 24], and Stochastic Timed Petri Net [1, 2, 3]. Queuing Networks were initially proposed to represent performance typical features of hardware or manufacturing systems; notations like Petri Nets and Process Algebras were first introduced in the software specification field and then exported to the performance domain.

Performance models based on the above notations can be solved by simulation or by analytical methods. “Simulation is a widely used general technique whose drawback is the potential high development and computational cost to obtain accurate results” [14]. On the other hand, analytical methods can often be applied to simple models only, which cannot adequately capture real systems behavior. Analytical solution of performance models is based on a stochastic process that is usually a continuous-time discrete-space homogeneous Markov Chains (MC) [31].

## 2.2.1 Queuing Networks

A Queuing Network (QN) [32, 34, 51, 30] model can be described as “a collection of *service centers*, which represent system resources, and *customers*, which represent users or transactions” [34]. It consists of a direct graph whose nodes are service centers. Nodes are connected by edges expressing the flow of customers’ service requests. The model has a graphical representation shown in Figure 1.

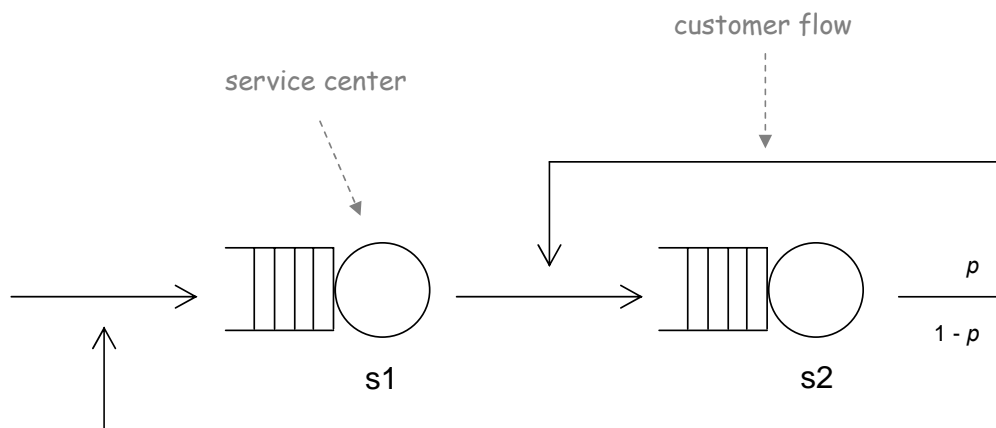


Figure 1: Example of QN model

QN models have been extensively applied to build performance models of hardware and software systems. The popularity of QN models for performance evaluation is due to their scalability and to their ability to express many of the important factors affecting performance mentioned in Chapter 1. Moreover efficient and accurate techniques for QN analysis are available, in particular for a class of QN referred to as *product-form*, which has been widely used to carry out performance analysis.

The *definition* of a “QN model of a particular system is made relatively straightforward by the close correspondence between the attributes of queuing network models and the attributes of computer systems” [34]. For instance service centers in QN models naturally map to hardware devices in computer systems, while

customers map to system users. It is also possible with QN to describe multiple customer classes, each with its own workload intensity and service demands.

*Parameterization* associates service rates to service centers, and workload information and service requests to customer classes. At this regard, “a major strength of queuing network models is the relative ease with which parameters can be modified to obtain answers to ‘what-if’ questions” [34]. Solution of a parameterized QN model returns a set of performance indices such as response time, system throughput, resource utilization, etc. These indices can refer to a given resource only or extend to the whole system.

Several extensions of classical QN are available for performance modeling. Among them, *Extended Queuing Networks (EQN)* [30, 34] introduce features that allow to represent several interesting characteristics of real systems, such as finite capacity queues, simultaneous resource possession, synchronization, concurrency constraints, and memory constraints,. EQN models can be solved by approximate solution techniques.

*Layered Queuing Networks (LQN)* [45, 54, 17] represent another extension of QN that is particularly suited to model concurrent and/or distributed software systems. The main different between QN and LQN is that LQN can model both logical and physical resources of a system. Additionally they allow representing nested services, where a server may become client of other servers while waiting for its own clients requests to be served. A recent extension of LQN allows for a software entity to be further decomposed into activities which can be connected in sequence, loop, parallel, and alternative configurations forming a directed graph. LQN models can be solved both by analytic methods and simulation methods.

## **2.2.2 Stochastic Timed Petri Nets**

Stochastic Timed Petri Net (STPN) [1, 2, 3] are extensions of Petri Nets (PN), a modeling notation that is mainly used to verify functional properties of software

systems. In particular, “Petri Nets can be used to formally verify the correct synchronization between various activities of concurrent systems” [14]. A PN model consists of places, transitions, and direct arcs connecting places with transitions. Places may contain any number of tokens. A distribution of tokens over the places of a net is called a *marking*. Transitions act on input tokens by a process known as *firing*. Each transition is instantaneous, i.e. once a transition is enabled, it fires in zero time. A PN has a graphical representation shown in Figure 2. Places are represented by circles, transitions by bars, and marking by the set of tokens depicted inside places.

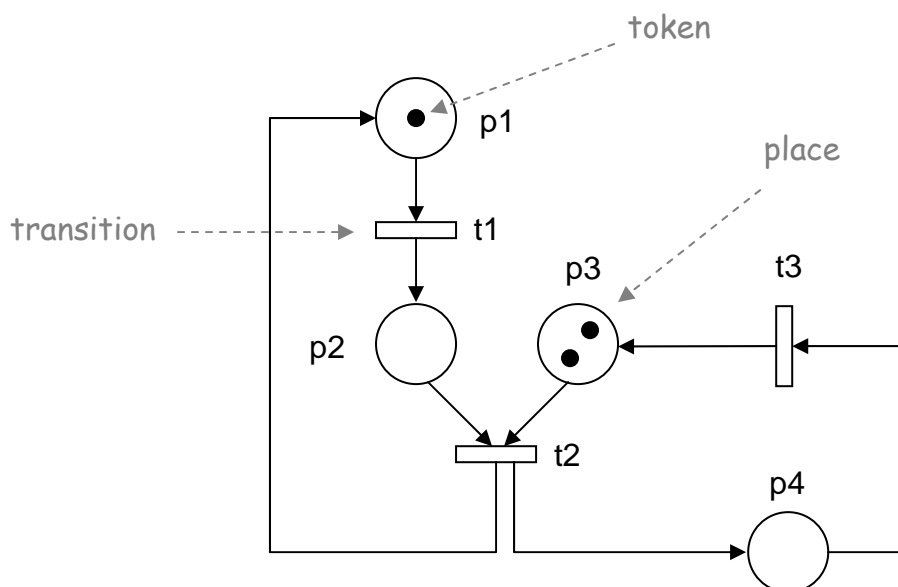


Figure 2: Example of PN model

STPN extend PN by associating a *firing time*, i.e., finite time duration, with transitions. The firing time is usually expressed by a random variable. Such variable may have an arbitrary distribution; however, in practice the use of non memoryless distributions can make the analysis unfeasible, unless other restrictions are imposed (e.g. only one transition is enabled at a time) to simplify the analysis. The quantitative evaluation of an STPN requires the identification and solution of the corresponding MC derived based on the net reachability graph. For this reason, the exact solution of

a STPN model may become infeasible due to the state space explosion problem. However, non-polynomial algorithm solution exists for a special class of STPN, known as *product-form*. Many approximated general solution techniques have also been defined.

Generalized Stochastic Petri Nets (GSPN) [2] represent another extension of classical PN, which allow both exponentially timed and immediate transitions. Immediate transitions fire immediately after they become enabled and have priority over timed transitions. They are associated with normalized weights, so that, in case multiple immediate transitions are concurrently enabled, the choice of the firing one is taken probabilistically. GSPN admit specific solution techniques reviewed in [3].

### **2.2.3 Stochastic Process Algebras**

Stochastic Process Algebras (SPA) [8, 23, 24, 37] are extensions of Process Algebras (PA), which allow to integrate qualitative (functional) and quantitative (temporal) aspects of software systems into a single modeling notation. A pure PA model describes a system in terms of its active components, and the interaction or communications between them. Components are called *agents* or *processes* and execute actions, which are assumed to be instantaneous.

SPA extend PA by incorporating temporal information into models. A duration is associated to actions using continuous random variables, often. Such addition makes it possible to evaluate system functional properties (e.g. liveness, deadlock), performance indices (e.g. throughput, waiting times), or combinations of them (e.g. probability of timeout, duration of action sequences).

A quantitative analysis of the modeled system can be performed by obtaining the stochastic process underlying the process algebra model, which is a MC when action durations are given as exponential random variables. Research has been made in order to avoid the problem of state space explosion associated to Markov modeling, which soon makes performance analysis unfeasible. Various methods to tackle the issue

have been proposed. A few authors suggested syntactic characterizations of PA terms whose underlying MC admits efficient product-form solution [22, 25].

## **2.2.4 Simulation Models**

Besides being a solution technique for performance models, simulation [6] is a modeling technique by itself, which allows reproducing the behavior of arbitrarily complex systems using different possible languages, libraries, and tools.

A simulation model is a conceptual representation of a system, which relies on a set of assumptions on the system operation, and on the workload driving it. The simulation model is translated into a simulation program. During the experimental phase the program is run in order to generate results. The number of runs and the length of simulation runs depend on the desired degree of accuracy for the results. If high confidence is required a high execution cost may be necessary. At the end of the simulation experiment performance results are evaluated, using appropriate analysis techniques.

## **2.3 Evaluation of Performance Models**

The goal of this section is to evaluate the suitability of the notations described in the previous section to define performance models of software systems in early phases of the software lifecycle. For each notation we consider:

1. the easiness to define models, to solve them, and to modify them based on possible feedback from the performance evaluation;
2. the adequacy to embed relevant factors affecting performance (e.g., system workload, system architecture, resource requirements, etc.).

### **2.3.1 Queuing Networks**

QN models are relatively easy to build, solve, and modify. QN are particularly well-suited for software modeling at the architectural level. In fact, the elements of a QN

model closely correspond to the elements of a software system (e.g., QN service centers map to system components, connections among QN service centers can be mapped to connections among system components). On the negative side, QN are not appropriate to represent the internal dynamics of software components. Therefore, in later stages of the development process, when more details about software behavior become available, QN may not be powerful enough to support performance evaluation. In this case LQN represent a good alternative, thanks to the availability of *activities* to specify software internal dynamics. Another limitation of classical QN is their ability to represent only asynchronous communications among service centers. This reduces the expressiveness of the notation in modeling modern distributed systems using different communication interactions (e.g., synchronous, asynchronous, deferred synchronous). To overcome this limit, extensions of QN can be used, such as LQN.

In general, QN and their extensions are able to embed many relevant factors affecting performance. In particular, in contrast to other notations we considered, they naturally model resource contention, which is a very important driver of system performance.

### **2.3.2 Stochastic Timed Petri Nets**

SPN are not particularly appropriate to model software at the architectural level, since a direct correspondence between software components and PN facilities (places, transitions, and tokens) cannot be established. In fact, SPN model the system from a functional point of view, thus making it difficult to represent system structure. As a modeling strategy, software components could be mapped to Petri subnets; the same could be done for hardware components. However, this approach is not straightforward and complicates model modifications in case any change has to be made to the software or hardware configuration of the system. In the unlikely case where SPN or their extensions are used for software specification, PN-based notations

can be used to represent software performance models in a straightforward way. Otherwise, performance models can be defined based on the close mapping between software behavioral models and SPN model structure; even though this increases the model complexity as the software description becomes more complex and detailed. SPN allow to model synchronous communications in a natural way. However, representing other types of interactions between software components (e.g., asynchronous) may require additional SPN structures or submodels. Models changes can require substantial effort. For instance in case a particular hardware component needs to be replicated, the whole subnet corresponding to that component needs to be identified, replicated, and then suitably reconnected to the rest of the model. Finally, analytical solution of PN-based performance model can be impractical for systems with a large number of concurrent states. In these cases models can be solved through simulation, at the expense of often high execution costs.

PN-based models are less suitable than QN to represent relevant factors affecting performance. In fact, aspects such as hardware component replicas, software component multithreading, or software and hardware contention are not directly representable.

### **2.3.3 Stochastic Process Algebras**

SPA represent a better candidate than SPN to model software performance in the early phases of the software lifecycle. In fact, SPA allow a natural mapping between processes and software components. SPA also allow to model software internal dynamics. As a drawback, hardware components and deployment of software components are not directly representable; particular modeling strategies have to be adopted to overcome this limit. Synchronous communication can be easily specified. However, other types of interactions between software components (e.g., asynchronous) may require additional actions, increasing model complexity and decreasing the correspondence between software model behavior and SPA model.



Model changes may require some effort because in SPA, as in SPN, since many types of performance information are not managed explicitly (e.g., hardware service rates, deployment of software, etc.). Finally, model solution can use analytical methods if the corresponding Markov model has a manageable number of states. Otherwise simulation has to be used.

The adequacy of SPA to embed relevant performance factors is medium. In fact, SPA can directly represent user requests, internal dynamics and (synchronous) interactions, replicas and threading of software components, software contention, etc. However, other factors such as allocation of software to hardware platforms, or hardware contention cannot be expressed.

### **2.3.4 Simulation Models**

Simulation represents the most general and powerful modeling technique. It can be used to represent early abstractions of a software system, provided that enough details about system behavior are known. Simulation models can be very expressive and embed all the relevant factors affecting performance. However, their usage also implies disadvantages. In fact, simulation models may require a high development cost, especially for complex systems. Model solution can also be time consuming. The output of simulation programs consists of streams of random variables and usually requires special skills to be analyzed, for instance using appropriate statistical techniques. It is also not possible to obtain performance results as a function of one or more model parameters (e.g., number of system users). Instead, a separate simulation model has to be performed for each different parameter value.

# Chapter 3: A Methodology for Early Software Performance Analysis

In this chapter we present our methodology to address the problem of early performance analysis of software systems. The methodology uses UML as the software modeling notation and LQN as the performance modeling notation. We devise a transformation to automatically derive a LQN model from a set of UML diagrams. The transformation is largely inspired by previous work presented in [20, 21, 39, 40, 41, 46]; however, our contribution is the adaptation of the existing techniques to a different set of UML diagrams, which are more suitable to be used in early stages of the software development lifecycle. We suggest extensions to the current UML Performance Profile to allow a more convenient specification of the performance characteristics of the system under examination. Extensions are also proposed to cover gaps in the current Profile, which does not cover UML 2.0 diagrams.

## 3.1 Software Specification Model

The Unified Modeling Language (UML) [10] provides the basis of our performance analysis methodology. The main reasons for its selection are the widespread diffusion and acceptance of UML as a *de facto* standard for software specification, and the need to integrate performance modeling and evaluation with standard practice development

environments [5]. An additional reason is that since the adoption by OMG of the “UML Profile for Schedulability, Performance and Time” [36], UML enables quantitative performance annotations that can be used to establish requirements for the generation of performance models.

This section provides a brief overview of the UML notation focusing on the features we use in our performance evaluation methodology.

### **3.1.1 UML**

UML [10] is a semi-formal language developed by the OMG to specify, visualize, and document software artifacts. The UML notation is quite rich, including a set of diagrams that can be used to model systems from different points of view and at different levels of detail. However, UML deliberately lacks a formal semantics. While on one hand this is an advantage, since it allows to use and combine UML models with few restrictions, on the other hand it is also a drawback because it makes any formal reasoning based on UML specifications very difficult.

Quantitative performance analysis of software systems based on annotated UML diagrams requires that system specification models are translated into performance models. To bridge the gap between software design and performance analysis this process should be automatic, possibly integrated within common software development tools and environments. Since the introduction of SPE a significant research effort has been devoted toward this direction and many techniques for manual or automatic derivation of performance models directly from UML software specifications have been proposed [4].

### **3.1.2 UML Diagrams**

UML includes two fundamental types of diagrams: structural diagrams and behavioral diagrams. UML 2.0 [78] provides better capabilities than its previous version to

model behavioral diagrams; for this reason hereinafter we will implicitly refer to the new release.

*Structural diagrams* model the logical or physical structure of system components and include Class Diagram, Component Diagram, Composite Structure Diagram, Deployment Diagram, Package Diagram, and Object Diagram.

*Behavioral diagrams* model system dynamics and include Use Case Diagram, State Machine Diagram, Activity Diagram, Sequence Diagram, Communication Diagram, Interaction Overview Diagram, and Timing Diagram.

We are not interested in considering all the diagrams above as possible software specification models. Rather, our focus is on a minimal subset of diagrams that allows capturing early performance-relevant information of software systems. In particular, we adopt Use Case Diagrams to identify performance-relevant system functions and workloads, Sequence Diagrams to model performance scenarios, and Deployment Diagrams to represent possible platform configurations for the system. For simplicity we assume that only one Use Case Diagram and one Deployment Diagram are associated with the system under study. However, this does not represent a serious limitation to the applicability of our methodology. In fact, in case multiple Use Case Diagrams or Deployment Diagrams were available, it would be sufficient to separately process each of them.

### *Use Case Diagram*

Use Case Diagrams capture high-level interactions between a system and users that invoke its functionalities. A *use case* is “a set of sequences of actions, including variants, that a system performs that yields an observable result of value to an actor” [10]. An *actor* identifies a significant system stakeholder i.e., a physical or logical entity requiring services.

A Use Case Diagram is graphically displayed as a rectangle, representing system boundary, filled with ellipses, representing use cases. Actors, shown as stick figures

or stereotyped icons, are connected to the use cases they generate or take part to. Both use cases and actors are associated with descriptive names. Figure 3 represents a simple example of Use Case Diagram for a simplified ATM system, which allows users to check their balance and to deposit or withdraw money.

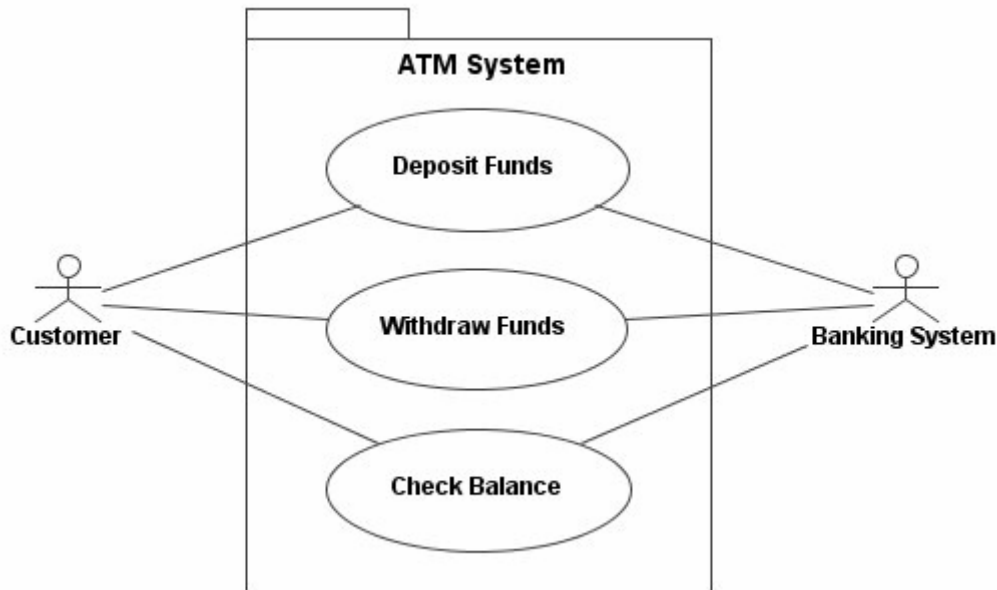


Figure 3: Example of Use Case Diagram

From a performance perspective Use Case Diagrams allow to identify performance-relevant functions of the system, i.e. interactions that “are critical to the operations of the system, influence user’s perception of responsiveness, or represent a risk that performance goal might not be met” [50]. They also help to identify significant user workloads.

### Sequence Diagram

Sequence Diagrams specify the dynamics of use cases in terms of interactions between system components. They represent the components involved in the interactions, and the set of partially ordered messages exchanged between them. A

message can express either an event or an invocation of an object's method. Both synchronous and asynchronous communication can be represented.

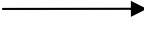
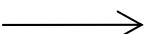

Since UML 2.0, Sequence Diagrams have better capabilities to model complex system dynamics than the previous UML version. In fact, so-called *fragments* have been introduced, which allow to clearly specify alternation, looping, concurrency, etc. “A *combined fragment* includes a portion of a Sequence Diagram surrounded by a frame, and contains one or more operand regions tiled vertically and separated by horizontal dashed lines. An operator shown in the upper-left corner of the frame prescribes how the operand regions of the combined fragment are handled. For instance, the operators `opt` and `alt` are used for branch selection, `par` for parallel execution, and `loop` for repetition. Another new feature allows for hierarchical decomposition of a scenario step into a more detailed subscenario. This is done by using an *interaction occurrence*, a fragment labeled with the operator `ref`, which refers to another interaction shown in a separate Sequence Diagram” [56].

An example of Sequence Diagram for the “*Check Balance*” function of the ATM system in Figure 3 is shown in Figure 4. We can observe that system components are laid out near the top of the diagram, from left to right. The lifeline of a component is rendered as a dashed line extending downward from the objects and representing the advancing of time. Along the lifeline are narrow rectangles representing the execution of component operations. Messages go from the sending component lifeline to the receiving component lifeline. They are displayed as arrows whose head shape indicates the type of the message. Table 1 shows the arrowheads available in UML 2.0.

Taking a performance perspective, we use Sequence Diagrams to model the dynamics of “the scenarios within each use case that have the greatest impact on performance” [50], i.e., the performance scenarios. Identification of performance scenarios using Use Case Diagram and specification of their dynamics using

Sequence Diagrams are essential steps toward our definition of a system performance model.

Table 1: Types of Sequence Diagram messages

	Synchronous message
	Asynchronous message
	Response to synchronous message

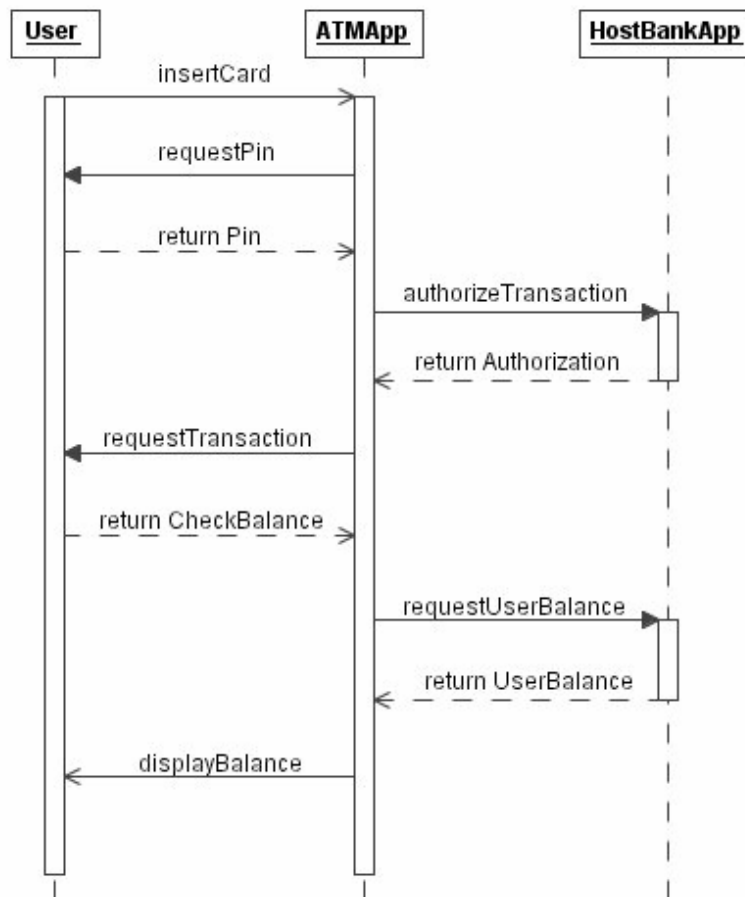


Figure 4: Example of Sequence Diagram

### Deployment Diagram

Deployment Diagrams model the platform configuration of the system and the allocation of its software components to the hardware devices in the configuration,

called *nodes*. Communication between different nodes is represented using communication paths.

Graphically, a Deployment Diagram consists of a graph of nodes connected by communication associations. Nodes may contain component instances; this indicates that the components execute on the node. Components may be connected to other components using dashed-arrow dependencies, implying that one component uses services of another component. Figure 5 shows a simple Deployment Diagram for the ATM system modeled by the Use Case Diagram in Figure 3 and the Sequence Diagram in Figure 4.

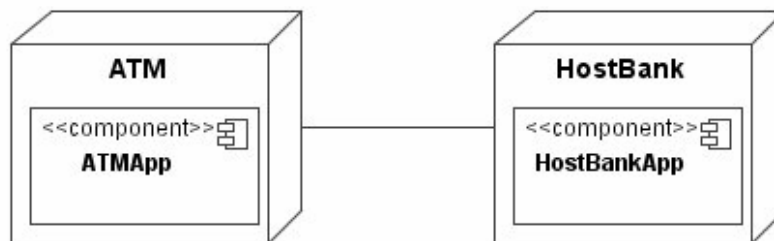


Figure 5: Example of Deployment Diagram

The use of Deployment Diagrams for performance modeling is motivated by the need to identify the hardware devices running a software system and the allocation of the software components of the system to those devices. This allows to estimate the resource demands of interactions represented within performance scenarios. Each interaction is potentially resource consuming, and only knowing the device executing the operation and its service rate we can associate a time requirement to the step, which is an essential datum to build a performance model of the system.

### 3.1.3 UML Performance Profile

The “UML Profile for Schedulability, Performance, and Time” [36] extends UML using standard mechanisms, i.e., stereotypes, tagged values, and constraints. Its goal is



to enable quantitative annotations that can be used to capture performance requirements for the system at the design level, and to associate performance-related characteristics with selected elements of a UML model [36].

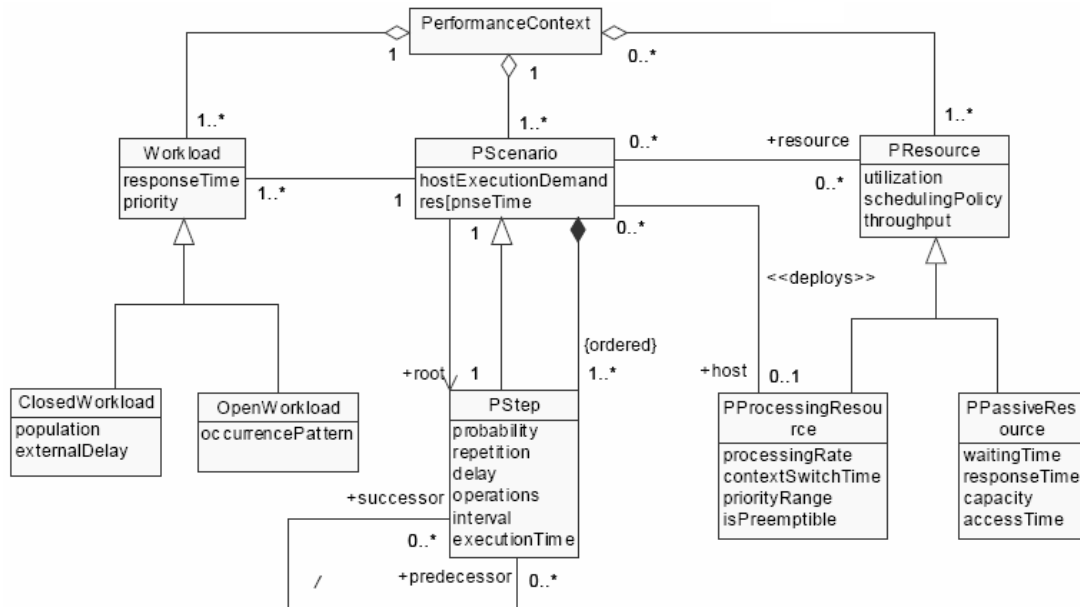


Figure 6: Performance analysis domain model

The Performance Profile defines a domain model, represented in Figure 6, which identifies basic abstractions that can be used to support the central concepts of performance analysis. Examples of these concepts are scenarios, workloads, and resources. *Scenarios* define system responses to user requests, and can have QoS requirements such as response time or throughput. Scenarios are executed by a job class or user class with certain load intensity, called workload. *Workloads* can be either open or closed. Open workloads are characterized by a certain arrival rate and distribution (e.g., Poisson); closed workloads have a fixed number of potential users cyclically requesting system functions, with a delay period – called Think Time – between the end of a system response and the issuing of the next user request. Each scenario is composed by scenario steps that can be joined in sequence, loops, branches, forks, and joins. A scenario step may be an elementary step, or a complex

sub-scenario, composed of many elementary steps. Each step has a mean number of executions, a host execution demand, demands to other resources (such as file I/O), and optionally its own QoS properties. *Resources* are another basic concept defined by the Profile. They can be active or passive, each resource type with its own attributes. Active resources have processing capabilities (e.g. CPU), while passive resources have not (e.g. I/O devices); they need to be acquired to execute an operation, and they usually have limited capacity.

The main stereotypes defined by the Profile include «*PAClosedLoad*», «*PAOpenLoad*», «*PAhost*», «*PAresource*», and «*PAstep*».:

- «*PAClosedLoad*» models a closed workload. Its main tags are: *PApopulation* and *PAextDelay*. The former defines the number of system users; the latter specifies the Think Time between successive user requests.
- «*PAOpenLoad*» models an open workload. Its main tag is *PAoccurrence*, which defines the arrival pattern of workload users. This usually corresponds to a random variable of given distribution.
- «*PAhost*» models a processing resource. Its tags include *PARate*, *PAschdPolicy*, and *PActxSwT*. The first one indicates the processing rate of the resource. The second one is the scheduling policy for the resource (e.g., FIFO, LIFO). The last one is the time needed to perform a context switch.
- «*PAresource*» models a passive resource. Its tags include *PACapacity* and *PAaxTime*. *PACapacity* defines the initial and maximum number of available instances of the resource. *PAaxTime* specifies the access time of the resource. Releasing a resource is assumed to require no time.
- «*PAstep*» models a step in a performance scenario. Its tags include: *PAdemand*, *PAextOp*, *PAprob*, and *PArep*. *PAdemand* indicates the total execution demand of the step on its host resource. *PAextOp* specifies operations on resources that are needed to execute the step, but which are not explicitly represented in the UML

model.  $PA_{prob}$  is the probability that the step will be executed. Finally,  $PA_{rep}$  is the number of times the step will be repeated.

## 3.2 Performance Model

Nowadays Queuing Networks (QN) are the preferred choice for performance modeling because of their abstraction level - which makes them suitable to express high-level software architecture models -, and because of the availability of efficient solution algorithms and tools to evaluate the models [5]. However, classical QN are constrained in the representation of behavioral details emerging from more detailed software design models [13]. This limitation is overcome by Layered Queuing Networks (LQN), which provide proper abstractions to express potentially complex operations performed by software components. Moreover, unlike classical QN, LQN can explicitly represent software components and their common characteristics (e.g., resource requirements, multithreading, allocation to hardware devices, etc.). Accordingly, it is also possible to obtain performance figures explicitly related to them, such as utilization, response time, and throughput. This allows to identify *software bottlenecks*, i.e., the overloading of one or more software components, while the underlying CPUs are lowly used. Another feature of LQN that is missing in QN models is the possibility to represent nested services, i.e., situations where servers issue requests to other servers, present in many distributed systems (e.g. three-tier software systems).

Because of all the advantages and properties mentioned above, LQN is the performance modeling notation we adopt within our performance evaluation methodology. The next subsections briefly review the notation and describe the software tools supporting the specification and solution of LQN models.

### 3.2.1 LQN

The LQN notation was developed as a combination of Stochastic Rendezvous Networks and the Method of Layers presented in [17, 45, 53, 54]. LQNs describe a system as a set of software and hardware resources. Software resources are processes, threads, semaphores, and other logical entities. Hardware resources are devices such as CPUs, disks, computing devices, etc. Resources can be modeled within LQNs using tasks and host processors.

A *task* models a logical resource that requires mutual exclusion. An *entry* models an operation that processes a distinct class of messages received by the task. For example, if a task models an object, entries can represent its methods. An entry is specified by its resource demands, which include the total average amount of host processing, and the average number of calls required for service operation to complete. A task is associated with a *host processor*, which represents the physical entity that carries out the operations. Tasks and processors include a queue, a discipline, and a multiplicity.

Interactions between software tasks are expressed as *service requests*, named as calls in LQN models. Tasks may send and receive service requests and play the client/server role. If tasks do not receive any request they are pure clients, called *reference tasks*, and they represent load generators or users of the system. Service requests between tasks can be made using three types of interactions: synchronous, asynchronous, and forwarding. LQN *synchronous* and *asynchronous* interactions are interpreted in the usual way. *Forwarding* interactions require that the sending task makes a synchronous call and blocks waiting for a response. However, the receiving task does not reply; in fact, after partially processing the call, it forwards the request to a third task, which either replies to the blocked client task or forwards the request further.

A recent extension to LQNs [18] introduces a new model primitive called *activity*. Activities allow detailing the sequence of operations executed when a task accepts a request at an entry. Activities can be connected in sequence, loop, parallel (AndFork/AndJoin) and alternative (OrFork/OrJoin) configurations. Just like entries, they have execution time demands and can issue service requests to other tasks.

A LQN model is graphically represented by an acyclic graph, whose nodes correspond to tasks and host processors. Tasks are depicted as parallelograms, and processors as circles. Arcs between tasks and processors indicate the allocation of software components to hardware devices. Arcs toward task entries denote service requests. They are labeled by the mean number of issued requests; in the absence of a label, a default value of one is assumed. The shape of the arc arrowhead expresses the type of the message (i.e., synchronous, asynchronous, forwarding).

Figure 7 shows an example of LQN model. *Users* is a non-reference task, i.e., a workload generator.  $n$  users are assumed to issue requests to the system with a Think Time of  $10s$ .  $p1$  and  $p2$  are host processors.  $A$  and  $B$  are tasks with entries  $s1$  and  $s2$ , whose associated service demands are  $0.5s$  and  $0.001s$ , respectively. Entry  $s1$  is detailed by activities  $A1$  and  $A2$ , which do not have associated service demands. The number of calls to entries is not indicated; this implies a default value of  $1$ .

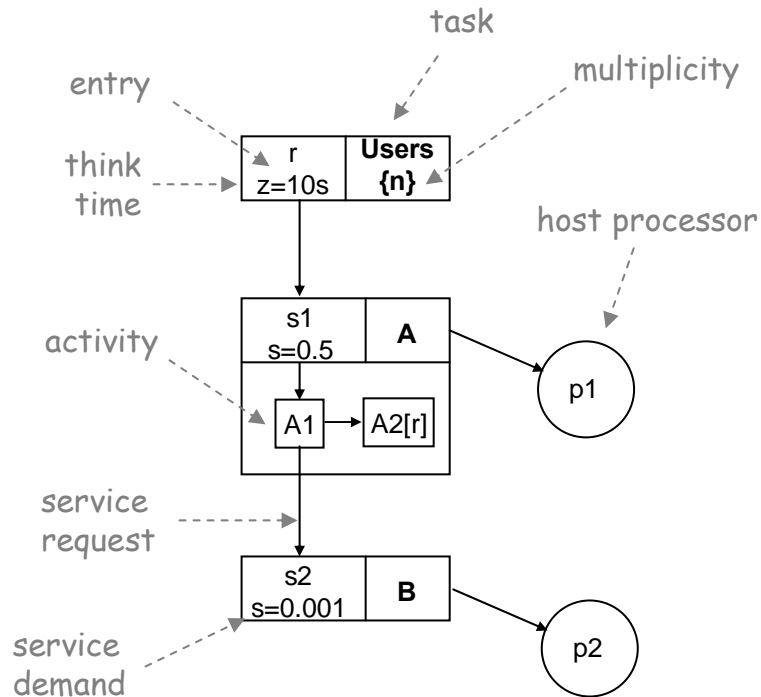


Figure 7: Example of LQN model

### 3.2.2 LQN Tools

LQN models can be created using the LQN modeling language [38], the XML grammar described in [19], or the visual software `jlqnDef` [55]. Both analytical and simulation tools are available to solve LQN models [19]. They all have been developed within the Department of Systems and Computer Engineering at Carleton University in Ottawa, Canada, and are freely available upon registration.

`lqns` is an analytical solver using mean-value queuing approximations. `lqsim` is a simulation solver using discrete-event simulation. `multisrvn` is an experiment controller that executes parameterized experiments over given ranges. All these software tools are textual; they can be only be executed at the command line.

## 3.3 UML to LQN Transformation

Several approaches to derive LQN performance models from UML software specifications have been presented in the research literature [20, 21, 39, 40, 41, 46]. In the next subsection we provide a brief overview of the assumptions, input information, and transformation methodologies they use; we also point out at their benefits and limits. Afterwards, we describe our approach for UML to LQN transformation.

### 3.3.1 Previous Work

In [39, 40] a graph grammar-based transformation from UML to LQN is described. The transformation assumes the availability of UML Collaboration Diagrams, Deployment Diagrams, and Activity Diagrams. The UML diagrams have to be annotated using standard extensions defined by the UML Performance Profile.

The structure of the LQN model is generated using Collaboration Diagrams and Deployment Diagrams. The former represent the high-level software architecture of the system and the interaction patterns between software components (such as client/server, master-slave, pipeline and filters, etc.). The latter specify the allocation of software components to hardware devices. The dynamics of the performance model is generated from detailed descriptions of key performance scenarios based on Activity Diagrams. Parameters for the LQN model are given by the performance annotations on the Activity Diagrams.

The actual transformation from UML to LQN has been implemented in different ways. In [40] an existing graph-rewriting tool called PROGRES [46] is adopted and a set of production rules to convert UML diagrams into LQN models is defined. The disadvantage with the approach is that it introduces an additional step in the software development process, i.e., the conversion of each UML Activity Diagram into a PROGRES graph to be used as the input for the transformation.

Another technique [41] implements an ad-hoc graph transformation in Java. The input graph is an XML representation of a UML model that is transformed into a set of Java objects. This approach is preferable to the previous one because it eliminates the step of creating a PROGRES input graph from the UML model. Instead, it is only necessary to convert the UML model into its XML format, which is easily obtainable using any UML software tool. The transformation is more efficient because it is tailored to the problem at hand. Another advantage is that it is possible to integrate the performance model builder with a UML tool.

The third methodology is presented in [21]. An XML representation of a UML model is again the input to the transformation, which is based on XML tree-manipulation techniques using XMLgebra. The advantage of the proposed transformation is its flexibility, since it can easily be applied to create performance models based on notations different than LQN.

The fourth and last solution, proposed in [20], is conceptually similar to the second one, in that the starting point of the transformation is again a XML representation of a UML model. However, the LQN model is generated from the XML file using XSLT.

From the point of view of a potential user the last three techniques are not different from each other. However, from the perspective of a solution developer the XSLT program is shorter and easier to create than implementing the Java program or defining the XML tree-manipulation rules.

### **3.3.2 Our Approach**

In this section we propose a UML to LQN transformation to derive a performance model of a software system modeled with UML diagrams. Our transformation is conceptually and methodologically similar to the ones reviewed in the previous section. However, we do not use Collaboration Diagrams to model architectural patterns of communication between software components, since we only focus on distributed systems using client/server interactions. We also adopt UML 2.0 Sequence



Diagrams instead of Activity Diagrams to model performance scenarios. The reason for our choice is that “performance is largely a function of the frequency and nature of intercomponent communication [...]” [12], and Sequence Diagrams are the most appropriate UML model to express cooperation between system components. Unlike Activity Diagrams, Sequence Diagrams are very good at showing which components are responsible for different actions, and the partial order of execution of scenario steps. Additionally, since UML 2.0, Sequence Diagrams can represent complex software dynamics, including non-sequential flows of control, very well. In fact, the introduction of the “combined fragment” feature, described in Section 3.1.2, allows to represent branches, loops, parallel execution, etc.

We annotate UML diagrams with performance-related information partly using the UML Performance Profile, partly using newly introduced stereotypes and tagged values, which we will explain later. The motivation for these extensions is the convenience of associating expected system workloads with different classes of system users, instead of with each performance scenario, as prescribed by the current Performance Profile. Given the user workloads and the set of probabilities of executing use cases and scenarios, it is then possible to “automatically” estimate the workload associated with each of them. This procedure involves adding performance annotation to Use Case Diagrams, and slightly modifying the annotations currently associated with Sequence Diagrams. Consistently with naming conventions used by the standard UML Performance Profile, we prefix the newly introduced performance-related UML extensions with the “PA” string.

A high-level description of our algorithm for UML to LQN transformation is shown in Figure 8. Next, we present details about its assumptions and methodological steps.

*INPUT: Use Case Diagram, Sequence Diagram, Deployment Diagram.*

*Performance annotations*

*TRANSFORMATION:*

*1. Generate the LQN model structure*

- a. Determine LQN devices from DD*
- b. Determine LQN tasks from UCD, DD, and SD*
- c. Determine the allocation of tasks to devices from DD*

*2. Generate details for LQN entries and activities*

- For each performance scenario process the corresponding SD*
  - a. Determine entries of reference tasks*
  - b. Determine entries for offered services*
  - c. Determine entries for external services*
  - d. Determine activities*
  - e. Determine request flow among entries and activities*

*3. Generate LQN parameters from UML performance annotations*

*OUTPUT: LQN model*

Figure 8: High-level algorithm for UML to LQN transformation

## **Input**

The definition of a complete LQN model of a software system requires the following information:

- high-level software architecture to determine the performance model structure, i.e., the configuration of the system software and hardware resources;
- detailed performance scenarios to determine the flow of service requests among software and hardware resources in the performance model;

- performance annotations to determine the workload and resource requirements associated with the performance model.

We adopt Deployment Diagrams to meet the first requirement. Sequence Diagrams are used to model performance scenarios. Finally, annotations on Use Case Diagram and Sequence Diagrams are used to parameterize the LQN model. In the next subsections we describe our assumptions about each type of UML diagram; we also explain which performance annotations defined in the UML Performance Profile we use, and which we introduce to address potential Profile incompleteness.

### **Use Case Diagram**

We adopt Use Case Diagrams to help performance analysts to identify performance-significant system actors and use cases, corresponding to the user groups and functions that are critical to the perceived performance of the system. We annotate performance-significant users in Use Case Diagrams with expected user workloads using the «*PAClosedLoad*» and «*PAopenLoad*» stereotypes introduced in Section 3.1.3. Associations between performance-significant actors and performance-significant use cases are annotated with the <<PAuse>> stereotype, whose tagged value,  $P_{Aprob}$  expresses the probability that a user invokes the linked use case. This allows to automatically compute user workloads on different performance scenarios based on the probabilities associated with the scenarios and with the related use cases.

Figure 9 represents a simple Use Case Diagram annotated for performance assessment purposes. The diagram indicates that the system has 10 potential or active users of type *User1*, using the system with an assumed Think Time of 30 seconds between successive requests. The system has an unlimited number of users of type *User2* (open workload), invoking system functions according to a Poisson distribution with average 0.5s.

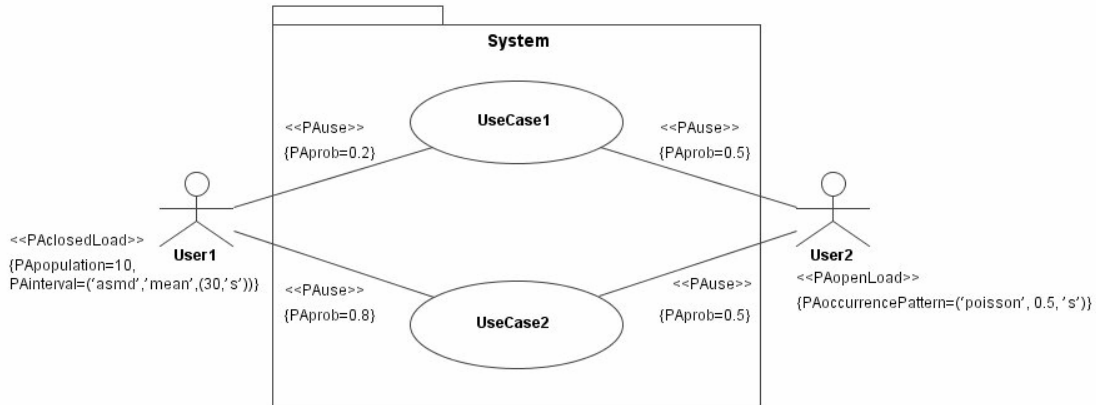


Figure 9: Annotated Use Case Diagram

Let  $m$  be the number of different performance-relevant users, and  $n$  the number of performance-relevant use cases within a Use Case Diagram. Let  $p_i(j)$  ( $i=1, \dots, m, j=1, \dots, n$ ) be the probability that the  $i$ th user makes use of the software system by executing the Use Case  $j$  ( $\sum_{i=1}^m p_i(j) \leq 1$ ). Then the workload generated by the  $i$ th user on UC  $j$  can be determined based on the user workload type. In fact, if the user generates a closed workload with population  $x$  and Think time  $t$ , the workload on UC  $j$  includes a population  $x$  with Think Time  $t_i(j) = t / p_i(j)$ . On the other hand, if the user generates an open workload with arrival distribution function  $f$ , the workload generated by user  $i$  on UC  $j$  is characterized by an arrival distribution function  $f_i(j) = f \cdot p_i(j)$ .

Referring to Figure 9 the workload generated by *User1* on *UseCase1* includes a population of 10 users, with think time  $150s = 30 / 0.2$ . On the other hand, the workload generated by *User2* on *UseCase1* is characterized by a Poisson arrival distribution with mean  $0.25s = 0.5s \cdot 0.5$ .

### Sequence Diagram

We use Sequence Diagrams to model the dynamics of performance scenarios, identified by the stereotype `<<PAcontext>>` of the UML Performance Profile. As explained in the previous section, we adopt Use Case Diagrams to identify performance-significant use cases. However, for each significant use case, not all

scenarios are performance scenarios, i.e., are relevant from a performance standpoint. For this reason, we associate with the `<<PAcontext>>` stereotype the tagged value `PAprob`, expressing the probability of executing the scenario in exam, with respect to other ones referring to the same use case.

We also introduce performance annotations for combined fragment regions and their operands. This is not possible using the standard UML Performance Profile, since it was defined for UML 1.4 and has not been upgraded for UML 2.0. In particular, we annotate the single operand of the *opt* fragment with the tagged value `PAprob`, expressing the probability that the set of scenario steps represented in the fragment is executed. Similarly, we annotate with `PAprob` each operand of the *alt* fragment, with the constraint that the sum of the given probabilities is equal to 1. Finally, we annotate with `PArep` the operand of the *rep* fragment, to specify the number of times the set of steps represented in the fragment is repeated.

Figure 10 shows a possible Sequence Diagram with performance annotations. We assume that the scenario refers to the use case *UseCase1* depicted in Figure 9. The diagram is labeled by `<<PAcontext>>`, hence it represents a performance scenario. The probability of execution of the scenario is expressed by the variable  $p$  associated with the tag `PAprob`. In the diagram the stereotype `<<PAresource>>` identifies system components. These are usually software components; however, other resource types are possible, such as passive resources or even human resources required to carry out system operations. The tag `PAcapacity` is optionally attached to the `<<PAresource>>` stereotype to indicate the number of replicas or the level of multi-threading of the corresponding resource. If the tag is omitted a default value of 1 is assumed.

Scenario steps are labeled by the stereotype `<<PAstep>>`, and annotated with the corresponding resource demand using the tag `PAdemand` and `PAextOpt`. `PAdemand` expresses the processing time required to execute the step. As with any performance

value the demand can be a required, assumed, estimated or measured value. In Figure 10 all values are assumed. They represent mean values and are expressed in milliseconds. Scenario steps are optionally associated with the `PAextOp` tag, which defines the time requirement of external operations, i.e., operations on resources that are needed to execute the step, but which are not explicitly represented in the UML model.

The workloads associated with performance scenarios can be computed using the user workloads calculated in the previous section and the execution probabilities of scenarios. In particular, if  $p_i(j, k)$  is the probability of user  $i$  executing scenario  $k$  of Use Case  $j$  ( $i=1, \dots, m, j=1, \dots, n, k=1, \dots, h$ ), the workload generated by user  $i$  on that scenario can be determined based on the user workload type. In particular, if the user generates a closed workload on UC  $j$  with population  $x$  and Think Time  $t_i(j)$ , the workload on scenario  $k$  has population  $x$  and Think Time  $t_i(j, k) = t_i(j) / p_i(j, k)$ . On the other hand, if the workload is open and the arrival rate is  $f_i(j)$ , the arrival rate for scenario  $k$  can be computed as  $f_i(j, k) = f_i(j) \cdot p_i(j, k)$ .

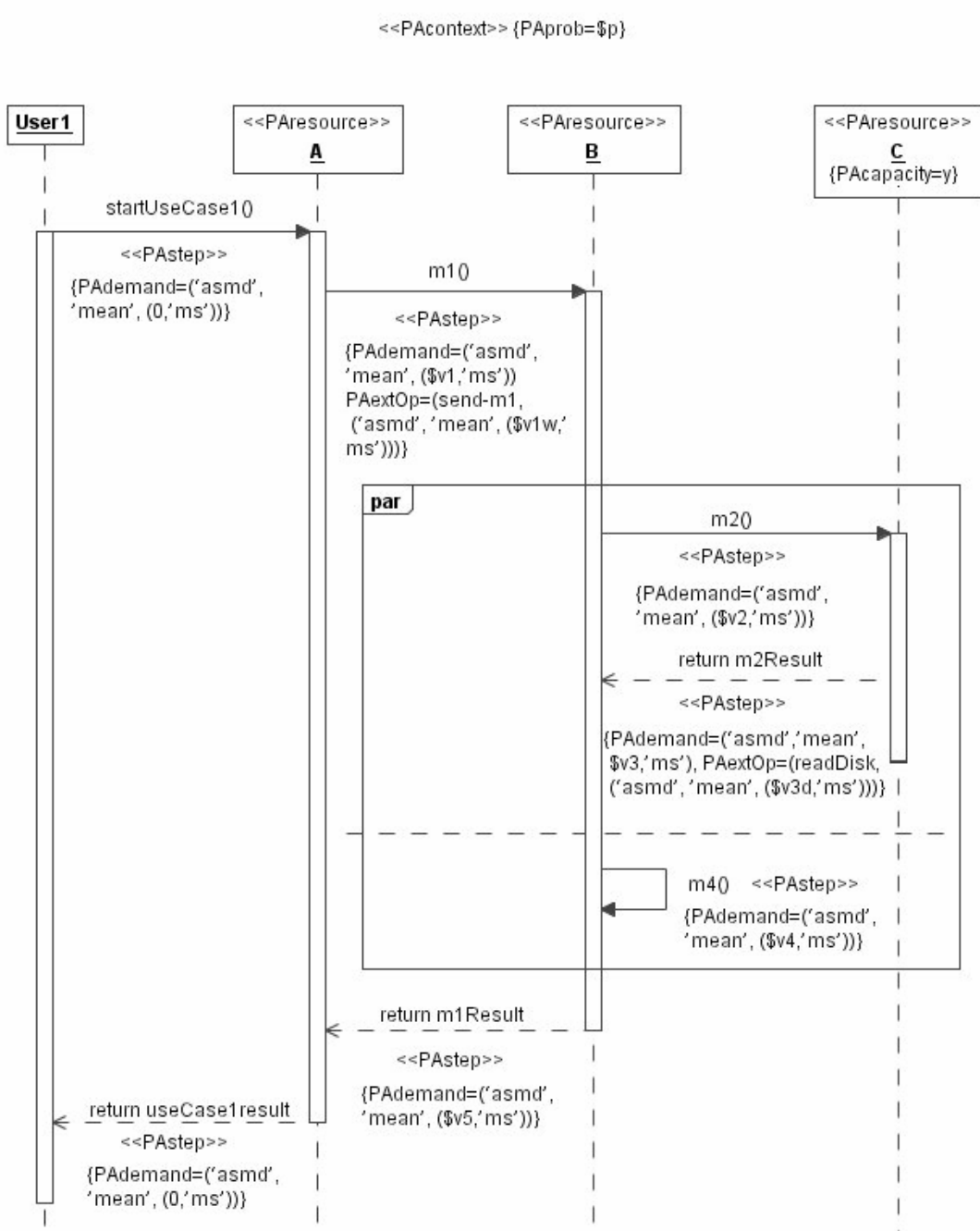


Figure 10: Annotated Sequence Diagram

### Deployment Diagram

We adopt Deployment Diagrams to represent the platform configuration where the application in exam is targeted to run. Deployment Diagrams allow to identify software and hardware resources within the system and the allocation of software components to hardware nodes. We use standard features of Deployment Diagrams. We also use standard extensions defined by the UML Performance Profile, with the

exception of the association of the tag `PAcapacity` not only with the `<<PAresource>>` stereotype, but also with `<<PAhost>>`, to represent the number of CPUs of a processing device. If the tag is absent, a default value of one instance is assumed. A special situation is represented by the specification of a  $\infty$  symbol for `PAcapacity`, which means that the associated device imposes no resource constraint, and no queues are formed to use its services (e.g., WAN).

Figure 11 shows an example of annotated Deployment Diagram. In the diagram the nodes labeled by the `<<PAhost>>` stereotype, i.e., *ClientCPU*, *ServerCPU*, *DBCPU*, represent processing devices. Nodes labeled by `<<PAresource>>`, i.e., *WAN* and *Disk*, correspond to non-processing devices; they cannot initiate events but only respond to them. If `<<PAresource>>` is associated with a software component instead of a hardware node, it indicates a software unit running under its own thread of control, e.g., *A*, *B* and *C*. The tag `PAcapacity` can optionally be associated with resources labeled by the `<<PAhost>>` and the `<<PAresource>>` stereotypes to indicate the number of CPU, or the number of replicas or threads of the corresponding software or hardware resource. In Figure 11 *ServerCPU* has *x* CPU, while software component *C* has *y* threads of control.



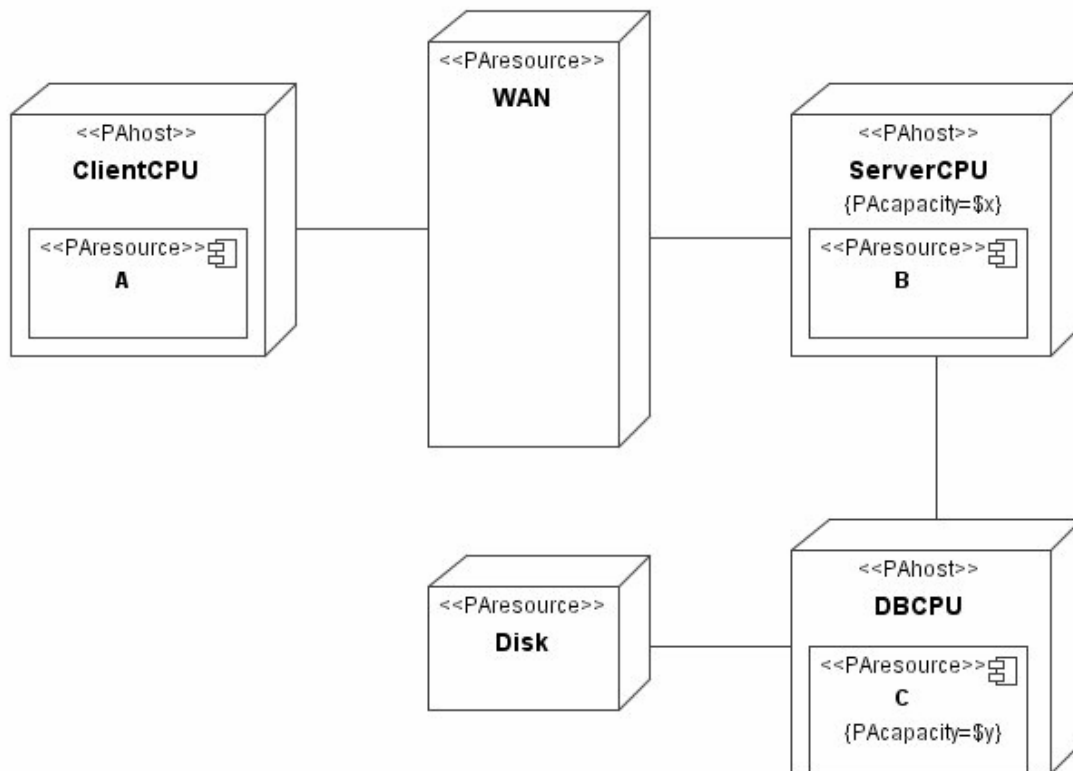


Figure 11: Annotated Deployment Diagram

### STEP 1:

The first step of the algorithm for UML to LQN transformation generates the LQN model structure (i.e., LQN tasks, devices, and connecting arcs between them). The step is rather straightforward. In fact, there is a close correspondence between elements of the Deployment Diagram used as input by the transformation algorithm, and LQN model entities. The correspondence is made even more explicit by the performance annotations attached to the Deployment Diagram, which allow to quickly identify tasks, devices, and their mappings.

#### STEP 1.a:

This step generates LQN devices of the performance model. It explores the annotations on the Deployment Diagram for the system, and creates LQN devices for each UML node. The optional tag `Pcapacity` is used to associate a number of

replicas to the identified devices. Figure 12 shows a graphical representation of the transformation step.

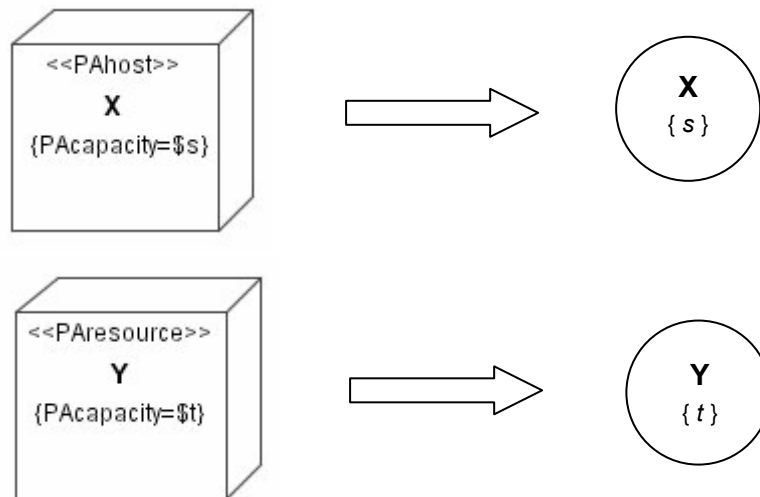


Figure 12: Mapping from Deployment Diagram elements to LQN devices

#### STEP 1.b:

This step, represented graphically in Figure 13, generates LQN tasks for the performance model.

Reference tasks are defined to represent significant user workloads in the Use Case Diagram for the system. If a closed user workload is assumed, the reference task is given multiplicity equal to the user population size; the Think Time of its entries will be specified in Step 3. If an open workload is considered, the multiplicity of the reference task is set to one; the arrival rate of its entries will be specified in Step 3. Referring to the Use Case Diagram in Figure 9, the reference tasks we identify for the system are *User1* and *User2*.

Non-reference LQN tasks are created by examining the Sequence Diagrams for the system and defining a new task for each component labeled by the <<PAresource>> stereotype. Tasks are also created for hardware nodes labeled by the same stereotype in the Deployment Diagram. In this case, the task takes the role of a software

controller implementing the access mechanism to the resource. The value of the optional tag `PAcapacity` is used to associate a level of multi-threading different from one to the task. A special situation is represented by the specification of a  $\infty$  symbol for the tag, which indicates that the corresponding resource is a delay server. Delay servers serve incoming user requests immediately; no wait time is required to access the resource.

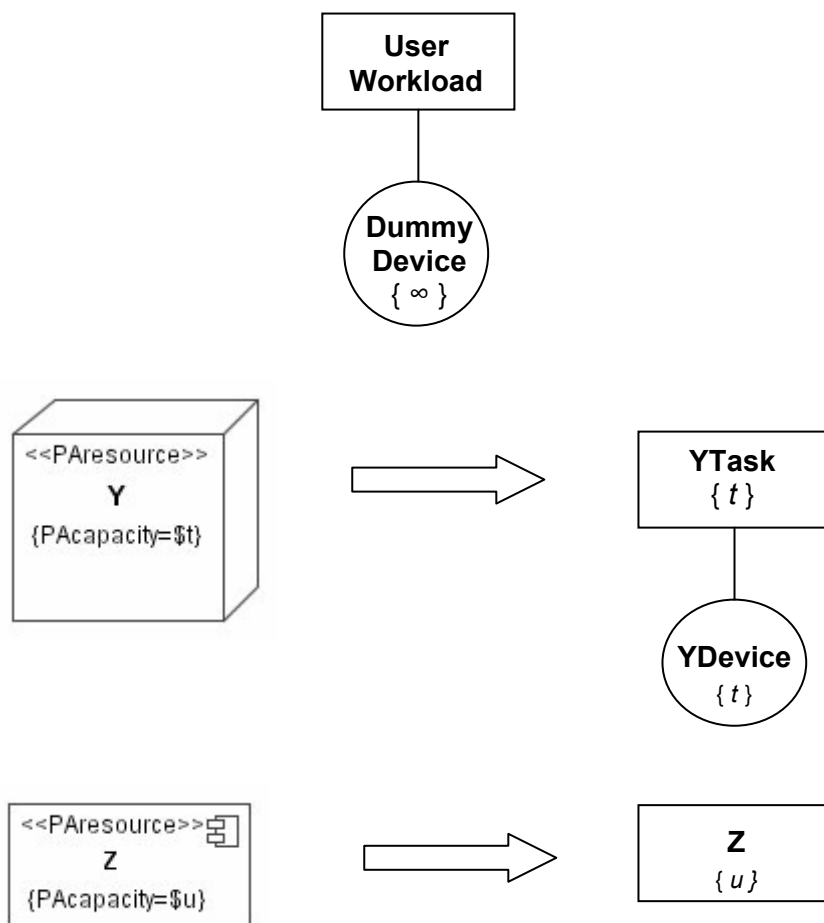


Figure 13: Mapping from Deployment Diagram elements to LQN tasks

**STEP 1.c:**

This step generates connecting arcs between LQN tasks and devices, based on the deployment relationships between software and hardware components represented in the annotated Deployment Diagram. Not all LQN tasks defined in Step 1.b correspond

to deployable resources; for instance, reference tasks are not associated with any system device, the same happens with <<PAresource>> components represented in Sequence Diagrams but not in the Deployment Diagram. However, in the LQN notation each task needs to be associated to a host processor. To meet this requirement we map the mentioned tasks to dummy LQN devices with infinite capacity. Figure 14 shows the result of the execution of this step on the Deployment Diagram in Figure 11.

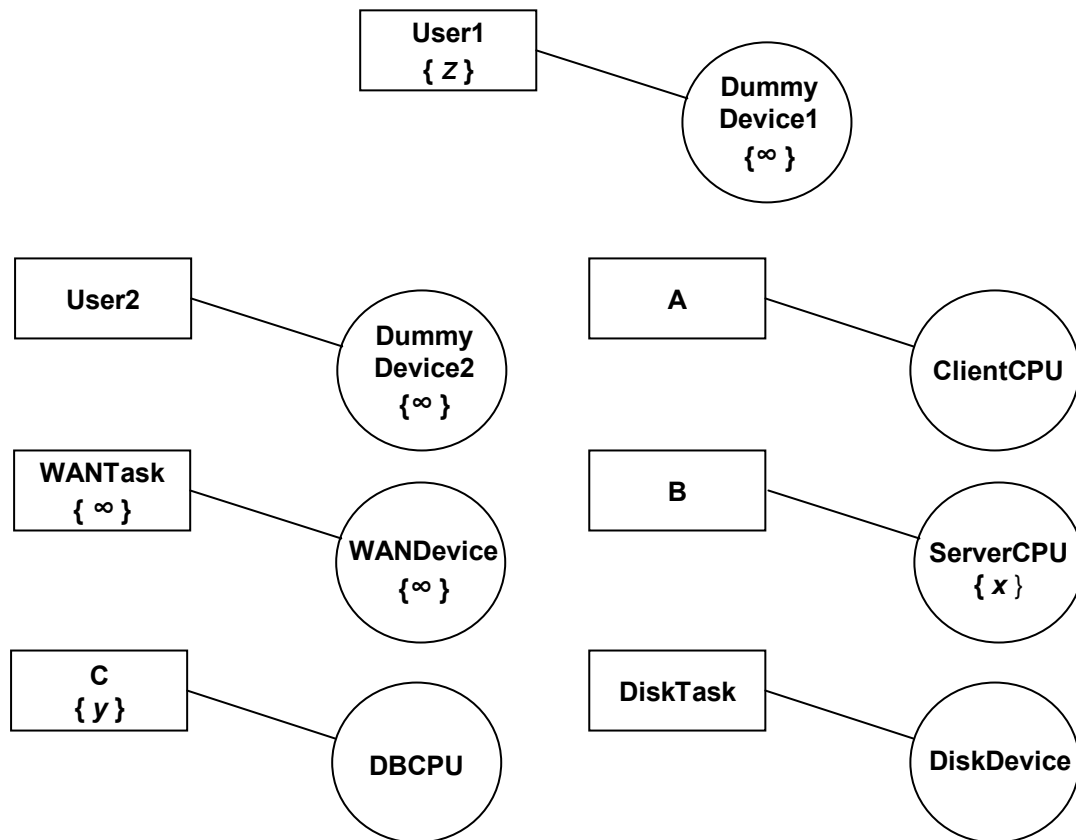


Figure 14: Mapping between LQN tasks and corresponding devices

## STEP 2:

The second step of the algorithm for UML to LQN transformation creates LQN entries, activities, and the request flow among them using the set of Sequence Diagrams (labeled by <<PAcontext>>) given in input to the transformation. The

step processes each Sequence Diagram, following its message flow and generating LQN model entities accordingly, as described in the following paragraphs.

**STEP 2.a:**

For each reference task defined in Step 1, a LQN entry is created for every performance-significant scenario the corresponding user initiates. Each entry corresponds to a workload generator for the scenario. Referring to the Use Case Diagram in Figure 9, and assuming that the performance scenario in Figure 10 is the only one for *UseCase1*, this step creates for the reference task *User1* the entry *UseCase1*.

**STEP 2.b:**

For every scenario a LQN task entry is generated for each type of service offered by a software component. We identify such services by looking at the operations invoked by the clients of the component, or, equivalently, by looking at the messages received by the software component in the considered scenario. The application of this procedure to the Sequence Diagram in Figure 10 leads to the identification of three LQN Entries: *startUseCase1* belonging to task *A*, *m1* belonging to task *B*, and *m2* belonging to task *C*.

**STEP 2.c:**

LQN task entries are generated within the task corresponding to a software controller for a passive resource, for each interaction represented in a Sequence Diagram that involves usage of that resource. In particular, LQN task entries are generated for scenario steps – labeled by the <<PAextOp>> stereotype – which require the use of a hardware device other than the host processor. Each entry models the demand on the external resource for a similar interaction. This means, for instance, that message *m1()* in Figure 10 requires the creation of a new entry. In fact, the message is exchanged between components connected by a WAN, labeled in Figure 11 as a passive resource

by the <PResource> stereotype. The entry models the request of the WAN resource for that interaction.

#### **STEP 2.d:**

LQN task activities are generated to represent internal computations of a software component, identified by self-addressed messages of the component corresponding to the task. Non-sequential flow of control, represented in UML 2.0 by the *combined fragment* feature of Sequence Diagram, also generates activities. In Section 3.1.2 we briefly reviewed the main types of combined fragments available to model complex software dynamics, i.e., *opt*, *alt*, *par*, and *loop*. Here we restrict ourselves to that subset. The next paragraphs describe how to process each fragment toward the generation of a LQN model.

The *opt* fragment corresponds to the optional execution of the set of scenario steps contained within the corresponding frame. Its translation within the LQN model generates an LQN “OrFork” within the task generating the first optional message. The “OrFork” connects two activities. One of them is used to model the set of optional steps; the other just connects to the activity merging the conditional branching. Figure 15 shows a very simple example of *opt* fragment, where software component *A* invokes service *m()* on component *B* depending on a guard with probability *p*. Figure 16 shows the LQN translation of the fragment, generated according to the above description. The translation is not connected to the rest of the model, since we do not know its full context.

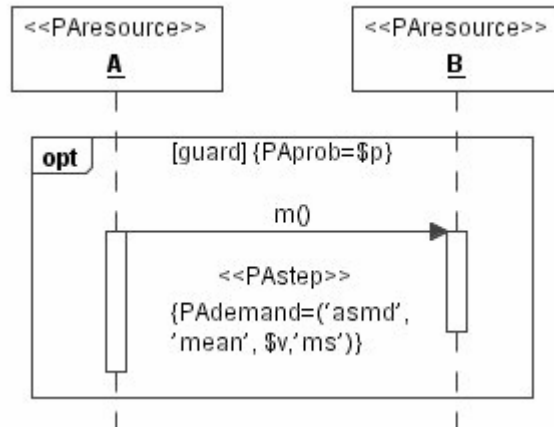


Figure 15: Example of *opt* fragment

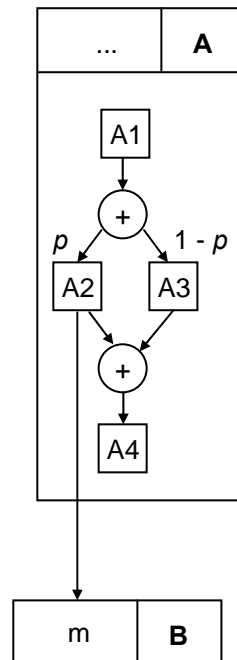


Figure 16: Translation of *opt* fragment in LQN notation

The *alt* fragment is very similar to the *opt* fragment. In fact, it is used to represent conditional branching. However, differently from the *opt* fragment, the *alt* fragment can represent multiple branches, each associated with a guard and a probability. The translation of an *alt* fragment in the LQN notation generates an “OrFork” within the task generating the first optional message. The “OrFork” connects a number of

activities equal to the number of conditional states represented in the fragment. Each activity is used to model the set of steps within a state. The probability of an activity corresponds to the probability of the corresponding state. Figure 17 shows an example of *alt* fragment, where software component *A* invokes service *m1()* on software component *B* depending on a guard with probability  $p1$ ; *A* invokes service *m2()* on *B* depending on another guard with probability  $p2$ ; if the previous guard conditions are not satisfied *A* executes operation *m3*. Figure 18 shows the LQN translation of the *alt* fragment, generated according to the description above. The translation is not connected to the rest of the model, since we do not know its full context.

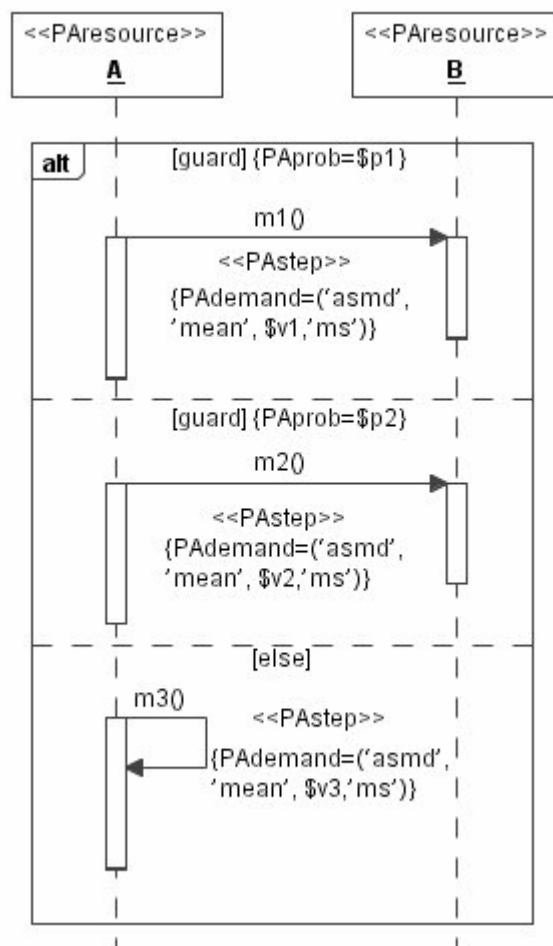


Figure 17: Example of *alt* fragment



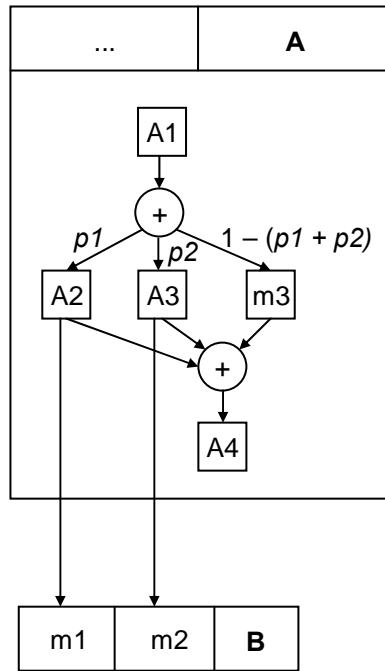


Figure 18: Translation of *alt* fragment in LQN notation

The *par* fragment is used to model the parallel execution of multiple sets of scenario steps contained within the corresponding fragment, each separated by a dashed line. Its translation within the LQN model generates an LQN “AndFork” within the task generating the first parallel message. The “AndFork” connects a number of activities equal to the number of parallel threads represented in the fragment. Each activity is used to model the concurrent thread of execution represented by the set of steps within a thread. Figure 19 shows an example of *par* fragment, where, in parallel, component *A* invokes service *m1()* on software component *B*, and executes operation *m2*. Figure 20 shows the LQN translation of the *par* fragment, generated according to the description above. The translation is not connected to the rest of the model, since we do not know its full context.

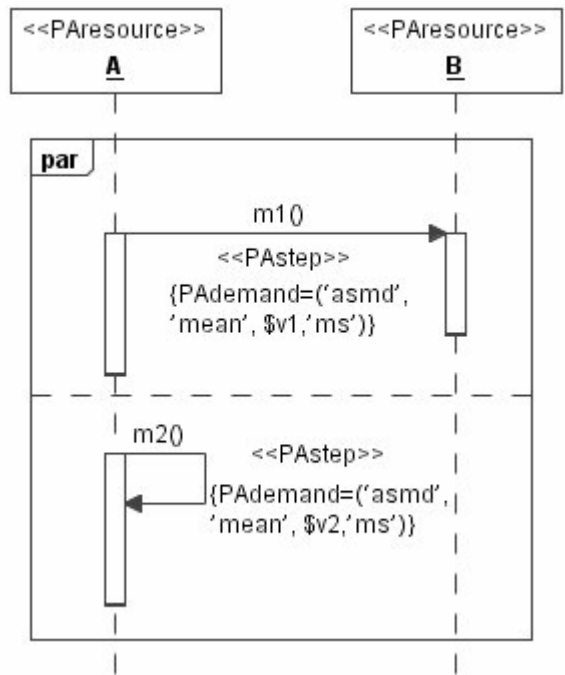


Figure 19: Example of *par* fragment

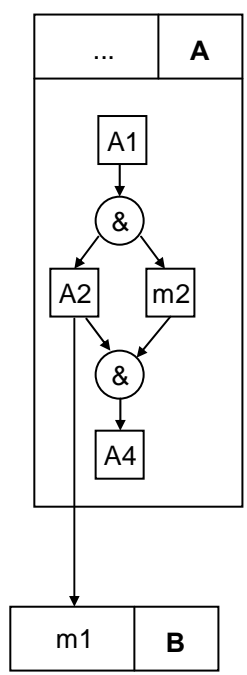


Figure 20: Translation of *par* fragment in LQN notation

The last type of fragment we consider is *loop*, which models the repeated execution of the set of scenario steps contained within the corresponding fragment. Its translation within the LQN model generates a LQN activity within the task executing the first operation of the sequence. The activity repeatedly invokes that operation for a number of time equal to number of loop repetition specified in the fragment the tagged value *PArep*. Figure 21 shows an example of *loop* fragment, where component *A* invokes service *m()* on component *B* for a number of time *n*. Figure 22 shows the LQN translation of the *loop* fragment, generated according to the description above. The translation is not connected to the rest of the model, since we do not know its full context.

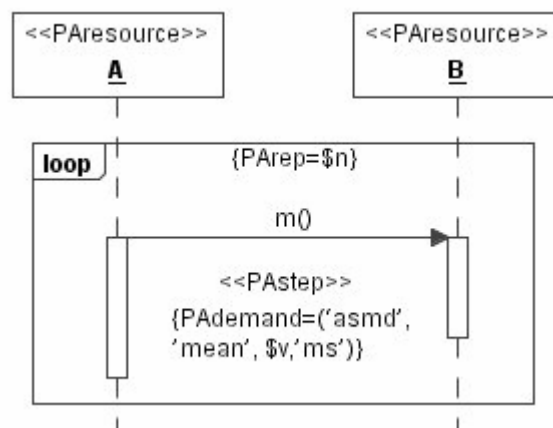


Figure 21: Example of *loop* fragment

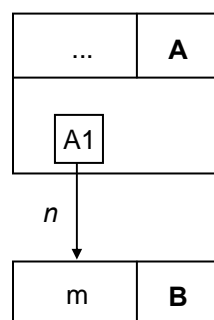


Figure 22: Translation of *loop* fragment in LQN notation

**STEP 2.e:**

The request flow among LQN entries and activities is clearly established from the sequence of messages represented in the Sequence Diagram.

A request arc is generated when a communication is detected between an entry or activity of a task playing the role of client, and the entry of another task, playing the role of server [20]. If a scenario step is associated with a `PAextOp` tagged value, denoting the usage of a hardware device other than the host processor executing the step, a request arc has to be generated to connect the entry requesting the use of the device with the entry created in the corresponding controller task for the interaction in exam; another arc has to be created to connect the receiving entry of the controller task to the destination entry of the server task.

A request can have different types. In fact, as reviewed in section 3.2.1, LQN service requests may be synchronous, asynchronous, or forwarding. Synchronous and forwarding interactions determine potential software blocking which may have significant performance implications; therefore it is important to determine them. With Sequence Diagrams synchronous and asynchronous messages are immediately identifiable based on the shape of the arrowhead corresponding to the interaction. Forwarding messages can instead be identified using the Call and Reply Stack (CRS) algorithm presented in [42], which follows the sequence of interactions between components and resolve their roles by examining the history of preceding messages.

Request arcs between activities are generated to connect them in sequence, loop, parallel, and alternative configurations, as seen with the translation of the combined fragments explained previously in this section. This leads to the creation of precedence graphs, which express for each task the internal and interaction dynamics of the corresponding software component in the system.

Figure 23 represents the outcome of the execution of Step 2 on the performance scenario represented in Figure 10.

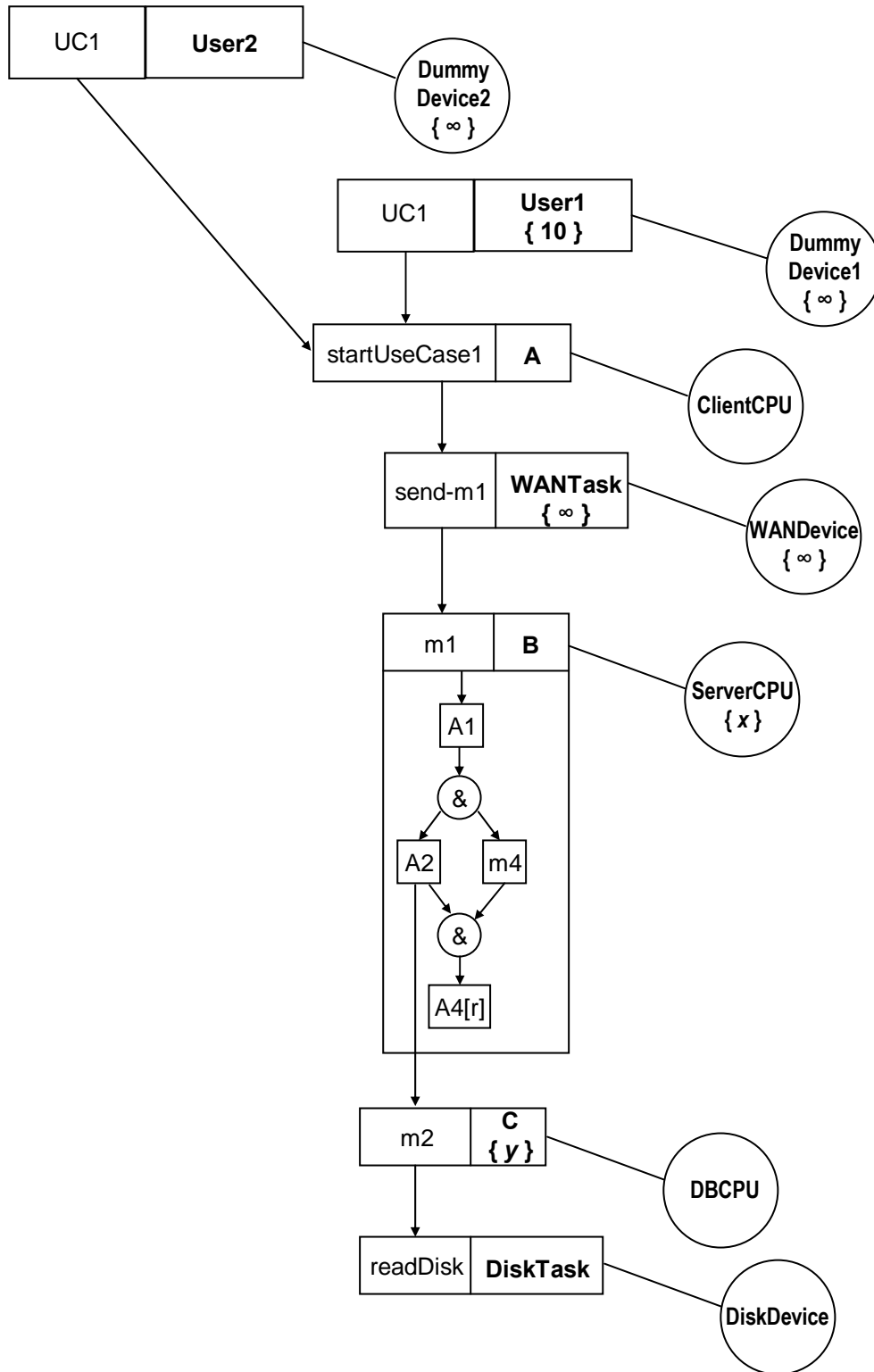


Figure 23: Sample LQN model at the end of Step 2

### **STEP 3:**

The LQN model obtained at this point needs to be parameterized with appropriate performance data, i.e., workload generated by reference tasks and service demands of entries and activities. These values are obtained using the adjusted workload information computed for performance scenarios, and the performance annotations on the UML Sequence Diagrams.

Reference tasks are parameterized depending on the associated user workload type. If a closed workload is considered, the think time for each entry in the task, is specified according with the think time calculated for the performance scenario corresponding to the entry. If an open workload is considered, the arrival rate for each entry in the task is specified according with the arrival rate calculated for the performance scenario corresponding to the entry.

Regarding service demands for entries and activities, they are defined using the *PAdemand* tag associated with the «*PAstep*» stereotype. We assume that the tag expresses the processing time required to prepare and send the message on the host processor. Performance requirements for non-processing resources are expressed by the *PAextOp* tag, which specifies the time demand of the software controller entry corresponding to the labeled interaction.

Figure 24 represents the outcome of the execution of Step 3 on the performance scenario represented in Figure 10.

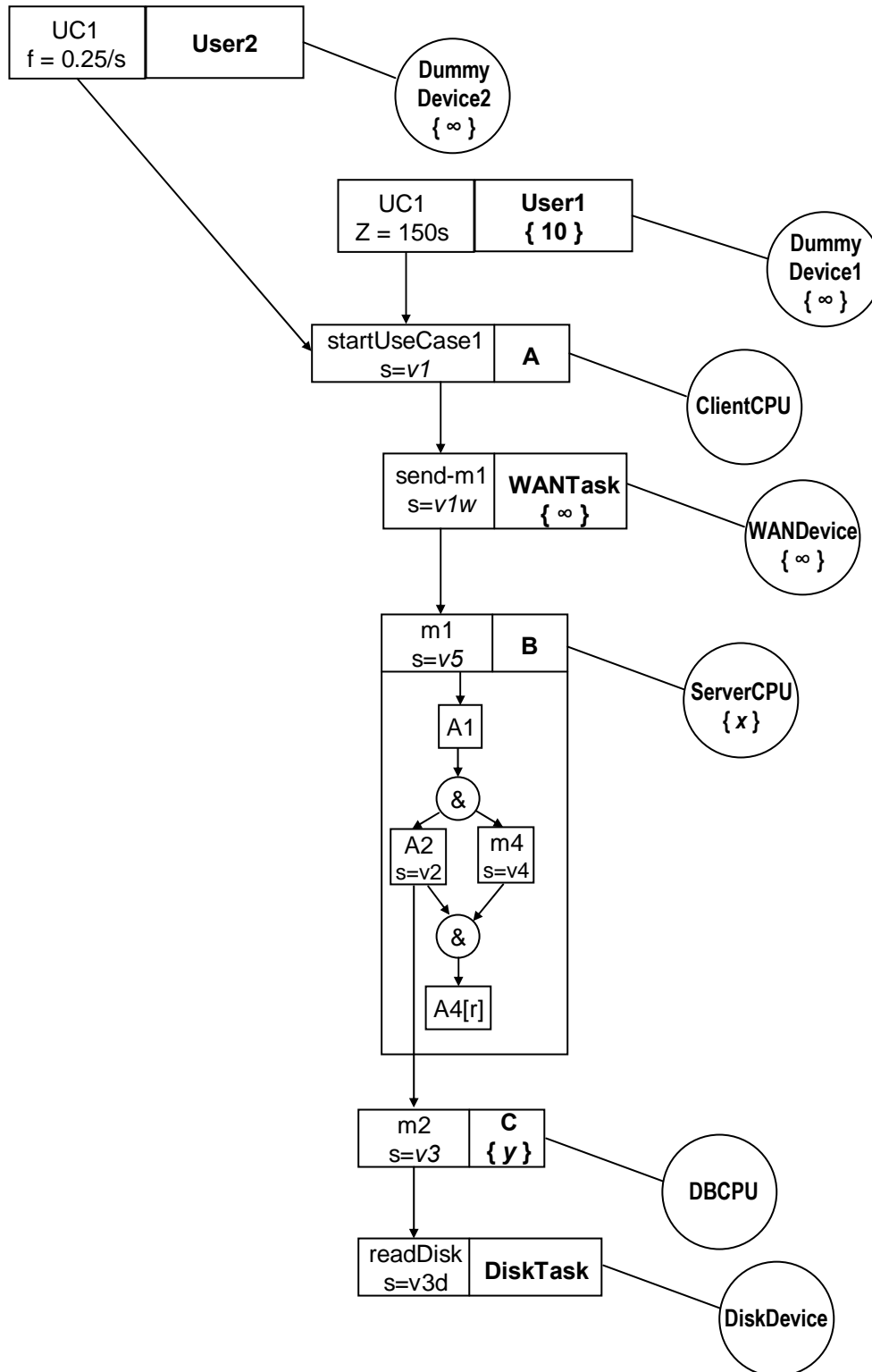


Figure 24: Sample LQN model at the end of Step 3

## Chapter 4: Case Study

In this chapter we present our experience with the application of the performance modeling methodology described in the previous chapter to the analysis of an airport inspection system that uses biometrically enabled, digitally signed travel documents. While the specific modeling parameters are hypothetical, system architecture resembles the systems being deployed at various US airports as part of the US-Visit program [48].

We first describe the system in terms of its structure and functionalities using high-level UML models based on typical requirements for similar applications. Hence, we build and parameterize performance models for the system. Finally, we report and analyze the obtained performance results.

### 4.1 System Description

An border inspection system is a complex combination of human processes and software systems used for traveler authentication at official Ports of Entry (POE) within a country. Hereinafter, we focus on airports since different POEs (i.e., land, sea) typically require different authentication protocols.

This section introduces context, structure, and functionalities of modern airport inspection systems. Our description is based on requirements for similar systems



emerging from technical reports and other documents released by U.S. government organizations [48] and the International Civil Aviation Organization (ICAO) [28, 29].

### **4.1.1 Context**

Increased security risk in international travel is resulting in new programs to determine the admissibility of foreign travelers at POEs within a country. Primary program goals are improving border security and, at the same time, facilitating the flow of legitimate travelers. Major program requirements include the adoption of Machine Readable Travel Documents (MRTDs) such as passports, visas, etc., the use of biometric identifiers, and the interoperability among multiple information systems for travelers' identity verification and background checks. In line with these emerging demands many countries have passed legislations that advance the incorporation of biometric and document authentication identifiers on MRTDs used at POEs for travelers' authentication (e.g., USA, New Zealand, Sweden, Pakistan, etc.).

*MRTDs* are international travel documents that contain both human-readable and machine-readable data. They contain world-wide standard data set by the ICAO. Simple forms of MRTDs are passports characterized by a machine readable strip at the bottom of the personal data page. The next level of MRTD, currently adopted by many countries, entails the incorporation a Secure Contactless Integrated Circuit (SCIC) [28, 29] that securely holds biometric data of the passport bearer.

*Biometrics* is a means of identifying a person by physiological or behavioral characteristics unique to an individual, using advanced computerized recognition techniques. It provides strong means of self-contained validation of the rightful MRTD bearer. Implementation of Digital Signatures (DSs) on MRTDs warrants integrity of the recorded data and avoids or minimizes fraud and counterfeit. Use of DSs requires the implementation of a Public Key Infrastructure (PKI) scheme, i.e., a framework to manage and enable the effective use of Public Key Encryption technology.

## 4.1.2 Structure

We assume that an airport inspection system consists of a series of identical traveler inspection facilities, to allow the inspection of multiple travelers at the same time. We call each inspection facility an airport inspection point. Our configuration for an inspection point, represented by the annotated Deployment Diagram in Figure 25, includes the following components:

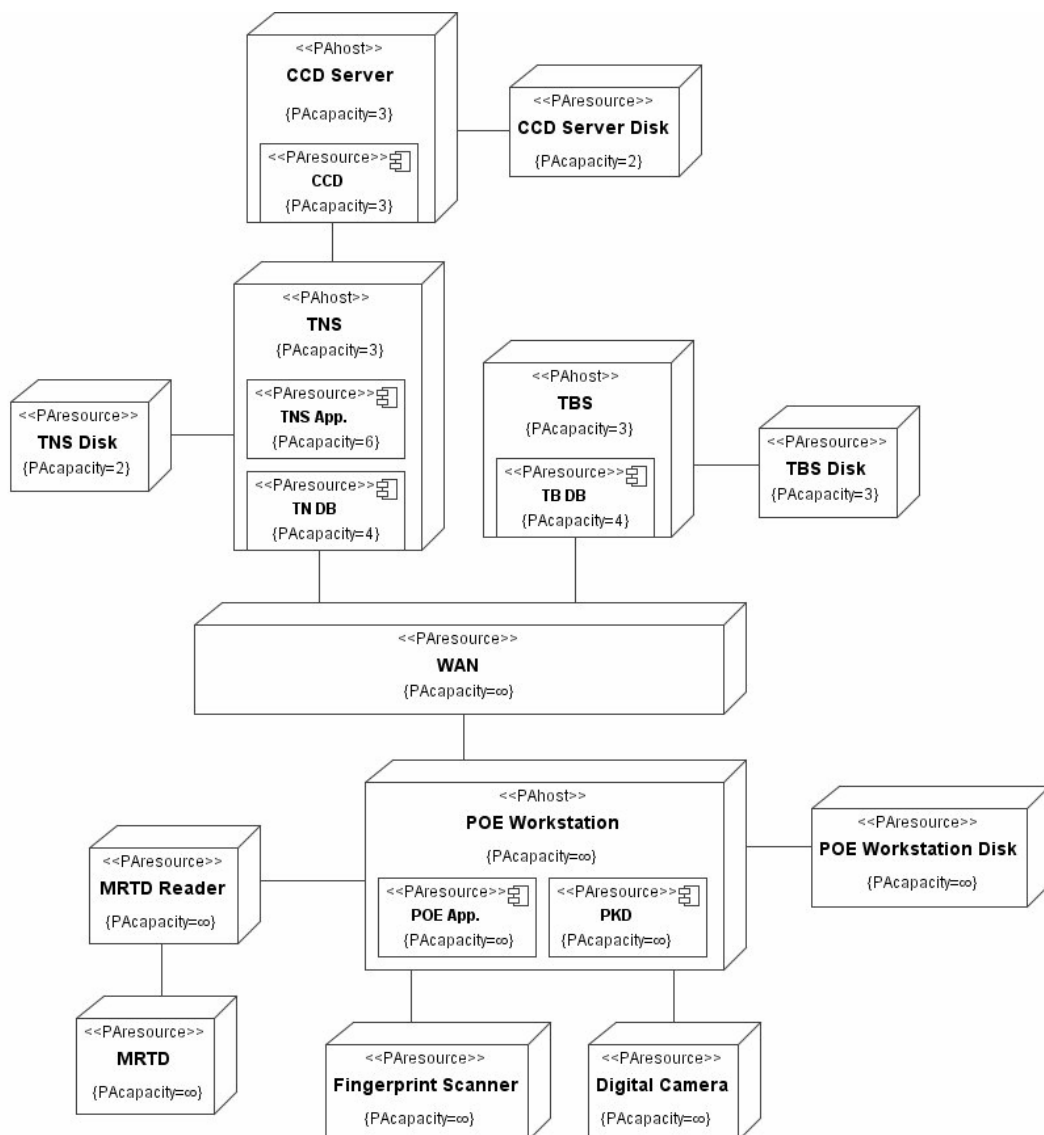


Figure 25: Possible Deployment Diagram for the airport inspection system

- The *Public Key Directory (PKD)* provides Public Key Certificates required to verify the authenticity of MRTDs handed by travelers at airport inspection points. The PKD is managed by a central authority (ICAO). Synchronized replicas are possible to reduce its workload and, accordingly, travelers' authentication time. Different options for the placements of the PKD can be considered: each airport inspection point, each POE, a regional, state, or national reference point, or combinations of them. Figure 25 represents a possible deployment of the PKD. Performance analysis, management concerns, and other issues and/or constraints emerging from system requirements and design will determine the convenience and efficiency of various architectural alternatives.
- The *Travelers' Names Server (TNS)* is a centralized server that provides access to a multiagency (law enforcement and other agencies) database of name-based lookout information. The database alerts officers of conditions that may make travelers inadmissible to the country. The database is also used by inspectors at POE to collect and modify traveler information.
- The *Travelers' Biometrics Server (TBS)* is a centralized server that stores and processes travelers' biometric data. During the authentication process the TBS can be used in verification or identification mode. In verification mode the system checks the validity of a claimed identity. In identification mode the system compares the individual's biometric with all stored biometric records. This provides an additional check to name-based checks and may help to detect travelers who have successfully established multiple identities.
- The *POE Workstation* is a computing device supporting the inspection officer in the collection and analysis of information coming from other components of the airport inspection point. Each POE Workstation accesses the PKD through a connection, whose exact type and capabilities depend on the location of the PKD itself. Communications with the TNS and with the TBS rely on a WAN.

Communication between the workstation and the MRTD Reader, and between the workstation and the biometric devices happens through a USB link.

- The *MRTD* is a document containing a chip with storage memory, which contains a digital photo plus optional fingerprints of the document bearer. A DS ensures the authenticity of data stored in the chip against unauthorized alteration or access. We assume that the Public Key Certificate of the MRTD issuing site, required to verify the authenticity of the signature on the MRTD, is stored on the MRTD itself, or in the PKD.
- The *MRTD Reader* is a computing device responsible for reading data from the MRTD and transferring it to the POE Workstation.
- The *Fingerprint Reader* is a biometric device responsible for capturing travelers' fingerprint data and transferring it to the POE Workstation.
- The *Digital Camera* is a device responsible for capturing travelers' face image data and transferring it to the POE Workstation.

In the Deployment Diagram for the inspection point we associate an infinite capacity with dedicated resources, i.e., resources that are exclusive of each inspection point, and used by one user at a time (the currently inspected traveler). Example of such resources are the POE Workstation, the MRTD Reader, and the Fingerprint Scanner. On the other hand, we associate a finite capacity to resources that are shared with other inspection points or inspection systems and serve multiple users at a time. Examples of these resources are the TNS, the TBS, and the CCD Server.

### **4.1.3 Functions**

Figure 26 shows a Use Case Diagram for the airport inspection system. The diagram represents two types of users: travelers, who require inspections, and other border inspection systems, which use system resources to perform name-based lookups, and biometric verification and identification. All user types and system functions are

considered to be relevant from a performance perspective, hence they are annotated with quantitative performance information.

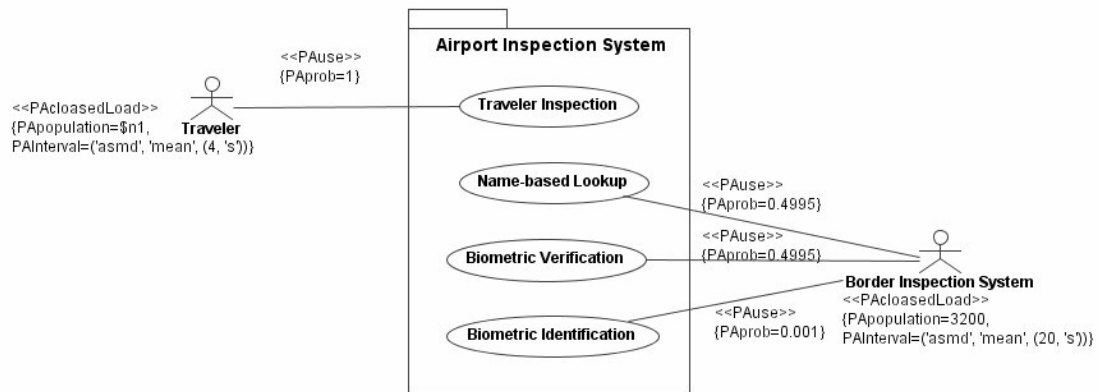


Figure 26: Use Case Diagram for the airport inspection system

The main function performed by the airport inspection system is travelers' inspection, whose dynamics is represented by the Sequence Diagrams in Figures 27 through 31. The diagrams are annotated with performance data. However, to make them more readable, and to list all the performance parameters for the system in a single location, we report resource demands for scenario steps in Table 5 of Section A.3.

When a traveler arrives at an airport inspection point, an inspection officer starts an authentication process by performing a primary inspection. The outcome of the authentication is access authorization for the vast majority of travelers. However, based on the results of watch list queries, behavioral observations, document reviews, etc., an officer may refer a visitor to a secondary inspection, consisting of multiple system queries, in-depth interviews, and thorough review of documentation and personal belongings (Figure 31).

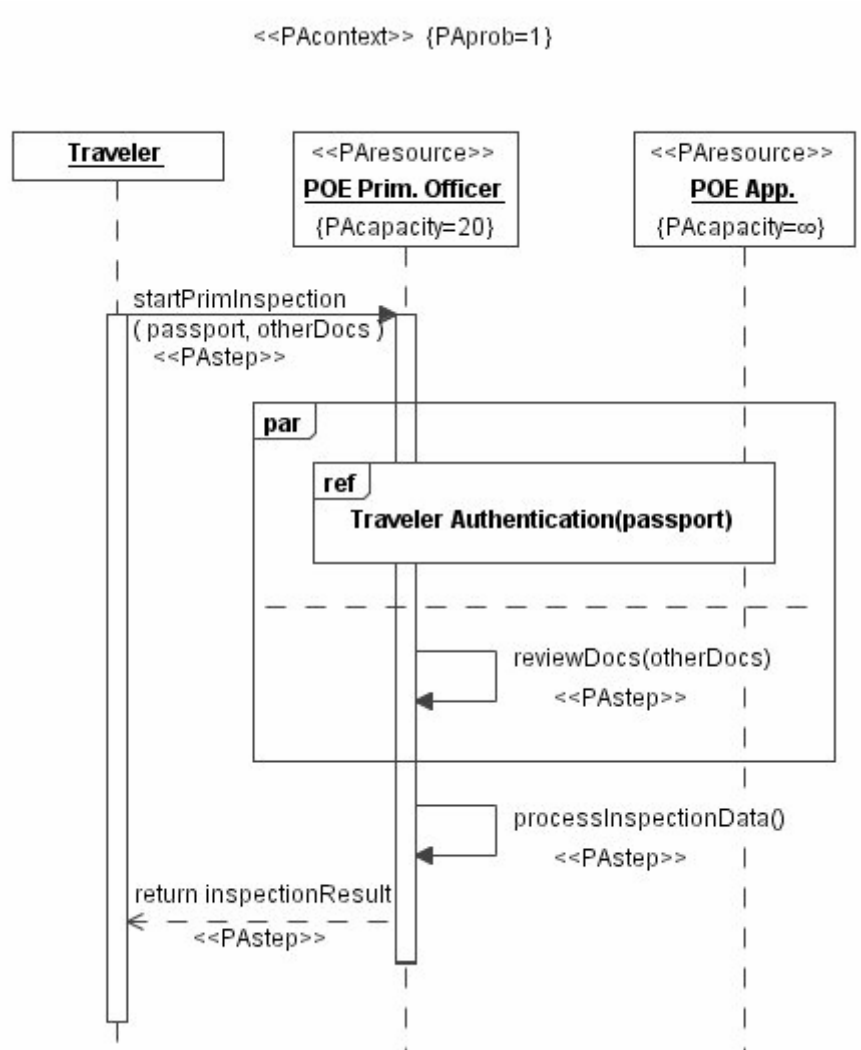


Figure 27: Sequence Diagram for the Traveler Inspection use case

Travelers' inspection, represented in Figure 27, consists of the parallel execution of an automated authentication process (e.g., MRTD check, name lookup, biometric verification) and a brief interview and manual revision of the traveler's documents by an inspection officer. The automated authentication process, shown in Figure 28, starts with the scanning of the traveler's MRTD through the MRTD Reader. The data on the card is read and its DS is verified using the Public Key Certificate recorded on the card itself or in the PKD. The authenticity of the retrieved Public Key Certificate is also checked. Hence DS of single MRTD data elements (MRZ and face image data) are verified (Figure 29).

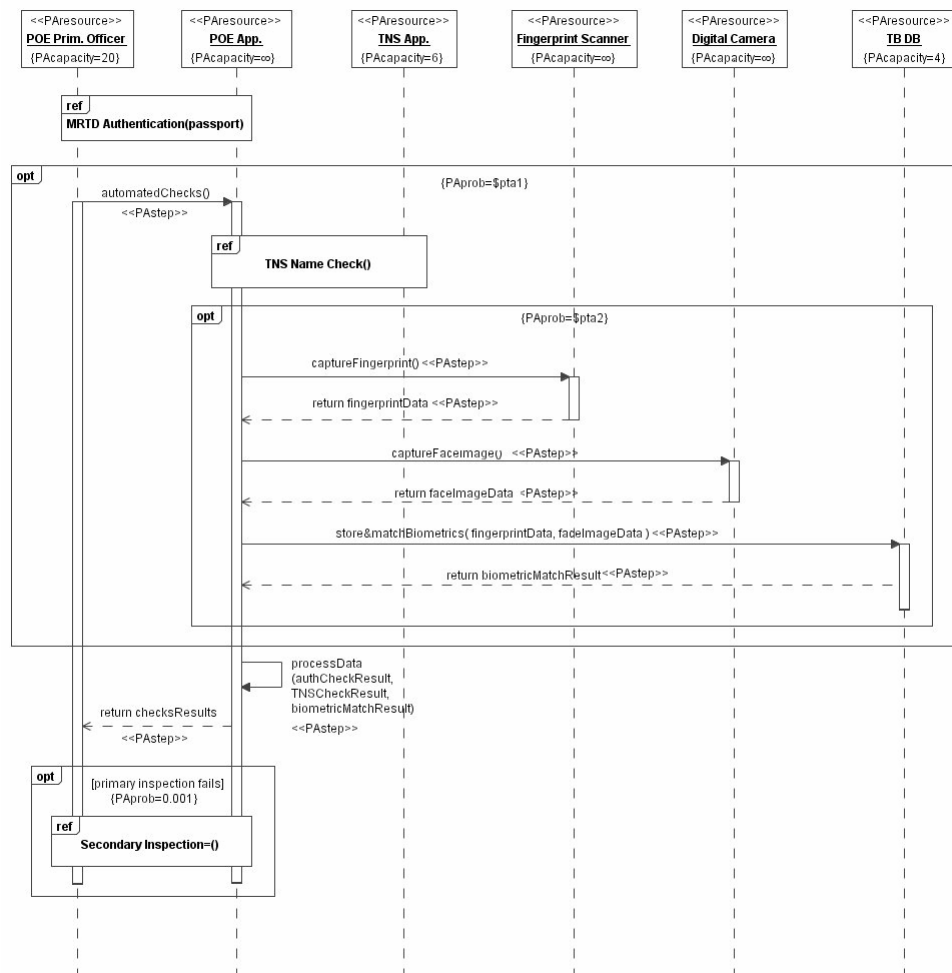


Figure 28: Sequence Diagram for the Traveler Authentication interaction

The TNS name check, represented in Figure 30, is performed next and returns any existing information about the traveler, including biographic lookout hits and a picture. Afterwards, the officer requests the traveler to scan his/her fingerprints (left and right index fingers), and captures his/her face image using a digital camera. The collected data is forwarded to the TBS, where it is checked against existing traveler's biometric samples (we assume that all travelers are pre-enrolled in the TBS, for instance at MRTD or visa request time). The system performs a 1:1 match to confirm that the person submitting his/her photo is the person on file. Results from the match, together with those from the previously described checks are finally reviewed by the inspection officer. Based on gathered information and observations, the officer decides whether sending the traveler to secondary inspection for further screening or processing, or granting him/her access to the country.



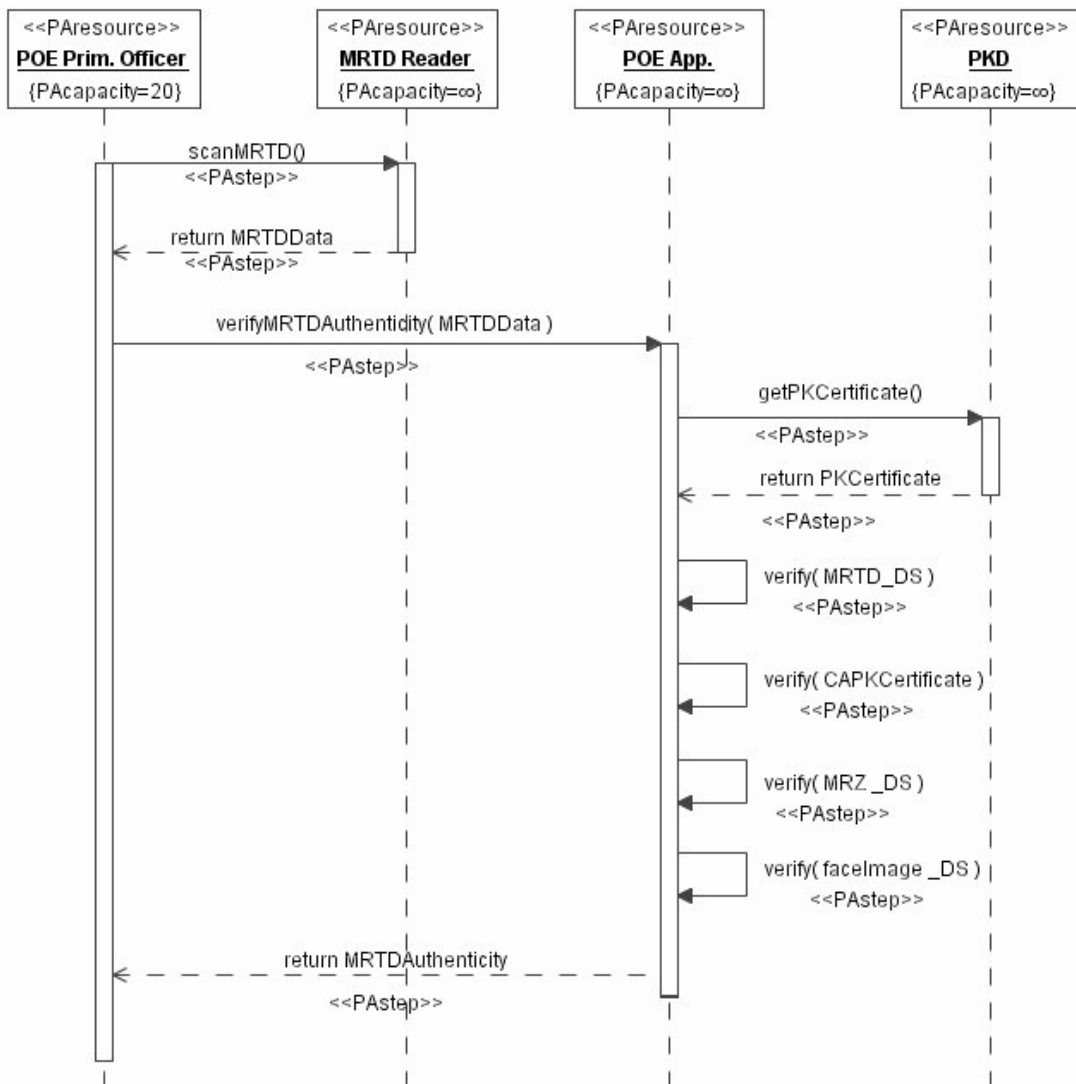


Figure 29: Sequence Diagram for the MRTD Authentication interaction

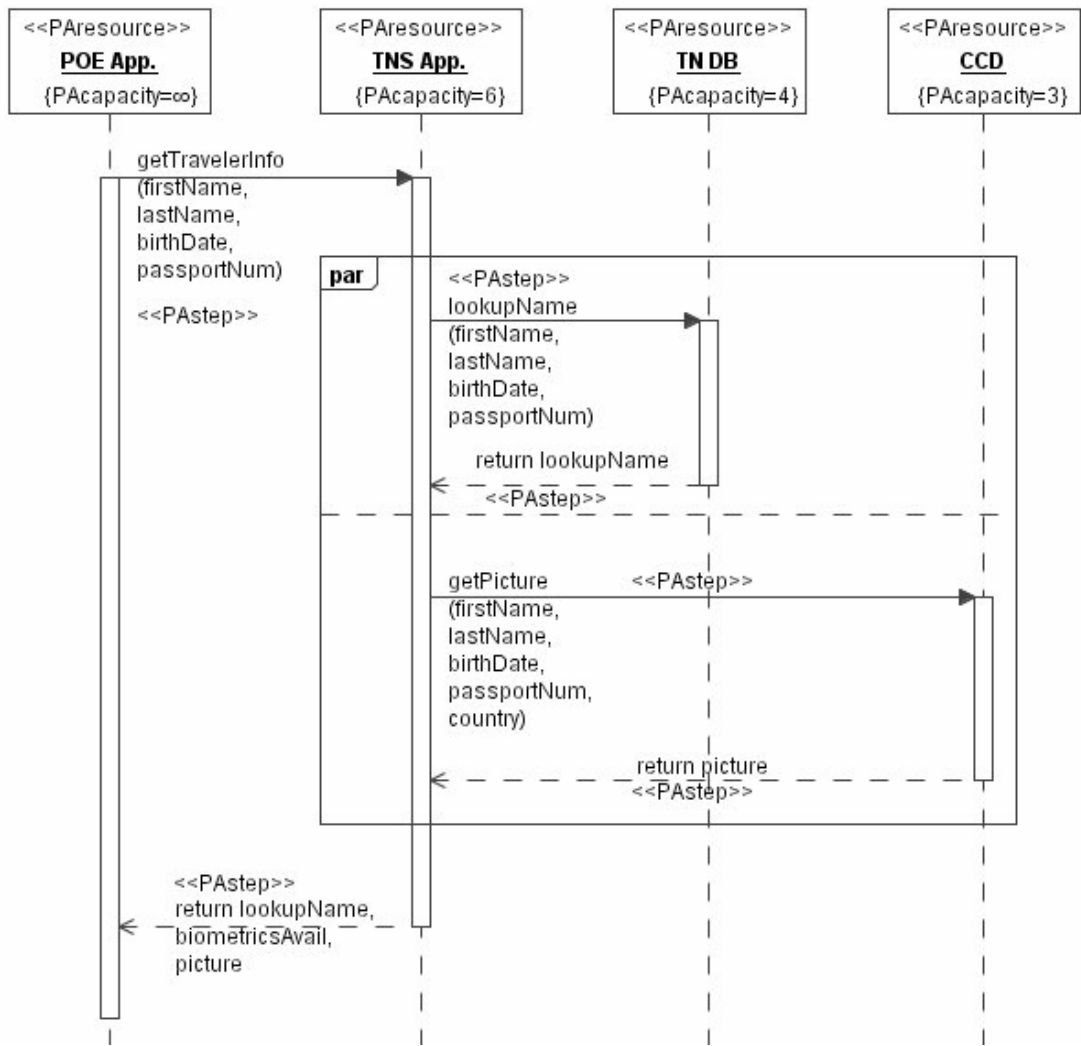


Figure 30: Sequence Diagram for the TNS Name Check interaction

The dynamics of the Secondary Inspection interaction occurrence is represented by the Sequence Diagram in Figure 31. Resource demands for scenario steps are reported in Table 5 of Section A.3.

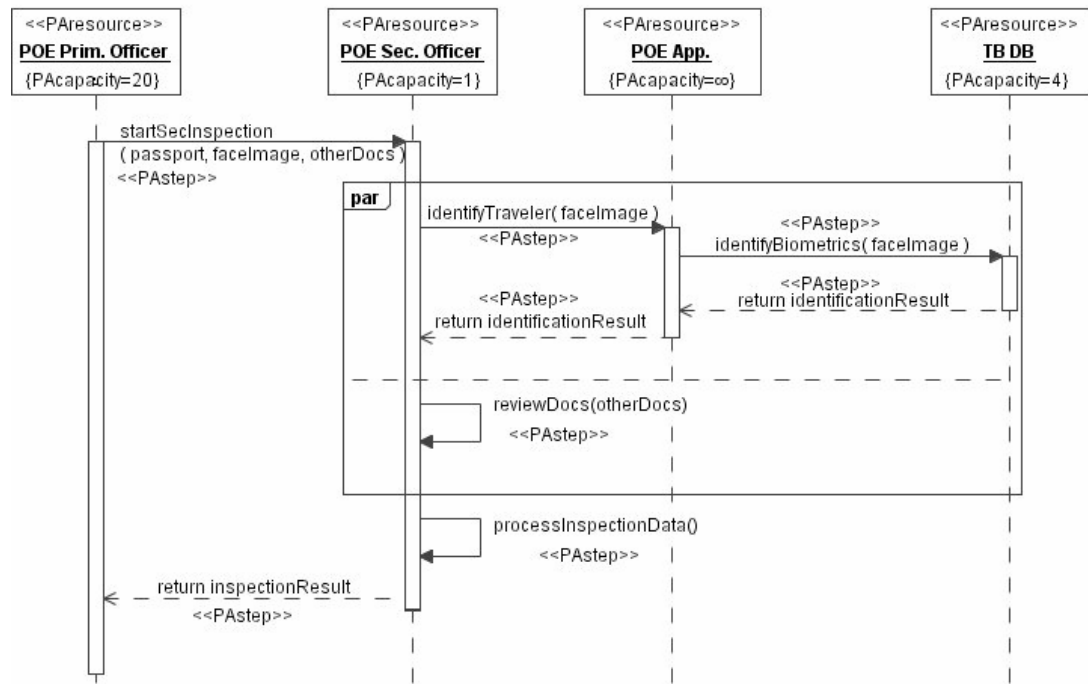


Figure 31: Sequence Diagram for the Secondary Inspection interaction

Figures 32-34 represent the dynamics of the Name-based Lookup, the Biometric Verification, and the Biometric Identification use cases, respectively. Resource demands for scenario steps are reported in Table 5 of Section A.3.

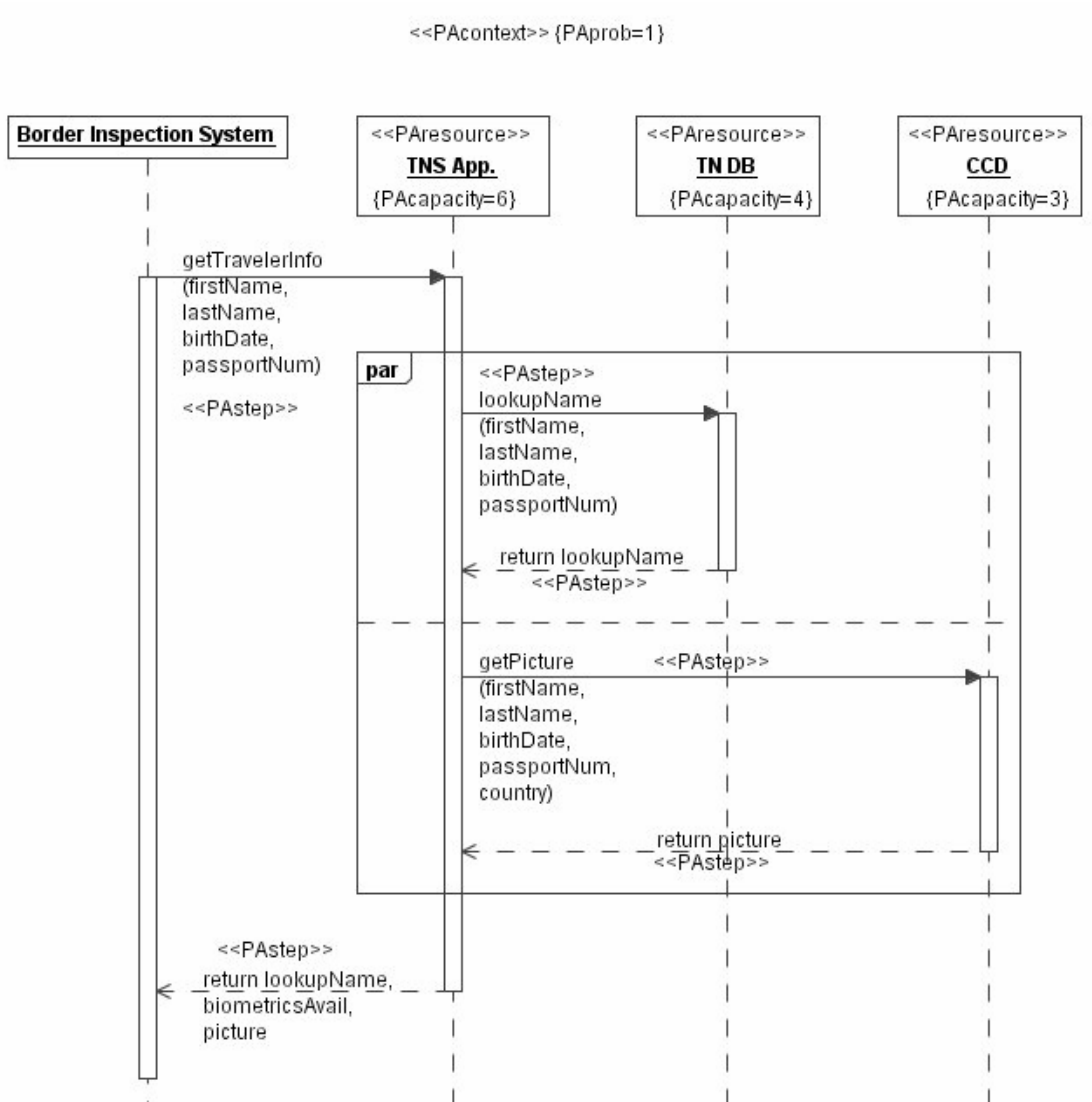


Figure 32: Sequence Diagram for the Name-based Lookup use case

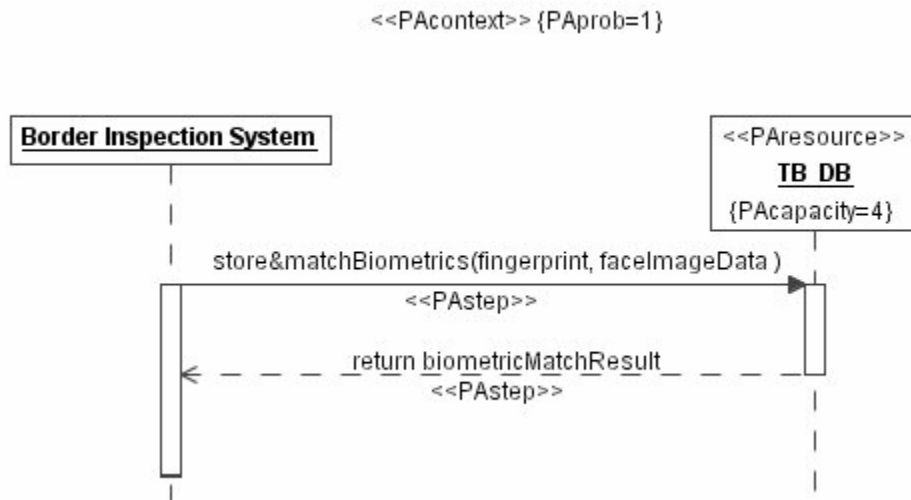


Figure 33: Sequence Diagram for the Biometric Verification use case

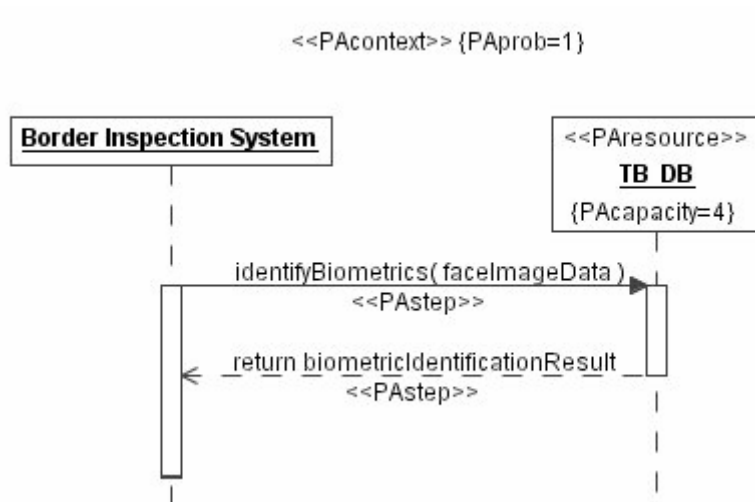


Figure 34: Sequence Diagram for the Biometric Identification use case

#### 4.1.4 Technical and Policy Options

We intend to evaluate the performance impact of different technical configurations and policy options that can be adopted to implement primary and secondary inspection processes within an airport inspection system. Results of the performance evaluation can be used to understand what the primary drivers affecting system performance are, and to enable policymakers to plan accordingly, in terms of infrastructure, scheduling system implementation, or policy changes.

The technical configurations we consider represent different alternatives for the architecture of the airport inspection system. Each configuration corresponds to a different possibility for the location of the PKD, which stores Public Key Certificates of MRTD issuing sites and of country Certificate Authorities (CAs). The latter are used to verify the authenticity of MRTD issuers' certificates. The configurations under exam are described below:

- MRTD: MRTDs store Public Key Certificates of the corresponding issuing sites; Public Key Certificates of country CAs are stored in the PKD, which is replicated at each POE workstation.
- PKD Local: Public Key Certificates of MRTD issuing sites and of country CAs are collectively stored in the PKD, which is replicated at each POE workstation.
- PKD Remote: Public Key Certificates of MRTD issuing sites and of country CAs are collectively stored in the PKD. The PKD may be available at a single location within the host country or it may be replicated at each POE, or region of POEs.

Options 1 and 2 share the same structure, represented by the Deployment Diagram in Figure 25. The difference between these options lies in the content of the MRTD, and the size of the PKD stored at the POE Workstation. To keep this chapter clear and readable we separately describe Option 3 in Appendix A.

The policy options we consider are intended to explore how variations in the authentication procedure, due for instance to the nature of the verified traveler's data, or to the traveler's nationality, affect authentication time and throughput. We considered three possible inspection scenarios:

- Scenario 1: A traveler is granted access based only on the validity of his/her MRTD, which is determined by verifying the MRTD digital signature, through access to the PKD.
- Scenario 2: The traveler authentication process includes the MRTD verification described in scenario 1. It also includes a name based check, to exclude that the

traveler is on a watchlist of inadmissible individuals, and a biometric based check, to verify that the biometric data collected from the traveler matches the biometric data stored in the TBS.

- Scenario 3: The traveler authentication process varies based on travelers' nationality. In fact, national travelers only require MRTD authentication and a name-based watchlist check. On the other hand, foreign travelers must follow the inspection process described in Scenario 2.

## 4.2 Performance Modeling

In this section we apply our performance evaluation methodology to the analysis of the airport inspection system described in the previous section. As we stated in Section 4.1.4, the given description actually represents two technical configurations for system: Options 1 and 2. These options share the same structure and functions; however, their MRTD-related operations have different resource demands. As a result, application of steps 1 and 2 of our UML to LQN transformation to those cases results in the same outcome. On the other hand, parameterization of the obtained LQN model, performed in step 3 of the transformation, is different; for this reason we will describe this operation for the two options separately.

### 4.2.1 Assumptions

To simplify our modeling task we have made several assumptions:

- all travelers bear MRTD with digitally signed data and picture stored in it;
- all travelers are aggregated into a single class, i.e., they are authenticated following the same process, through the same facilities;
- all travelers are pre-enrolled in the biometric system, i.e., at least one biometric sample is stored in the TBS for each traveler;

- only a 1-to-1 verification check against the biometric sample stored in the TBS is performed at primary inspection. A 1-to-n check against the biometric watchlist is conducted at enrollment time and repeated at secondary inspection;
- in our airport inspection system the number of inspection points for traveler authentication is constant. We assume one traveler queue for primary inspection, and a separate traveler queue for secondary inspection.

## 4.2.2 Model Structure

The structure of the LQN model (i.e., tasks, devices, and their mappings) for the airport inspection system is generated by Step 1 of our UML to LQN transformation. The next subsections describe the execution of this step based on the outcome of its substeps.

### **STEP 1.a:**

This step creates LQN devices for each hardware node – whether stereotyped as <<PAhost>> or as <<PAresource>> – in the annotated Deployment Diagram for the system. The application of the step to the inspection system generates the LQN devices represented in Figure 35.



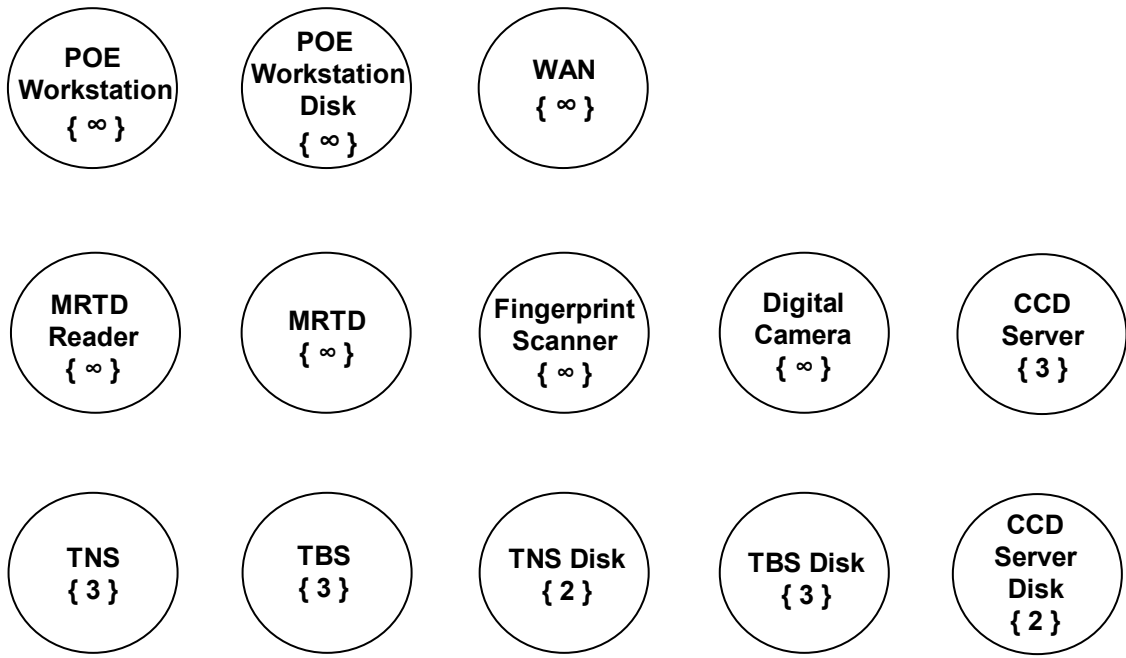
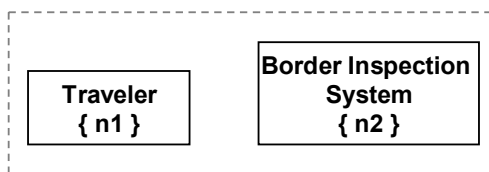


Figure 35: LQN devices for the airport inspection system

**STEP 1.b:**

This step creates LQN reference tasks to represent different user workloads. It also creates LQN non-reference tasks for each each system component labeled by the <<PResource>> stereotype in the Deployment Diagram or the in Sequence Diagrams for the system. Figure 36 shows the outcome of the application of this step to the airport inspection system.

Reference Tasks



Non-reference Tasks

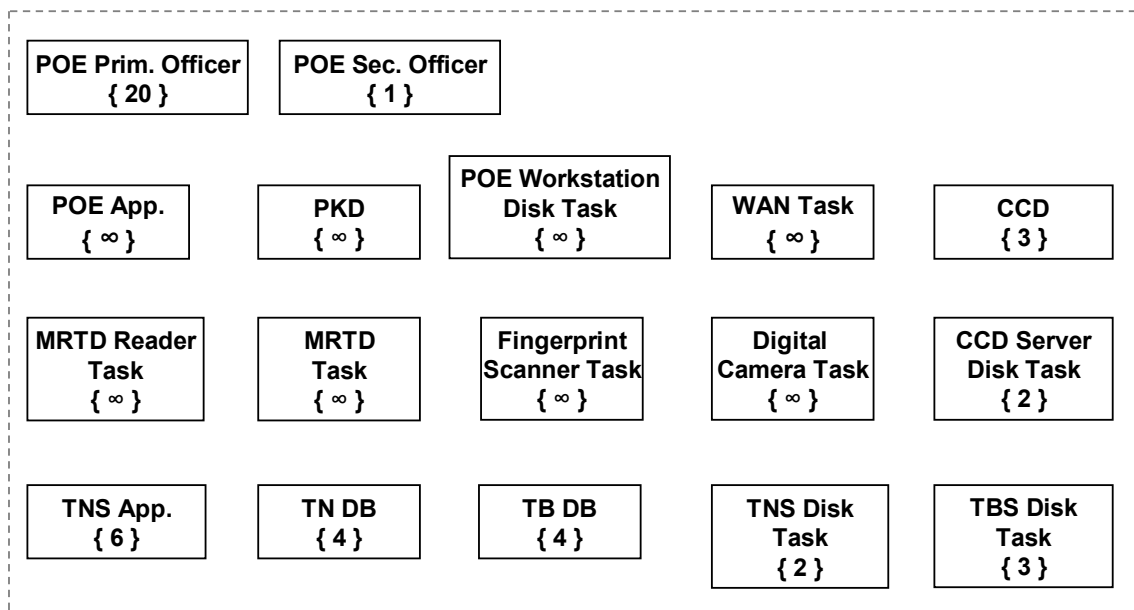


Figure 36: LQN tasks for the airport inspection system

**STEP 1.c:**

This step creates connecting arcs between the LQN tasks and devices generated in the previous steps. Figure 37 represents the result of the application of the step to the airport inspection system.

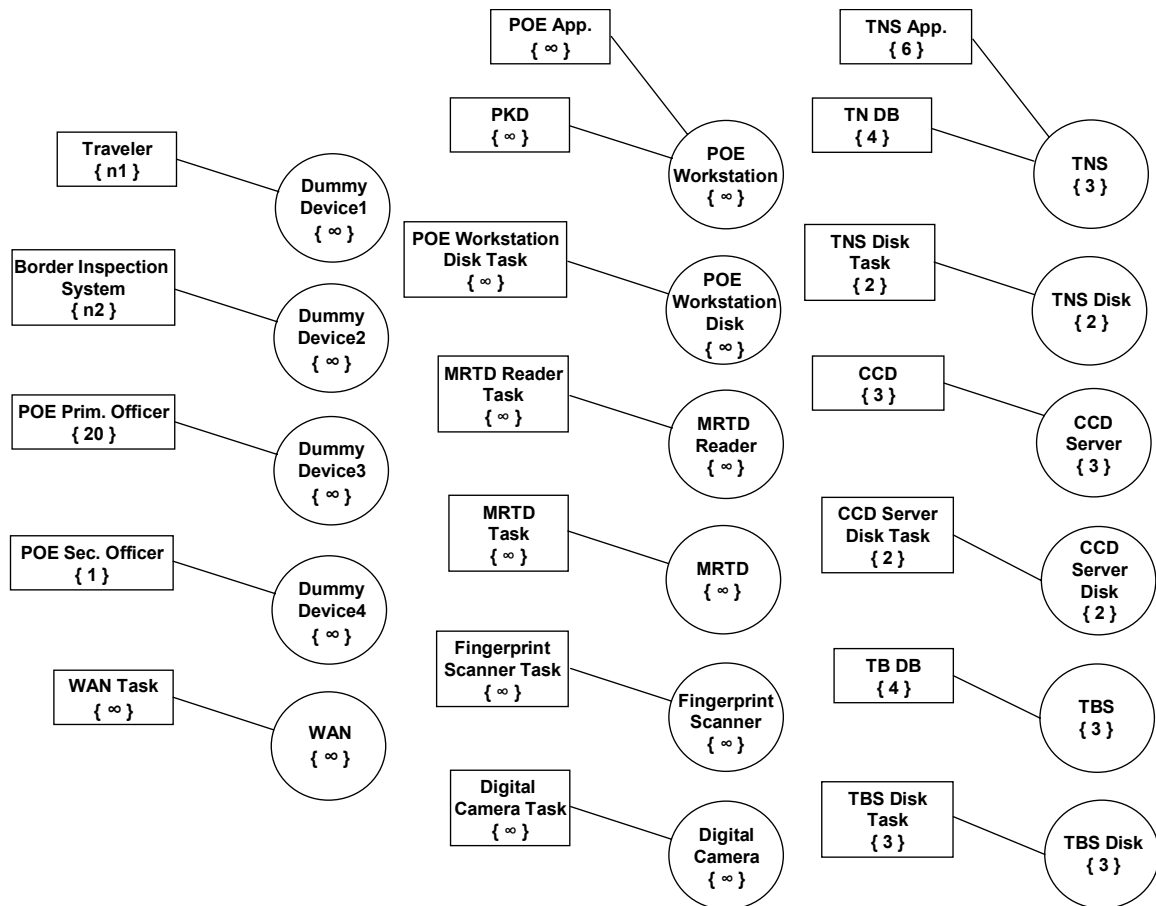


Figure 37: LQN tasks, devices, and their mappings for the airport inspection system

### 4.2.3 Model Dynamics

The dynamics of the LQN model (i.e., entries, activities, and request flow among them) is generated by Step 2 of our UML to LQN transformation. The next subsections describe the execution of this step based on the outcome of its substeps.

#### STEP 2.a:

This step creates entries for the LQN reference tasks defined in Step 1.b. Each entry matches a performance scenario invoked by the user corresponding to the reference task. In the case of the inspection system the value 1 associated with the  $P_{Aprob}$  tag in each Sequence Diagram implies a single performance scenario per use case. This leads to the following entries:

- *travelerInspection* for the *Traveler* reference task;

- *name-basedLookup*, *biometricVerification*, and *biometricIdentification* for the *Border Inspection System* reference task.

**STEP 2.b:**

This step creates entries for each LQN task corresponding to a system component receiving service requests from other components. In the case of the inspection system the following entries are identified:

- *startPrimInspection* for the *POE Prim. Officer* task;
- *startSecInspection* for the *POE Sec. Officer* task;
- *automatedChecks*, *verifyMRTDAuthenticity*, and *identifyTraveler* for the *POE App.* task;
- *scanMRTD* for *MRTD Reader Task*;
- *getPKCertificate* for the *PKD* task;
- *captureFingerprint* for *Fingerprint Scanner Task*;
- *captureFaceImage* for *Digital Camera Task*;
- *getTravelerInfo* for the *TNS App.* task;
- *lookupName* for the *TN DB* task;
- *getPicture* for the *CCD* task;
- *store&matchBiometrics* and *identifyBiometrics* for the *TB DB* task;

**STEP 2.c:**

This step generates entries of LQN tasks corresponding to passive resources whose usage is required to perform certain operations. Examples of such resources for the inspection system are the WAN and storage disks. Their use is explicitly represented by performance scenarios through the `PAextOp` tag optionally associated with scenario steps\* .

Execution of step 2.c on the inspection system leads to:

---

\* Annotations for scenario steps of the airport inspection system are reported in Appendix A, Table 4.

- the creation of entries *send-getTravInfo*, *send-store&matchBiom*, and *send-identifyBiom* for *WAN Task*. These entries correspond to the network operations required to invoke the services provided by the TNS and by the TBS. In fact these servers are connected to the POE Workstation through a WAN link.
- the creation of entry *readMRTDData* for *MRTD Task*. The entry is required by the `PAextOp` tag associated with the *scanMRTD()* interaction in the MRTD Authentication fragment. The tag indicates that the interaction requires a reading operation on the MRTD chip.
- the creation of entry *readPKCertData* for *POE Workstation Disk Task*. The entry is required by the `PAextOp` tag associated with the *getPKCertificate()* interaction in the MRTD Authentication fragment. The tag indicates that the interaction requires a reading operation on the disk of the POE Workstation.
- the creation of entry *readLookupData* for *TNS Disk Task*. The entry is required by the `PAextOp` stereotype associated with the *lookupName()* interaction in the TNS Name Check fragment. The tag indicates that the interaction requires a reading operation on the disk of the TNS.
- the creation of entry *readPictureData* for *CCD Server Disk Task*. The entry is required by the `PAextOp` tag associated with the *getPicture()* interaction in the TNS Name Check fragment. The tag indicates that the interaction requires a reading operation on the disk of the CCD Server.
- the creation of entry *readWriteBiomData* and *readWatchlistData* for *TBS Disk Task*. The former is required by the `PAextOp` stereotype associated with the *store&matchBiometrics()* interaction in the Traveler Authentication fragment. The tag indicates that the interaction requires a reading and a writing operation on the disk of the TBS. The latter is required by the `PAextOp` tag associated with the *identifyBiometrics()* interaction in the Secondary Inspection scenario. The tag

indicates that the interaction requires the reading of a set of biometrics samples (i.e., a watchlist) on the disk of the TBS.

**STEP 2.d:**

This step generates activities of LQN tasks, to represent internal computations of the system components corresponding to those tasks. Such computations are represented in Sequence Diagrams by self-addressed messages sent out by components.

The activities we identify for the inspection system are:

- *reviewDocs* and *processInspectionData* for the *POE Prim. Officer* task;
- *reviewDocs* and *processInspectionData* for the *POE Sec. Officer* task;
- *processData* and *verify* for the *POE App.* task.

We also identify other LQN activities to specify non-sequential flow of control, expressed in Sequence Diagrams by combined fragments. The fragments found in the Sequence Diagrams for the inspection system lead to the creation of different sets of activities to represent:

- the *par* fragment in the Primary Inspection scenario;
- the two *opt* fragments in the Traveler Authentication interaction occurrence;
- the *par* fragment in the TNS Name Check interaction occurrence;
- the *par* fragment in the Secondary Inspection scenario;
- the *par* fragment in the Name-based Lookup scenario.

The LQN translation of the above structures follows the mapping rules explained in Chapter 3. The names of the activities created in the translation are not relevant; we display the interconnections of those activities with the rest of the LQN model in the next subsection.

**STEP 2.e:**

This step generates the request flow among LQN entries and activities identified in the previous steps. The generation process follows the sequence of messages represented in each performance scenarios for the inspection system. In this section we gradually determine and display the process outcome.

Figure 38 shows a high-level framework of the LQN model for the airport inspection system. The framework represents the requests of the workload generators, i.e., the reference tasks *Traveler* and *Border Inspection System*, toward LQN submodels representing functions invoked by them. In general the LQN submodels are not disjoint. Rather, they usually overlap since different use cases may use the same system resources and invoke the same system services.

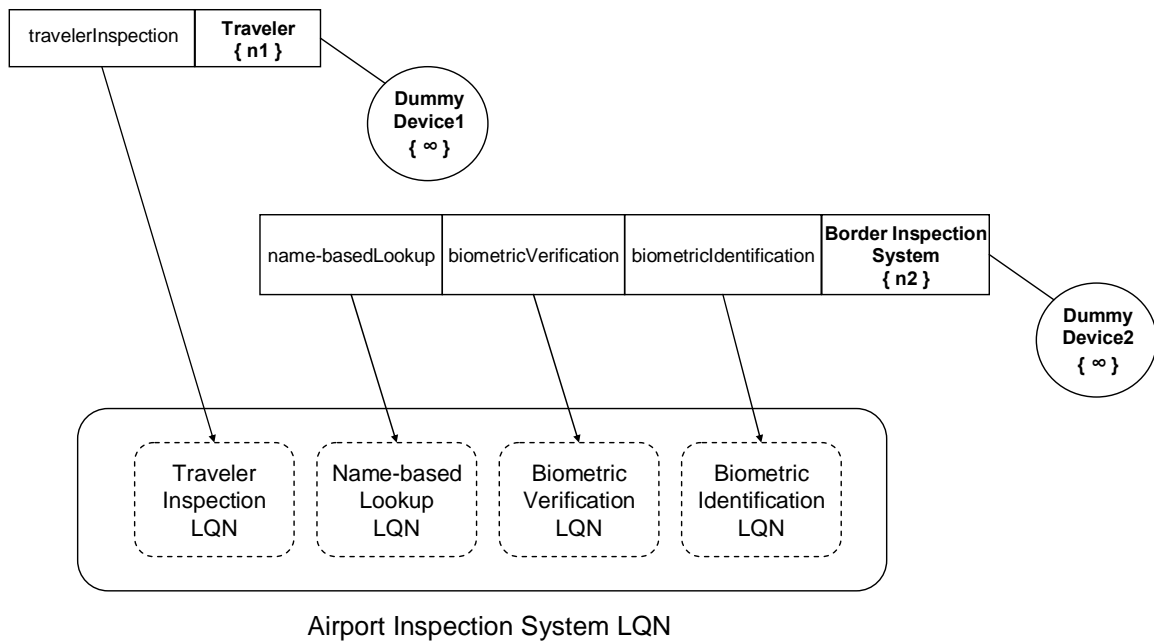


Figure 38: High-level framework of the LQN for the airport inspection system

In the next paragraphs we explain how to complete the framework for the LQN model. We generate (possibly overlapping) LQN submodels, which we later merge into a single LQN model for the whole system. To make the models more readable

and understandable we only represent tasks, entries, and activities of interest within the context under study.

The activities we introduce to represent control flow are given generic names, i.e.,  $A_i, i \geq 1$ . We assume the values of  $i$  to be unique within a single task, but not across different tasks. Values of  $i$  for a set of activities do not represent the order of executions of the activities. Rather, they express their order of creation, based on the order of processing of the interactions represented in Sequence Diagrams.

We now focus on how to define the *Traveler Inspection LQN*, represented in Figure 38. With this purpose, we process the set of Sequence Diagrams modeling the corresponding scenario. We start with the most general one (Figure 27), obtaining the submodel displayed in Figure 39, which represents the first draft of the *Traveler Inspection LQN*. The submodel contains a placeholder for the *Traveler Authentication* interaction occurrence. The submodel is refined by examining the Sequence Diagram(s) specifying that occurrence.

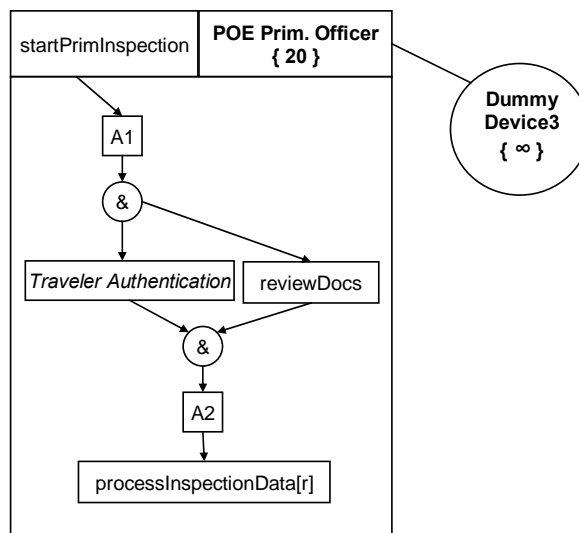


Figure 39: *Traveler Inspection LQN* after Traveler Inspection scenario

Processing the Traveler Authentication interaction occurrence augments the current *Traveler Inspection LQN* with tasks, entries, activities, and service requests modeling



the interactions represented in that occurrence. The outcome of the process is represented in Figure 40.

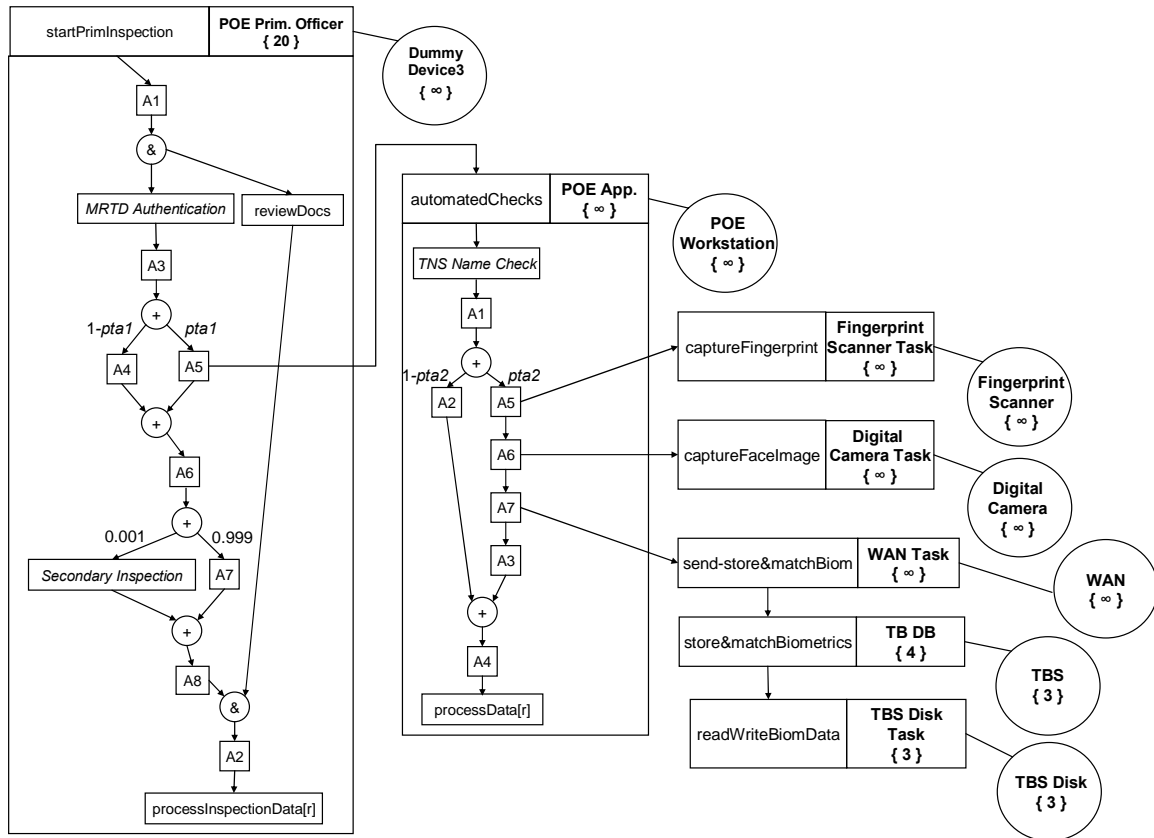


Figure 40: Traveler Inspection LQN after Traveler Authentication

Processing the MRTD Authentication interaction occurrence augments the current Traveler Inspection LQN as represented in Figure 41. For the sake of clarity we omit to reproduce again the activities invoked by the *automatedChecks* entry of the POE App. Task, as well as the tasks and entries invoked by those activities.

Processing the TNS Name Check interaction occurrence augments the current Traveler Inspection LQN as represented in Figure 42. We can notice that the abstract TNS Name Check activity, represented in Figure 41, is refined to generate a service request towards the TNS. The TNS generates service requests to the CCD Server and to the TN Database.

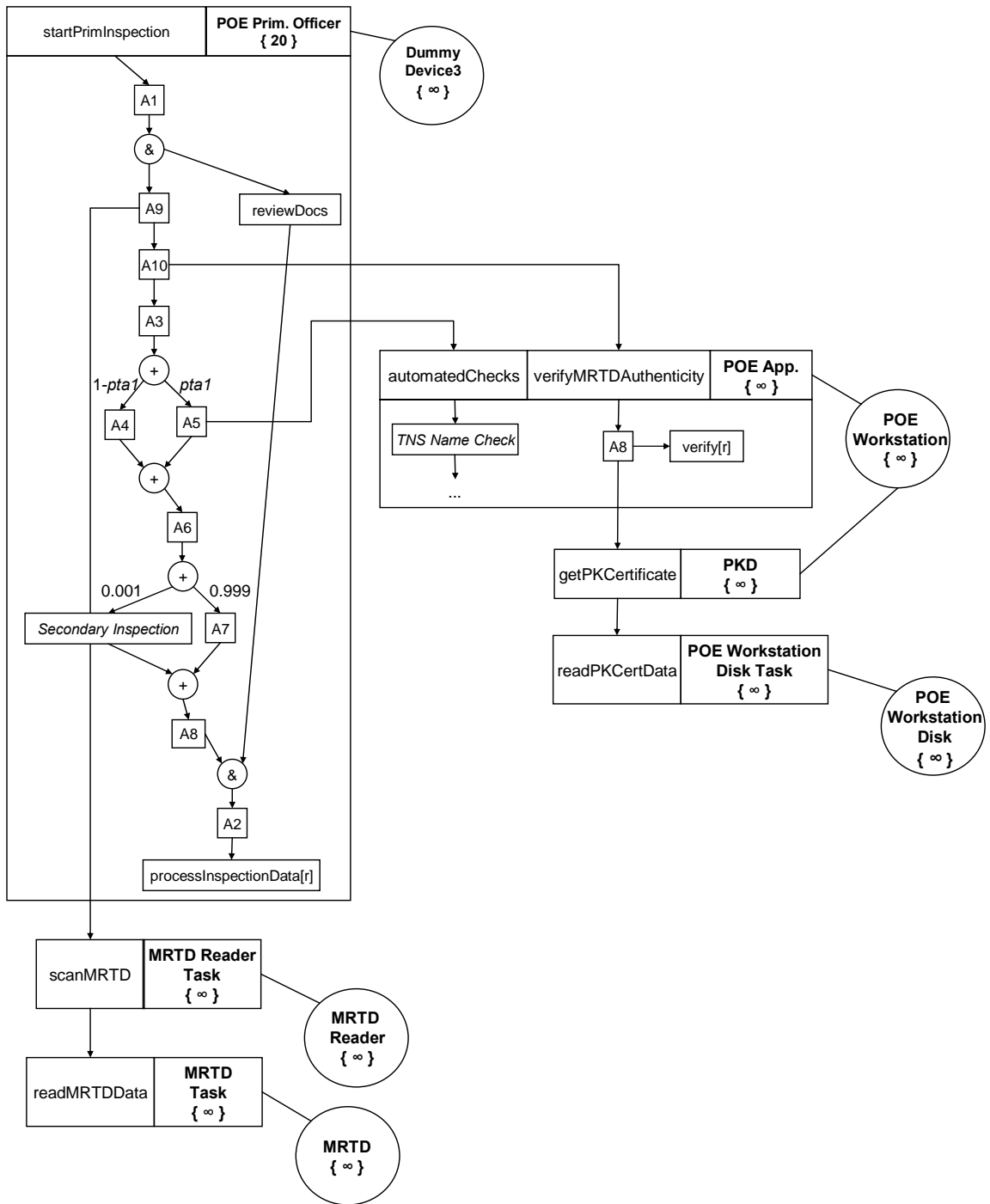


Figure 41: Traveler Inspection LQN after MRTD Authentication

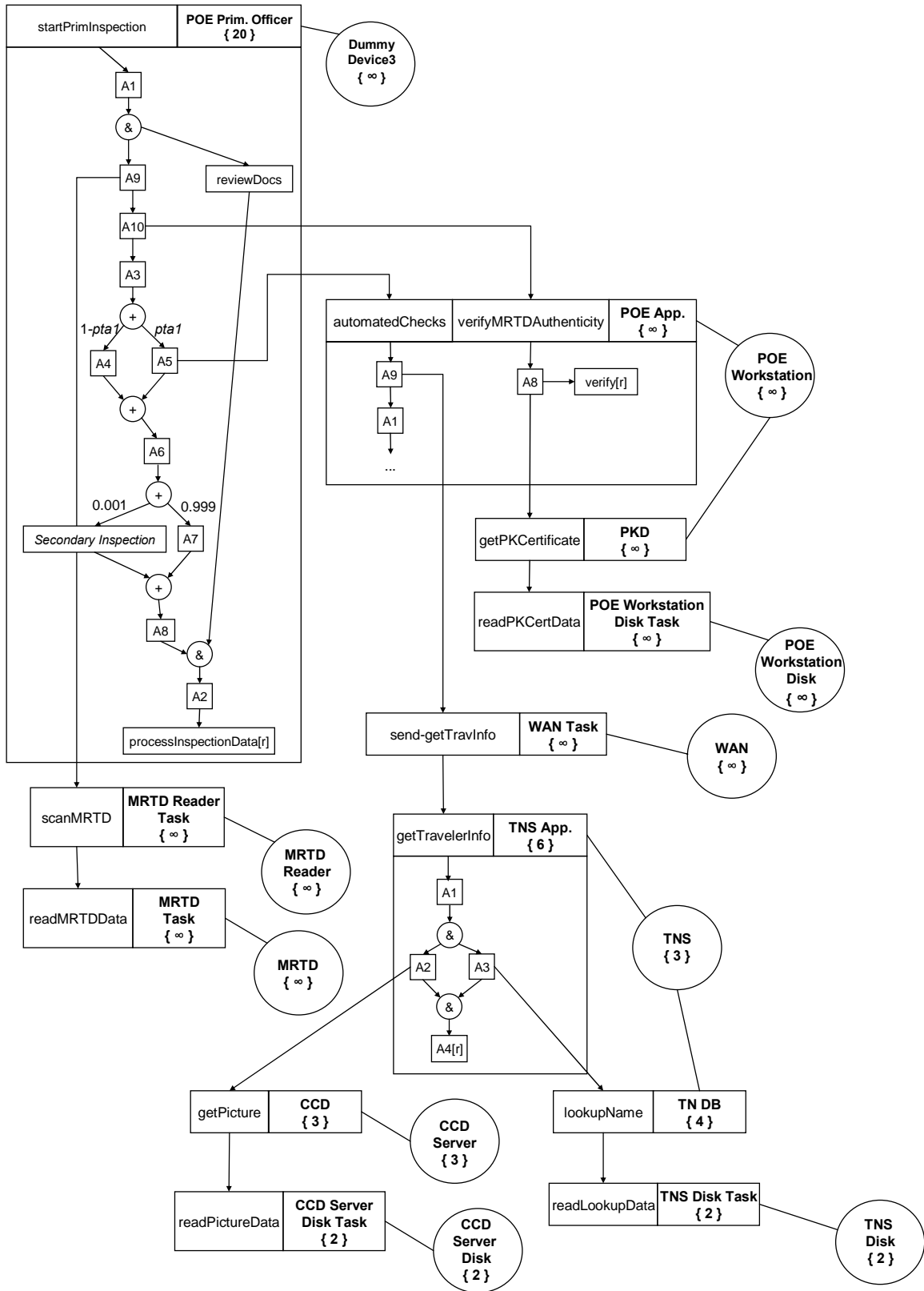


Figure 42: Traveler Inspection LQN after TNS Name Check

Finally, processing the Secondary Inspection interaction occurrence augments the current *Traveler Inspection LQN* as represented in Figure 44.

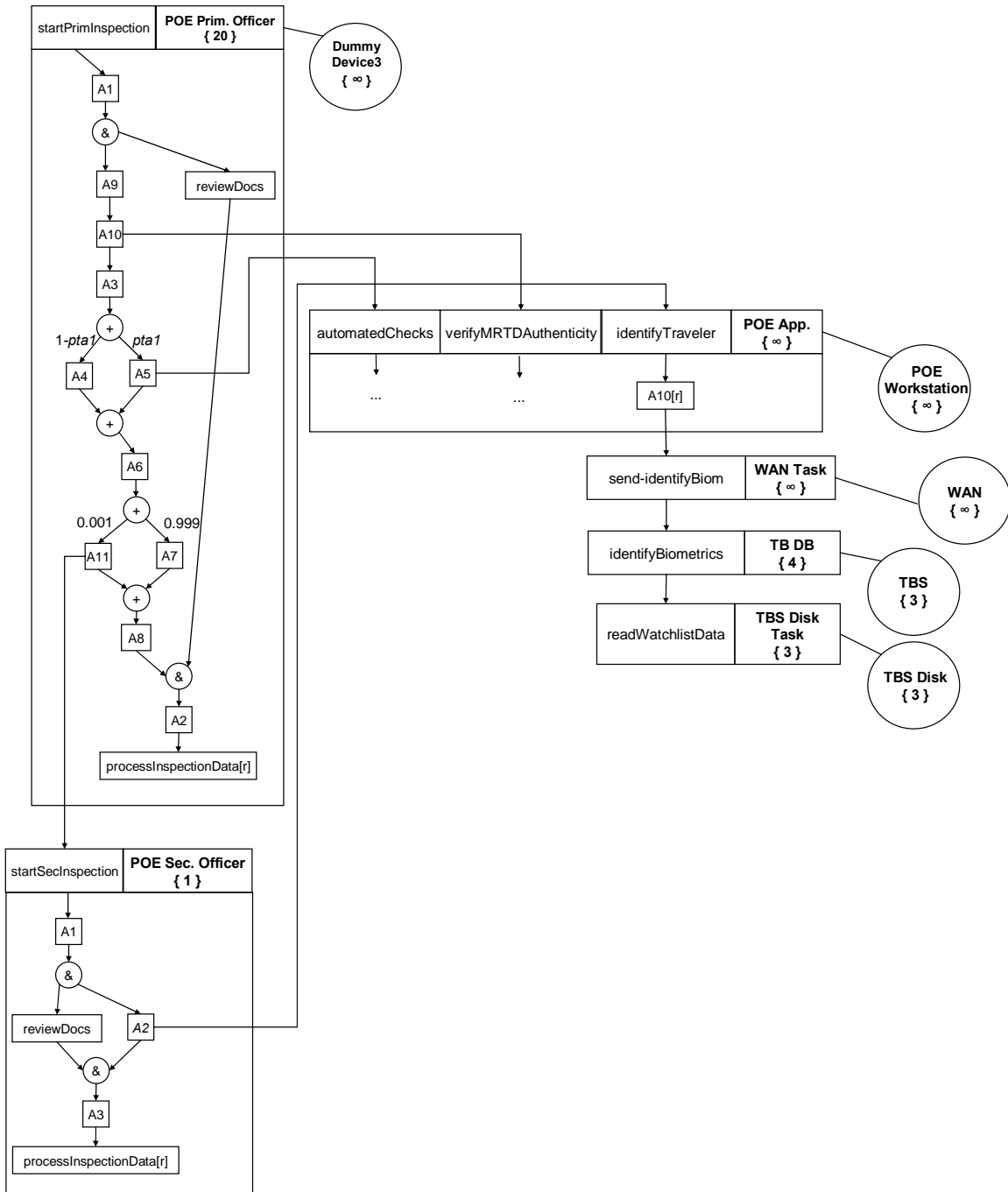


Figure 43: Primary Inspection LQN after Secondary Inspection

Figure 44 shows the final high-level layout of the *Traveler Inspection LQN*. For the sake of clarity we only represent LQN tasks, entries and devices. LQN activities within tasks are assumed to be the same as those represented in Figures 39 through 42. We represent a service request from an activity connected to a certain entry toward another entry, as a service request from the entry itself toward the destination entry. For instance, the service request from activity *A7* of the *startPrimInspection* entry of the task *POE Prim. Officer* toward the *scanMRTD* entry of the *MRTD Reader Task* is displayed as a service request from the *startPrimInspection* entry toward the *scanMRTD* entry.

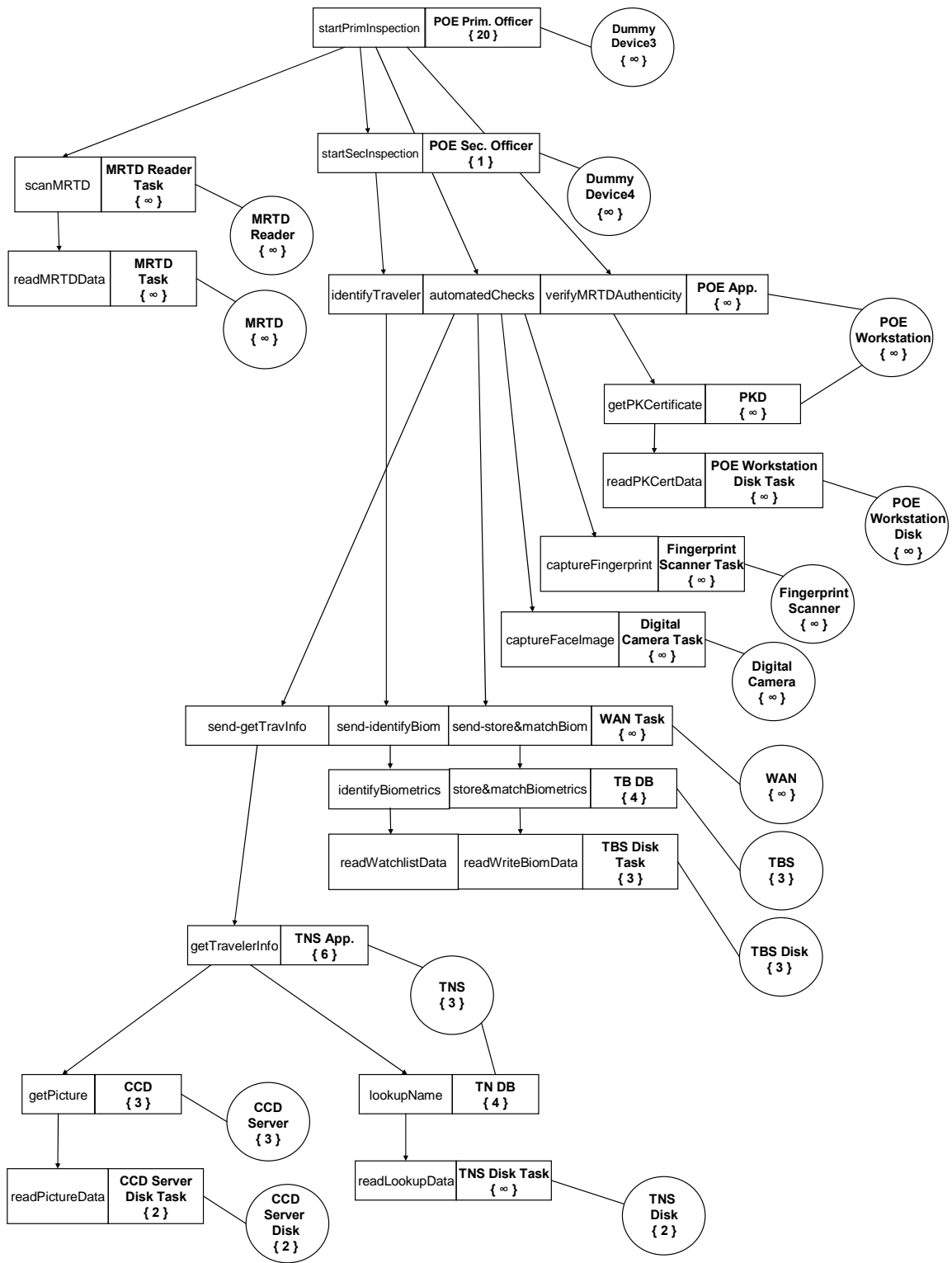


Figure 44: High-level layout of the *Traveler Inspection LQN*

Processing the Sequence Diagram for the Name-based Lookup use case generates the *Name-based Lookup LQN*, depicted in Figure 45. As we can notice, no new LQN entities were added to those generated by the processing of the Primary Inspection use case.

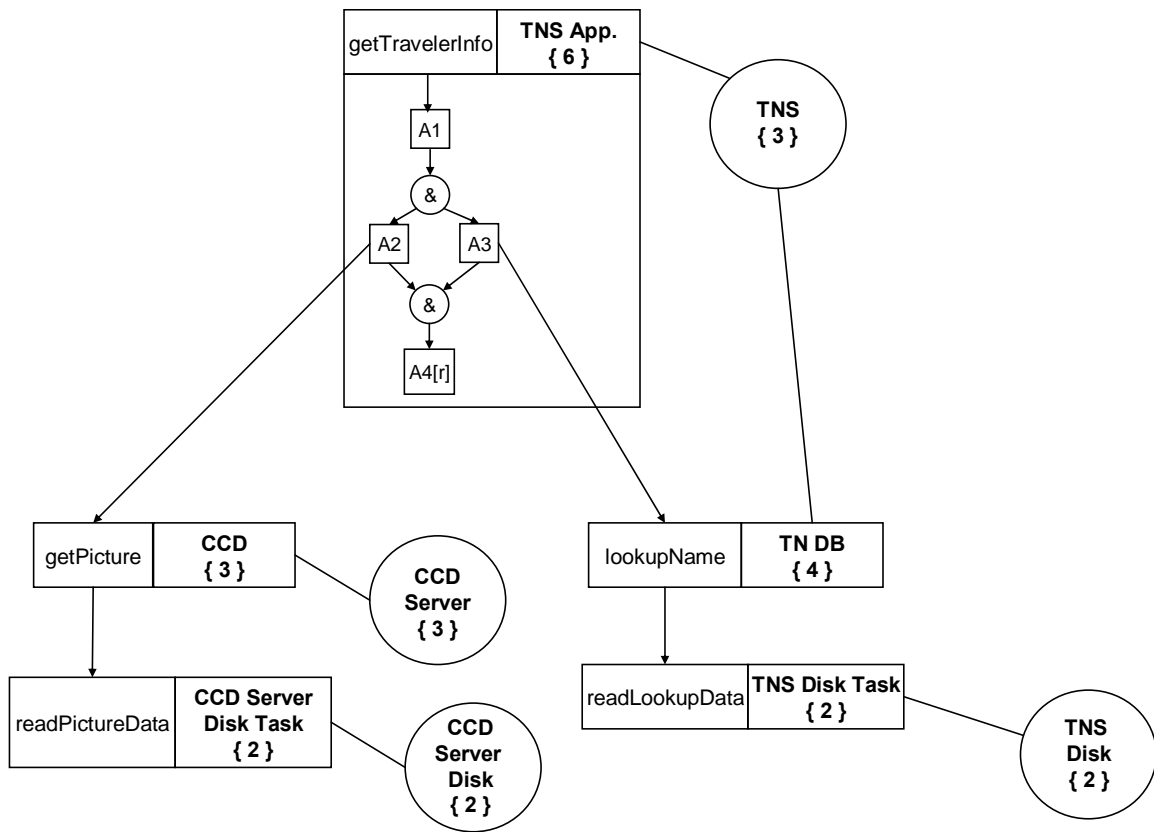


Figure 45: *Name-based Lookup LQN*

Processing the Sequence Diagram for the Biometric Verification use case only generates a service request toward the *identifyBiometrics* entry of the *TB DB* task. Similarly, processing the Biometric Identification use case generates a request toward the *store&matchBiometrics* entry of the same task.

The final LQN model for the airport inspection system is represented in Figure 46 and is obtained by merging the LQN submodels obtained by processing each performance scenario into a single LQN model. This is performed by starting with the high-level

model framework (Figure 38) and processing the LQN submodels obtained for the LQN black-box in the framework, one at a time. Each LQN submodel augments the current LQN model with devices, tasks, entries, activities, and request flow. However only LQN entities that are not already in the current model are added to it.

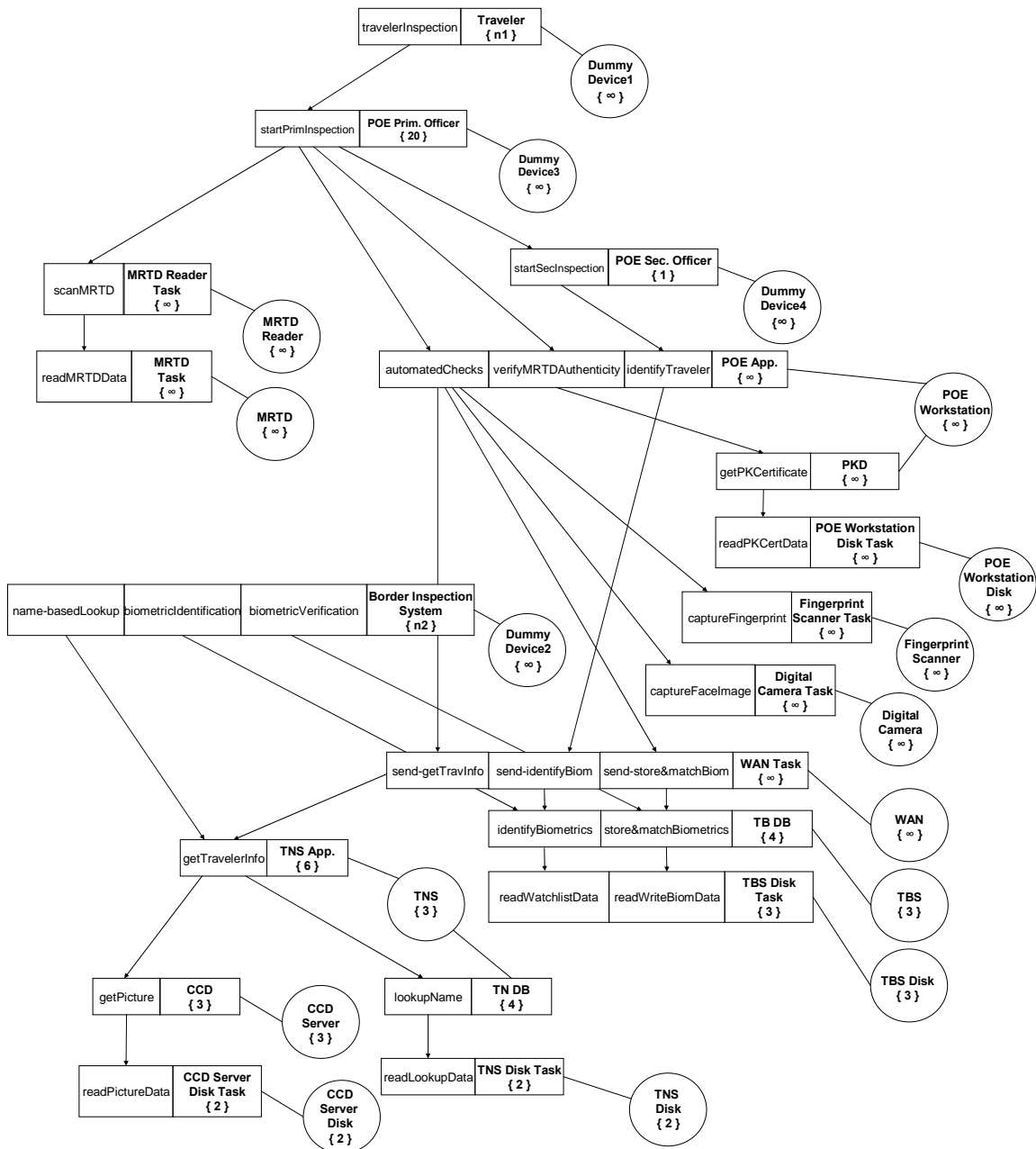


Figure 46: High-level layout of the LQN for the airport inspection system



## 4.2.4 Model Parameters

We derive parameter information for our LQN model partly from available technical reports for similar systems (e.g., [28, 29, 48]), partly from our estimates or assumptions.

We estimate the service time of *human components* (e.g., inspection officers) of the inspection system by guessing the amount of time they take to perform operations such as reviewing travelers' documents, processing data gathered from the TNS and the TBS, etc. The service time required by *processing devices* is very difficult to estimate since at this stage we only have very coarse-grained information about system operations and their complexities. For this reason, we usually assume the processing time needed by processing devices to carry out different tasks. Finally, we estimate the time taken by *I/O devices* based on the type of device. For instance, we estimate file I/O time as the ratio between the size of the data to be transferred and the throughput of the device storing the data. On the other hand, we estimate network I/O as the ratio between the size of the exchanged data and the throughput of the network link used for data communication.

Appendix A reports details on how we annotated performance scenarios for the inspection system with resource demands of scenario steps. It also explains how these values are used to derive parameters for the LQN model of the system.

## 4.3 Performance Experiments

We defined several performance experiments on the LQN models for the airport inspection system configurations described in this chapter and in Appendix B. The goal of the experiments is to evaluate the performance effects of the technical and policy options for the system described in Section 4.1.4. Each experiment selects one or more model parameters as independent variables of the analysis and establishes a set of possible values for each of them. Execution of the experiment returns a set of

performance results by solving the non-parameterized LQN models obtained varying the independent variables through their ranges.

All our experiments assume a constant population size representing the load on the airport inspection system at a given time. We vary the population size from 100 to 2000 to evaluate system performance for different workload intensities, such as peak hour, average hour, off hour, and so on. Response time is a very important performance measure for our system; therefore for each experiment we plot response time against traveler population.

### 4.3.1 Technical Options

The goal of this experiment is to evaluate the technical design options for the system described in Section 4.1.4. The alternatives in exam consider several possible locations for the Public Key Certificates of authorities issuing MRTDs, i.e., each MRTD, a database for each POE Workstation, or a database shared by multiple POE Workstations. We want to select the option that provides the best performance. Below we report parameter values for each option:

#### **MRTD:**

This option results in the following LQN model parameters, defined in Appendix A:

- *readMRTDData*: 1.0831s
- *scanMRTD*: 0.2708s
- *verifyMRTDAuthenticity*: 0.0093s
- *readPKCertData*: 0.0065s
- *verify*: 0.0044s

#### **PKD Local:**

This option results in the following LQN model parameters, defined in Appendix A:

- *readMRTDData*: 0.9472s
- *scanMRTD*: 0.2368s
- *verifyMRTDAuthenticity*: 0.0082s

- *readPKCertData*: 0.0065s
- *verify*: 0.0044s

### **PKD Remote:**

This option results in the following LQN model parameters, defined in Appendix B:

- *readMRTDData*: 0.9472s
- *scanMRTD*: 0.2368s
- *verifyMRTDAuthenticity*: 0.0082s
- *send-getPKCert*: 0.0017s
- *readPKCertData*: 0.0065s
- *verify*: 0.0044s

Within this technical option we consider different sub-options, to express the intensity of the request load on the PKD, in the case where the PKD is not locally stored at each airport inspection point. We devise four different values for the size of the *PKI System* population. Each size corresponds to a certain number of airports, each with 20 airport inspection points, issuing MRTD request authentications to the PKD:

- 1) 20 (1 airport);
- 2) 800 (40 airport);
- 3) 1600 (80 airport);
- 4) 3200 (160 airport).

### **4.3.2 Authentication Policies**

This experiment intends to evaluate the policy options for the system described in Section 4.1.4. Different policies require different authentication procedures, optionally based on the traveler's nationality. We want to assess how each authentication policy affects the performance provided by the airport inspection point. Below we report parameter values for each option. *pta1* and *pta2* appear in the Traveler Authentication interaction occurrence. The former expresses the probability of executing name-based and biometric-based checks during Primary Inspection. The

latter corresponds to the probability of collecting traveler biometric samples and verifying them.

**Scenario 1:**

- $pta1=0$ ;
- $pta2=0$ ;

**Scenario 2:**

- $pta1=1$ ;
- $pta2=1$ ;

**Scenario 3:**

- $pta1=1$ ;

Within Scenario 3 we identify three values for the probability  $pta2$  of executing biometric-based checks during travelers' inspection:

- 1) 0.5;
- 2) 0.7;
- 3) 0.9.

### **4.3.3 Manual Inspection Times**

This experiment considers different values for the manual inspection time required by the primary inspection officer. The values we consider are:

- 0s
- 30s
- 60s.

These correspond to the following parameterization of LQN entries:

**Scenario 1:**

- $reviewDocs: 0s$

**Scenario 2:**

- $reviewDocs: 30s$

**Scenario 3:**

- *reviewDocs*: 60s

### 4.3.4 Biometric Sampling Times

This experiment considers different values for the time required to capture fingerprint scans and a face image of the traveler. The values we consider are:

- 10s
- 15s
- 20s.

These correspond to the following parameterization of LQN entries:

**Scenario 1:**

- *captureFingerprint*: 5s
- *captureFaceImage*: 5s

**Scenario 2:**

- *captureFingerprint*: 10s
- *captureFaceImage*: 5s

**Scenario 3:**

- *captureFingerprint*: 15s
- *captureFaceImage*: 5s

## 4.4 Results and Analysis

Analytical solutions for our LQN models were obtained using the *LQNS* and *MultiSRVN* applications [17, 19]. Our results give insights into the performance of the technical and policy options in exam, taking both the point of view of a traveler experiencing the authentication process and that of an officer executing the process. For each option we evaluate the average total waiting time for a traveler to complete the authentication process, from the moment he/she arrives to the inspection queue, to

the moment he/she is granted or denied entry into the country. We also evaluate the average inspection time, which is the time required for the manual (performed by the POE officer) and automated authentication processes to determine the admissibility of a traveler. Finally we obtain system throughput during a 12 hours period, which makes it possible to estimate whether the inspection system is able to match the expected volume of incoming travelers. We also analyze the utilization of software and hardware resources to identify possible software or hardware bottlenecks. In all cases we solve our models for a traveler population size varying from 100 to 2000. This allows us to explore system response in conditions of light to heavy traffic.

#### **4.4.1 Technical Options**

Figures 47 through 50 show results related the technical configurations for the inspection system described in Section 4.3.1.

At the very beginning project sponsors did not know if the system would exhibit an acceptable performance and if it would experience bottlenecks. Therefore, we built LQN models for a baseline configuration that uses a single copy of the PKD server. As it can be noticed from the diagram in Figure 47, the inspection time for Options 1, 2 and 3, where up to 40 airports refer to the same PKD, is about the same and in every case is about 40s. This time is due mostly to the manual inspection process performed by the POE officer, assumed to last for 30 seconds (exponentially distributed with mean 30s). The manual process happens in parallel with the automated inspection process, which in all cases completes in less than 20s. When the number of airports referring to the same PKD is 80 we start noticing an increase in the inspection time, due to a slower response from the PKD server that becomes overloaded. The performance issue becomes even more evident when the PKD server supports 160 airports. In this case the total inspection time almost doubles and most of it is spent in the automated inspection process.

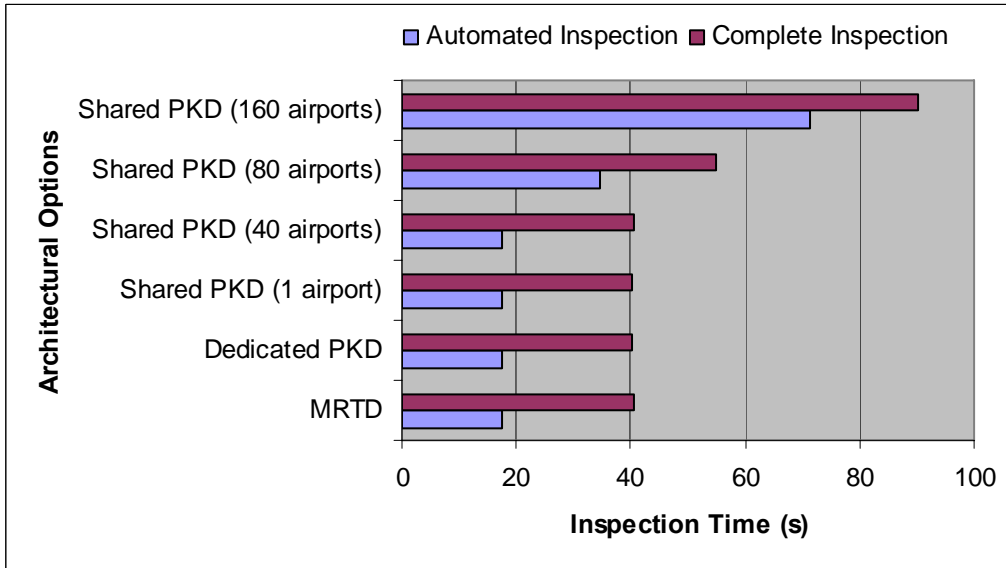


Figure 47: Primary inspection time for different technical options

Figure 48 shows the throughput provided by different system options during a 12 hours period.

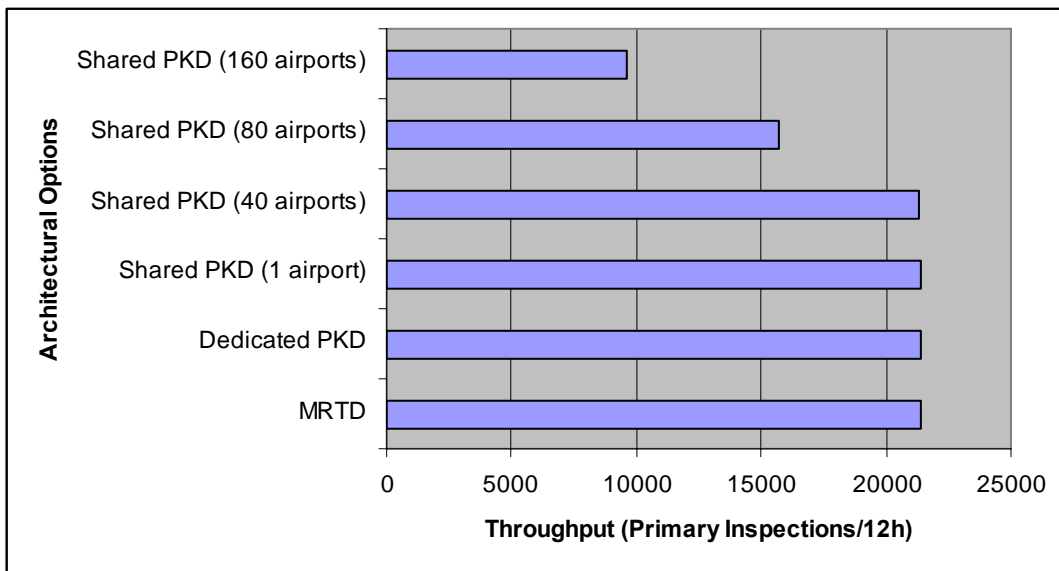


Figure 48: Primary inspection throughput for different technical options

Figure 49 shows the average total waiting time experienced by travelers at inspection facilities. For each technical option, the request load on the PKD is bounded. In fact, regardless of what the traveler population at the airports referring to the same PKD is, the maximum number of travelers that can be inspected in a certain moment is limited by the number of inspection facilities at those airports. This makes the average inspection time constant, i.e., not affected by the traveler population at POEs. Therefore, the total waiting time increases linearly with the traveler population. We notice that for Options 1, 2 and 3 (with up to 40 airports referring to the same PKD), the travelers' waiting time is about the same (the lines in Figure 49 overlap). In all cases the total waiting time is largely dominated by the travelers' queuing time due to the limited availability of POE border inspection points. We assume only 20 inspection facilities per airport, which quickly become over-saturated and lead to a system bottleneck as the traveler population increases. For Option 3, with 80 and 160 airports issuing requests to the same PKD, the total waiting time becomes sensibly larger. As we observed from Figure 47, this is due to request overloading on the PKD server, which causes a slower system response.



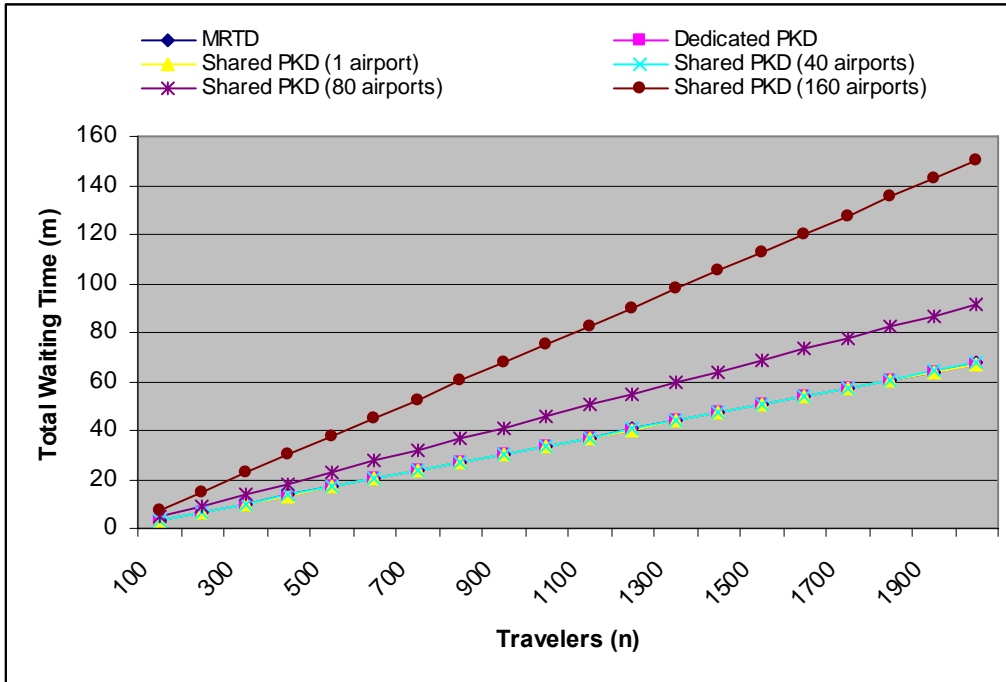


Figure 49: Primary total waiting time for different technical options

Figure 50 shows the average total waiting time for Option 3 as the load on the PKD server increases from 1 to 160 airports. We considered four possible traveler populations. From the diagram we can observe that, as the request load on the PKD increases, the system response increases non-linearly. The increase rate is higher as the traveler population increases.

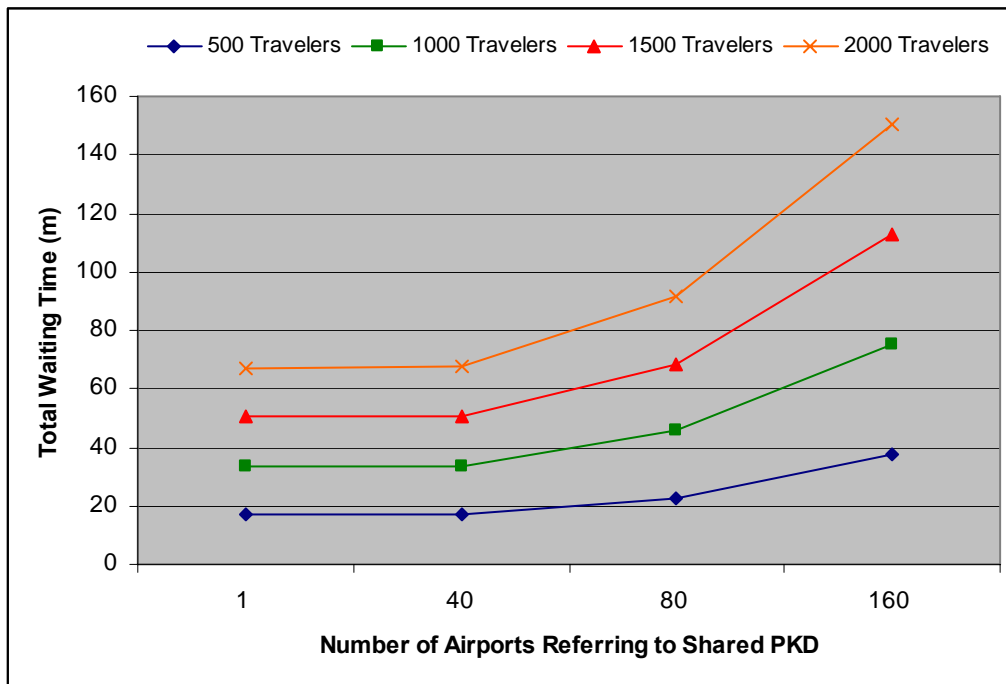


Figure 50: Primary total waiting time vs. airports served by a remote PKD

Table 2 shows detailed system response time and utilization for situations where 80 and 160 airports issue authentication requests to the PKD server. As we observed in Figures 47 through 50 in those cases system performance seems to be a problem. Therefore, we want to evaluate how introducing PKD replicas alleviates response time problem. In particular we are interested in identifying the optimal number of replicas to be introduced. Table 2 includes two sections, the first one reports results for 80 airports, the second one refers to 160 airports. Analyzing the first section, we can observe that when only one PKD exists the PKD server utilization reaches 99.95%, i.e., the PKD server becomes saturated leading to increased inspection and waiting times. On the other hand, the PKD processor utilization is quite low (43.45%), as it happens in the case of data intensive applications. As we introduce a PKD replica, utilization of the PKD server drops to 91.92%, PKD processor utilization drops to 39.96%, and PKD disk utilization is down to 51.95%. The system inspection time and waiting time reach their lower bound in with three PKD replicas.

In fact, introducing additional replicas only minimally decreases those parameters due to inherent inspection time delays. We can conclude that in the case of 80 airports referring to the same PKD, three is an optimal number for PKD replication. From the second section of the table we can observe that when only one PKD is present for 160 airports, the PKD server utilization is very high, 99.99%. When two or three replicas are present, utilization does not decrease sensibly. Introducing a fourth replica decreases the PKD server utilization to 91.91%. Five replicas lead to a PKD server utilization of 73.71%, while the PKD processor utilization becomes 32.05% and the PKD disk utilization is 41.66%. In this case, five is an optimal number of PKD replicas. In fact we observe that introducing additional replicas, while being more expensive, does not change system response in terms of inspection time and waiting time.

Based on the results discussed above we can conclude that the best configuration cannot be identified from a pure performance standpoint. Other factors have to be considered such as management issues for public key certificates and PKD replicas, choices or constraints emerging from system requirements and design, etc. Based on pure performance analysis alone none of the options give a performance that is appreciably better than the others. However, for the cases where 80 or 160 airports refer to the same PKD, the PKD has to be replicated, as described in Table 2. The reason is the bottleneck due to the limited availability of airport inspection points, assumed to be 20 per airport. System devices within a airport inspection point are dedicated and/or under-utilized and therefore highly responsive. Different technical options imply minimal variations in the system inspection time. These variations turn out to be an irrelevant component of the total average waiting time.

Table 2: Response time and resource utilization for *PKD Shared Option*

<b>80 Airports</b>					
# PKD	Insp. Time(s)	Wait. Time(m)	PKD Ut.	PKD Serv. Ut.	PKD Serv. Disk Ut.
1	54.8836	45.73733	0.999572	0.434597	0.564976
2	40.4675	33.7235	0.9192	0.399652	0.51955
3	40.4085	33.67433	0.616257	0.267938	0.34832
4	40.4061	33.67233	0.4623	0.201	0.2613
5	40.4057	33.67217	0.369852	0.160805	0.209046
<b>160 Airports</b>					
# PKD	Insp. Time(s)	Wait. Time(m)	PKD Ut.	PKD Serv. Ut.	PKD Serv. Disk Ut.
1	90.1481	4507.5	0.999919	0.565172	0.565172
2	54.7181	2735.96	0.99999	0.434782	0.565215
3	43.4452	2172.3	0.99997	0.43477	0.5652
4	40.4353	2021.8	0.919175	0.399643	0.519535
5	40.409	2020.49	0.737196	0.32052	0.416676
6	40.4064	2020.36	0.614483	0.347317	0.347317
7	40.4059	2020.33	0.526727	0.229011	0.297716
8	40.4057	2020.33	0.460893	0.200388	0.260505

#### 4.4.2 Authentication Policies

Figures 51 and 52 show how traveler waiting time and inspection time change as we differentiate the authentication procedure for country’s citizens and visiting aliens. We assume that all travelers undergo the inspection process from the baseline scenario, but country’s citizens are exempt from biometric data collection and verification.

We considered different values for the percentage of citizens and non citizens arriving at inspection points. In the baseline scenario all travelers undergo the same authentication procedure. Other values we considered are 50%, 70%, and 90%, corresponding to increasing rates of country’s citizen population. We can observe from the diagrams that modifying the authentication policy can reduce the average traveler waiting time, especially when a large traveler population is waiting to be authenticated.

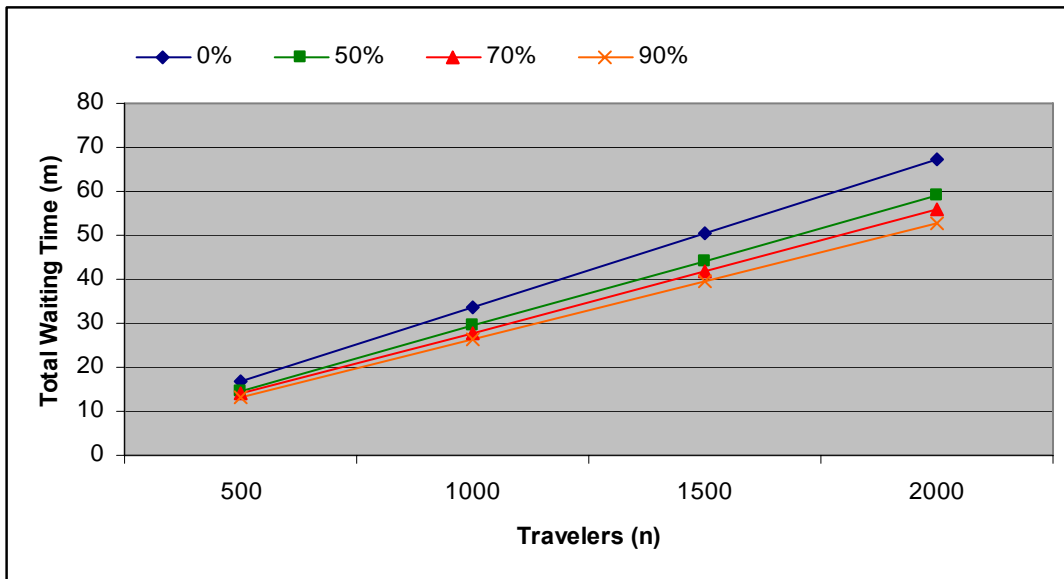


Figure 51: Primary total waiting time for different authentication scenarios

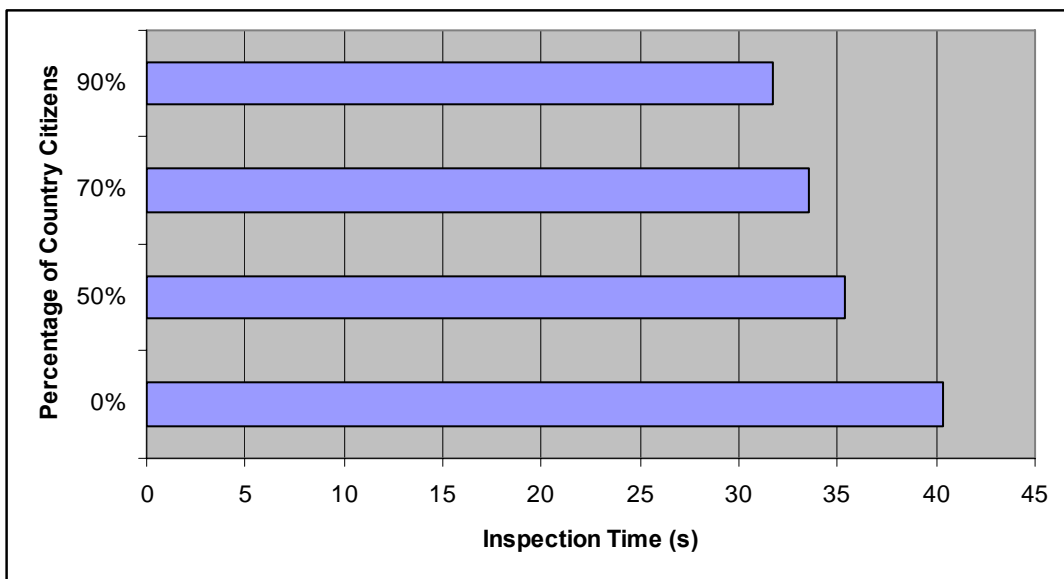


Figure 52: Primary inspection time for different authentication scenarios

### 4.4.3 Manual Inspection Times

Figures 53 and 54 display results for different requirements of the POE Officer inspection time. For the baseline authentication scenario we assumed a manual inspection time of 30s. Now we perform a sensitivity analysis on that parameter, giving it the values 0s, 30s, and 60s. These values may correspond to different authentication policies and requirements. A 0s inspection time corresponds to a totally

automated inspection process. A 30s and 60s inspection times correspond to increasing requirements for manual inspection time, which can be caused by the necessity to review less/more documents, to ask the traveler some questions. Figures 53 and 54 show that the passengers' waiting time and inspection time are more sensitive to changes in the officer inspection time, as opposed to changes in the biometric acquisition time, which we explore below. In fact, the difference between different options is significant.

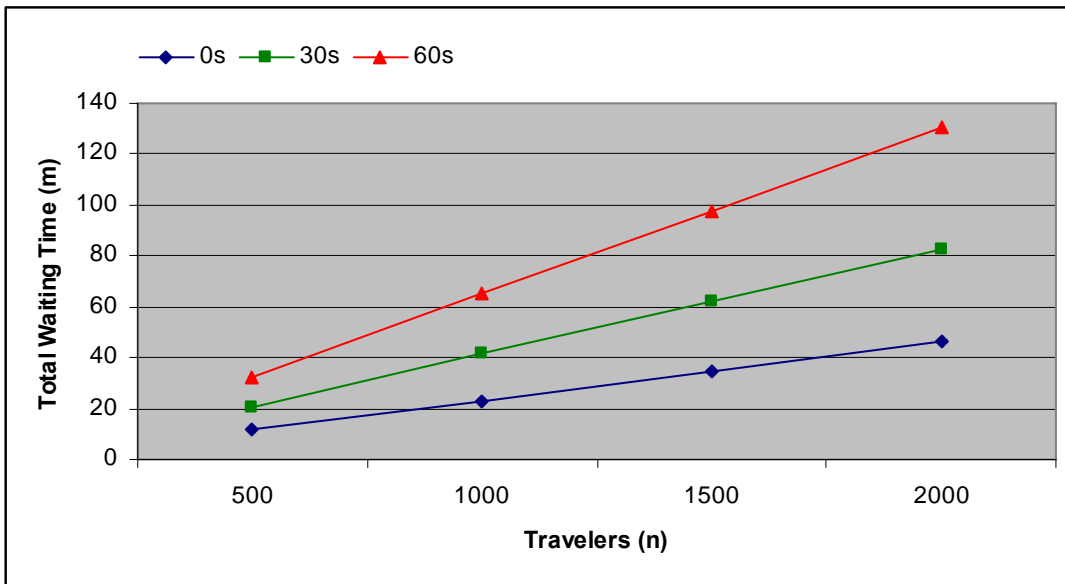


Figure 53: Primary total waiting time for different manual inspection times

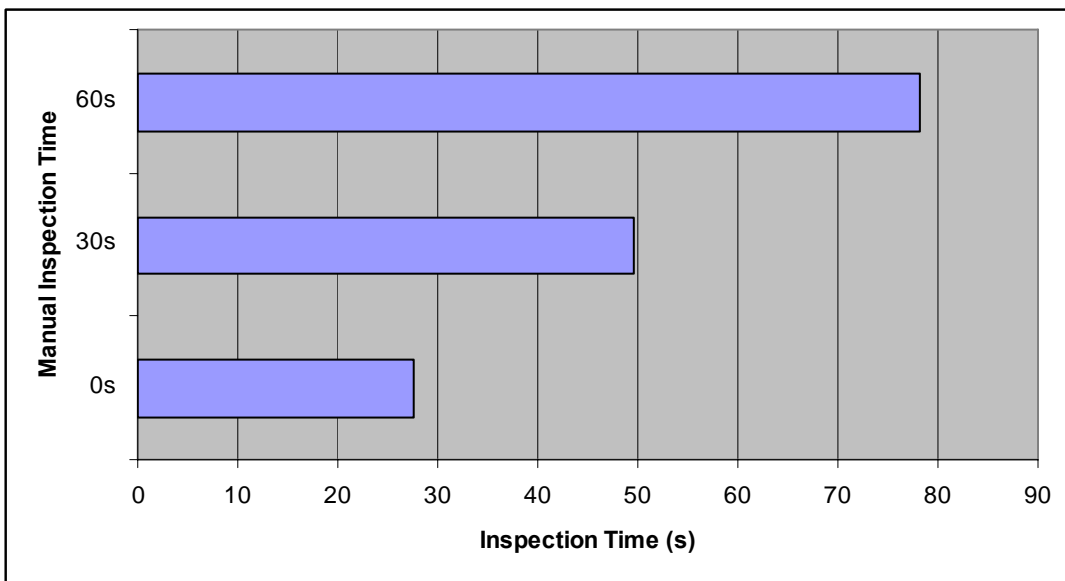


Figure 54: Primary inspection time for different manual inspection times

#### 4.4.4 Biometric Sampling Times

Figures 55 and 56 represent system performance as we vary the time to capture travelers' biometric samples. Fifteen seconds is the biometric acquisition time assumed for the baseline scenario and given by the sum of the time to acquire fingerprint and face image data. We evaluate how system waiting time and throughput change as we consider a shorter acquisition time of 10s and a longer acquisition time

of 20s. These different times may correspond to requests for a lower/higher sample quality, to the adoption of sampling devices with increased/decreased performance, or to a requirement of an increased/decreased number of samples. The diagram in Figure 55 shows the difference in average travelers' waiting time for the given biometric capture time options. The difference increases as the traveler population increases. However, in all cases for adjacent options, as long as the population is less than 2000, it is less than 5 minutes. Figure 56 displays the average inspection time for different biometric sampling times. As it is natural, the inspection time increases as the biometric collection time increases.

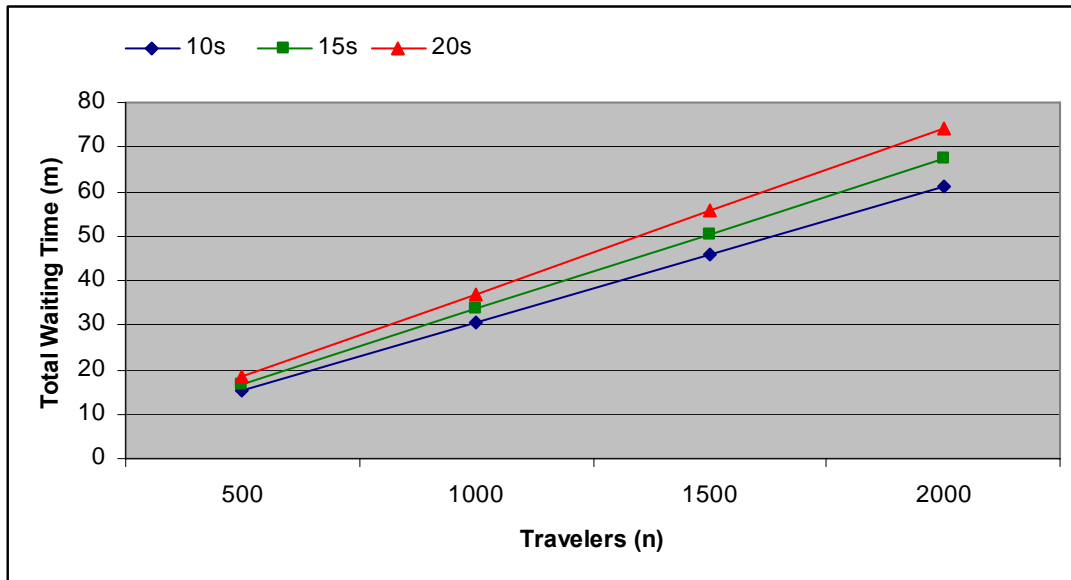


Figure 55: Primary total waiting time for different biometric sampling times



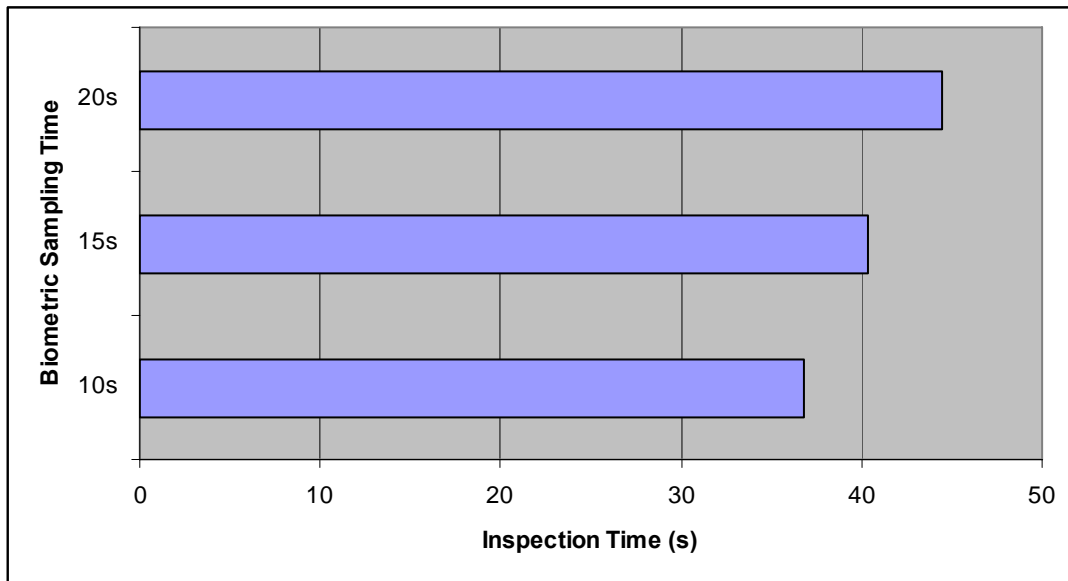


Figure 56: Primary inspection time for different biometric sampling times

## 4.5 Validation

Validation of performance results aims at checking whether the performance figures obtained by solving a performance model are close to those obtainable by observing the system in action. This task, in the absence of a system prototype or implementation, is a difficult matter. However, we were able to indirectly validate our performance results for an airport inspection system without having access to a real system. In fact, we validated our performance results against validated results coming from a simulation analysis of the inspections of international travelers at Los Angeles International Airport [15]. The simulation analysis is based on a discrete-event simulation model implemented using a commercial software package, Extend [33]. The model is quite complex and detailed, including approximately 400 modules from the Extend libraries. To obtain performance measures the simulation model needs to be run for 24 hours. To estimate the mean and variance of performance measures 10 simulation runs are required.

Based on the performance results produced by the simulation model, the average wait time for travelers going through primary inspection is  $43.2 \pm 5.4$ , and the

average queue length is  $1374 \pm 108$ . Incorporating this information on the diagram in Figure 49, which represents total wait time against the number of travelers waiting for primary inspection, we observe that our results are compatible with those obtained from the simulation study. In fact, in the cases where the PKD is not overloaded, i.e., all technical options except for *Shared PKD* with 80 and 160 airports referring to the same PKD, the waiting time returned by our LQN models is within the range returned by the simulation model.

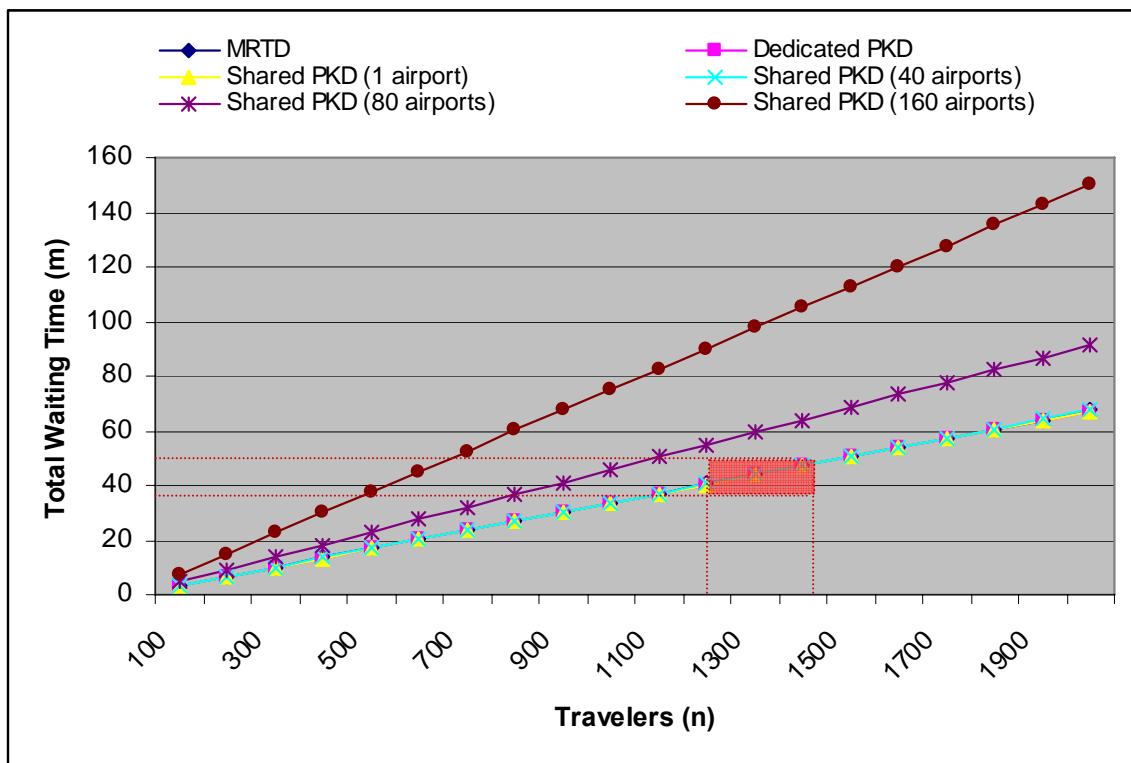


Figure 57: Validation of travelers' total waiting time

## Chapter 5: Conclusions

In this thesis we presented a methodology for modeling and evaluating the performance of software systems in the early stages of the software lifecycle.

We selected UML 2.0 as our notation for software specification. In particular, we adopted Use Case Diagrams, Deployment Diagrams, and Sequence Diagrams to model system users and functions, system hardware and software resources, and system dynamics, respectively. We annotated UML diagrams with quantitative performance-oriented information using standard extensions defined in the “UML Profile for Schedulability, Performance, and Time” [36]. We also introduced additional extensions to allow a more convenient specification of the system performance characteristics. Other extensions were proposed to address gaps in the current Performance Profile, which does not cover UML 2.0 diagrams.

The notation we selected for performance modeling is LQN. This choice was motivated by several factors. One of them is the suitability of LQN to express high-level software architecture abstractions, which makes it easier to define performance models and to trace back performance results into the original UML software specifications. Another factor is the ability of LQN to explicitly model software components and to express potentially complex operations performed by them. Additionally, LQN models are highly scalable, and efficient solution algorithms and tools are available for their evaluation.

We proposed a transformation methodology to automatically derive LQN models from annotated UML models. The transformation is largely inspired by earlier work presented in [20, 21, 39, 40, 41, 44]. However, our contribution is the adaptation of existing performance modeling techniques to a different set of UML diagrams. In particular, we adopted Sequence Diagrams instead of Activity Diagrams to express the dynamics of performance scenarios. We believe that Sequence Diagrams provide a better way to define a performance model for several reasons:

- with UML 2.0, they can represent very well complex system dynamics, including non-sequential flow of control,
- they naturally specify which system components are responsible for different operations,
- they are very good at expressing intercomponent communication.

We tested the applicability of the proposed transformation to the performance modeling and evaluation of a complex software system used at international airports within a country to grant or deny access to incoming travelers. The case study showed that our methodology is expressive and easy to apply. It is also modular; in fact, once we defined the high-level layout of the LQN model for the system, it was possible to separately process different performance scenarios, and then merge the corresponding LQN submodels to obtain the LQN model for the system. The LQN models obtained for different technical and policy options for the inspection system were easily solved by an analytical solver with a very limited resource usage on the host machine.

On the list of future work is the extension of our transformation to cover other architectural patterns besides the client/server one, and to address more features from UML 2.0 Sequence Diagrams. Another objective is the formalization of the additions and modifications suggested in the UML Performance Profile. We also find it desirable to develop an automated tool to implement our methodology, possibly

integrable with common practice development environments. This would allow to fill the gap between software development and performance analysis, and to integrate the validation of performance requirements in the software lifecycle. Finally, while our experience with the case study is positive, a validation of the methodology on additional and more complex systems is desirable.

# **Appendix A: Parameterization of LQN Model for Options 1 and 2**

This appendix explains how we parameterized the LQN model obtained in Chapter 4 for two different design alternatives that can be adopted to build an airport inspection system. We introduced the alternatives in exam in Section 4.1.4, and we called them as Options 1 and 2, respectively.

The next sections specify the service rates we assume for the execution environment of the inspection system and for the expected size of the data exchanged during key system operations. This information is used to motivate the resource demands attached to steps of performance scenarios. Resource demands are used to derive parameters for the LQN model of the system.

## **A.1 Assumed Execution Environment**

Table 3 summarizes the basic characteristics of the platform configuration we assume for the airport inspection system, including service rates of hardware devices and links between them. Parameters we do not explicitly use to estimate resource demands for system operations are left unspecified. Example of such parameters are the CPU rate and RAM of processing devices other than POE Workstation, or the throughput of the fingerprint scanner and of digital camera.

Table 3: Execution environment (Options 1 and 2)

<b>POE Workstation</b>	<i>CPU:</i> Pentium 2.40 GHz	
	<i>RAM:</i> 512 MB	
	<i>Disk:</i>	Command Overhead: 1 ms
		Access Time: 3.5 ms
		Latency: 2 ms
Transfer Time: 75 MB/s		
<b>MRTD Reader</b>	<i>Reading Rate:</i> 424 kilobits/s	
<b>MRTD Card</b>	<i>Data Transfer Rate:</i> 106 kilobits/s	
<b>Fingerprint Scanner</b>	Not specified	
<b>Digital Camera</b>	Not specified	
<b>LAN</b>	<i>Bandwidth:</i> 100 Mbits/s	
<b>WAN</b>	<i>Bandwidth:</i> 16.6 Mbits/s (avg)	
<b>TNS</b>	<i>CPU:</i> Not specified	
	<i>RAM:</i> Not specified	
	<i>Disk:</i>	Command Overhead: 1 ms
		Access Time: 2.93 ms
		Latency: 2 ms
Transfer Time: 85 MB/s		
<b>TBS</b>	<i>CPU:</i> Not specified	
	<i>RAM:</i> Not specified	
	<i>Disk:</i>	Command Overhead: 1 ms
		Access Time: 2.93 ms
		Latency: 2 ms
Transfer Time: 85 MB/s		
<b>CCD Server</b>	<i>CPU:</i> Not specified	
	<i>RAM:</i> Not specified	
	<i>Disk:</i>	Command Overhead: 1 ms
		Access Time: 2.93 ms
		Latency: 2 ms
		Transfer Time: 85 MB/s

## A.2 Expected Size of Data

Table 4 lists the expected size of the data exchanged within the airport inspection system during key system operations, such as MRTD authentication, collection of name-based lookup information, etc. This information was mostly gathered from technical reports available for similar systems (e.g., [28, 29]).

Table 4: Expected size of data (Options 1 and 2)

<b>MRTD Data</b>	<i>MRZ</i> : 88 bytes	
	<i>Picture</i> : 12704 bytes	
	<i>DS</i> : 20 bytes	
	<i>Public Key Certificate</i> : 1.8 KB	
	<i>Total Size</i>	(without Public Key Certificate): 12852 bytes (with Public Key Certificate): 14695.2 bytes
<b>TBS Data</b>	<i>Fingerprint scans</i>	10 KB
	<i>Face Image</i>	20 KB
	<i>Watchlist Size</i>	1000 Face images
<b>TNS Data</b>	5 KB	
<b>CCD Data</b>	20 KB	

## A.3 Performance Annotations

Table 5 specifies the performance annotations we assume to be attached to the scenario steps represented in the set of Sequence Diagrams for the system. For each step we report the parameter associated with the `Pademand` tag, and optionally with the `PextOp` tag. We state the type of the parameter (i.e., required, assumed, estimated, measured), its numeric value, and most of the times a rationale for it. A *par* value indicates that the exact resource demand for the corresponding scenario step is dependent on which design alternative is assumed for the system (i.e., Option 1 or 2).



We assume all service demands to have exponential distributions, with mean values equal to the specified values. We also assume that a system component always performs the same operation with the same service demand.

As explained in Section 4.2.4, we estimate the I/O time required by system operations as the ratio between the data to be transferred and the throughput of the involved I/O device. On the other hand, in the absence of more information, we assume that the processing time needed to perform system operations is 0.005s. The only exception to this assumption is represented by the time taken by cryptographic operations performed on the POE Workstation to verify the validity of a MRTD. In fact, the performance of these functions was benchmarked on a machine with the same configuration as the POE Workstation using *openssl*, an application available with the *OpenSSL* libraries. The *speed* option of the *openssl* binary returns performance results for a wide range of cryptographic algorithms, including SHA-1 and RSA. For the SHA-1 algorithm, it returns the number of bytes that can be processed per second. For the RSA algorithm it returns the times needed by sign/verify cycles for different values of key length. We used this information to estimate the processing time required by SHA as a function of the amount of data to be processed. The processing time required by RSA was instead estimated as a function of the length of the used key.

Table 5: Resource demand of scenario steps (Options 1 and 2)

Scenario Step	Tag	Source	Value (s)	Rationale
<b>Scenario: Primary Inspection</b>				
<i>startPrimInspection</i>	PAdemand	asmd	0.0	(1)
<i>reviewDocs</i>	PAdemand	asmd	20.0	(2)
<i>processInspectionData</i>	PAdemand	asmd	5.0	(3)
<i>return inspectionResult</i>	PAdemand	asmd	5.0	(4)
<i>automatedChecks</i>	PAdemand	asmd	0.005	
<i>captureFingerprint</i>	PAdemand	asmd	0.005	
<i>return fingerprintData</i>	PAdemand	pred	10.0	(5)

<i>capture faceImage</i>	PAdemand	asmd	0.005	
<i>return faceImageData</i>	PAdemand	asmd	5.0	(6)
<i>store&amp;matchBiometrics</i>	PAdemand	asmd	0.005	
	PAextOp ( <i>send-store&amp;matchBiom</i> )	pred	0.0142	(7)
	PAextOp ( <i>readWriteBiomData</i> )	pred	0.0064	(8)
<i>return biometricMatchResult</i>	PAdemand	asmd	0.005	
<i>processData</i>	PAdemand	asmd	0.005	
<i>return checksResult</i>	PAdemand	asmd	0.005	
<i>scanMRTD</i>	PAdemand	asmd	1.0	(9)
<i>return MRTDData</i>	PAdemand	pred	par	(10)
	PAextOp ( <i>readMRTDData</i> )	pred	par	(11)
<i>verifyMRTDAuthenticity</i>	PAdemand	pred	par	(12)
<i>getPKCertificate</i>	PAdemand	asmd	0.005	
<i>return PKCertificate</i>	PAdemand	asmd	0.005	
	PAextOp ( <i>readPKCertData</i> )	pred	par	(13)
<i>verify(MRTD_DS)</i>	PAdemand	pred	par	(14)
<i>verify(CAPKCertificate)</i>	PAdemand	pred	0.0015	(15)
<i>verify(MRZ_DS)</i>	PAdemand	pred	0.0009	(16)
<i>verify(faceImage_DS)</i>	PAdemand	pred	0.0006	(17)
<i>return MRTDAuthenticity</i>	PAdemand	asmd	0.005	
<i>getTravelerInfo</i>	PAdemand	asmd	0.005	
	PAextOp ( <i>send-getTravInfo</i> )	pred	0.0118	(18)
<i>lookupName</i>	PAdemand	asmd	0.005	
<i>return lookupName</i>	PAdemand	asmd	0.005	
	PAextOp ( <i>readLookupData</i> )	pred	0.006	(19)
<i>getPicture</i>	PAdemand	asmd	0.005	
<i>return picture</i>	PAdemand	asmd	0.005	
	PAextOp ( <i>readPictureData</i> )	pred	0.0062	(20)
<i>return lookupName, picture</i>	PAdemand	asmd	0.005	
<i>startSecInspection</i>	PAdemand	asmd	0.0	(21)
<i>identifyTraveler</i>	PAdemand	asmd	5.0	(22)
	PAextOp ( <i>send-identifyBiom</i> )	pred	0.0094	(23)
<i>return identificationResult</i>	PAdemand	asmd	0.5	(24)
	PAextOp ( <i>readWatchlistData</i> )	pred	0.2357	(25)
<i>identifyBiometrics</i>	PAdemand	asmd	0.005	
<i>return identificationResult</i>	PAdemand	asmd	0.005	
<i>reviewDocs</i>	PAdemand	asmd	300.0	(26)
<i>processInspectionData</i>	PAdemand	asmd	10.0	(27)
<i>return inspectionResult</i>	PAdemand	asmd	3.0	(28)
<b>Scenario: Name-based Lookup</b>				
<i>getTravelerInfo</i>	PAdemand	asmd	0.0	(29)

<i>lookupName</i>	PAdemand	asmd	0.005	
	PAextOp ( <i>readLookupData</i> )	pred	0.006	(30)
<i>return lookupName</i>	PAdemand	asmd	0.005	
<i>getPicture</i>	PAdemand	asmd	0.005	
	PAextOp ( <i>readPictureData</i> )	pred	0.0062	(31)
<i>return picture</i>	PAdemand	asmd	0.005	
<i>return lookupName, picture</i>	PAdemand	asmd	0.005	
<b>Scenario: Biometric Verification</b>				
<i>store&amp;matchBiometrics</i>	PAdemand	asmd	0.0	(32)
<i>return biometricMatchResult</i>	PAdemand	asmd	0.005	
<b>Scenario: Biometric Identification</b>				
<i>identifyBiometrics</i>	PAdemand	asmd	0.0	(33)
<i>return biometricIdentificationResult</i>	PAdemand	asmd	0.005	

**Rationale for resource demand values:**

- (1) It is the time required by the traveler to generate a request for primary inspection. We assume this time to be null.
- (2) It is the time required by the primary inspection officer to interview the traveler and to review his/her documents. We assume this time to be 20s.
- (3) It is the time required by the primary inspection officer to decide if authorizing the traveler to enter the country based on the outcome of manual and automated checks. We assume this time to be 5s.
- (4) It is the time required by the primary inspection officer to communicate to the traveler the outcome of the inspection process. We assume this time to be 5s.
- (5) It is the time required to capture fingerprint scans of the traveler. We assume this time to be 10s.
- (6) It is the time required to take a picture of the traveler using a digital camera. We assume this time to be 5s.
- (7) It is the time required to send the biometric data collected from the traveler (fingerprint scans plus face image) to the TBS, through the WAN connecting the POE Workstation to that server. We compute the average data transfer time as:  
 $30\text{KB} / 16.6 \text{ Megabits/s} = 0.0142\text{s}$

- (8) It is the time required to write the biometric data collected from the traveler (fingerprint scans plus face image) to the TBS disk, and to read a previously stored face image file (20 KB) from it. We compute the average data transfer time as:  
 $1 \text{ ms} + 2.93 \text{ ms} + 2 \text{ ms} + (50 \text{ KB} / 85\text{MB/s}) = 0.0064\text{s}$
- (9) It is the time required by the primary inspection officer to swipe the MRTD through the MRTD Reader. We assume this time to be 1s.
- (10) It is the time required by the MRTD Reader to read the data stored in the MRTD. The actual duration of the operation depends on the size of the MRTD data, which in turn depends on the technical configuration assumed for the system. This leads to the following values for Options 1 and 2:  
*Option 1:*  $15695.2 \text{ bytes} / 424 \text{ kilobits/s} = 0.2708\text{s}$   
*Option 2:*  $12852 \text{ bytes} / 424 \text{ kilobits/s} = 0.2368\text{s}$
- (11) It is the time required by the MRTD to transfer its data to the MRTD Reader. The actual duration of the operation depends on the size of the MRTD data, which in turn depends on the technical configuration assumed for the system. This leads to the following values for Options 1 and 2:  
*Option 1:*  $15695.2 \text{ bytes} / 106 \text{ kilobits/s} = 1.0831\text{s}$   
*Option 2:*  $12852 \text{ bytes} / 106 \text{ kilobits/s} = 0.9472\text{s}$
- (12) It is the time required to transfer the MRTD data from the MRTD Reader to the POE Workstation through a 12 Mbits/s USB link. The actual transfer time depends on the size of the MRTD data, which in turn depends on the technical configuration assumed for the system. This leads to the following values for Options 1 and 2:  
*Option 1:*  $15695.2 \text{ bytes} / 12 \text{ MB/s} = 0.0093\text{s}$   
*Option 2:*  $12852 \text{ bytes} / 12 \text{ MB/s} = 0.0082\text{s}$
- (13) It is the time required to read Public Key Certificates from the Disk of the POE Workstation. The actual reading time depends on the number of certificates to be read (one or two), which in turn depends on the technical configuration assumed for the system. This leads to the following values for Options 1 and 2:  
*Option 1:*  $1 \text{ ms} + 3.5 \text{ ms} + 2 \text{ ms} + (1.8 \text{ KB} / 75\text{MB/s}) = 0.0065\text{s}$   
*Option 2:*  $1 \text{ ms} + 3.5 \text{ ms} + 2 \text{ ms} + (3.6 \text{ KB} / 75\text{MB/s}) = 0.0065\text{s}$

- (14) It is the time required to verify the authenticity of the DS on the MRTD. This requires to compute a hash function (SHA-1) of the MRTD data, and eventually to verify the authenticity of the DS by applying the RSA algorithm using the Public Key of the MRTD signer (2048 bits) [28, 29]. The time to perform the operation depends on the amount of data stored in the MRTD, which in turn depends on the technical configuration assumed for the system. This leads to the following values for Options 1 and 2:

$$\text{Option 1: } t[\text{SHA\_1}(14695.2 \text{ bytes})] + \\ t[\text{RSA}(2048 \text{ bits})\text{-verify}(20\text{bytes})] = 0.0091\text{s}$$

$$\text{Option 2: } t[\text{SHA\_1}(12852\text{bytes})] + \\ t[\text{RSA}(2048 \text{ bits})\text{-verify}(20\text{bytes})] = 0.0091\text{s}$$

- (15) It is the time required to verify the authenticity of the DS on the Public Key Certificate of the MRTD issuer. This requires to compute a hash function (SHA-1) of the certificate data itself, and eventually to verify the authenticity of its DS by applying RSA using the Public Key of the Country CA (3072 bits) [28, 29]. The time to perform the operation can be estimated as:

$$t[\text{SHA\_1}(1.8 \text{ KB})] + \\ t[\text{RSA}(3072 \text{ bits})\text{-verify}(20\text{bytes})] = 0.0015\text{s}$$

- (16) It is the time required to verify the authenticity of the MRZ portion of the MRTD. This requires to compute a hash function (SHA-1) of the MRZ data, and eventually verify the authenticity of the DS on the MRZ by applying RSA with the Public Key of the document signer (2048 bits) [28, 29]. The time to perform the operation can be estimated as:

$$t[\text{SHA\_1}(88 \text{ bytes})] + \\ t[\text{RSA}(2048 \text{ bits})\text{-verify}(20\text{bytes})] = 0.0009\text{s}$$

- (17) It is the time required to verify the authenticity of the face image portion of the MRTD. This requires to compute a hash function (SHA-1) of the image data, and eventually to verify the authenticity of the DS on the face image by applying RSA with the Public Key of the document signer (2048 bits) [28, 29]. The time to perform the operation can be estimated as:

$$t[\text{SHA\_1}(12704 \text{ bytes})] + \\ t[\text{RSA}(2048 \text{ bits})\text{-verify}(20\text{bytes})] = 0.001\text{s}$$

- (18) It is the time required to exchange the TNS traveler's biographic and lookup information and a picture of him/her. We assume an average size of 5 KB for the biographic and lookup data, and an average size of 20 KB for the picture. This leads to an average data transfer time of:  
 $25\text{KB} / 16.6 \text{ Megabits/s} = 0.0118\text{s}$
- (19) It is the time required to retrieve biographic and lookup data for the traveler from the TNS disk. We compute the average data transfer time as:  
 $1 \text{ ms} + 2.93 \text{ ms} + 2 \text{ ms} + (5 \text{ KB} / 85\text{MB/s}) = 0.006\text{s}$
- (20) It is the time required to retrieve a traveler's picture from the CCD server disk. We compute the average data transfer time as:  
 $1 \text{ ms} + 2.93 \text{ ms} + 2 \text{ ms} + (20 \text{ KB} / 85\text{MB/s}) = 0.0062\text{s}$
- (21) It is the time required by the POE Primary Officer to generate a request for secondary inspection. We assume this time to be null.
- (22) It is the time required by the secondary inspection officer to start an identification process. We assume this time to be 5s.
- (23) It is the time required to send a face image of the traveler to the TBS. We compute the average data transfer time as:  
 $20\text{KB} / 16.6 \text{ Megabits/s} = 0.0094\text{s}$
- (24) It is the time required by the TBS to match the traveler's face image with the set of 1000 face images in the biometric watchlist. We assume this time to be 0.5s
- (25) It is the time required by the TBS disk to read the set of 1000 face image templates in the biometric watchlist. The actual time required to perform the computation depends on the size of the watchlist. We compute the average data transfer time as:  
 $1 \text{ ms} + 2.93 \text{ ms} + 2 \text{ ms} + (1000 \times 20 \text{ KB} / 85\text{MB/s}) = 0.2357\text{s}$
- (26) It is the time required by a secondary inspection officer to thoroughly review the traveler's documents and belongings and to question him/her. We assume this operation to last 5 minutes.
- (27) It is the time required by the secondary inspection officer to decide if authorizing the traveler to enter the country based on the outcome of manual and automated checks. We assume this time to be 10s.
- (28) It is the time required by the secondary inspection officer to communicate to the traveler the outcome of the inspection process. We assume this time to be 3s.

- (29) It is the time required by the border inspection system to generate a request for name-based lookup. We assume this time to be null.
- (30) As in (19)
- (31) As in (20)
- (32) It is the time required by the border inspection system to generate a request for biometric verification. We assume this time to be null.
- (33) It is the time required by the border inspection system to generate a request for biometric identification. We assume this time to be null.

## A.4 Model Parameters

Tables 6 and 7 define the parameterization of the LQN model for the airport inspection system obtained in Chapter 4. Table 6 parameterizes the reference tasks for the system based on the information attached to the Use Case Diagram in Figure 26. We leave the traveler population parametric; we defined a value for it in the experiment section of Chapter 4.

Table 6: LQN parameters for system workloads (Options 1 and 2)

Reference Task	Multiplicity	Entry	Think Time (s)
Traveler	$n1$	<i>travelerInspection</i>	4/1 = 4 s
Border Inspection System	3200	<i>name-basedLookup</i>	20/0.4995 = 40.04 s
		<i>biometricVerification</i>	20/0.4995 = 40.04 s
		<i>biometricIdentification</i>	20/0.001 = 20,000 s

Table 7 defines resource demands for entries and activities of non-reference tasks using the performance annotations specified in Table 5. We list our parameters in a tabular format, instead of displaying them with the graphical representation of the parameterized LQN, for the sake of readability. We assume that underlined elements in the table denote task activities.

Table 7: LQN parameters for resource demands (Options 1 and 2)

Task	Entry/Activity	Service Time (s)
POE Prim. Officer	startPrimInspection	5.0
	<u>reviewDocs</u>	20.
	<u>processInspectionData</u>	5.0
	<u>A5</u>	0.005
	<u>A9</u>	1.0
	<u>A10</u>	see rationale (12)
POE Sec. Officer	startSecInspection	3.0
	<u>reviewDocs</u>	300.0
	<u>processInspectionData</u>	10.0
	<u>A2</u>	5.0
POE App.	verifyMRTDAuthenticity	0.005
	automatedChecks	0.005
	identifyTraveler	0.005
	<u>processData</u>	0.005
	<u>Verify</u>	see rationale (14) through (17)
	<u>A5</u>	0.005
	<u>A6</u>	0.005
	<u>A7</u>	0.005
	<u>A8</u>	0.005
	<u>A9</u>	0.005
	<u>A10</u>	0.005
MRTD Reader Task	scanMRTD	see rationale (10)
MRTD Task	readMRTDData	see rationale (11)
PKD	getPKCertificate	0.005
Fingerprint Scanner Task	captureFingerprint	10.0
Digital Camera Task	captureFacelImage	5.0
TNS App.	getTravelerInfo	0.005
	<u>A2</u>	0.005
	<u>A3</u>	0.005
TN DB	lookupName	0.005
CCD	getPicture	0.005
TB DB	store&matchBiometrics	0.005
	identifyBiometrics	0.005
WAN Task	send-getTravInfo	0.011765813
	send-store&matchBiom	0.01418976
	send-identifyBiom	0.009412651
POE Workstation Disk Task	readPKCertData	see rationale (13)
TBS Disk Task	readWriteBiomData	0.006416791
	readWatchlistData	0.235709412
TNS Disk Task	readLookupData	0.005987445



CCD Server Disk Task	readPictureData	0.006159779
----------------------	-----------------	-------------

The only *probabilities* attached to performance scenarios through  $P_{\text{Aprob}}$  tagged values are *pta1* and *pta2*. They appear in the Traveler Authentication interaction occurrence. The former expresses the probability of executing name-based and biometric-based checks during Primary Inspection. The latter corresponds to the probability of collecting traveler biometric samples and verifying them. We leave both probabilities parametric; we defined a value for them in the experiment section of Chapter 4.

# Appendix B: Option 3

This appendix describes a design alternative that can be adopted for the airport inspection system. We introduced the alternative in exam in Section 4.1.4, and we called it as Option 3. The next sections derive its LQN performance model largely reusing the outcome of the application of our UML to LQN transformation to the design Options 1 and 2, which we described in Chapter 4.

## B.1 Description

The Deployment Diagram in Figure 58 represents the configuration of Option 3. We can observe that this is almost identical to the one represented in Figure 25 of Chapter 4. The only difference lies in the location of the PKD, which is remote, and not stored at the POE Workstation. The PKD is connected to the inspection system through a network link whose exact type and capability are dependent on the exact location of the PKD. This may be a POE, a regional, state, or national reference point, or combinations of them.

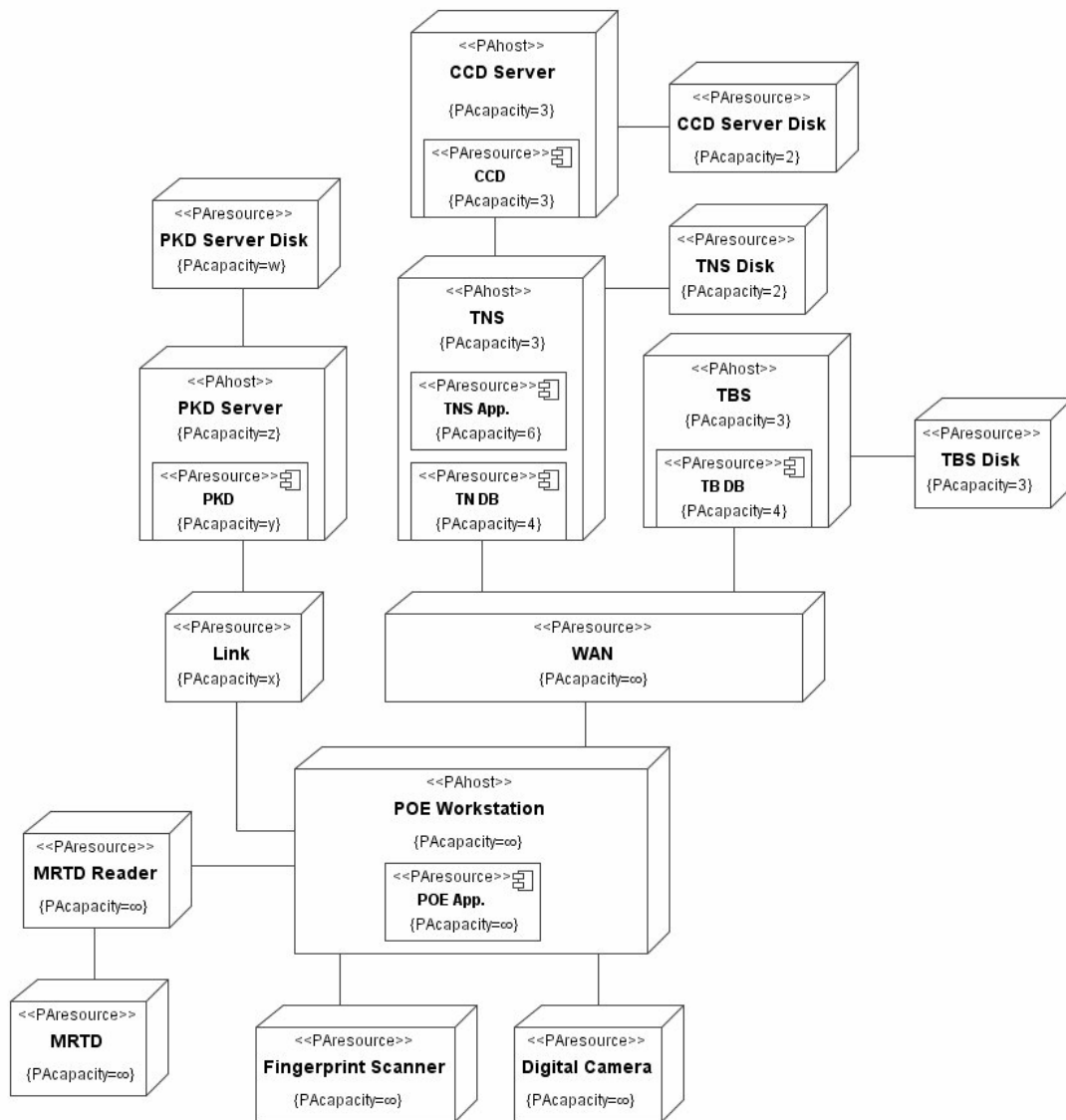


Figure 58: Deployment Diagram for Option 3

Figure 59 shows a Use Case Diagram for Option 3. The diagram displays the same users and functions represented in Figure 26 of Chapter 4, with the exception of the PKI System user. This corresponds to a system outside the scope of the inspection system, which issues requests for Public Key Certificates to the PKD. The population size for the workload generated by PKI System is left parametric. Its actual value determines a different load on the PKD and corresponds to a different location of it.

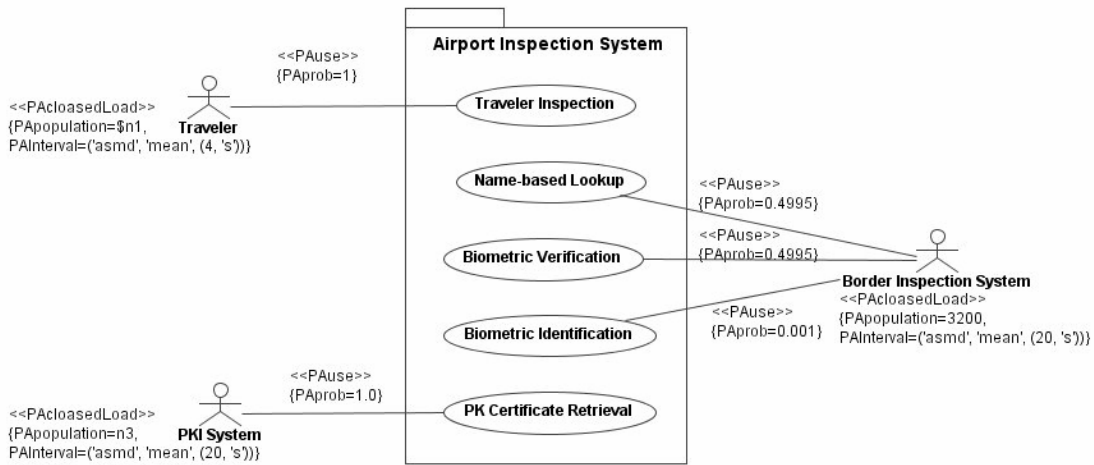


Figure 59: Use Case Diagram for Option 3

Figure 60 shows a Sequence Diagram for the PK Certificate Retrieval use case.

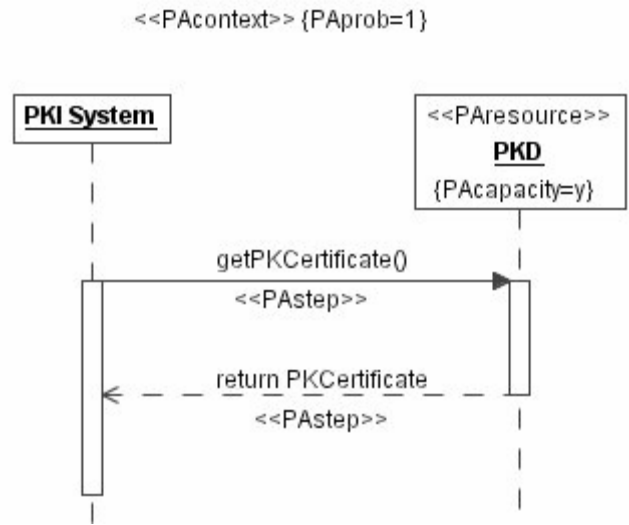


Figure 60: Sequence Diagram for the PK Certificate Retrieval use case

## B.2 Performance Modeling

In this section we describe the outcome of the application of our UML to LQN transformation to the configuration for the inspection system described in the previous section. Since the overall structure, dynamics, and parameterization of the resulting LQN are almost identical to the ones obtained Chapter 4, we omit to report the

complete output of every step of the transformation. Instead, we only describe the differences between the two models.

### **B.2.1 Model Structure**

The application of Step 1.a to the Deployment Diagram in Figure 58 generates the same LQN devices obtained in Chapter 4 for the other configurations of airport inspection system. However, compared to that set, the *POE Workstation Disk* device has been removed, since the deployment in exam does not use the Workstation to store any data, but just to access external information and then process it. The *Link* device has instead been added, together with the *PKD Server* and *PKD Server Disk* devices.

The application of Step 1.b also creates a set of LQN tasks very similar to the ones obtained in Chapter 4. The only differences consist in the removal of *POE Workstation Disk Task*, and the introduction of the reference task *PKI System*, for which a dummy device (*Dummy Device5*) is created, and of *Link Task* and *PKD Server Disk Task*.

Finally, compared to Chapter 4, the set of mappings between LQN tasks and devices obtained with Step 1.c, does not include the association between *POE Workstation Disk Task* and *POE Workstation Disk*. It instead contains the associations between *PKI System* and *Dummy Device5*, *Link Task* and *Link*, *PKD* and *PKD Server*, and *PKD Server Disk Task* and *PKD Server Disk*.

### **B.2.2 Model Dynamics**

The application of Step 2.a generates the same outcome produced by the that step in Chapter 4, except for the introduction of the entry *PKCertificateRetrieval* for the reference task *PKI System*. Step 2.b is identical to the same step performed in Chapter 4. Step 2.c creates the entry *send-getPKCert* of *Link Task*, and the entry *readPKCertData* of *PKD Server Disk Task*. Finally, the outcome of Step 2.e is the

same as described in Chapter 4, with the exception of the result of processing the MRTD Authentication interaction occurrence, displayed in Figure 61. Step 2.e also generates a service request from *PKI System* toward the *getPKCertificate* entry of the *PKD* task, displayed in Figure 62.

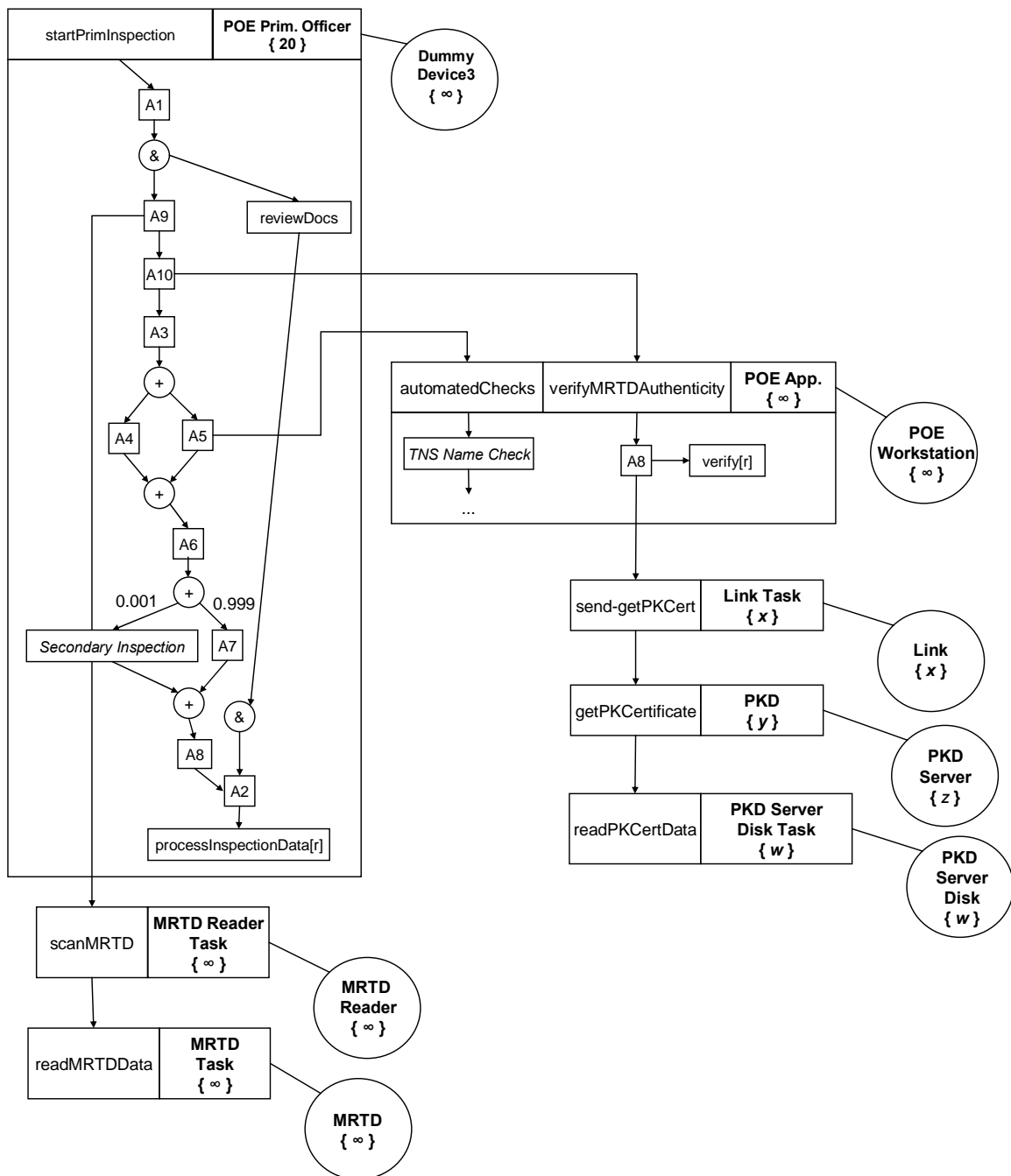


Figure 61: LQN request flow after MRTD Authentication

The final LQN model for Option 3, obtained as described in Chapter 4, is shown in Figure 62.

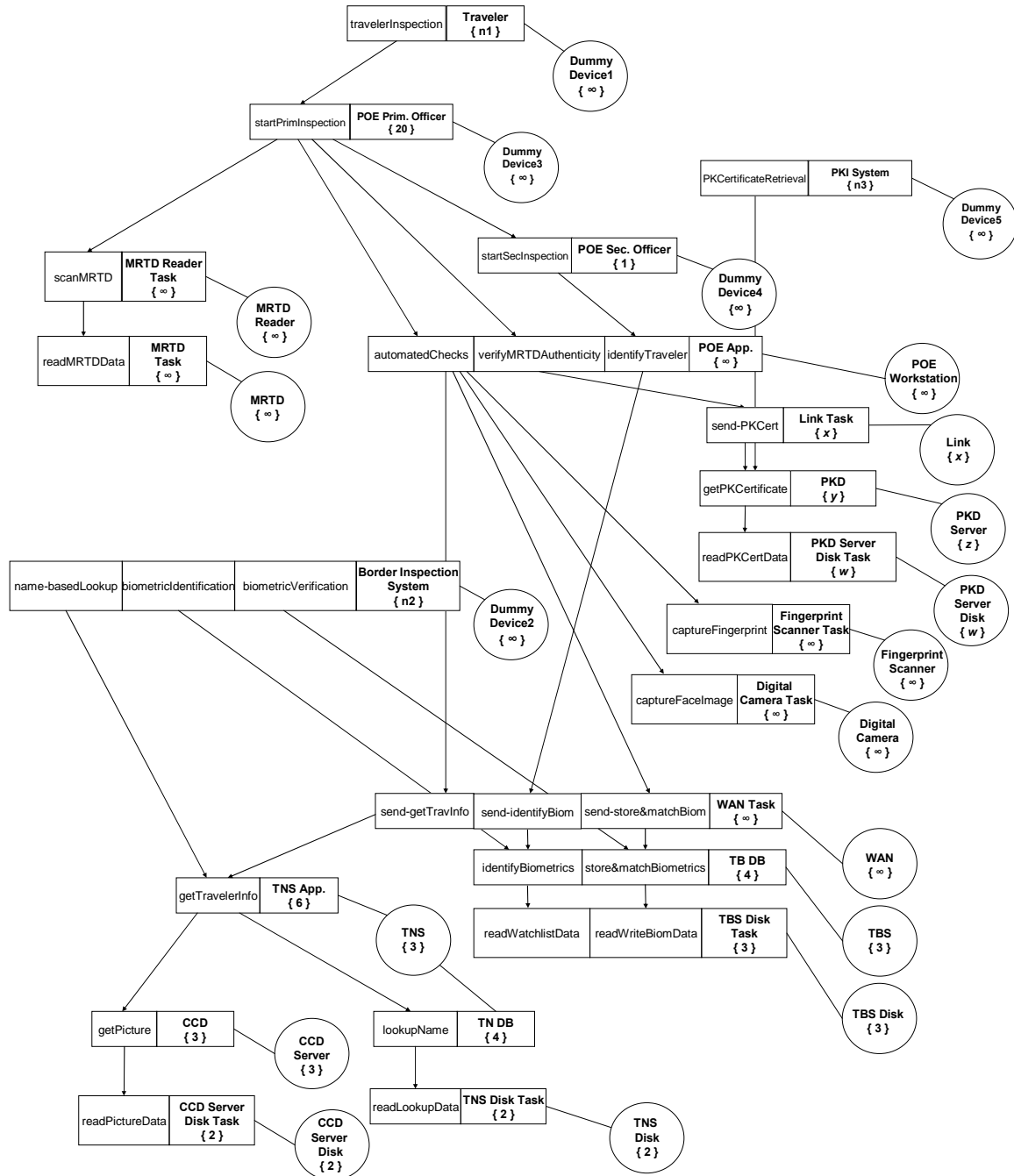


Figure 62: High-level layout of the LQN model for the airport inspection system (Option 3)

### B.2.3 Model Parameters

The parameterization of the LQN model for Option 3 is almost identical to the parameterization of the LQN model for Options 1 and 2. For this reason in this section we only cover the differences between them.

Table 8 summarizes the basic characteristics of the hardware nodes in the platform configuration for Option 3 that are not included in Options 1 and 2, i.e., the PKD Server and the Link (we assume it to be a WAN) between the POE Workstation and the PKD server. We do not specify the CPU rate and RAM of the PKD Server since we do not explicitly use that information to estimate resource demands for system operations.

Table 8: Execution environment (Option 3)

<b><i>PKD Server</i></b>	<i>CPU</i> : Not specified	
	<i>RAM</i> : Not specified	
	<i>Disk</i> :	Command Overhead: 1 ms
		Access Time: 3.5 ms
		Latency: 2 ms
Transfer Time: 75 MB/s		
<b><i>Link (WAN)</i></b>	<i>Bandwidth</i> : 16.6 Mbits/s (avg)	

The expected size of the data exchanged within the system is the same as reported in Appendix A for Options 1 and 2. The MRTD does not contain the Public Key Certificate of the document issuer therefore its size is 12852 bytes.

Table 9 lists the performance annotations attached to the steps of the MRTD Authentication interaction occurrence, the only one containing MRTD and PKD-related operations. The table also reports the annotations attached to the PK Certificate Retrieval use case, not provided by Options 1 and 2 of the inspection system.



Table 9: Resource demand of scenario steps (Option 3)

Scenario Step	Tag	Source	Value (s)	Rationale
<b>Scenario: Primary Inspection</b>				
scanMRTD	PAdemand	asmd	1.0	(1)
return MRTDData	PAdemand	pred	0.2368	(2)
	PAextOp (readMRTDData)	pred	0.9472	(3)
verifyMRTDAuthenticity	PAdemand	pred	0.0082	(4)
getPKCertificate	PAdemand	asmd	0.005	
	PAextOp (send-getPKCert)	pred	0.0017	(5)
return PKCertificate	PAdemand	asmd	0.005	
	PAextOp (readPKCertData)	pred	0.0065	(6)
verify(MRTD_DS)	PAdemand	pred	0.0091	(7)
verify(CAPKCertificate)	PAdemand	pred	0.0015	(8)
verify(MRZ_DS)	PAdemand	pred	0.0009	(9)
verify(faceImage_DS)	PAdemand	pred	0.001	(10)
return MRTDAuthenticity	PAdemand	asmd	0.005	
<b>Scenario: PK Certificate Retrieval</b>				
getPKCertificate	PAdemand	asmd	0.0	(11)
return PKCertificate	PAdemand	asmd	0.005	
	PAextOp (readPKCertData)	pred	0.0065	(12)

**Rationales for resource demand values**

(1)	See rationale (9) in Appendix A, Option 2
(2)	See rationale (10) in Appendix A, Option 2
(3)	See rationale (11) in Appendix A, Option 2
(4)	See rationale (12) in Appendix A, Option 2
(5)	It is the time required to receive from the PKD the Public Key Certificate of the MRTD issuer and of the Country CA for the MRTD issuer through the WAN connecting the POE Workstation with the PKD. We compute the average data transfer time as: $(2 \times 1.8\text{KB}) / 16.6 \text{ Megabits/s} = 0.0016942771\text{s}$
(6)	See rationale (13) in Appendix A, Option 2
(7)	See rationale (14) in Appendix A, Option 2

(8)	See rationale (15) in Appendix A, Option 2
(9)	See rationale (16) in Appendix A, Option 2
(10)	See rationale (17) in Appendix A, Option 2
(11)	It is the time required by the PKI system to generate a request for retrieval of Public Key Certificate. We assume this time to be null.
(12)	See rationale (13) in Appendix A, Option 2

Table 10 and 11 completely specify the parameterization of the LQN model for Option 3. The size of the *Traveler* population and of the *PKI System* population was defined in the *Performance Experiments Section* of Chapter 4.

Table 10: LQN parameters for system workloads (Option 3)

Reference Task	Multiplicity	Entry	Think Time (s)
Traveler	$n1$	<i>travelerInspection</i>	4s / 1 = 4s
Border Inspection System	3200	<i>name-basedLookup</i>	20s / 0.4995 = 40.04s
		<i>biometricVerification</i>	20s / 0.4995 = 40.04s
		<i>biometricIdentification</i>	20s / 0.001 = 20,000s
PKI Sysetm	$n3$	<i>PKCertificateRetrieval</i>	20s / 1 = 20s

Table 11: LQN parameters for resource demands (Option 3)

Task	Entry/Activity	Service Time (s)
POE Prim. Officer	<i>startPrimInspection</i>	5.0
	<i>reviewDocs</i>	20.
	<i>processInspectionData</i>	5.0
	<i>A5</i>	0.005
	<i>A9</i>	1.0
	<i>A10</i>	0.0082
POE Sec. Officer	<i>startSecInspection</i>	3.0
	<i>reviewDocs</i>	300.0
	<i>processInspectionData</i>	10.0
	<i>A2</i>	5.0
POE App.	<i>verifyMRTDAuthenticity</i>	0.005
	<i>automatedChecks</i>	0.005

	identifyTraveler	0.005
	processData	0.005
	verify	0.0124
	A5	0.005
	A6	0.005
	A7	0.005
	A8	0.005
	A9	0.005
	A10	0.005
MRTD Reader Task	scanMRTD	0.2368
MRTD Task	readMRTDData	0.9472
PKD	getPKCertificate	0.005
PKD Server Disk Task	readPKCertData	0.0065
Fingerprint Scanner Task	captureFingerprint	10.0
Digital Camera Task	captureFacelImage	5.0
TNS App.	getTravelerInfo	0.005
	A2	0.005
	A3	0.005
TN DB	lookupName	0.005
CCD	getPicture	0.005
TB DB	store&matchBiometrics	0.005
	identifyBiometrics	0.005
WAN Task	send-getTravInfo	0.0118
	send-store&matchBiom	0.0142
	send-identifyBiom	0.0094
	send-getPKCert	0.0017
TBS Disk Task	readWriteBiomData	0.0064
	readWatchlistData	0.2357
TNS Disk Task	readLookupData	0.006
CCD Server Disk Task	readPictureData	0.0062

## References

- [1] AJMONE, M., BALBO, G., and CONTE, G. A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems* 2 (1984), 93–122.
- [2] AJMONE, M., BALBO, G., and CONTE, G. *Performance Models of Multiprocessor Performance*. The MIT Press, 1986.
- [3] BACCELLI, F., BALBO, G., BOUCHERIE, R., CAMPOS, J., and CHIOLA, G. Annotated bibliography on stochastic Petri nets. In *Performance Evaluation of Parallel and Distributed Systems-Solution Methods* (Amsterdam, 1994), C. Tract, Ed., no. 105, pp.1–24.
- [4] BALSAMO, S., and SIMEONI, M. On transforming UML models into performance models. *Workshop on Transformations in the Unified Modeling Language*, April 2001.
- [5] BALSAMO, S., DI MARCO, A., INVERARDI, P., and SIMEONI, M. Model-based performance prediction in software development: A survey. *IEEE Transactions of Software Engineering* 30, 5 (2004), 295–310.
- [6] BANKS, J., II, J. C., NELSON, B., and NICOL, D. *Discrete-event System Simulation*. Prentice-Hall, 1999.
- [7] BASS, L., CLEMENTS, P., and KATZMAN. *Software Architecture in Practice*. Addison Wesley, 1998.

- [8] BERNARDO, M., and BRAVETTI, M. Performance measurement sensitive congruencies for markovian process algebras. *Theoretical Computer Science* 290 (2003), 117–160.
- [9] BOEHM, B. W. Verifying and Validating Software Requirements and Design Specifications. *IEEE Software*, vol. 1, no. 1, pp. 75-88, 1984.
- [10] BOOCH G., RUMBAUGH J., and JACOBSON I. *The UML Reference Guide*, Addison Wesley, 1999.
- [11] BUHR, R., and CASSELMAN, R. *Use Case Maps for Object-Oriented Systems*. Prentice-Hall, 1996.
- [12] CLEMENTS, P. C. Coming Attractions in Software Architecture. *Technical Report No. CMU/SEI-96-TR-008*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1996.
- [13] CORTELLESSA, V., DI MARCO, A., and INVERARDI, P. Three performance models at work: a software designer perspective, *2nd International Workshop on Foundations of Coordination Languages and Software Architectures*, September 2003.
- [14] DI MARCO, A. *Model-based Performance Analysis of Software Architectures*, PhD Thesis, Università degli Studi di L'Aquila, June 2005.
- [15] EDMUNDS, T., SHOLL, P., YAO, Y., GANSEMER J., CANTWELL, E., PROSNITZ D., ROSENBERG, P., and NORTON, G. Simulation Analysis of Inspections of International Travelers at Los Angeles International Airport for US-VISIT. *Technical Report*, Lawrence Livermore National Laboratory, CA, 2004.
- [16] FERRARI, D. *Computer Systems Performance Evaluation*. Prentice Hall, 1978.

- [17] FRANKS, G., HUBBARD, A., MAJUMDAR, S., PETRIU, D., ROLIA, J., and WOODSIDE, C. A toolset for performance engineering and software design of client-server systems. *Performance Evaluation* 24, 1-2 (1995), 117–135.
- [18] FRANKS, G. *Performance Analysis of Distributed Server Systems*. PhD Thesis, Carleton University, Canada, 2000.
- [19] FRANKS, G., MALY, P., WOODSIDE, M., PETRIU, D., and HUBBARD, A. *Layered Queuing Network Solver and Simulator User Manual*. Carleton University, Ottawa, Canada, 2005.
- [20] GU, G., and PETRIU D. XSLT Transformation from UML Models to LQN Performance Models. *Proc. of 3rd Int. Workshop on Software and Performance WOSP'2002*, pp.227-234, July 2002.
- [21] GU, G., and PETRIU D. From UML to LQN by XML algebra-based model transformation. *Proc. of 5th ACM Workshop on Software and Performance WOSP'2005*, pp.99-110, July, 2005.
- [22] HARRISON, P., and HILLSTON, J. Exploiting quasi-reversible structures in markovian process algebra models. *Computer Journal* 38, 7 (1995), 510–520.
- [23] HERMANNNS, H., HERZOG, U., and KATOEN, J. P. Process algebra for performance evaluation. *Theoretical Computer Science* 274, 1–2, pp. 43–87, Mar. 2002.
- [24] HILLSTON, J. Pepa-performance enhanced process algebra. Tech. Rep., Dept. of Computer Science, University of Edinburgh, 1993.
- [25] HILLSTON, J., and THOMAS, N. Product-form solution for a class of pepa models. *Performance Evaluation* 35, 3 (1999), 171–192.
- [26] HOARE, C. *Communicating Sequential Processes*. Prentice-Hall International, London, 1985.
- [27] HOPCROFT, J., and ULLMAN, J. *Introduction to automata theory, languages and computations*. Addison-Wesley, 1979.

- [28] INTERNATIONAL CIVIL AVIATION ORGANIZATION. PKI Digital Signatures for Machine Readable Travel Documents. *Technical Report*, Version 4, 2003.
- [29] INTERNATIONAL CIVIL AVIATION ORGANIZATION. PKI for Machine Readable Travel Documents Offering ICC Read-Only Access. *Technical Report*, Version 1, 2004.
- [30] KANT, K. *Introduction to Computer System Performance Evaluation*. McGraw-Hill, 1992.
- [31] KEMENY, J., and SNELL, J. *Finite Markov Chains*. Springer, New York, 1976.
- [32] KLEINROCK, L. *Queuing Systems Vol. 1:Theory*. Wiley, 1975.
- [33] KRAHL, D. Extend: the extend simulation environment, *WSC '01: Proceedings of the 33nd conference on Winter simulation*, 217-225, USA, 2001.
- [34] LAZOWSKA, E., KAHORJAN, J., GRAHAM, G. S., and SEVCIK, K. C. *Quantitative System Performance: Computer System Analysis Using Queuing Network Models*. Prentice-Hall, Inc., Englewood Cliffs, 1984.
- [35] MILNER, R. *Communication and Concurrency*. Prentice-Hall International, International Series on Computer Science, 1989.
- [36] OMG. *UML Profile for Schedulability, Performance, and Time*. OMG document ptc/2002-03-02, <http://www.omg.org/cgi-bin/doc?ptc/2002-03-02>.
- [37] OMG. *UML 2.0 Infrastructure Specification*. OMG document formal/05-07-05, <http://www.omg.org/cgi-bin/doc?formal/05-07-05>.
- [38] PETRIU, D., FRANKS, G., and HUBBARD, A. *SRVN input file format*. Carleton University, Ottawa, Canada, 1998.

- [39] PETRIU, D. C., and WANG, X. Deriving Software Performance Models from Architectural Patterns by Graph Transformations. *Lecture Notes in Computer Science* Vol. 1764, pp. 475-488, Springer, 2000.
- [40] PETRIU, D.C., and WANG, X. From UML description of high-level software architecture to LQN performance models. *Lecture Notes in Computer Science* Vol. 1779, pp. 47-62, Springer, 2000.
- [41] PETRIU, D., and SHEN, H. Applying the UML Performance Profile: Graph Grammar based derivation of LQN models from UML specifications. *Lecture Notes in Computer Science* 2324, pp.159-177, Springer Verlag, 2002.
- [42] PETRIU, D., and WOODSIDE, M. Software Performance Models from System Scenarios in Use Case Maps. *Proc. of 12th Int. Conf. on Modeling Tools and Techniques for Computer and Communication System Performance Evaluation TOOLS 2002*, pp. 141-158, April 2002.
- [43] PETRIU, D., and WOODSIDE, M. Analysing Software Requirements Specifications for Performance. *Proc. Third Int. Workshop on Software and Performance*, July 2002.
- [44] REISIG, W. Petri nets: an introduction. *EATCS Monographs on Theoretical Computer Science*, Vol.4, 1985.
- [45] ROLIA, J., and SEVCIK, K. The method of layers. *IEEE Transaction on Software Engineering* 21/8 (1995), 622–688.
- [46] SCHÜRR, A. Introduction to PROGRES, an attribute graph grammar based specification language. *Graph-Theoretic Concepts in Computer Science*, M. Nagl (ed.), LNCS 411, pp. 151-165, Springer, 1990.
- [47] SECTOR, I. T. S. *Message Sequence Charts, ITU-T Recommendation Z.120(11/99)*. 1999.
- [48] SEGHETTI, M., and VIÑA, S. U.S. Visitor and Immigrant Status Indicator Technology Program (US-VISIT), *CRS Report RL32234*, February 2004.



- [49] SMITH, C. U. *Performance Engineering of Software Systems*. Addison-Wesley, 1990.
- [50] SMITH, C. U., and WILLIAMS L. G. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, Addison-Wesley, 2002.
- [51] TRIVEDI, K. S. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley and Sons, 2001.
- [52] WILLIAMS, L. G., and SMITH C. U. PASA<sup>SM</sup>: a Method for the Performance Assessment of Software Architectures, *Proc. of 3th ACM Workshop on Software and Performance WOSP'2002*, pp. 307-320, July 2002.
- [53] WOODSIDE, C. M. Throughput Calculation for Basic Stochastic Rendezvous Networks. *Performance Evaluation*, Vol. 9, No. 2, April, 1989.
- [54] WOODSIDE, C., NEILSON, J., PETRIU, S., and MJUMDAR, S. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Transaction on Computer* 44 (1995), 20–34.
- [55] WOODSIDE, M., and FRANKS, G. *Tutorial Introduction to Layered Modeling of Software Performance*. Carleton University, Ottawa, Canada, 2005.
- [56] WOODSIDE, M., PETRIU, D. C., PETRIU, D. B., SHEN, H., ISRAR, T., and MERSEGUER, J. Performance by Unified Model Analysis (PUMA). *Proc. of the 5th ACM Workshop on Software and Performance WOSP'2005*, pp. 1-12, July 2005.
- [57] XU, J., WOODSIDE, M., and PETRIU, D. Performance Analysis of a Software Design using the UML Profile for Schedulability, Performance and Time. *Proc. of 13th Int. Conf. on Modeling Techniques and Tools for Computer Performance Evaluation*, September 2003.