A Flexible Framework for Modeling Middleware Completions

by

Adnan Faisal

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

Doctor of Philosophy

in

Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

Abstract

Middleware has a major impact on the performance of distributed software applications. To predict the performance (e.g., response time, throughput) of an application using a performance model, it is essential to capture not only the characteristics of the application, but also the platform and the middleware. Model elements, which are added to an application software model for performance analysis are known as Completions. This research proposes a unified and flexible framework, called MMLQ (Middleware Modeling for Layered Queues), for modeling middleware completions and automatically composing them into software performance models. The existing approaches have many limitations. Some of them require to build and compose models manually, others lack generality, while others use component-based modeling which creates many elements that are not essential for performance analysis and complicate the model. MMLQ overcomes the limitations of the existing approaches and can model a wide range of middleware platforms. The contributions of this thesis include identifying the commonalities and differences among different middleware platforms, modeling their features, specializing middleware models to cater to the needs of an application, calibrating the specialized models and a proposing a process to compose calibrated models with the application models. The software application models and the middleware models are all built using Layered Queuing Network Model (LQN) developed at the RADS lab of Carleton University. The MMLQ framework is validated by showing its ability to model three different categories of middleware and by comparing the predicted performance with measured performance metrics.

Acknowledgements

In the name of God, the most beneficent and the most merciful.

Behind this thesis there are patience, inspiration and hard work of many. A page or two is not enough to describe the gratitude that I have for them in my heart, yet I would try to summarize my feelings for them here.

I would begin by expressing my endless thankfulness to my two co-supervisors: Prof. Murray Woodside and Prof. Dorina Petriu. They always made sure that I never lost my track and I am so grateful for getting them as my supervisors during this long journey.

It is said that behind every success story there is a woman, and I know how true it is. There is a lady who never loses her confidence on me, who believes in me more than I believe in myself, she is my wife, my love and my best friend – Farhana. It would have never been possible without all the tips, advices and mental support she gave, and her hard work for taking care of our little family. Without her love and inspiration, I could never become a proud doctorate.

PhD is a long, hardworking road in which times of joy and amusement are needed to keep oneself going. My only son Farhaz brought that joy and amusement to my life. The moments in the mornings when I used to leave him to his daycare/school so that I could come and work in my lab were heartbreaking. But, the joy that I used to receive when we got back home in the evenings and played hours with superhero characters are the moments that make life worthy.

I firmly believe that nothing is more powerful than sincere prayers. My mother is the one who was consistent in her prayers throughout my PhD studies. Though she lives thousands of miles away from me, in a different country in a different continent, but she is always

closest to my heart. A person who has such a loving mother, can never fail in his life. I wish my father were alive to see this day who had a dream that I would become a great Computer Engineer one day.

It is easier to learn things by practically seeing in front of your own eyes rather than reading thousands of pages. The moments that I spent with my father-in-law, who was a medical doctor by profession, I learned how to keep myself cool and calm in complex situations. I am so grateful to him for his patience, support and trust on my abilities. My two sisters, and my sisters-in-law – their love and prayers greatly inspired me. I am thankful to my brother, brothers-in-law and, other members of my extended family and my friends, both in Ottawa and abroad, for their inspirational words and prayers.

I am especially grateful to the reviewers and panel members of Doctoral Symposium of MODELS'15 Conference. Their feedback and comments helped me a lot to identify and resolve the various gaps that I had in my thesis.

I could have never accomplished this thesis without funding. This research was funded by the Natural Sciences and Engineering Research Council of Canada through its Strategic Network SAVI (Smart Applications on Virtual Infrastructure) and its Discovery Grant programs. I am grateful to those projects for the funds they provided me.

This work is dedicated to

my lovely wife Farhana,

my cute son Farhaz

and

my beloved mother Halima Khatun

Table of Contents

List of Tables

List of Figures

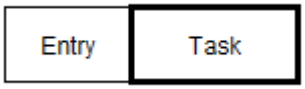
List of Listings

List of Acronyms

AOS	Aspect Oriented Modeling
ARS	Airline Reservation System
AWS	Amazon Web Services
BAM	Base Application Model
BaseFM	Base Feature Model
BMM	Base Middleware Model
CBM	Container Based Middleware
CORBA	Common Object Request Broker
DICE	Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements
DoBM	Distributed Object Based Middleware
EC2	Elastic Compute Cloud
FCD	Feature Composition Descriptor
FRM	Feature Realization Model
LQN	Layered Queuing Network
MBA	Middleware Benchmark Application
MBM	Message Based Middleware
MCD	Middleware Composition Descriptor
MCE	Middleware Composition Engine
MDE	Model Driven Engineering
MMLQ	Middleware Modeling for Layered Queues

MODAClouds	MOdel-Driven Approach for design and execution of applications on multiple Clouds
MQ	Message Queue
PCM	Palladio Component Model
POSAML	Pattern-Oriented Software Architecture Modeling Language
QoS	Quality of Service
QPN	Queuing Petri Nets
RADS	Real-Time and Distributed Systems
RAM	Reusable Aspect Models
REST	Representational State Transfer
RMI	Remote Method Invocation
SAM	Specialized Application Model
SEI	Service Endpoint Interface
SMM	Specialized Middleware Model
UML	Unified Modeling Language
U-QASAR	Universal Quality Assurance & Control Services for Internet Applications with Volatile Requirements and Contexts
VM	Virtual Machine

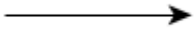
List of Symbols (used in LQN models)



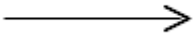
Task (with entry)

Host

Deployment



Blocking call



Non-blocking call

Unknown call type

| Single bar denotes an element that will remain in the composed model

|| Double bar denotes an element that will not remain in the composed model

List of Appendices

1 Chapter: Introduction

Models used for software development commonly represent the functional properties of the software, without modeling the platforms on which the software will run. The performance impact of the platforms can be addressed by both performance testing (e.g., benchmarking) and modeling. This thesis prefers the modeling approach because it supports the analysis of many issues in a single model including changes in software, deployment and workload.

The information about the underlying platform infrastructure (e.g., middleware, operating system, hardware, networks) on which the software will be deployed is essential for the analysis of software performance. This issue was identified by Woodside et al. [98] and the authors called these added performance model elements *Performance Completions* or simply *Completions*. A key completion for a distributed application is its middleware [1], [21] . This thesis proposes a framework called MMLQ (Middleware Modeling for Layered Queues) to model middleware and to compose it with software performance models. The framework is particularly useful during the early software development phase to predict the performance of an application under various possible middleware configurations for a given host deployment. MMLQ is flexible, as it covers a number of middleware platforms and their features, and it also provides a mechanism to control the size of the performance models. This chapter presents the motivation, objectives and scope of this work, as well as expected contributions.

1.1 Motivation

There are three main reasons that motivated the development of a framework for modeling middleware completions, as described below.

1. Middleware has a major impact on software performance: Different middleware require the 'transferred message' to be packaged in different format, giving different numbers of messages and message sizes. In [49], [48], [35], it is shown that the transferred message sizes change not only from one middleware to another, but also depend on what services are used (e.g., encryption, compression). Juric et al. [48] observed that the SOAP messages of web services were on average ~4.3 times larger than Java Remote Method Invocation (RMI) JRMP messages (See Table 1-1). Encrypted communication with the Triple-DES encryption algorithm did not increase the RMI message size, but it greatly increased the SOAP message size due to the use of textual encoding.

Table 1-1 Relative message sizes for different middleware [48]

RMI	RMI with Encryption	WS	WS with Encryption
1	1.5	4.3	30

The impact of middleware on system response time is shown in Table 1-2. Web services were ~9.6 times slower than RMI for simple types (e.g., boolean, integer etc.) and strings, and ~14% slower for instantiation. RMI–SSL was ~25% slower than RMI for simple types, strings, and instantiation.

Table 1-2 Approximate response times [ms] for different middleware with respect to workload [48]

Workload	RMI	RMI with Encryption	WS	WS with Encryption
Instantiation	0.75	0.93	0.85	0.85
Invoke basic data type	0.4	0.5	3.85	450
Invoke String	0.4	0.5	3.85	400

2. Separation of concerns: Though any distributed application is highly dependent on the middleware on which it is being deployed, the development and maintenance of the middleware is not the job of the application development team. With MMLQ, the application development team can use completions prepared by middleware experts, and focus on their job, without losing sight of middleware impact.

3. Middleware models are reusable by nature: In any distributed system, a middleware is shared by many components of the application. Since the middleware is reused, its models can also be reused in many places within one application, and across applications. This gives more efficient modeling.

The data in Table 1-2 show the importance of modeling the middleware, since its impact depends on what functions are performed. If the performance analyst can estimate the performance impact of different middleware on the application software, the system designer can choose the right middleware in order to meet not only the functional requirements but also the performance requirements.

1.2 Objective and Scope

The objective of this thesis is as follows:

To develop a systematic, flexible framework that supports model completions for a wide range of middleware for software performance models.

Let us examine the research objective bit by bit. We examine the research objective sentence starting from the end, and move towards the beginning, describing the scope of the research along with.

- **… for software performance models:** This thesis deals with performance models of software. In other words, the models used in this thesis neither have the purpose of representing software design nor software architecture. The models are built solely for the purpose of performance evaluation, intending to predict different performance metrics such as Throughput, Response Time, Utilization etc. (see Chapter 2 for the definitions of terms). A logical consequence of this scope is using a *Domain-Specific Language* (DSL) that is developed solely for the purpose of performance modeling. This thesis uses Layered Queuing Network (LQN) [32] as the modeling language.
- **… for a wide range of middleware:** This thesis aims to model middleware, assuming that a performance model for the application is already given. It also assumes that other elements of the underlying platform, apart from the middleware, are also given. In this thesis, *Middleware* [21] is defined as a software layer supporting distributed communications that provides a programming abstraction as well as masks the heterogeneity of the underlying networks, hardware, operating systems and programming languages. "A wide range" means that middleware can be used to model many kinds of middleware platforms. Variability between different middleware platforms is due to their differences in architecture and implementation. The middleware platforms covered in this thesis are discussed in the next section.

- **… framework that supports model completions**: *Completions,* in general*,* refer to model elements of a software that are not required for functional specification but essential for performance analysis. Completions may range from execution cost of operations and details of deployment, up to missing subsystems and layers. In this thesis completions are limited to middleware. The thesis proposes a framework named MMLQ (Middleware Modeling for Layered Queues) to model middleware completions, customize and calibrate them according to the deployment, and compose them with the software performance model.
- **A systematic, flexible framework** …: "Systematic" means that the proposed framework has concrete steps that describe building and composition of the completions. "Flexible" means that the framework is able to model variabilities across many middleware platforms, which are caused by the presence of optional features (e.g., security, compression etc.), and is extendible to new middleware types. It also means that the framework offers a way to control the size of the performance models.

This thesis classifies middleware in three groups (as described in the next section) and covers models of all three groups. It defines a descriptive framework for the architecture and features of any middleware, and a process for developing a model for any middleware. It also includes models for a wide range of popular distributed web-application middleware.

1.3 Middleware Platforms Covered

Middleware may refer to a variety of software and while this thesis models a wide range, it does not cover all kinds of middleware. The thesis models the following three kinds of middleware:

- **Distributed object based (or point-to-point) middleware** in which a caller makes direct calls to the receiver, without any intermediary or service container. Both caller and receiver use the same application software (i.e., middleware) for distributed communication. Today's point-to-point middleware platforms are all object based and they are known as Distributed Object Based Middleware [63]. RMI and CORBA are such examples.
- **Container based middleware** in which a *Container* is deployed to the server side to handle its services. A huge number of middleware platforms of today, e.g., Servlet, EJB, Spring, SOAP, employ web containers. Though REST is not a pure middleware (but rather, an architectural style) it can be modeled by our framework because REST services are deployed to web containers, and from a performance modeling perspective, they are similar to other container based middleware platforms. The performance of REST was modeled using this framework in Faisal et al. [28] and will be further demonstrated with example in the chapter of experimental results (Chapter 6). This thesis models web containers, not containers in general.
- **Message based middleware** where callers transfer their messages to a message queue that takes the responsibility of delivering those messages to the subscribed

receivers. JMS, ActiveMQ, and RabbitMQ are some message based middleware platforms.

Any application that uses a combination of the above middleware platforms can be modeled using the MMLQ framework.

The following kinds of middleware platforms are excluded from this study:

- **Group communication,** in which a message is sent to a group server and then delivered to all members of the group. Group communication may either be application layer multicast or network assisted multicast. The former does not scale well [21], and in the latter, copies of the message are automatically created in other network elements, such as routers, switches and cellular network base stations, located in network segments that currently contain members of the group. In order to model the performance of such middleware, network layer modeling is required which is out of the scope of the thesis. If a network layer model would be given, the process defined here could be applied to group communication middleware.
- **Streaming middleware** [21] transmits continuous streams of data in real time. Streams of data can be large quantities of audio, video and other time-based data elements, and the timely processing and delivery of the individual data elements (audio samples, video frames) is essential. A flow specification for a multimedia stream is expressed in terms of bandwidth (i.e., acceptable values for the rate at which data passes from a source to destination), response time (i.e., the delivery delay of each element), jitter, and the rate at which elements are lost or dropped. The LQN tool used in this thesis do not address jitter or lost packets, so multimedia middleware was not addressed in this research.

- **Peer-to-peer systems** [21] represent a paradigm for the construction of distributed systems and applications in which data and computational resources are contributed by many hosts on the Internet, all of which participate in the provision of a uniform service. As a result, every participating node in a peer-to-peer system acts as both a caller and a receiver. Moreover, nodes leave and join their peer-to-peer network dynamically, changing the structure of the system at runtime. Structural dynamics are outside the scope of LQN models, although a single state of the structure could be modeled using the present methods.
- Middleware platforms for **database transaction and concurrency control** [95] are used in servers to manage shared resources. They are used to maintain a system's integrity (typically a database or some modern filesystems) in a known, consistent state, by ensuring that interdependent operations on the system are either all completed successfully or all canceled successfully. While MMLQ does model transaction processing as a feature of a middleware, it does not model middleware platforms specially designed for transaction and concurrency control The reason is that the primary purpose of such middleware platforms is not message transmission, but rather maintaining the ACID properties [95] in the server. Therefore, a different set of features are needed to model them. The modeling process of MMLQ could be applied to create such models, in future work.
- **Hard real-time systems** [21] is hardware or software that must operate within the confines of a stringent deadline. The application may be considered to have failed if it does not complete its function within the allotted time span. Examples include nuclear power systems, some medical applications such as pacemakers, a large

number of defense applications, avionics, etc. have this requirement. The thesis does not model hard real-time systems but all its models are for soft real-time systems.

- The thesis does not model **virtualization middleware** such as virtualization containers and hypervisors. However, the proposed feature models can be extended with the features of such middleware.

1.4 MMLQ (Middleware Modeling for Layered Queues) Framework Overview

The proposed MMLQ (Middleware Modeling for Layered Queues) framework starts from an application software model (called the *Base Application Model (BAM)*) and its deployment, but without middleware. The goal is to model any middleware (including its different features) that the BAM will use. Each interaction with middleware is first extended with a minimum middleware model, called a *Base Middleware Model (BMM)*. Each chosen middleware provides a number of services (e.g., serialization, compression, security etc.) that we call *features*. A feature is an abstract concept and its model is called *Feature Realization Model (FRM)*. The specification of which FRM are used by a BMM are written in a text file called *Feature Composition Descriptor (FCD)*. The *Middleware Composition Engine (MCE)* takes BMM, FRM and FCD as input to produce a *Specialized Middleware Model (SMM)* for each middleware instance. The SMM is calibrated to get the service demand of the SMM under different workload. Finally each calibrated SMM is composed to the BAM to obtain the complete software performance model.

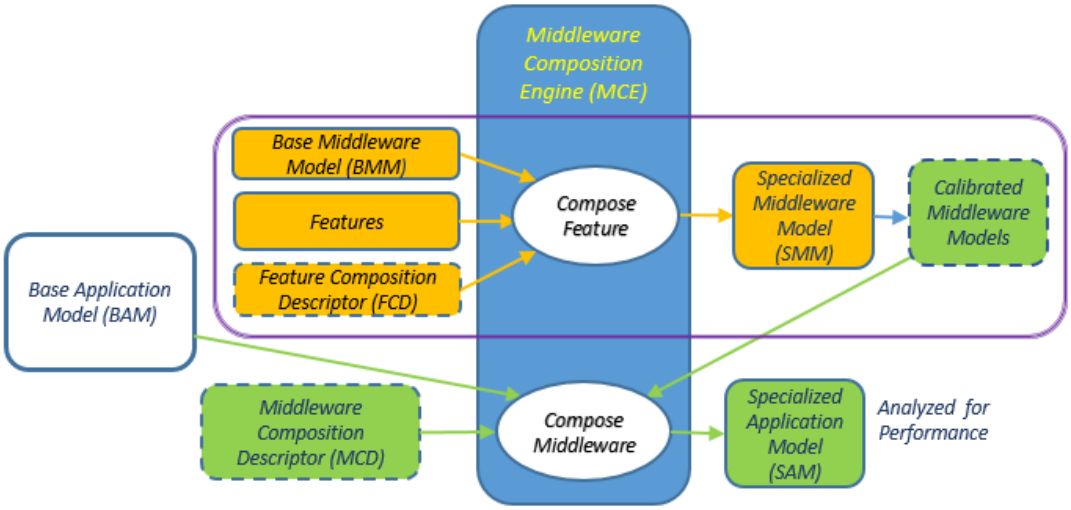


Figure 1.1 The MMLQ Framework

Where and how the SMM is to be composed to the BAM is specified in a text file called *Middleware Composition Descriptor (MCD)*. The Engine takes the BAM, SMM and MCD as input to produce a complete software model (that includes the middleware submodels) called the *Specialized Application Model (SAM)*. All the models in the process, including the final SAM, are modeled in LQN language. The final model is executed using the LQNS [32] tool to obtain performance metrics (e.g., response time, throughput) of the complete system.

Use of the MMLQ Framework in Software Performance Engineering (SPE):

MMLQ is useful for building a distributed application following the principles of *Software Performance Engineering (SPE)*. SPE [83] refers to the building of performance models for an application as early as possible, preferably during the architectural design, when a large number of design options are still available.

Following the process of SPE, a performance modeler builds the performance models for his/her application at design time. During this time, the project manager also decides, or at least narrows down the choices, about the deployment environment (e.g., local server,

rented cloud etc.) of the application. Based on the information above, a performance modeler can estimate [83] the execution demand and latency over the network for that application. But, the end to end response time heavily depends on the middleware as well [49], [48], [35]. Another considerations is that, before beginning the actual development of the application, the project manager needs to make crucial decisions about the application's middleware and services. This is where the MMLQ framework comes into play.

A performance modeler can use MMLQ to model a wide range of middleware platforms and their services (i.e., features) as shown in Chapters 3 and 4. Using the process described in Chapter 6 these models can be calibrated. Then these middleware models are composed to the application model following the process described in Chapter 5. The final model is executed to predict the performance of the application for the composed middleware platforms. The entire process can be repeated for many middleware platforms and their services to make predictions about how a distributed application is going to perform under various middleware platforms and their features.

Therefore, the MMLQ framework is helpful in predicting the performance of distributed applications at an early development phase. The framework offers the benefit of modeling the middleware once in order to reuse it later for different modules of the same software, or even for different software systems. This is particularly suitable in today's cloud-based infrastructure as there are many virtual machines that essentially use the same software and hardware configurations. In addition, MMLQ is highly compatible with the cutting-edge DevOps technologies due to its automated feature and middleware composition processes.

1.5 Contributions

The main contribution of this research is a systematic approach to model the performance impact of many middleware platforms and their features. This is achieved through the following *contributions to knowledge*:

1. Modeling Middleware Features (Chapter 3): Every middleware platform offers a set of services that are common between different middleware platforms. Yet, every middleware has variability in terms of its features. The service demand of a middleware depends on the features chosen for that middleware. This thesis proposes a systematic, flexible framework called MMLQ, that can capture the commonality and variability (from a performance view-point) of various middleware platforms. This is achieved by means of the following sub-contributions:
    - Proposing a Base Middleware Model (BMM) and identifying its properties, so that it can be adapted to a large range of middleware.
    - Identifying and grouping the different features present in various middleware platforms.
    - Describing two kinds of realizations for a middleware feature, so that either compact or detailed middleware models can be built as required.
2. Specialization of the Middleware Model (Chapter 4): The BMM needs to be customized based on the features required by it for the Base Application Model (BAM). This is achieved by means of the following two sub-contributions:
    - Feature Composition Descriptor (FCD) notation to specify what features of the middleware to be used with what properties.

- Feature Composition Process of the Middleware Composition Engine (MCE) to compose the BMM with the features described in FCD, producing a Specialized Middleware Model (SMM).

3. Obtaining a middleware-aware software application model (Chapter 5): The BAM is composed with SMM to obtain the desired application model that also includes the middleware model. Again, this contribution can be divided into two sub-contributions:
    - Middleware Composition Descriptor (MCD) notation to specify what middleware to be used for what group of application calls with what properties.
    - Middleware Composition Process of the Middleware Composition Engine (MCE) to compose the BAM with the MCD-specified middleware, producing the Specialized Application Model (SAM). The SAM is finally solved by the LQNS tool to obtain performance results.

This work also has the following *practical contributions*:

1. Calibration of SMM (Chapter 6): A process to calibrate the SMM to obtain its service demand under various workloads is described. A synthetic middleware benchmark application was built for this purpose.
2. Model Validation (Chapter 6): Validating the developed framework and the tool by comparing the model performance results against the measurements obtained from a real system. The system was built in Java with different middleware platforms and deployed to the Amazon Web Services cloud.
3. Composition Tool: A Middleware Composition Engine (MCE) for composing features models to middleware models and middleware models to application models

is built. The current tool takes simple input from keyboard instead of parsing FCD and MCD files.

The following papers are the outcomes of this research work so far:

1. **Adnan Faisal,** Unified Approach for Adding Middleware Completions to Software Performance Models, Doctoral Symposium, MoDELS 2015, Ottawa, ON, 29 Sept 2015.
2. **Adnan Faisal,** Dorina Petriu, Murray Woodside, A Systematic Approach for Composing General Middleware Completions to Performance Models, in "Computer Performance Engineering" (Proc. European Performance Engineering Workshop EPEW14, Florence Sept. 2014), LNCS vol. 8721, Springer, pp.30-44
3. **Adnan Faisal,** Dorina Petriu, Murray Woodside, Network Latency Impact on Performance of Software Deployed Across Multiple Clouds, CASCON 2013, November 18-20, 2013, Markham, Ontario, Canada.

1.6 Thesis Content

This thesis is structured as follows:

Chapter 1: Introduction. This chapter motivates the research, presents the objective, scope and contributions of this thesis and also gives an overview of the proposed solution.

Chapter 2: Background and State of the Art. This chapter briefly describes the foundational knowledge upon which this thesis depends, including a brief introduction to Queuing Networks, Layered Queuing Networks and software performance metrics. This chapter also defines feature, middleware, their classification and describes in brief the middleware platforms that are modelled in the thesis. The chapter continues with a discussion of the state

of the art, describing existing approaches found in literature for middleware completion. The chapter concludes by mentioning some weaknesses of the previous approaches that the MMLQ framework overcomes.

Chapter 3: Modeling Middleware Features. This chapter describes the base middleware model, and then presents a unified feature model to model the differences and similarities between different middleware platforms.

Chapter 4: Specializing Middleware. This chapter describes the process for obtaining the Specialized Middleware Model (SMM) with features of the middleware to be modeled.

Chapter 5: Obtaining the Specialized Application Model (SAM). The process of composing the BAM with the SMM to obtain the desired SAM is described in this chapter.

Chapter 6: Experimental Analysis: Calibration and Validation. Calibration process for different kinds of middleware platforms are presented. The MMLQ framework is validated by comparing the model results with actual system results.

Chapter 7: Conclusions. This chapter concludes the thesis, summarizing its achievements, limitations and open directions for future research.

2 Chapter: Background and State of the Art

This chapter describes background knowledge and discusses the state of the art in the field of performance modeling of middleware completions.

2.1 Queuing Network (QN)

In our daily life we are all familiar with some sorts of queues. From bank to coffee shop, wherever there is some service, there is a queue. In a queuing network, service is provided by a *node* which contains two parts: a *Queue* and a *Server*. *Arriving Requests* or *Jobs* coming to a node for service join the queue and at some point are served by the Server. Upon receiving its service, a Request leaves the node and is a *Completed Request*.

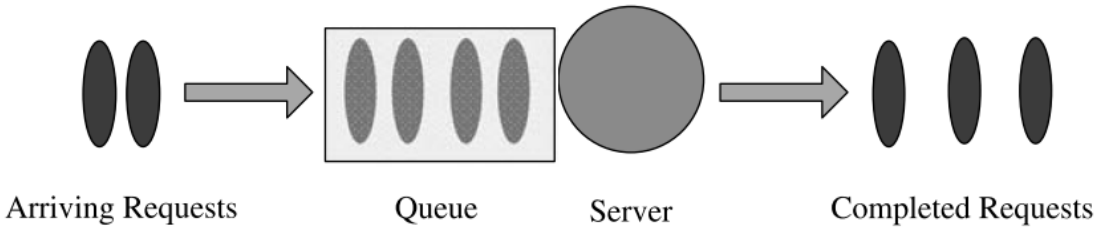


Figure 2.1 Graphical representation of a queuing node (taken from [62])

Queues can be chained to form networked queuing systems where the departures from one queue enter the next queue. Queuing systems can be classified into two categories: *open queuing systems* and *closed queuing systems*. Open queuing systems have an external input and an external final destination. Closed queuing systems are completely contained and the customers circulate continually, never leaving the system. Of course, there could also be a situation in-between, for example, an open queuing system that contains internal feedback loops.

Queuing models can get very complicated. Figure 2.2 is an example where a network of Zigbee sensors is modeled. In this model there are two sensor nodes with open arrivals and each of them having a Processing Center (PC) and a Transmission Center (TC). The model

also uses other elements such as source, sink, fork, join and router. For a description of this model please refer to [11].

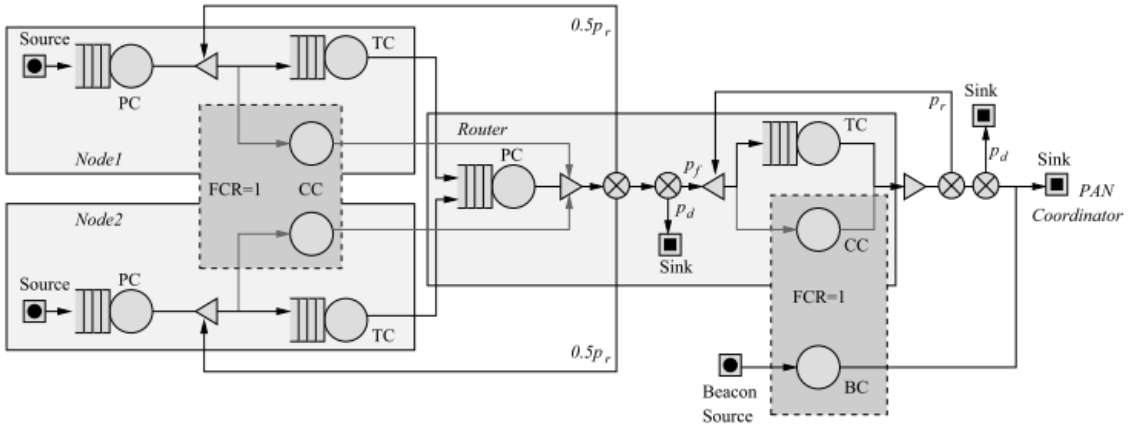


Figure 2.2 A network of Zigbee sensors represented as a Queuing Network [11]

Some of the frequently used QN terms are summarized below, using an example of a simple coffee-shop where customers come to buy and drink coffee. The terms that are input to a QN model are described first with the help of this example.

Server: This is the coffee-shop fulfilling the customer's service request.

Customer: This is the initiator of service requests. A Customer is sometimes also referred as **Job** or **Request**.

Service Time: This is the time duration from when a coffee-maker starts to service a customer to when the customer is leaving the shop and the next customer is called in for service.

Service Rate: This is the rate at which customers are serviced by the coffee-maker. Mathematically, it is the inverse of the service time.

Service Demand: If a customer needs to visit the coffee-maker more than once to get his/her coffee then service demand is the sum of all the service times. Mathematically, service demand is computed as service time multiplied by a *visit count*.

Arrival Rate: This is the rate at which customers arrive at the coffee-shop. An arrival rate exists only in open queuing systems.

Number of Customers: Total number of customers (both in queue and in service) in a coffee-shop. For a closed queuing system, this is a fixed number.

Think time: It is the idle time of a customer when instead of joining the queue of a server, he/she is rather busy thinking what kinds of coffee to order. Think times are used in closed systems to represent user time, and sometimes they are used in both open/closed systems to represent pure latency.

QN models can be solved either analytically or by simulation [62] to obtain P*erformance Metrics*. Some key performance metrics are described below.

Utilization: This is the portion of a coffee-maker's time actually servicing customers rather than idling, expressed as a fraction or percentage. The node of a queuing network that has utilization nearest to 100% is called the *bottleneck*.

Wait Time: This is the time duration a customer has to spend waiting in line.

Queue Length: This is the total number of customers waiting or being serviced or both.

Response Time: This is the sum of wait time and service time for one visit to a coffee-shop.

Residence Time: This is the total response time if the coffee-maker is visited multiple times for one transaction.

System Response Time: Sum of residence times over all queuing nodes in a queuing network.

Throughput: This is the rate at which customers are serviced in a queuing node. When an open queue system is in equilibrium, then its throughput equals its arrival rate.

System Throughput: This is the rate at which customers are serviced in a queuing network.

It is obvious that in terms of performance, the goal of the customer would be to get faster service and the goal of the coffee-maker would be to have a busy shop with many customers. But, the coffee-maker does not like the shop to be overloaded with customers as it would frustrate the customers due to long waiting lines. In QN terms, these goals can be translated into following sentences:

- Customers want low response time.
- Coffee-maker wants high throughput.
- Coffee-maker wants medium to high utilization. A small utilization (<20% approx.) means the coffee-shop has only few customers. A very high utilization (>90% approx.) means the coffee-maker has become bottleneck, the coffee-shop is saturated and response time may be too high.

The most important performance metrics of any queuing system are response time, throughput and utilization. In "Chapter 6: Experimental Analysis: Calibration and Validation", the models produced by MMLQ are validated by comparing their performance metrics with those of actual systems.

Queuing networks are excellent for modeling computer systems because they are made up of chain of service nodes. But, traditional queuing networks (as described so far) do not model many scenarios including blocking calls (e.g., RPC), simultaneous resource possession (e.g., semaphore) etc. that are common in computer systems. This shortcoming moti-

vates the creation of many extended queuing networks such as the Layered Queuing Network (LQN) developed at Carleton University. The models built in this thesis are based on the LQN domain-specific language. The next section gives a brief introduction to LQN.

2.2 Layered Queuing Network (LQN)

The central feature of an LQN is a kind of structured "simultaneous resource possession" [97]. Software systems often have software servers with a queue of requests. When a request is accepted by the software server process, the process must then request and wait for the CPU. When it has the CPU, the operation can be executed and a reply returned for the request. This is "simultaneous resource possession" with two layers of resources, the software server and the CPU. This scenario is common in layered and client-server architectures, which gives it the name Layered Queueing.

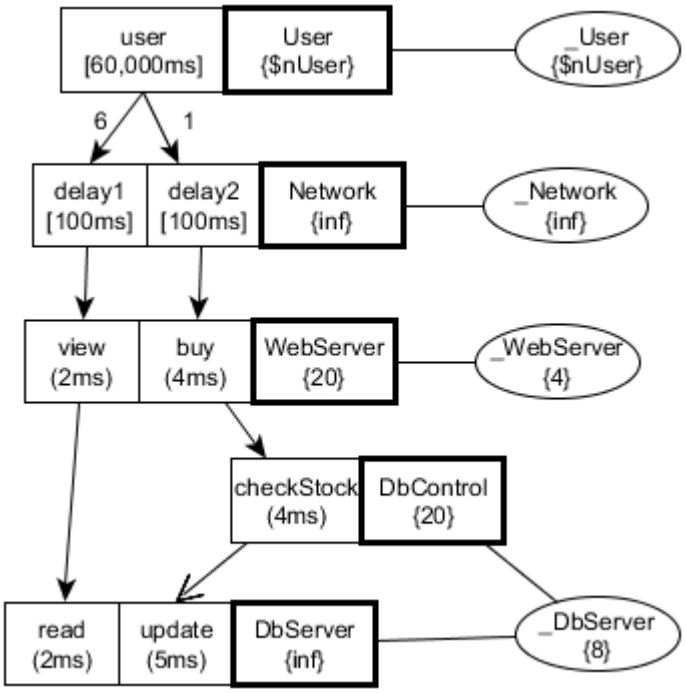


Figure 2.3 An LQN model for a 3-tier web application

The LQN performance models [32] offer the advantage that they directly represent the software components and their deployment. LQN treats software entities as resources, and captures inter-component communications, and resource interactions between layers of the application. LQN models can be written using LQML language and can be solved either analytically or by simulation using the LQNS solver. Figure 2.3 shows a generic layered model of a web application.

The main players of an LQN model are the following:

Tasks represent software processes. A task has a queue for requests. A task is defined by its name, thread-pool multiplicity and queueing discipline. A task is drawn as two or more rectangles where the right-most rectangle contains task name, with the multiplicity in braces. In Figure 2.3, there are in total five tasks (e.g., User, Network, WebServer, DbControl and DbServer) where User task generates traffic, Network task models network latency and the rest of the tasks provide services. Typically in LQN, network latency is modeled using a *Network* task that has infinite thread and is deployed to a host of infinite multiplicity [1], [97]. The default task multiplicity is 1 and queuing discipline is FIFO. In this thesis, all task names start with a capital letter. Service demands are given in round parentheses. Think times / delays are given in square brackets.

Hosts are processors where tasks are deployed. Hosts are physical entities that carry out operations. Each host has a queue of its own for the requests it receives. A host provides service (i.e., actual CPU execution) to the jobs that comes from the queues of tasks to its own queue. A host is defined by its name, core-multiplicity, queuing discipline and speed ratio [97]. Default multiplicity of a host is 1, queuing discipline is FIFO and speed ratio is 1. A host is drawn as an oval and its multiplicity is given in braces. Figure 2.3 has 3 hosts

namely *_User, _Network, _WebServer, _DbServer*. In this thesis, all host names start with an underscore (_) followed by a capital letter.

Entries represent operations of software processes (i.e., tasks). One or more entries may belong to a task. Every entry has a name and has a time duration that it demands from its host to carry out its operation completely. In LQN, this time duration is called the service demand (not service time) of the operation and it is given for one invocation of the entry. An entry may also have optional think time. Think time and delay values of entries are given in square brackets. In Figure 2.3, the think time of *user* is 60,000 ms. The delay values of the entries *delay1* and *delay2* are 100 ms each. In this figure there are 8 entries and the service demand of every entry is given in round parentheses (e.g., service demand of *view* is 2ms). In this thesis, entry names always start with a small letter.

Calls are used to flow jobs from one entry to the other. Every call has a *mean number of calls*, *call-type* and optional *think time* associated with it. The mean number of calls describes how many times an entry is being called. Call types can be either *Blocking, Non-Blocking* or *Forwarding [97]*, and they are shown in diagram using different kinds of arrows. A blocking call has a solid arrowhead, a non-blocking call has an empty arrowhead and a forwarding call has an empty arrowhead plus the call line is dotted. In this thesis, a diamond arrowhead is used when the call type is unknown.

In Figure 2.3, the mean number of calls from *user* to *delay1* and *delay2* are 6 and 1 respectively, meaning that in each cycle the user executes the view operation (through the delay1 path) 6 times and the buy operation (through the delay2 path) 1 time. The call from *checkStock* to *update* is non-blocking (drawn using an empty arrowhead) and rest of the calls are blocking. The default mean no. of calls is 1.

As mentioned earlier, LQN has the capability of modeling blocking calls and concurrency. As a result, it is possible that two tasks compete with each other for the same processor and block each other to get that processor. This may cause a rise in the utilization of a task, because in LQN the utilization of a task, unlike the utilization of a host, is computed considering both its 'wait time' and 'busy time' over the entire period. This makes it possible for a task to become **software bottleneck** while the host to which it is deployed is underutilized (because host utilization is computed considering only 'busy time' over the entire period). LQN behaves like ordinary queuing networks when there are only non-blocking calls and infinite thread pools. LQN models presented in this thesis are closed models.

2.3 Middleware

At one time distributed application developers had to rely on low-level TCP/IP communication mechanisms, such as sockets, in order to establish communication sessions between a client and a server. Such low-level mechanisms require a protocol (i.e., a set of rules) that a client and a server must adhere to in order to carry out their communications successfully. But, design of protocols is error-prone. This motivated the invention of a higher level of abstraction, namely Middleware. In [63], Mahmoud defined middleware as:

"Middleware is a distributed software layer that sits above the network operating system and below the application layer and abstracts the heterogeneity of the underlying environment. It provides an integrated distributed environment whose objective is to simplify the task of programming and managing distributed applications and also to provide value-added services such as naming and transactions to enable distributed applications development. Middleware is about integration and interoperability of applications and services running on heterogeneous computing and communications devices."

2.4 Middleware Classification

The middleware platforms covered in this thesis are classified into three categories: distributed object based middleware, container based middleware and message based middleware.

Distributed Object Based Middleware (DoBM): In a DoBM, communication interfaces are abstracted to the level of local procedure call, or a method invocation. The simplest DoBM-based application comprises two separate programs, a server and a client. The server program creates some remote objects, makes reference to these objects accessible, and waits for clients to invoke methods on these objects. The client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them.

DoBM-based systems are usually synchronous. The client must block and wait until the server method completes execution – and therefore they don't support parallelism except through multiple threads at the server. These systems require the client and the server to be available at the same time. But, such tight coupling may cause a performance bottleneck.

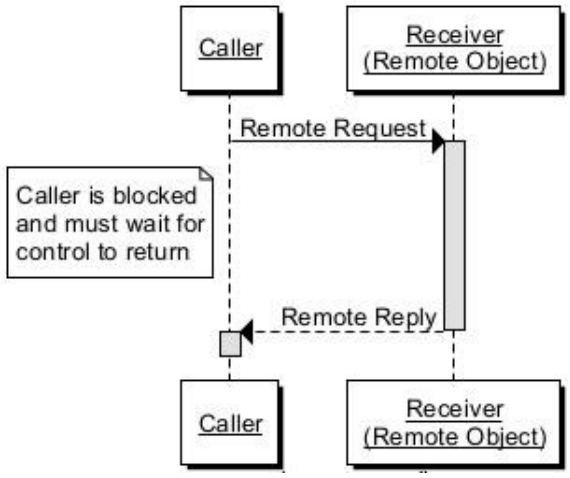


Figure 2.4 DoBM call scenario

The most popular DoBM is Java's Remote Method Invocation (RMI). RMI is suitable for building distributed desktop-applications such as distributed drawing board or chat applications. RMI has two major limitations. First, the communicating objects must run on Java Virtual Machines (JVM). This platform dependency gave rise to platform-independent standard such as Common Object Request Broker (CORBA). But CORBA has limited popularity due to its inherent complexity. More recently, platform-independence is usually achieved through web services. The second limitation of RMI is that, it was not developed for web applications that require efficient object lifecycle management and advanced configurations [21]; these are not easily programmable using the RMI API.

Note that, the Remote Procedure Call (RPC) middleware follows the client-server architecture similar to RMI but does not have the notion of objects. But, because of its architectural similarity with RMI, it is modeled as a DoBM.

Container Based Middleware (CBM): As mentioned, web applications require services for distributed objects such as efficient lifecycle management, URL mapping, dynamic deployment etc. These services are provided by *Web Containers* (or simply *Containers*) supplied by container-based middleware. Popular container-based middleware deriving from *Java Servlet* technology are: Servlet/JSP [44], EJB [26], Spring [86], and Struts [6]. Containers such as Tomcat [90] and Glassfish [70] have been developed by implementing Java Servlet [44] specification. Therefore applications that are developed using any of the aforementioned middleware can be deployed to these containers. Alternatively, applications that use the .Net Framework are deployed to a Microsoft Internet Information Services (IIS) [65] container.

Web Services are also deployed to containers. Web services are services that are platform-independent and consumable by other applications. Web Services are of two kinds: SOAP [85] web services and REST [30] web services. Both of the web services are deployed to web containers to be consumable. SOAP and REST are not strictly considered as middle-ware (they are called "message exchange protocol" and "architectural style" respectively), however for the performance modeling purpose they will be treated as middleware, as they incur overheads similar to other middleware.

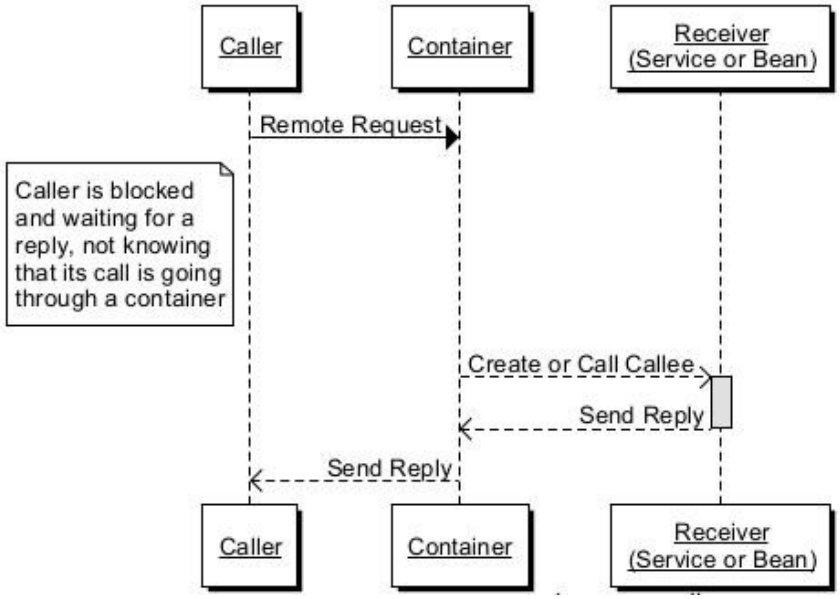


Figure 2.5 CBM call scenario

It is important to note from a performance modeling perspective that in a CBM, for a call to go from a caller to a receiver, it must go through a container that works as a controller. The receiver is only expected to handle certain parts of the total communication process. For example, in case of a Servlet deployed to a Tomcat container, the servlet code itself will never listen for requests on a certain port, nor will it communicate directly with a client, nor is it responsible for managing its access resources. Rather, these responsibilities are managed by Tomcat container [90].

Message Based Middleware (MBM), or Message Oriented Middleware (MOM), is a sub-system that provides the abstraction of a message queue that can be accessed across a network. It generalizes the well-known operating system construct 'mailbox'. MBM systems are flexible, as they allow applications to communicate with each other without requiring same-time availability or blocking the participating processes. A large range of MBMs have been developed adhering to the JMS API, including: Apache ActiveMQ [5], [84] IBM MQ [39]. Some widely known non-JMS messaging middleware are Pivotal RabbitMQ [72], and Microsoft MSMQ [66].

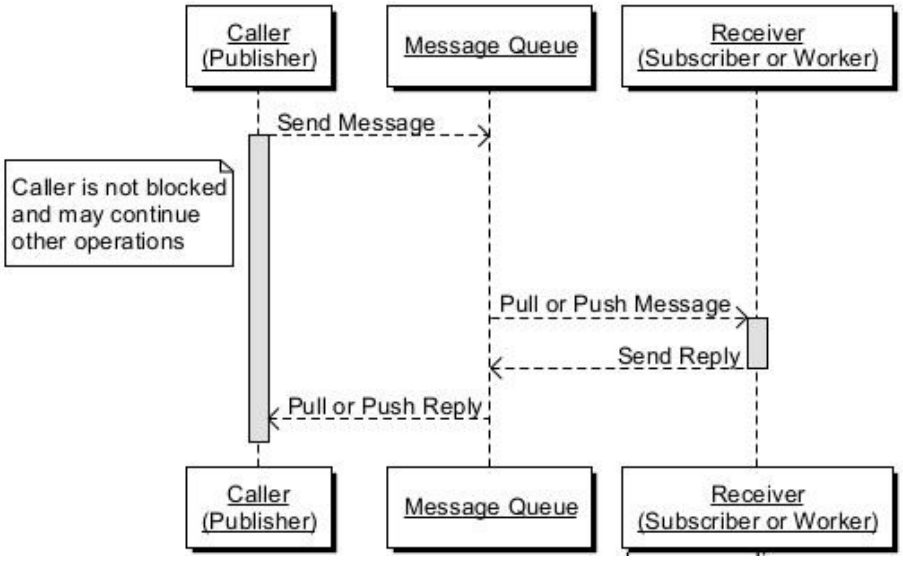


Figure 2.6 MBM call (asynchronous) scenario

Similar to a Container Based Middleware (CBM), the Message Queue of an MBM also intercepts every call from a caller to a receiver. But the difference is: the Message Queue does not wrap the services offered by the receiver, and therefore the Message Queue and the receiver do not appear as a single entity to an outsider. Rather, the Message Queue

'stores' all the incoming requests and dispatches them to the receivers. The communication between the Caller, Message Queue and Receiver is generally asynchronous but can be also configured as RPC-style synchronous [72], [5] and [84].

2.5 Feature Models

Zave et. al. [71] defines that "a *feature* is an increment in program functionality". In this thesis, a feature is defined as "an increment in middleware functionality". Feature models are widely used in Software Product Lines (SPLs) to represent the commonality and variability of a product line [8][22][50]. This thesis uses feature models to represent the possible specializations of a middleware. A middleware specialization is defined by a unique combination of features. One feature cannot appear more than once in a feature model.

Feature Relationship:

A feature model is a tree in which every node represents a feature. The possible relationships between a node and its children are categorized as:

- Mandatory – child node is required.
- Optional – child node is optional.
- Or – at least one of the child nodes must be selected.
- Alternative (xor) – exactly one of the child nodes must be selected

In addition, cross-tree constraints are allowed. The most common are:

- A requires B – The selection of A in a product implies the selection of B.
- A excludes B – A and B cannot be part of the same product.

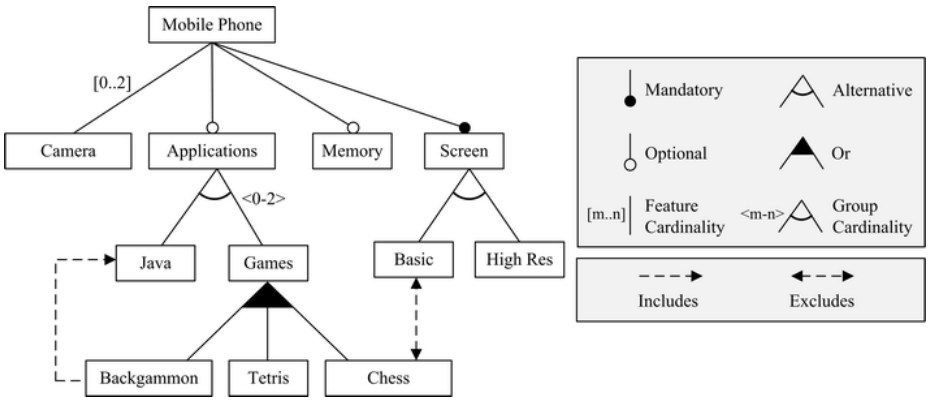


Figure 2.7 Feature Model for a mobile phone family (from [53])

Extended Feature Model:

Feature models do not have any standard specification, so the notations found in the literature sometimes slightly differ from one another. Many researchers have proposed extensions to the feature models described above. In this thesis some of the extensions given by [13], [12], [53] are used:

- Types can be attached to the nodes of a feature [13]. In MMLQ, two such types are used: features that have direct LQN realization models are drawn with solid rectangle, whereas features that do not have direct LQN realization model but their behavior is indirectly captured by the model are drawn with dotted borders. Chapter 3 has examples.
- A child node can be made the default choice of its parent node. This is shown in diagram by adding the keyword ‘default’ in square brackets.
- The child nodes of a feature can be shown in a separate sub-feature model [87]. This is very useful when a feature model becomes too large. A feature that has sub-feature model is shown in thick border and the root node in the child sub-feature model has a reference to its parent feature model.

- A feature-attribute may be written as a value or as a numeric range with an (optional) unit [12], [53]. A default value can be given to a feature-attribute by using the 'default' keyword.
- Some features are conditionally mandatory (shown in figure as a dashed circle, e.g., Figure 3.3). For example, if a performance modeler is using CBM, then the *Container* feature becomes mandatory. The conditions are stated as textual notes to the feature model.

Examples of these extensions can be found in the referred papers, as well as in Chapter 3.

2.6 State of the Art in Middleware Performance Modeling

Many works have been done in performance modeling of middleware [7], [19], [56]. This section addresses only the work that is the most relevant, in terms of approach and context, to this thesis.

Wu et al. [100], [101] proposed a mechanism and a tool for composing subsystems into LQN models. Subsystem models may have numerical parameters, but this is the only form of variability. While this capability may be used to model some kinds of middleware, it has a limited ability to adapt to the context of each interaction. If compared to the proposed framework, Wu et al. modeled only a particular rather complicated kind of structure-modifying realizations.

Verdickt et al. [93], [94] modeled CORBA middleware at the UML level, automatically combined it with a UML model of the application, and finally derived LQN performance models. That is, they did not have any automated composition process for features. The authors solved the automatic composition problem only for middleware, and did that at the UML level. While this has the advantage that it can address middleware issues within the

software design, it has the disadvantage that the performance model must then come from UML (MMLQ can exploit LQN models from any source). Also, they showed the application of their process only for CORBA and an extension for other middleware was not obvious.

Gokhale et al. [34] proposed POSAML, a visual modeling framework for provisioning (i.e., configuration and QoS validation) of different middleware. The POSAML model interpreter takes a visual model of the middleware as input, converts it to XML and obtains performance metrics by simulation, and thus automates middleware-specific QoS validation. Like Verdickt, authors of POSAML also build the middleware models manually and did not consider middleware in general, only CORBA.

Reussner and his co-workers have addressed communications infrastructure within a Component Based Software Engineering framework called the Palladio Component Model (PCM) [9]. Completions are components with variants defined by a feature model, with applications to Message Based Middleware. In [38], this idea gave pattern-based performance completions for MBM, with annotations to drive a model-to-model transformation to configure the completion. In [9], [37] coupled transformations are used to compose completions into both generated code and a performance model. In, [87], [88] an abstract connector model was developed into a particular communications infrastructure in PCM using completions and feature models. In [51], variability is incorporated in the model transformation process (rather than just in the model instances), using generators based on higher-order transformation patterns (configured transformations). These PCM-related works form the most general effort on the problem. While, MMLQ builds models directly using

layered queue performance models that require less information to construct than component-based PCM models. This implies that MMLQ does not require any other models (such as PCM or UML) to transform to performance models.

Sachs et al. [77] built a detailed performance model for a message-based event-driven system. In particular, they deployed an implementation of the SPECjms2007 standard benchmark (which represents a supermarket supply chain system) on JMS middleware, and modeled it using Queuing Petri Nets (QPNs). They also proposed a library of patterns to model different features.

Chen et al. [15] proposed an empirical approach to determine the performance characteristics of component-based applications with CBM. The authors designed a test-suite for measuring the overhead of the container for a class of middleware, and derived a performance model for them. They did not separately model the various components of the container, rather modeled the container as a single performance entity. While this model can be used to predict the saturation points of the system, it does not identify a bottleneck component within the container.

The use of an intermediate model is a popular way to transform software models to performance models as it reduces their coupling. Petriu & Woodside et al. [99] annotated UML software model using a performance profile, transformed the annotated model to an intermediate model which can be easily transformed to multiple performance models. Mirandola et al. [36] did not annotate the software models at the beginning, rather injected platform information to the intermediate model. Cortellesa et al. [20] composed software model with target platform model to produce integrated model, then they annotated the integrated model in UML-RT and directly simulated the annotated model without any need

of transforming it. While these approaches deal with the composition of platform information to software models, they don't have any systematic mechanism to generate middleware models depending on features.

Another line of research is based on Aspect Oriented Modeling (AOM). Alhaj et al. [1], [2] proposed an approach to transform platform independent-model to platform-dependent model in the context of a model transformation chain that generates queueing-based performance models from UML design models of service-oriented applications. The Reusable Aspect Model (RAM) proposed by Keinzle et al. [54] can be used to model middleware variability of functional requirements, whereas this research aims to model the variability of non-functional requirements for performance prediction.

In recent years, a number of works for modeling cloud systems to meet QoS (Quality of Service) requirements have emerged. MODAClouds [67] offers a quality-driven approach and offers basic tools to support DevOps. But, this approach is focused on multi-clouds and NoSQL database scenarios. SeaClouds [78] performs a "seamless adaptive multi-cloud management of service-based applications by developing cloud service orchestrators and a set of tools to manage complex applications". SeaClouds give emphasis on runtime management and it is not specific to performance modeling. The main objective of the U-QASAR [91] project is to "create a flexible Quality Assurance, Control and Measurement Methodology to measure the quality of Internet-related software development projects and their resulting products". The methodology is based on a knowledge service, while the present work is based on completions. DICE [14], [23] considers the question of how quality-aware MDE should support data-intensive software systems. The project aims to better understand big data properties such as volume, velocity and data location from a modeling

perspective, and to develop quality engineering toolchain for them, while the present work is not related to big data. None of these cloud-related projects is concerned with the performance impact of middleware as defined here.

The present work is different in three important ways from its predecessors.

First, it considers general performance model completions, not just completions as components. A model completion is both more general and simpler than a component or connector completion, in that it often penetrates and modifies the model of the software entities that use it, besides possibly providing additional components. A component is both a system structural object and a model object; here the completion is only considered in the performance model and its implementation is not a complicating concern. This in general may give a simpler (often a very much simpler) model to be solved and understood.

The second difference is that the completion is defined and composed at the performance model level, not in the software specification. It does not require a specification model to be available.

The third difference is that the proposed approach gives the option of modeling a feature as either an added structure or as a property. As a result, the performance modeler has control over the model size which is absent in the previous works.

The differences above give two advantages. First, the resulting performance model is smaller and simpler. Second, design details may be confidential or simply unavailable, and for such cases a performance model may be constructed as a “black box” and its middleware then can be composed using the methods proposed here.

Finally, the proposed framework can model a wider range of middleware (middleware platforms of three different categories) comparing to the existing techniques.

3 Chapter: Middleware Feature Modeling

The fundamental operation of every middleware is to carry out a distributed call. It is features added to this basic operation that make one middleware different from another.

MMLQ considers that features are added to some Base Middleware Model (BMM), as indicated in Figure 3.1 (which is repeated from Chapter 1, Figure 1.1). The first and simplest BMM is an empty model (EMM) that just represents the call itself. This chapter starts with the EMM and builds up models of the various features present in different middleware.

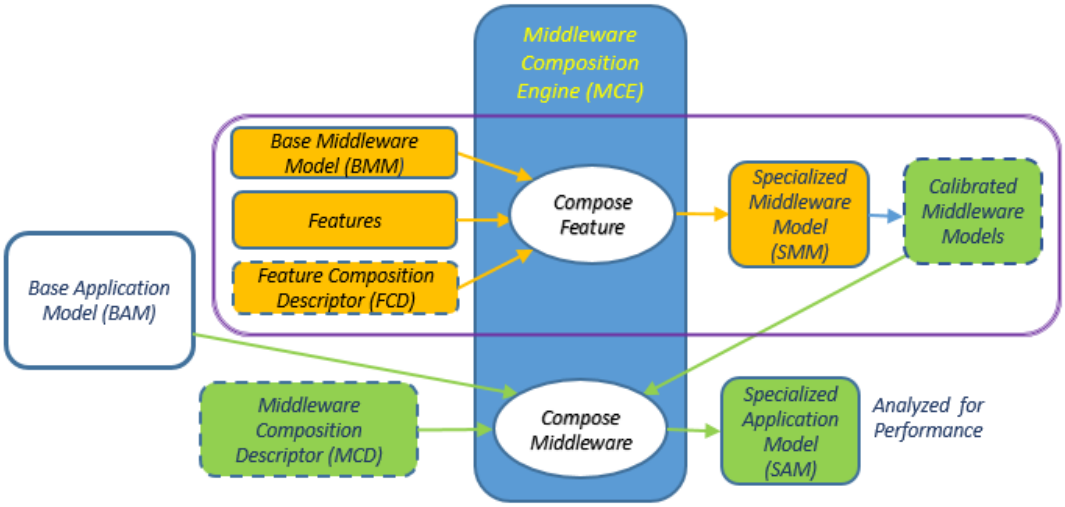


Figure 3.1 The MMLQ framework

3.1 Empty Middleware Model (EMM)

An EMM represents a single distributed call with no middleware. It contains all the parameters that are related to a single call in an LQN model. As described in Chapter 2, a call requires at least a caller task and a receiver task. These calls are deployed to their corresponding hosts, as it was shown in Figure 2.3.

MMLQ models follow several conventions. All task names start with a capital letter, all entry names start with a small letter and all host names start with an underscore (_). The double bar (||) before the name of a model element tells that this element is a placeholder and it is going to be replaced by the corresponding caller or receiver element in the composed model. A dollar sign ($) before a variable name signifies a placeholder variable which requires an input value. Think time is represented in square brackets. All the tasks and hosts in this model have default multiplicity of 1. A diamond arrowhead is used for the call to represent the fact that the call type is unknown and will be derived from the base application model, to which this middleware will be composed.

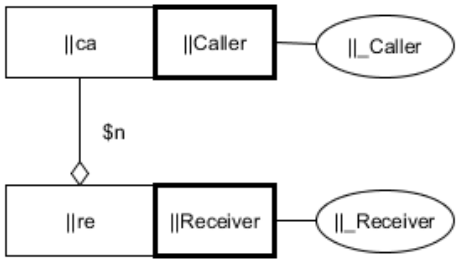


Figure 3.2 Empty Middleware Model

The elements of an EMM are described below.

- **Caller:** A caller task *Caller* has a single entry *ca*. The service demand of *ca* is 0 (and hence not shown in the diagram) in the EMM since no feature has been introduced yet. Task *Caller* is deployed to host *_Caller*.
- **Receiver:** A receiver task *Receiver* has a single entry *re*. The service demand of *re* is 0 as well since no feature has been introduced yet. Task *Receiver* is deployed to host *_Receiver*.
- **Call:** There is a single call in the EMM that goes from entry *ca* to entry *re,* identified as *ca>re*. A call has four properties. They are:

i) call type – which can be either blocking, non-blocking or undetermined. The default call-type is blocking, represented as a solid arrow-head. Figure 3.2 uses a diamond arrowhead to indicate that the call type in EMM is undetermined, it is determined when the middleware model is composed into an application.

ii) think time – which can take any real value (represented within square brackets). The default value is 0.0. Omitting think time from the figure indicates the absence of any think time.

iii) mean number of calls – which can take any real value (represented as $n in Figure 3.2). The value of $n is determined when the middleware model is composed into an application. If the mean number of calls is not mentioned, it is assumed to be 1.

iv) message size represented as $msz (not shown in this figure)

The LQN properties (e.g., service demand, task and host multiplicities) of the role entities for the caller task/host and receiver task/host and the call are determined when these roles are bound into an application model.

3.2 Middleware Features

In Chapter 2, middleware platforms were classified into three categories. In this chapter the commonality and variability of these three categories are described. To aid the discussion, features will be described as they occur in one middleware platform from each category. The categories and the middleware platforms to be described are:

Table 3-1 Middleware to be described

Category	Description	Middleware Platform
DoBM	Distributed Object-Based Middleware	Java Remote Method Invocation (RMI) [42], [21]
CBM	Container-Based Middleware	SOAP (Service-Oriented Access Protocol) [85]
MBM	Message-Based Middleware	Apache ActiveMQ [84], [5]

3.2.1 Description of Features

In MMLQ, a feature is defined as an increment of middleware functionality. Some features have direct LQN realizations (shown by solid borders in feature models), while others do not have direct LQN realizations (shown by dotted borders in feature models) but they are implicitly modeled by LQN through other elements. Feature realizations are described in Chapter 4.

The features described in this chapter were identified by a study of middleware specifications made by the author, seeking commonalities between their specifications.

Wrapper: A remote invocation requires the caller to send request to the receiver in a specific format, a format the receiver understands. The Wrapper feature needs to be used by a client to make distributed calls to any middleware.

In Java RMI caller and receiver communicate via two interfaces called *stubs*. A stub on each side works as a proxy of its remote object. The caller sends requests to its stub as if it is calling the remote object. The caller stub marshals/serializes the request to a format understandable by the receiver stub. The receiver stub unmarshals/deserializes the received request and passes it to the receiver. The receiver carries out its operation and replies using the same marshal/unmarshal mechanism. Therefore, for RMI, marshal/unmarshal are the wrapper operations. Similarly, for RPC, serialize/deserialize are the wrapper operations.

In container based middleware platforms such as Servlets, EJB, SOAP, REST, etc., requests are wrapped as HTTP messages by the caller and unwrapped by the receiver container. In case of SOAP, web services can be exposed as a single java class known as *Service Endpoint Interfaces (SEI)* and deployed in a container. In Message based middleware platforms (e.g., JMS, ActiveMQ), the caller may wrap its message following different network protocols (e.g., TCP, NIO, UDP etc.) that the message queue is able to unwrap and operate on.

In a distributed object based middleware, both the caller and the receiver must use Wrapper feature to model the overhead of distributed calls. In case of a message based middleware and container based middleware, the client that makes calls to those middleware platforms require Wrapper feature, but the wrapper operations at the servers are taken care by the Message Queue and the Container respectively; therefore an explicit composition of the Wrapper feature is not required.

Name Service: Name services store information in a central place that users, workstations, and applications may require to communicate across the network. Name services are fundamental to any computing network.

Among other features, a name service provides functionality that associates (binds) names with objects, resolves names to objects, removes bindings etc. In RMI, the name service is provided through an *RMI Registry*. In SOAP, information about a service is published in the *WSDL* (Web Service Description Language) format and accessed through a *UDDI* (Universal Description, Discovery, and Integration) registry. The Java Naming and Directory Interface (JNDI) is used by name services for a large array of middleware platforms including JMS, ActiveMQ, etc.

Security: A middleware can provide communication security in many ways and sending encrypted messages is one of them. Encryption allows sensitive information such as credit card numbers, social security numbers, and login credentials to be transmitted securely. In distributed applications, encryption is often provided by the use of the SSL (Secure Socket Layer) protocol, which determines variables of the encryption for both the link and the data being transmitted [25]. The caller and the replier need an *SSL Certificate* to be able to establish a secure connection. Every middleware offers encrypted message transmission over the SSL protocol.

Middleware platforms allow the programmers to use different kinds of security implementations. Three such kinds [79], [80], [81] are commonly found: *Simple Security*, *Subject Delegation and Fine-Grained Security*. In Simple Security, each different user requires one secure connection for every operation it performs. In Subject Delegation, a single secure connection for some user is created, and this connection can be used to perform related operations on behalf of any number of users. This is useful when the application requires the client end of the connection to perform different operations on behalf of multiple users or applications. Subject Delegation reduces the load on the client system. In Fine-Grained Security, permission to perform individual operations is controlled.

In Chapter 6 of this thesis, experimental results using SimpleSecurity feature are presented. SimpleSecurity can be implemented using many different techniques such as Custom Socket [92], Simple Authentication / Basic Authentication [84], Digest Access Authentication [24], Asymmetric Cryptography [76], OAuth [69], JSON Web Token [47] etc.

MMLQ considers the SimpleSecurity's Basic / Simple Authentication as the default implementation of Security feature. Therefore, when a realization of the Security feature will

be talked about, it will refer to SimpleSecurity feature's Basic / Simple Authentication (unless stated otherwise).

Compression: In many distributed systems, sometimes it has been observed that the bottleneck is neither the server nor the client, rather the data transfer [17]. In such circumstance, message compression can reduce the load on the network. Every middleware has a message compression feature, often built in. If a built-in compression feature is not provided, it can still use custom compression features. RMI has ZipInputStream and ZipOutputStream classes, ActiveMQ has a doMessageCompression flag, and SOAP has a GZip-based compression service.

Session: HTTP is a stateless protocol. But, quite often the server needs to know that the subsequent requests are made by the same client; the sequence of requests is called a Session. A session feature keeps user information at the server side. In RMI the session feature is provided by a singleton RMI object, bound in the Registry, with nothing but a *login* and a *logout* method. In a container based middleware, the session feature is provided by the container. Containers such as Apache Tomcat, Oracle Glassfish, JBoss container and Spring container all provide a session feature through an instance of the *HttpSession* class. In ActiveMQ, a session is managed through an instance of the *ActiveMQSession* class.

Transaction: A transaction is a set of operations in which either all the operations are carried out or none of the operations are carried out, according to some criterion [21]. In other words, only a partial set of operations can never carry out. For example, transferring money from one account to another account is a transaction containing two operations: executing a debit operation in one account and executing a credit operation in another account.

Some middleware comes with in-built transaction service feature and others do not. Distributed object based middleware (DoBM) such as Java RMI does not offer a transaction service, but the callee of the DoBM can be deployed to a container that offers transactions (e.g. an RMI based application can be deployed to WebLogic container that provides a transaction service). If this is the case, then that DoBM acts like a CBM (Container Based Middleware) and its performance models should be similar to a CBM. All container based middleware supports transaction services through their container and this service is one of the most significant aspect of any container. ActiveMQ provides transaction services through JMS Transaction API.

It is to be noted that a feature is used not only to represent an added operation, but sometimes also an inverse operation. For example, when a Compression feature is added to a call, the caller compresses (i.e., operation) the message before sending and the receiver decompresses (i.e., reverse-operation) the message upon receiving it. However, in our description of features, instead of giving two names (e.g., Compression and Decompression) for the two operations of the same feature, they are referred by a single name (e.g., Compression). The operation of the feature is understood by the context.

3.2.2 Conditional Mandatory Features

While Wrapper is a *Conditional Mandatory Feature* for DoBM, it is also used by all the distributed clients to make calls to the server. There are some other features that are mandatory only for a certain sub-group of middleware platforms. Two such conditional mandatory features are as follows:

- the *MessageQueue* feature is mandatory for every message based middleware.
- the *Container* feature is mandatory for every container based middleware.

Message Queue: A message queue acts as a mailbox and allows applications to communicate with each other without requiring same-time availability. At one end of the message queue, one or more caller processes called *producers* send messages to it. At the other end, one or more receiver processes called *consumers* consume and process each message. In message oriented middleware platforms such as JMS, ActiveMQ, RabbitMQ, etc., the message queue is deployed as a server, and producers and consumers connect to the message queue through a protocol (e.g., AMQP, TCP, NIO, etc.) and exchange messages.

Container: A container is responsible for creation, destruction and lifecycle management of the objects at the server. Every container based middleware uses the services provided by a container to carry out remote invocations. There are many containers such as Tomcat, Glassfish, WebLogic etc. A container based middleware is usually not restricted to be deployed to a particular container but every container based middleware has one or more recommended containers. For example, Servlet and JSP are usually deployed to Apache Tomcat, SOAP web services in Java to GlassFish and RESTful web services in Java to Apache Tomcat, as well. Apart from creating and destroying objects, a container may offer many other features such as compression, security, session, name service, message storage, etc.

Some container based middleware platforms come with their own specialized containers. For example, Spring has its built-in *IoC (Inversion of Control)* container. Quite often in business applications an object contains the references of other objects, and thus is dependent on those objects. Spring takes the responsibility of creating the referenced objects first and then to inject them to the parent object (known as *dependency injection*), reducing programmer's responsibility of instantiating new objects respecting their dependencies.

Every external request coming to the Spring server has to go through the IoC container. So, from a performance modeling perspective, the IoC container has similar overhead patterns to other containers.

For a message based middleware and a container based middleware, the Message Queue and the Container features handle the server operations that a Wrapper feature does in a distributed object based middleware server.

3.2.3 Message Queue Specific Features:

There are some features that are specific to message queues and therefore only found in message based middleware. Most of the popular message based middleware platforms are JMS compatible (e.g., ActiveMQ, Websphere MQ, IBM MQ) and they all have a broker cluster feature.

Broker-Clustering: Many message queue brokers can work together as a federated network of brokers for scalability purposes. This is known as a network of brokers and can support many different topologies.

Distribution Policy: A message queue can distribute a received message in two ways. First, it can work as a *load balancer* and dispatch the message to one of the consumers that is available to process the message. This kind of message distribution is known as *queue*. Second, a message queue can work as a *broker* and dispatch the message to all the consumers that have expressed an interest in that message. This kind of message distribution is known as *topic* (and is the MMLQ default). The broker may be also configured as a broadcaster, in which case it broadcasts any message it receives.

Dispatch Policy: A message queue can dispatch messages to the consumers either by a pull policy or a push policy. In a *pull* policy the consumer requests message to the message-

queue and remain blocked until a message is received. If this consumption policy is adopted by a consumer, in JMS/ActiveMQ vernacular it is said the consumer is *synchronous* (MMLQ default). In a *push* policy, the consumer subscribes as a listener to the dispatch event of the message queue and a message is pushed to the consumer whenever the message queue is ready. If this consumption policy is adopted by a consumer, in JMS/ActiveMQ vernacular it is said the consumer is *asynchronous*.

Prefetch Limit: A message queue may be configured to keep sending messages to a consumer without getting acknowledgement of the previous messages. The maximum number of messages that a message queue can send without requiring to receive any acknowledgement is known as its *Prefetch Limit*.

Message Storage Mode: A message can be delivered either as *persistent* or *non-persistent*. In a persistent delivery, messages are stored in a disk or database so that they will survive a broker restart. When using non-persistent delivery, if the broker fails then all in-transit messages are lost. The default storage mode for all JMS compatible middleware is persistent.

3.2.4 Container Specific Features:

Some features that are found in most web containers are described below.

Clustering: Clustering refers to a group of containers that appears as a single entity to its clients. Clustering provides scalability by spreading work out over a many workers. Usually, clustering is achieved through a concept known as *virtual hosting* in which each container is deployed to its virtual host.

Load-balancing: Load-balancing is similar to clustering. A load balancing implementation attempts to route requests to the server with the least amount of current load, for faster

processing. The load balancer also provides *session persistence*, which ensures that the information from an individual user's session is always available, even if the server currently hosting their session goes down, so that application state is maintained. To achieve session persistence, many load-balancers, including Apache's *mod_proxy* and *mod_jk*, uses *session stickiness*, in which the load balancer remembers which cluster worker is storing the session information for each client's request, and proxies all concurrent requests from the same client to the same worker. From a modeling perspective, load balancing is a very special feature that can change the scheduling type of the host. In this thesis, such a feature is referred as a *ReschedulingFeature.* To model a ReschedulingFeature, the scheduling algorithm (default is FIFO) of a task in the LQN model has to be changed accordingly [31][97].

Content Negotiation: Content negotiation is a mechanism defined in the HTTP specification that makes it possible to serve different versions of a document (or more generally, a resource representation) at the same URI, so that user agents can specify which version fit their capabilities the best. One classical use of this mechanism is to serve an image in GIF or PNG format, so that a browser that cannot display PNG images (e.g. MS Internet Explorer 4) will be served the GIF version.

It is to be noted that if a distributed object based middleware is deployed on a container (e.g., web server) then that middleware can take benefit of the services provided by the container and its performance behavior would resemble a container based middleware.

3.3 A Unified Feature Model of Middleware

A single feature model has been constructed to cover all the types of middleware discussed in this thesis. It starts with a base feature model (BaseFM) that shows all the features offered by a simple middleware, as described in Section 3.2.1. A performance analyst will describe his / her choices about middleware by choosing features from the feature model. Since, distributed object based middleware is the simplest type of middleware, the features presented in BaseFM (Figure 3.3) are also the features of a distributed object based middleware. Sometimes a feature has multiple sub-features as children and one of the child is designated default by using the *default* keyword. Features that have a direct LQN realization have solid borders. On the contrary, features that do not have direct LQN realizations, having their behavior implicitly captured by other model elements, are shown with dashed borders.

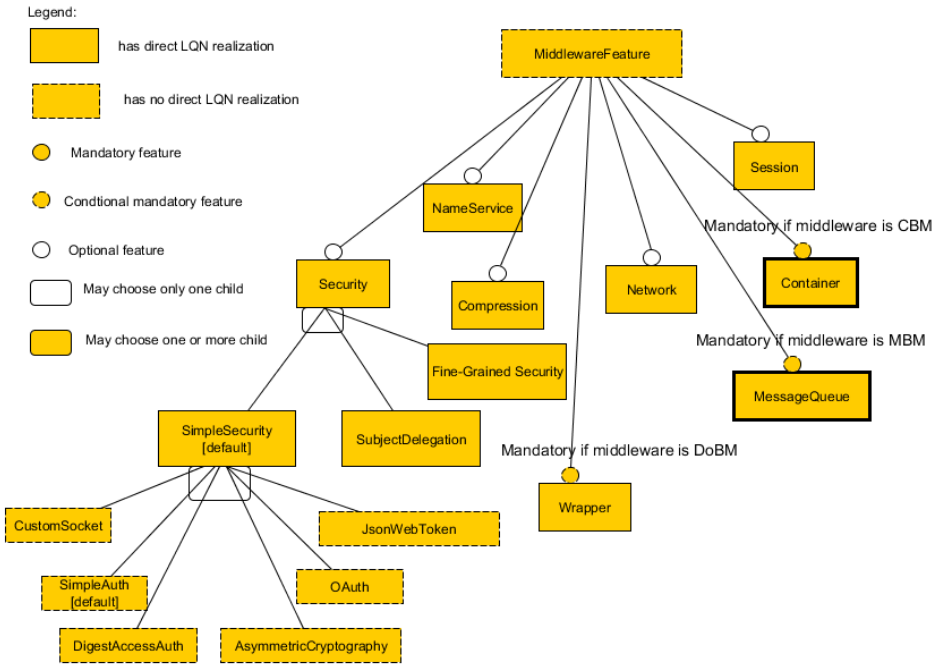


Figure 3.3 Base Feature Model (BaseFM)

There are three conditional mandatory features: *Wrapper* is mandatory for DoBM; if one is using either a message based middleware or a container based middleware, one must choose either the *MessageQueue* or the *Container* feature, respectively. The Security feature can be used either as SimpleSecurity, SubjectDelegation or Fine-Grained. SimpleSecurity can be implemented using many different techniques which are shown as sub-features.

The Network feature, though not a middleware feature but rather a property of the platform, is included in the feature model because middleware platforms are used through networks and network latency is treated as a feature property in MMLQ. Network latency is a necessary element to properly model the performance of any distributed application [29].

MessageQueue and *Container* have all the basic features shown in Figure 3.3, plus the sub-feature models shown in Figure 3.4 and Figure 3.5, based on the discussion of their features in Sections 3.2.3 and 3.2.4.

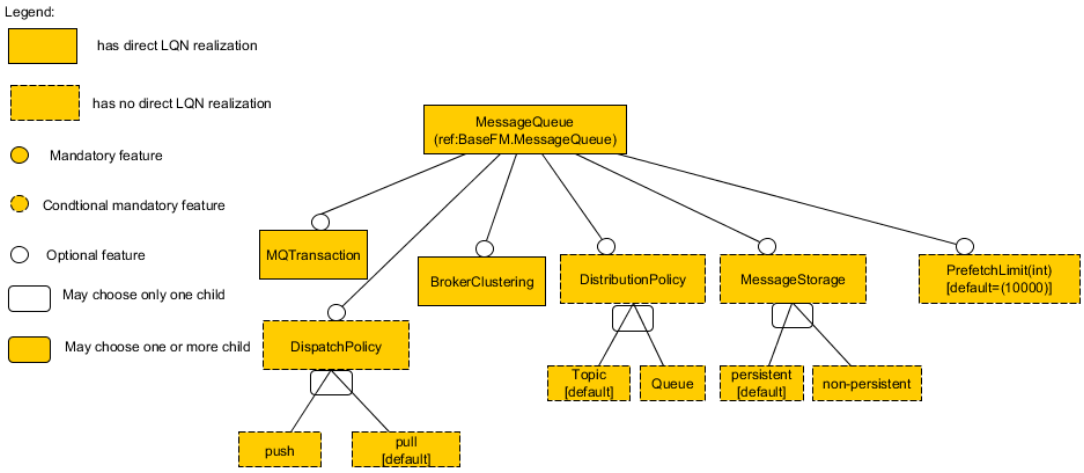


Figure 3.4 Message Queue Feature Model

The *ref* keyword followed by a feature name tells which node is the parent of the root node of the corresponding subtree. In Figure 3.4, the parent node of the root node *MessageQueue*

is the base feature model's *MessageQueue* node. *MessageQueue*'s *PrefetchLimit* feature takes an integer as parameter that specifies the prefetch limit of the message queue. The default value of this variable is taken from ActiveMQ, which is 10,000.

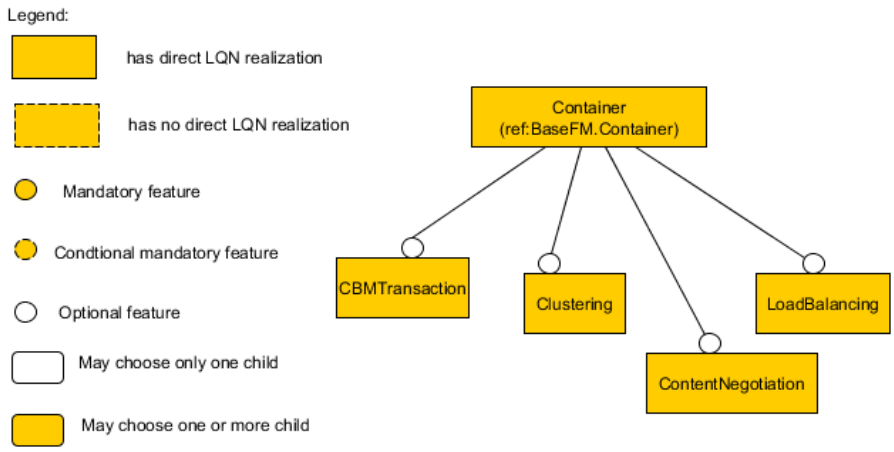


Figure 3.5 Container Feature Model

Pros and Cons of using a Feature Model: Feature models convey a lot of information about choices as simple trees. The feature model presented above do not cover all possible middleware features, but do include the popularly used features and the ones that may significantly impact middleware performance. Other features could be added. For example, if a performance modeler feels that whether the producer has a flow-control policy or not has a significant impact on the performance of a *MessageQueue*, he/she can add a *FlowControl* feature as a child under *MessageQueue*. The chosen features represent a trade-off between model complexity and completeness.

4 Chapter: Feature Realization and Composition

The features described in the previous chapter are realized as LQN models in this chapter. A process for composing the realized features to any base middleware model is described. The discussion includes feature composition properties, a feature composition algorithm, and feature composition descriptors.

4.1 Feature Realization

The Base Application Model (BAM) and Base Middleware Model (BMM) of MMLQ are LQN models, and the Feature Realization Models (FRM) are also defined in LQN.

A feature can be realized in two ways: a Property-Modifying Realization (PMR) which does not introduce new components, and a Structure-Modifying Realization (SMR) which does. Some features are naturally one or the other (e.g., a NameService feature has a structure-modifying realization by nature), and sometimes the modeler has a choice. A PMR can also approximate an SMR. The two kinds of realizations are described below.

4.1.1 Property-Modifying Realization (PMR)

A property-modifying realization does not modify the structure of a middleware model but only modifies the parameters of entries involved in a call. Many property-modifying realizations are defined simply by an additional demand in some model entry, which might be represented as a linear function of message size:

```
demand = (fixed part) + (message size)*coefficient
```

For property-modifying realizations defined by entry demands and for this form of function we will use the conventions illustrated here for the Wrapper feature:

```
Wrapper.demand = $WRAPPER + $msz * $wrapper
```

where `$msz` will be used consistently as a variable representing message size. This message size is the summation of request and reply messages sizes.

A property-modifying-realization modifies one or more model parameters. The property-modifying-realizations of Security (i.e., SimpleSecurity), Compression and Session (using capital letters for the fixed part and lower-case for the variable part, and `$msz` for message size) are given below. The parameter values are determined by calibration experiments (see Section 6.2).

```
Security.demand = $SECURITY + ($msz * $Security)

Compression.demand = $COMPRESSION + ($msz * $compression)

Session.demand = $SESSION + ($msz * $session)
```

Service demands of realizations are not limited to this format. Some realizations may have fewer or more parameters, for example, the service demand of a message queue depends on number of incoming and outgoing calls as well.

PMRs keep models compact. They are useful when modeling the concurrency of a feature is not essential. This will be later demonstrated by examples in Section 4.3.

4.1.2 Structure-Modifying Realization (SMR):

In a structure-modifying realization, a feature is modeled as a separate thread, meaning it is executed by its own entry on its own task, optionally deployed to its own host. These new components (i.e., entry, task and host) are referred as *feature entry*, *feature task*, and *feature host* respectively. Therefore, at the cost of larger models, structure-modifying realizations allow to model concurrency of the feature implementation (if it is present in the system). The structure-modifying realization of the Security feature is shown below:

Figure 4.1 Security Structure-Modifying Realization

Many structure-modifying realizations also take the same structure as Figure 4.1, with suitable parameters. The following figures present the structure-modifying realizations for the Container, MessageQueue (MQ) and NameService features.

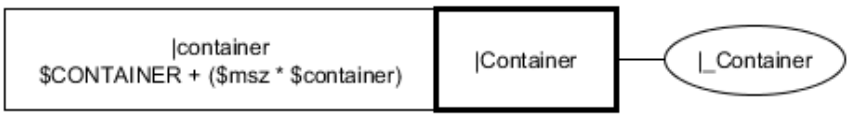


Figure 4.2 Container Structure-Modifying Realization

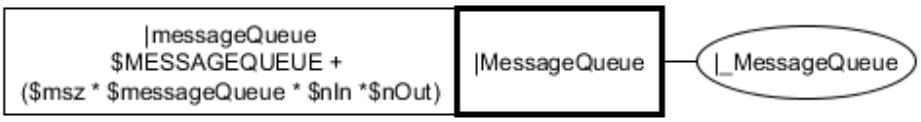


Figure 4.3 MessageQueue (MQ) Structure-Modifying Realization

Figure 4.4 NameService Structure-Modifying Realization

The Container feature has exactly the same structure and service demand format as Security. However, the service demands of MessageQueue and NameService are different from the common pattern. A MessageQueue processes the incoming messages (which may come from one or more callers) and dispatches them to many receivers. So, its service demand depends also on the number of incoming (*$nIn*) and outgoing (*$nOut*) participants to a MessageQueue. This effect is captured in its service demand equation.

A NameService works as a service discovery feature that takes a service name and returns its URI. The source of delay caused by such a service depends on locating the service but not on the message size. Therefore, the variable part of NameService feature is removed. A NameService feature does not add overhead to existing calls, rather a new call is added

to the middleware model to invoke the NameService. In MMLQ, this property of a feature is known as *unbound* property. The feature composition algorithm (Section 4.2.4) treats unbound features specially.

Network feature is modeled as an infinite server, following the recommendation of [59], [97]. An infinite server provides a pure time delay ($NETWORK) and is modeled as an infinite task running on an infinite host (using the multiplicity *{inf}*).

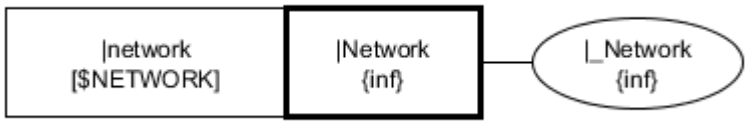


Figure 4.5 Network Structure-Modifying Realization

It would also be possible to incorporate a model with network contention by using not an infinite task but single task with a proper discipline.

4.1.3 Choosing between Property-Modifying Realization (PMR) and Structure-Modifying Realization (SMR):

One may ask how the performance modeler would know whether to use the PMR or the SMR of a feature for the middleware in use. The answer depends on the nature of the application and also on the goal of performance modeling. In other words, the choice of realization depends on the context. Some pros and cons of both kinds of realizations are presented below. This should help the performance modeler in making the proper decision about the feature realization.

- PMRs keep the model small, easier to maintain, easier to understand and faster to solve. Therefore, in general it is recommended to use PMR, unless otherwise required. The benefits of having a smaller model was described in [40].

- In SMR, a feature is modeled as a separate task. Therefore it gets a separate queue and it can request processor time in competition with other tasks. In other words, SMRs can model the concurrency of requests. If the concurrency of the feature is crucial to the performance of the middleware, then it needs to be modeled as an SMR, otherwise a PMR is preferred.
- When it is known from experience that a feature has almost no possibility to becoming a bottleneck, then it can safely be modeled as a PMR.
- If it is impractical to measure the exclusive overhead caused by the feature, then it may be modeled as a PMR.
- If a feature is a very large and distinct software entity with heavy execution demands, then it is preferable to model it as an SMR. A MessageQueue is an example.
- If a feature is likely to cause an important amount of delay, then it is recommended to model it as an SMR (e.g., the Compression feature for video files).
- If a feature is deployed to its own host, then it should be modeled as an SMR (e.g., NameService).

Before describing the feature composition process, an important property of a feature will be introduced, known as its 'order-value'. Order-value controls the order of composition operations.

4.1.4 Feature order

Careful study of the specifications [18], [26], [41], [42], [44], [73] and the literature [21], [89] related to the various middleware platforms show that their operations (i.e., features) follow a certain order. This has been found true for all three kinds of middleware platforms

of interest: Distributed Object Based Middleware (e.g., RMI), Container Based Middleware (e.g., SOAP) and Messaged Based Middleware (e.g., ActiveMQ).

The order of operations found for most middleware are as follows. NameService feature is used for service discovery, therefore it must be invoked before using any other feature of a middleware. Similarly, no further operation can take place on a secured (e.g., encrypted) message, so security is the last operation by the sender before sending an encrypted message. Compression is the operation that takes place just before that. The Wrapper operation (i.e., marshalling) refers to the packing of the final message to be sent, therefore it is the last operation that take place before security and compression. Session and Transaction features may precede or follow one another depending on the implementation of the middleware. LoadBalance is one of the first operations carried out by a message controller, such as Message Queue or Container, and therefore it should immediately follow the basic operations of a Message Queue and Container.

Following the description above, the features are composed in the order of execution, using an "order-value" as shown in Table 4-1. At the calling end of the call, the features are composed by ascending order-values, and at the accepting end, by descending order-values, since features tend to be layered (the highest layer is composed first at the sender and last at the receiver).

The network feature is never used in conjunction with any other feature, rather it is kept separate finally to be composed directly into the base application model. Therefore, an order-value of network feature is not needed.

The default order-values of the features are presented in the table below.

Table 4-1 Default order values of features

Feature	**Order-value**
NameService	10
MessageQueue	20
Container	30
LoadBalance	40
Session	50
Transaction	60
Wrapper	70
Compression	80
Security	90

Note that:

- The order-value has impact in composition only in the case of Structure-Modifying Realizations (SMR), because only SMR (but not PMR) preserves the order of operations;
- Apart from any feature that has a high probability of becoming a bottleneck or drastically changes one of the performance model parameters (e.g., message size), it is highly unlikely that the order of operations will change the performance metrics (i.e., output) of a model;
- If required, the default partial order can be overridden during the composition process, as discussed in Section 4.2.1.

4.1.5 Feature Realization Properties

From the discussion of the feature realizations, it is understood that every feature realization contains some properties. These properties are useful as they will be used by the composition process while composing the features. The feature properties are discussed below.

- Name: name of the feature;
- Type: PMR or SMR;
- Bound: UnBound or InBound;
- Feature order: an integer value.

Name refers to the name of the feature (e.g., Wrapper, Security etc.). Type is the type of the realization (property-modifying realization or structure-modifying realization). Bound refers to the bound type of the feature. All features discussed here are InBound except for the NameService which is UnBound. An UnBound realization itself becomes the receiver of a call and therefore does not require any existing call to be composed with. Feature order was described in the last section.

4.2 Feature Composition

The goal of the feature composition process is to compose one or more feature realizations to a Base Middleware Model (BMM) to obtain a Specialized Middleware Model (SMM) that is suitable to the application context where it is going to be used. In order to carry out the composition process one must state: a) what to compose, b) what "structured way", c) where to compose it. To carry out these tasks, three elements are introduced: *Feature Composition Properties*, *Feature Composition Descriptor* and a *Feature Composition Algorithm*.

Feature Composition Properties describe where and how a feature will be composed to a middleware model. For example, whether it is property-modifying or structure-modifying realization, whether the feature is to be deployed to its own host, whether the call from the feature is blocking or non-blocking etc. Each feature has a list of feature composition properties.

When a feature is specified for composition, its composition properties are assigned values, taken from a range of predefined values. The composition properties along with their values are described below.

4.2.1 Composition Properties of a Feature

The following set of composition properties governs each feature in each middleware model. SMR feature realizations introduce a new task, called its "feature task", which is referenced by some of these properties.

Feature realization type: The intended realization type of a feature can be either **PMR** or **SMR**. If a realization type is not mentioned for a feature composition, PMR is taken as the default realization type.

Feature destination: The intended call or task to which a feature realization should be composed. The destination can be either a call's caller task, call's receiver task, both caller and receiver tasks (which is default), or to a call itself, where a feature task is added as a separate entity (for example, a MessageQueue is not attached to any single caller or receiver). So, its values are **caller**, **receiver**, **both** or **call**. An SMR can take any of these four values, while a PMR can take all values but call, because it does not have a feature task.

Feature host: For SMR describes on which host the feature task is deployed. The host value can be **self**, **bound** or **shared. '**Self' means the realization task is deployed to its own

separate host. ‘Bound’ (which is the default) means the realization task is deployed to its destination task’s host. ‘Shared’ is used in a special case when a feature is composed more than once in a model and these feature tasks are deployed to the same shared host (for example, when a dedicated compression server is used by two different server side components). Note that, PMRs are always of ‘bound’ type.

Outgoing call type: While the call type of any incoming call to a feature task is determined by the task that is calling a feature (i.e., it comes from the base application model), the outgoing call type of a feature can depend on how the feature is used in the middleware. A call-type can take one of the two values: **block** (default) or **nonblock**, respectively, referring to blocking and non-blocking call types.

Mean number of outgoing calls: The mean number of incoming calls is determined by the task that calls a feature task, but the feature may send multiple outgoing calls (for example, if a message queue broadcasts incoming messages). The mean number of outgoing calls of a feature may take any positive real number (default is 1.00).

Feature task multiplicity: Number of threads of a feature task. May take any integer value (default value is infinity).

Feature host multiplicity: If a feature is deployed to a dedicated host (i.e., if the feature host type is ‘self’ or ‘shared’), then the multiplicity of that host can be changed from the default value of 1 to any integer number.

Feature scheduling: Some feature such as a Load Balancer may require to change the default host scheduling. MMLQ allows to change the scheduling of a feature destination (default scheduling is FIFO). Acceptable scheduling in LQN tasks are FIFO (First-In. First-Out), PPR (Priority, Pre-emptive Resume), HOL (Head Of Line priority) [59], [97].

Order-value: The default order-value of a feature can be overridden by any positive integer value.

4.2.2 Feature Composition Descriptor (FCD)

The features to be composed to a base middleware model, along with the composition properties of each one, are described in a **Feature Composition Descriptor (FCD)** for each middleware model. The FCD is provided as a text file to the Middleware Composition Engine (MCE), in the format given below. The keywords are shown in green colors. Optional parameters are written inside square brackets. An element starting with a dollar sign ($) is a placeholder for the appropriate model name, feature name, property or value. Listing 4.1 shows the template of an FCD. A formal description of FCD in extended BNF form is presented in Appendix A

Listing 4.1 Template of a Feature Composition Descriptor (FCD)

```
FCD [$nInOutBlock]?
in $BaseMiddlewareModel  at $callerEntry $receiverEntry
    put $FeatureName [-$PropertySwitch $Value]*
    /* use "&&" to specify more features */
out $SpecializedMiddlewareModel
/* provide an in-out block for each call */
```

The first line of the descriptor has the keyword 'FCD', indicating the descriptor type. The keyword 'FCD' is followed by an optional integer `$nInOutBlock` that describes how many *in-out blocks* will be there in this FCD. The attached superscript question sign ($^?$) in the template indicates there can be either zero or one number to follow the keyword 'FCD'. The next line starts an in-out block, which defines a composition step to add a set of features to a call of a middleware model. Another in-out block is needed to add a set of features

to another call of the middleware model. An FCD may contain as many in-out blocks as required, but, the number of in-out blocks has to be specified by the `$nInOutBlock` variable. If a value for `$nInOutBlock` is not mentioned, then it is assumed to be 1.

An in-out block starts with a line beginning with the "in" keyword and ends with a line beginning with the 'out' keyword. The 'in' line defines the input middleware model as the value of the placeholder `$BaseMiddlewareModel`. The call to which the middleware will be composed, called the *composeCall*, is identified by the names of its sending and receiving entries, which are the values of the placeholders `$callerEntry` and `$receiverEntry`, separated by a whitespace. If one of these entries is not required, the value is given as "`?`" (e.g., for the NameService feature, as described later).

The list of required features is specified between the 'in' and 'out' declarations. Multiple features are separated by "`&&`". The feature specification list starts with the 'put' keyword, followed by a feature name, an optional list of composition properties (an asterisk (*) in the template indicates that zero or more composition properties can be specified) and their values given by `$Value` of the placeholder pair `(-$PropertySwitch, $Value)`. `$PropertySwitch` is a single alphabetic character identifying the composition property listed in Table 4-2. For a description of these properties, please refer to Section 4.2.1.

Table 4-2 Feature Composition switches and values

Feature composition Property	**Switch**	**Possible values**	**Default value**
Feature realization type	-t	PMR, SMR	PMR
Feature destination	-d	caller, receiver, both, call	both
Feature host	-h	self, bound, shared	bound
Outgoing call type	-c	block, unblock	block

Mean number of outgoing calls	-n	Any positive real number	1.00
Feature task multiplicity	-m	Any positive integer number	infinity
Feature host multiplicity	-M	Any non-negative integer number, use 0 for infinity	1
Feature task scheduling	-j	FIFO, PPR, HOL	FIFO
Order-value	-o	Any positive integer number	Depends on the feature

The ‘out’ line of the FCD template defines the name of the output middleware model as the value of placeholder `$SpecializedMiddlewareModel`. Multiple in-out blocks can be used to build the middleware model in stages. Giving a name to every intermediate model could be tedious, therefore the FCD offers an easier alternative to specify this re-curring case: not to give any specific name to the output model of the former in-out block, rather use the keyword ‘auto’ for it and for the input model of the latter in-out block. The 'auto' keyword directs the parser to auto-generate the name of the intermediate model and to use it for the next composition. The examples of multiple feature compositions (Section 4.3.9) demonstrate the use of the ‘auto’ keyword.

Multiple in-out blocks are frequently used for the Message Based Middleware (MBM). Because in the case of MBM the message queue acts like a controller that intercepts every call from caller to receiver, splitting it into two calls (see Section 2.4). It is possible in a middleware that a certain feature (e.g., security) is applied only between the caller and the message queue, but it is not applied between the message queue and the receiver. This can happen when the message queue and the receiver belong to the same host. Therefore, separate in-out blocks are needed if the performance modeler wants to add features between

caller and message queue, or between message queue and receiver. Such examples can be found in Section 4.3.

The next section presents the feature composition process, followed by the feature composition algorithm. The chapter concludes with a number of examples of feature compositions using the feature composition algorithm.

4.2.3 Feature Composition Process

The feature composition process is executed by the Model Composition Engine (see Figure 1.1). The MCE has three elements: a parser, a composition controller and a feature composer which executes the feature composition algorithm. The parser reads an FCD, identifies the in-out blocks (see Section 4.2.2) and builds a *FeatureList* for each block. After parsing all the in-out blocks, the parser then passes to the composition controller the *FeatureLists*, names of the corresponding base middleware models and the *composeCalls* (the calls in BMM where the feature will be composed). The composition controller passes each *FeatureList* in turn with its base middleware model and *ComposeCall* to the feature composer. The feature composer, using the feature composition algorithm, composes the features to the specified call and returns the composed model to the controller. The end result is a set of SMMs held by the controller. The proposed composition process is very similar to the composition of aspects found in aspect-oriented modeling [54].

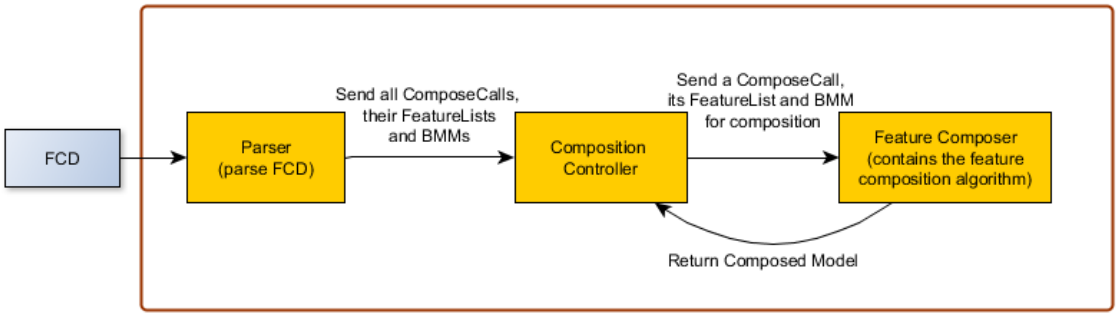


Figure 4.6 The Feature Composition Process

4.2.4 Feature Composition Algorithm

The feature composition algorithm, upon receiving a base middleware model, its *compose-Call* and a *FeatureList* (list of feature realization models) as input, sorts the features in descending order of their order values, building the *SortedFeatureList*. The features in this list are then composed in the indicated order.

The composition of an SMR requires to split the *composeCall* (e.g., ca->re) into two calls and add a feature task in between them. The *composeCall* is replaced by a call *PrevCall* (with the same type and parameters as the *composeCall*) which calls the entry of the feature task, and a call *NextCall* from the feature task's entry to *re*. Both *PrevCall* and *NextCall* direct the composition process to the locations of the next feature compositions. After a feature task is composed, its host is added if needed and the feature is deployed to its specified host. Finally, the properties (e.g., multiplicity, call type) related to that feature are updated as specified by the switches.

The algorithm is presented below in pseudocode. The pseudocode has close resemblance to Java programming language as our tool is implemented in that language. If an 'if' or 'for' block is too long to follow, colors are applied to matching pairs in order to help the

reader. Note that there are a few duplicate codes in this algorithm, which are left as is, in order to avoid function calls which may distract the readers from the flow of the algorithm.

Listing 4.2 Feature Composition Algorithm

```
ComposeFeatures (Model BMM, ArrayList<FeatureModel> FeatureList, Call
ComposeCall):Model{
    PrevCall := ComposeCall /* initialize PrevCall */
    NextCall := ComposeCall /* gets a new value when an SMR is added */
    SMM := BMM /* Make a copy of BMM to keep BMM unchanged. */

   SortedFetureList = ArrayList.SortDescending(FeatureList)

  /*All following model modifications take place in SMM. */
    For each FeatureModel in SortedFeatureList
        If FeatureModel.type = PMR
            If FeatureModel.destination = Caller
                Add FeatureModel.demand to PrevCall.Caller.demand
                Update demand variables by PrevCall.Caller.name
            End If /* if -d=both then both If parts are executed */
            If FeatureModel.destination = Receiver
                    Add FeatureModel.demand to NextCall.Receiver.demand
                    Update demand variables by NextCall.Receiver.name
            End If
        Else If FeatureModel.type = SMR
            /* Add two Feature Tasks if -d=both, one otherwise */
            Add Feature Task(s) (with its entry) to SMM
            Update demand variables by Caller and Receiver names

            If ComposeCall is unbound /* ca>? */
```

```
            Add a call from Caller to Feature Entry
            Add Feature Host (if specified)
            Deploy Feature Task to its Host
            Continue
        End If
        If FeatureModel.destination = Caller or Call
            C1 := Add a call from PrevCall.Caller to Feature Entry
            C2 := Add a call from Feature Entry to PrevCall.Receiver
            If NextCall = ComposeCall /*Composing SMR for first time */
                NextCall := C2
            End If
            Delete PrevCall
            PrevCall := C1
            /* update variable names */
            If FeatureModel.destination = caller
              Prepend Feature task/entry/host name by PrevCall.Caller
              If name already exists in model, append a counter number
              Update demand variables by PrevCall.Caller.name
            Else If FeatureModel.destination = call
              Update message size by PrevCall/Nextcall's Caller.name
            End If
            If Feature Host is required
                Add a Host to SMM
            End if
            Deploy Feature Task to the specified Host
            If other switches are specified
                Update outgoing call type,
                       mean number of outgoing calls,
                       Feature task multiplicity
```

```
                            Feature host multiplicity
                End If
            End If
            If FeatureModel.destination = receiver
                Add call from NextCall.Caller to Feature Entry
                C2 := Add call from Feature Entry to NextCall.Receiver
                Delete NextCall
                NextCall := C2
                Prefix Feature task/entry/host name by NextCall.Receiver
                 If name already exists in model, append a counter number
                Update demand variables by NextCall.Receiver.name
                If Feature Host is required
                    Add a Host to SMM
                End If
                Deploy Feature Task to the specified Host
                If other switches are specified
                    Update outgoing call type,
                           mean number of outgoing calls,
                           Feature task multiplicity
                           Feature host multiplicity
                End If
            End If
        End If
    End For
Return SMM
}
```

4.3 Examples of Feature Composition

This section presents examples of how the Feature Composition Descriptor (FCD), feature realization models and the feature composition algorithm can be used together to obtain the Specialized Middleware Models (SMM). This will also introduce some middleware models that will be used in later chapters.

To carry out the composition process, the Middleware Composition Engine (MCE) must identify to which call the feature is to be composed, locate the realizations of the required features, and compose the feature realizations to the base model respecting the provided switches.

All feature overhead formulae are updated in the composed model by i) prepending the destination entry name followed by an underscore for the fixed part and the coefficient, and ii) prepending the names of both the communicating entries followed by an underscore for the message size. So, the formula:

```
$COMPRESSION + $msz * $compression
```

becomes for caller entry `ca`:

```
$ca_COMPRESSION + $care_msz * $ca_compression
```

and for receiver entry:

```
$re_COMPRESSION + $care_msz * $re_compression
```

Note that the message size variable is same for the caller and the receiver because the sum of the request and reply message sizes will be used for the calibration. If there is a feature task, the name of the task, its entry and its host (if applicable) are also updated following the same approach.

The following specializations of middleware models are presented here.

- Modeling DoBM

- Composing an SMR
- Composing an unbound feature
- Composing a PMR and an SMR
- Composing a feature with shared host
- Modeling a Network
- Modeling CBM
- Modeling MBM
- Many features composed to a DoBM
- Many features composed to an MBM

4.3.1 Modeling DoBM

The first example uses the Empty Middleware Model (EMM) presented in Chapter 3 as its Base Middleware Model (BMM). For the reader's convenience, the figure showing the EMM is repeated below. A diamond arrowhead indicates that the call-type (e.g., blocking or non-blocking) is unknown and will be derived from a base application model.

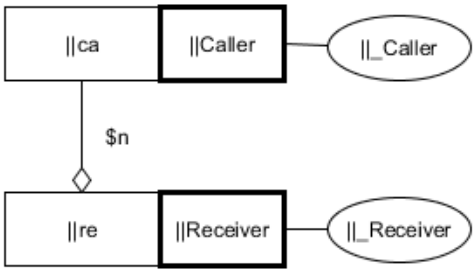


Figure 4.7 Empty Middleware Model (EMM)

Wrapper is a mandatory feature (see Section 3.2.1), whose operation is usually embedded within a distributed call. Therefore, the author suggests to model it as a property-modifying realization. Composing this feature to the caller and receiver of a call produces the middleware model of any Distributed Object Based Middleware (DoBM) such as RMI.

Listing 4.3 FCD to obtain the middleware model of DoBM

```
FCD
in  EMM  at ca re
    put Wrapper
out SMM_DoBM
```

The FCD above directs the composition process to compose EMM with the Wrapper feature (with all its default property values). Note that, by default a feature's property modifying realization is composed to both *ca* and *re*.

The resultant Specialized Middleware Model, called SMM_DoBM is shown below. The variables in the demand formulae are updated by prepending the destination entries name. Since the message size in the formula is the sum of the message sizes of the caller and the receiver, therefore both of their names are prepended before the $msz variable. The use of PMR keeps the composed model compact, yet captures the overhead of the feature.

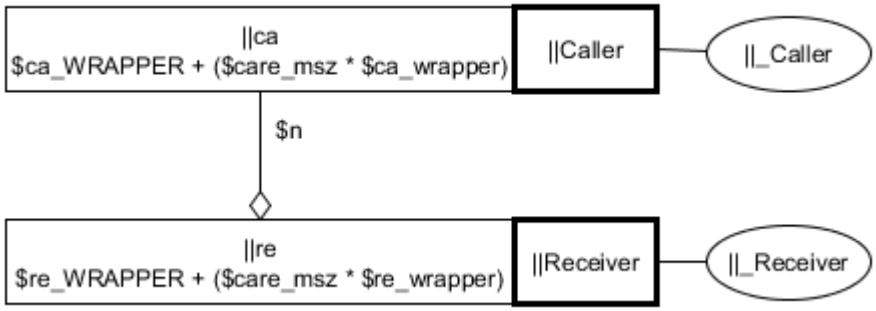


Figure 4.8 SMM_DoBM

SMM_DoBM is used by MMLQ as the base middleware model for RMI. Other features may be composed to it as required by an application.

4.3.2 Composing an SMR

The code and output model when SMM_DoBM is composed with Security as SMR are presented below. An SMR executes the security operations in a concurrent process giving a pipeline of middleware operations.

Listing 4.4 FCD for Security (SMR) composition to RMI

```
FCD
in  SMM_DoBM  at ca re
    put Security -t SMR
out SMM_DoBM_Security
```

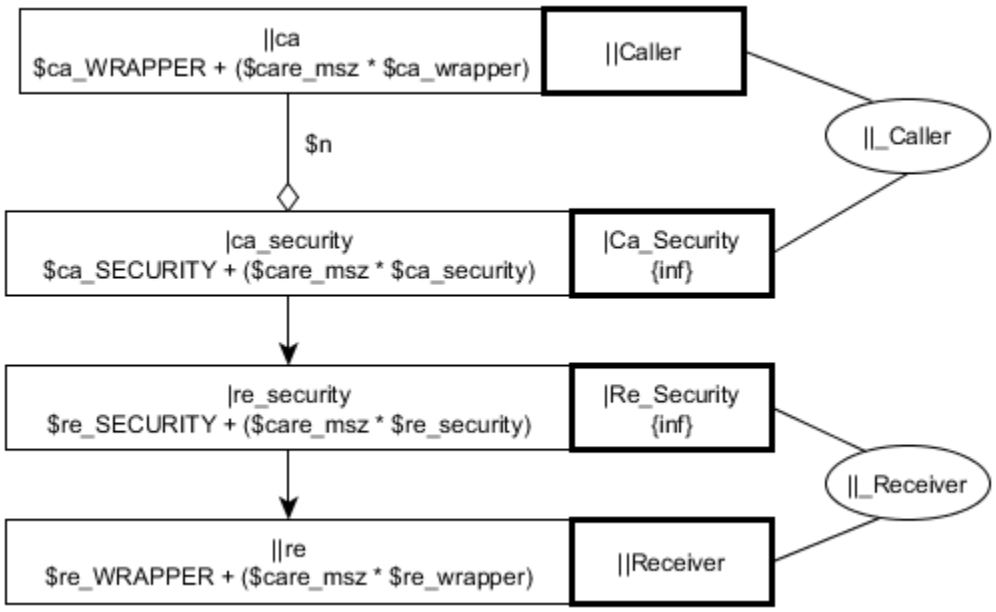


Figure 4.9 DoBM composed with Security (SMR)

Here, two realizations of the Security feature are composed to both destinations, i.e., to the caller and receiver, as separate tasks, and due to the use of default binding type 'bound', the tasks are deployed to the host of their corresponding destinations. The composed feature tasks, apart from updating their variable names, their entry and task names are also

updated. Both task names and entry names are prepended by their calls' destination entry names, but in case of tasks, the first letter of the final name is capitalized in order to follow the naming convention of MMLQ models (see Section 3.1). The calibration and usage of the models shown in Figure 4.8 and Figure 4.9 will be presented in Chapter 6.

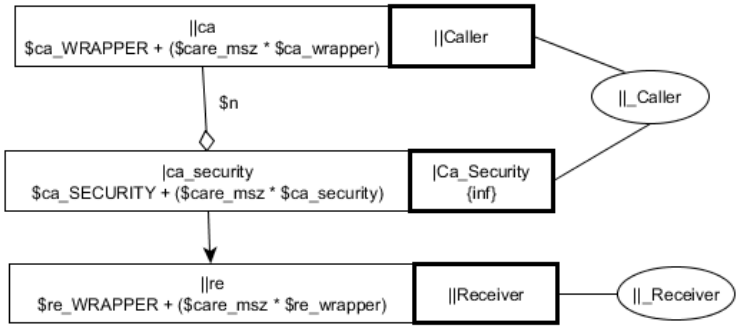


Figure 4.10 Security composed using 'put Security –t SMR –d caller'

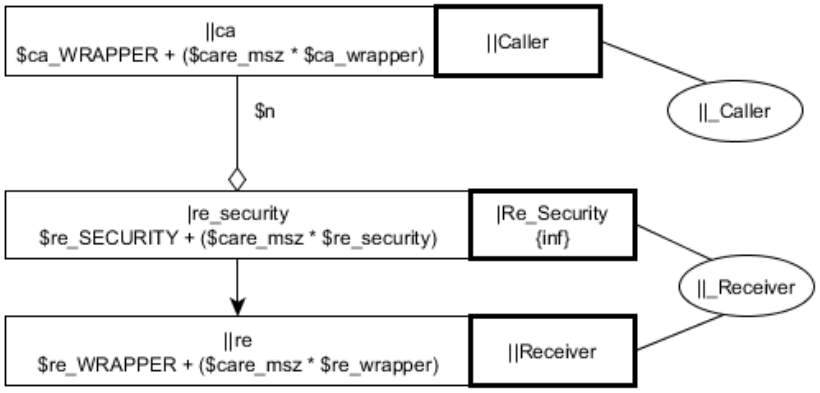


Figure 4.11 Security composed using 'put Security –t SMR –d receiver'

Situations may require to compose a feature either to a caller or to a receiver task only. This may happen if the other one was already composed with that feature from a previous composition. To compose a feature either to a caller or receiver only, the '–d' switch followed by the keywords 'caller' or 'receiver' is used. The results of such compositions are shown in Figure 4.10 and Figure 4.11.

4.3.3 Composing an Unbound Feature

In the following example, a NameService feature is composed as SMR. Typically, an inclusion of name service does not modify an existing call, rather it adds a new, separate call to the caller. This characteristic is captured in this example.

Listing 4.5 FCD for NameService Composition to DoBM

```
FCD
in  SMM_DoBM  at ca ?
    put NameService -t SMR -d call -h self
out SMM_DoBM_NameService
```

Note the use of question mark (?) in place of receiver entry name, indicating that this feature is not added to an existing call, but rather has to be introduced as a separate new call from ca. In this example, the NameService feature is composed to the call (`-d call`) as it is a separate service and is deployed on its own host (`-h self`).

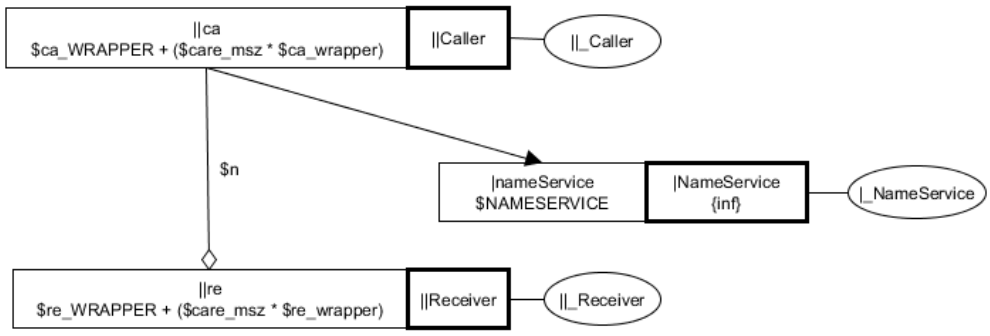


Figure 4.12 DoBM composed with NameService

4.3.4 Composing a PMR and an SMR

Suppose in an application the Security feature is carried out by a dedicated host in the server, but for the client it takes place in the client's host. In this case, the performance

modeler would compose the Security feature as SMR at the receiver and as PMR at the caller. This specification can be written as below.

Listing 4.6 DoBM is composed with two different (PMR and SMR) Security feature realizations

```
FCD
in  SMM_DoBM  at ca re
    put Security -d caller
    && Security -t SMR -d receiver
out SMM_DoBM_Security_v2
```

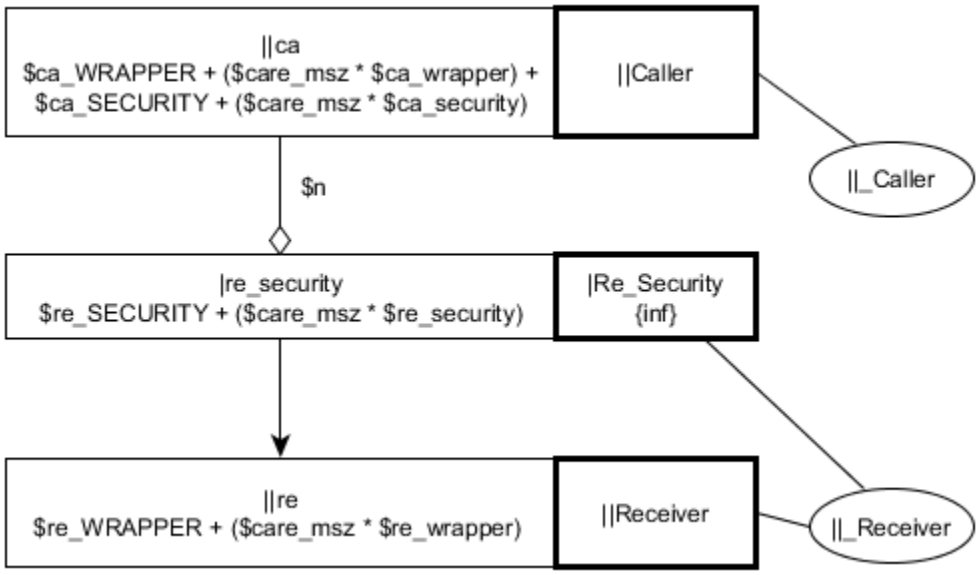


Figure 4.13 DoBM composed with two different (SMR and PMR) Security feature realizations

Note that the service demand of the caller entry will be incremented, while for the receiver the operation is carried out by an added security feature task. This is due to the use of `-t SMR` tag in the line 4 of FCD.

4.3.5 Composing a Feature with Shared Host

Within a web server, some highly demanding feature shared by a set of calls may be executed by a single dedicated host. For this, the feature needs to be composed as an SMR

deployed to its own host. An example of such server is shown below where the Compression service is a shared service. The usage of this SMM will be shown in Chapter 5 example SAM2.

Listing 4.7 Composing Compression features that share a host

```
FCD
in  SMM_DoBM  at ca re
    put Compression -t SMR -h shared
out SMM_DoBM_Compression_shared
```

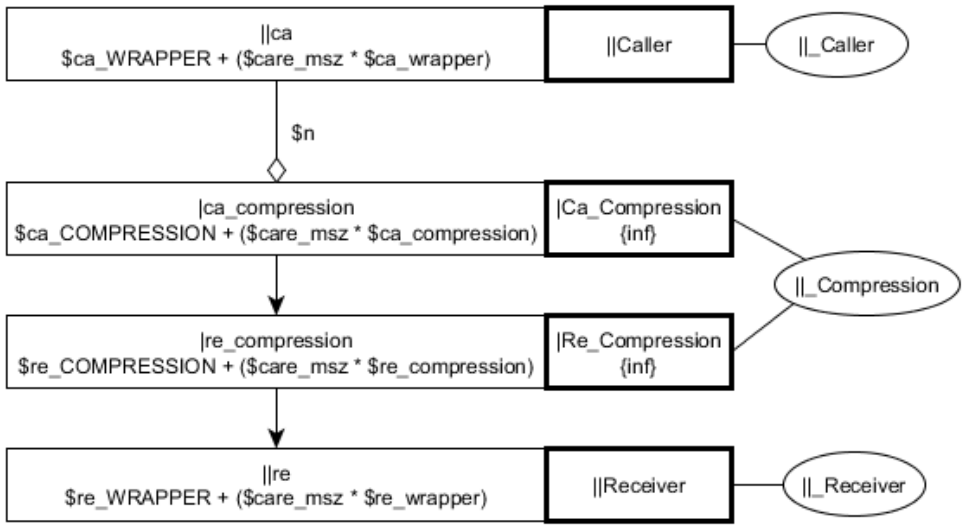


Figure 4.14 Compression features having a shared host

4.3.6 Modeling Network Latency

The Network feature is composed to the EMM to model the latency of a network. Recall that the Network Feature is modeled as an infinite task deployed in an infinite host. The FCD and the SMM for this composition are presented below. The use of '-d call' switch tells that this feature is neither attached to the caller or receiver, therefore the name of feature task Network remain unmodified. The delay variable of the Network feature task

retains its name because it does not depend on the context (more about this in Section 6.2.1). The use of '-M 0' switch directs the composer to set the feature host multiplicity as infinite.

Listing 4.8 FCD to get the Network Model

```
FCD
in  EMM  at ca re
    put Network -t SMR -d call -h self -M 0
out SMM_Network
```

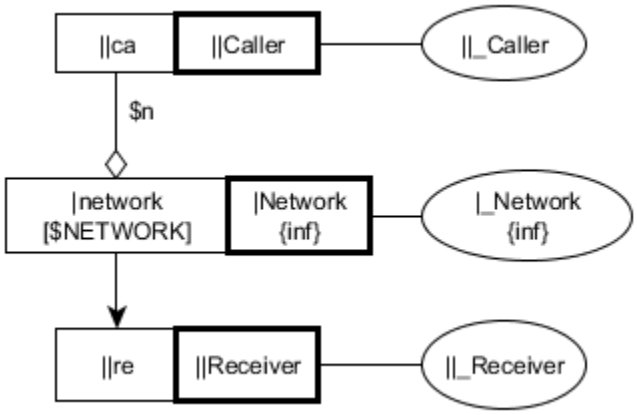


Figure 4.15 Model for network latency

In MMLQ, a single task is used to model the network latency of an entire application with an entry for each call, each with the appropriate delay value. This is a standard approach for LQN models [97].

4.3.7 Modeling CBM

All the Container Based Middleware (CBM) platforms require that the Container feature be composed with the server. The Container feature acts as a wrapper for the services offered at the server [4]. Therefore, it is proper to model a Container as a property modifying realization. Since the caller makes a distributed call to the container, its overhead is mod-

eled using the Wrapper feature. The following FCD is used to get the specialized middleware model of a container-based middleware. This model will be used as the base middleware model for all message-based middleware.

Listing 4.9 FCD to model a container based middleware

```
FCD
in  EMM  at ca re
     put Wrapper -d caller
     && put Container -d receiver
out SMM_CBM
```

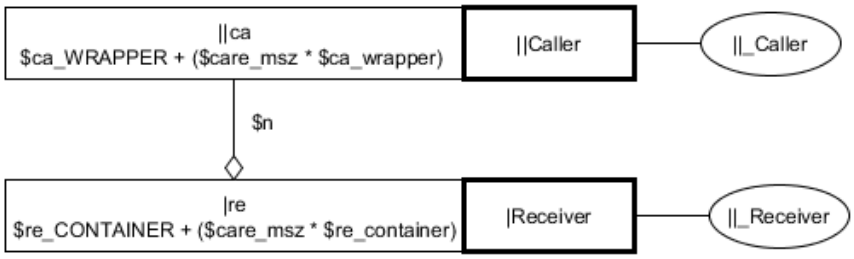


Figure 4.16 SMM_CBM is the base middleware model of CBM

This model is the starting point for specializing any Container Based Middleware. Features may be composed to it as required by the application.

Composing any feature to CBM is very similar to the one that is done for DoBM. The composition of the Security feature as an SMR is shown below.

Listing 4.10 CBM is composed with Security

```
FCD
in  SMM_CBM  at ca re
    put Security -t SMR
out SMM_CBM_Security
```

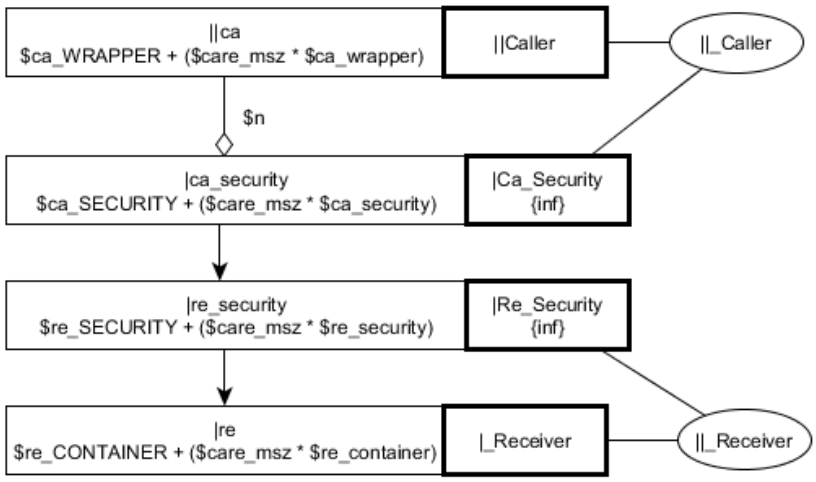


Figure 4.17 SMM_CBM_Security

4.3.8 Modeling MBM

A message queue works as a mailbox between the caller and receiver. To model a message based middleware, the EMM is composed with the MessageQueue feature. A message queue can be deployed either on the receiver (i.e., in which case that is the server) or (most commonly) on its own dedicated host. When a message queue has its own dedicated host, then it acts as a separate entity neither attached to the caller or receiver. Therefore, the switch '-d call' is used. This implies that the names of the MessageQueue feature task and its entry remain unmodified. However, the `$msz` variable is prepended by the names of the two communication entries as the message queue is handling the messages of the caller and the receiver.

The overheads at the caller and the receiver are modeled using the Wrapper feature. The message queue can be used either in synchronous style (known as RPC or request-reply style [84], [73]), or in asynchronous style too. The FCD and the composed models are presented below.

Listing 4.11 FCD to obtain a message based (synch) middleware model

```
FCD
in  EMM  at ca re
     put Wrapper
     && put MessageQueue -t SMR -d call -h self
out SMM_MqSynch
```

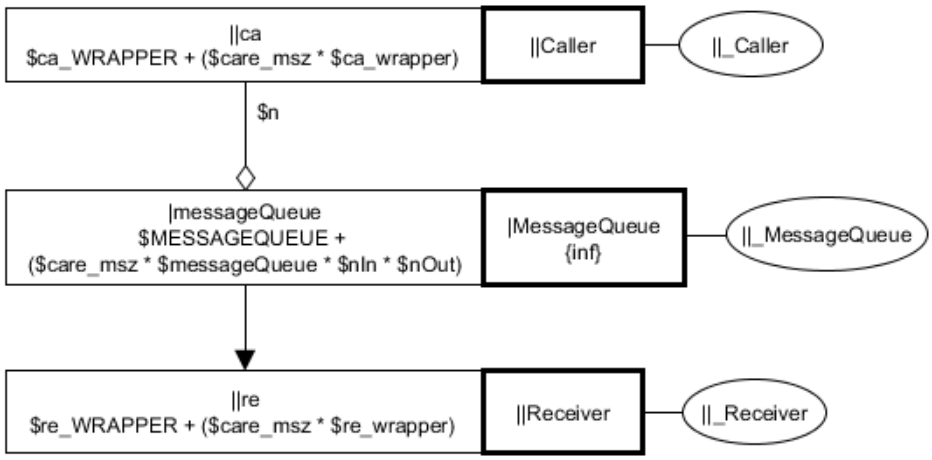


Figure 4.18 SMM_MqSynch is the base middleware model for Synchronous MBM

.

Listing 4.12 FCD to obtain a message based (asynch) middleware model

```
FCD
in  EMM  at ca re
     put Wrapper
     && put MessageQueue -t SMR -d call -h self -c unblock
out SMM_MqAsynch
```

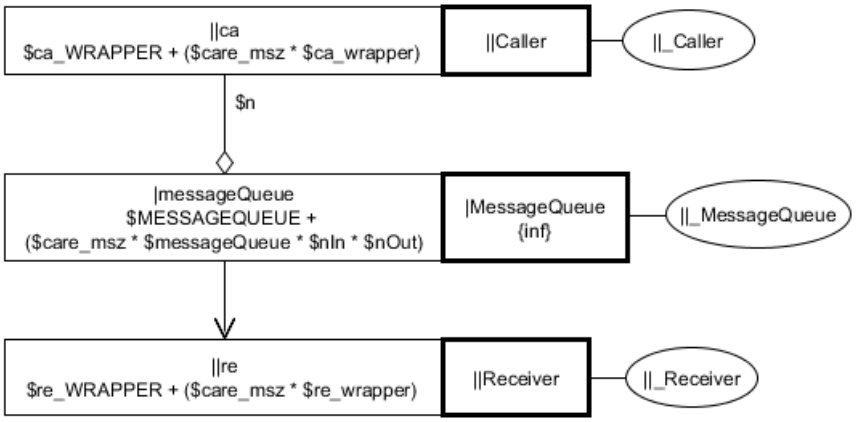


Figure 4.19 SMM_MqAsynch is the base middleware model for Asynchronous MBM

Figure 4.18 and Figure 4.19 are used as the base middleware model for MBM. The MessageQueue can be enhanced by composing different features to it. The following example composes the Security feature to it. The asterisks direct the MCE to compose the Security feature to all calls (here two) of the MBM model, in and out of the MessageQueue server.

Listing 4.13 Composing Security to Message Queue

```
FCD
in   SMM_MqSynch  at * *
     put Security -t SMR -d both
out  SMM_MqSynch_Security
```

The output composed model is given in Figure 4.20.

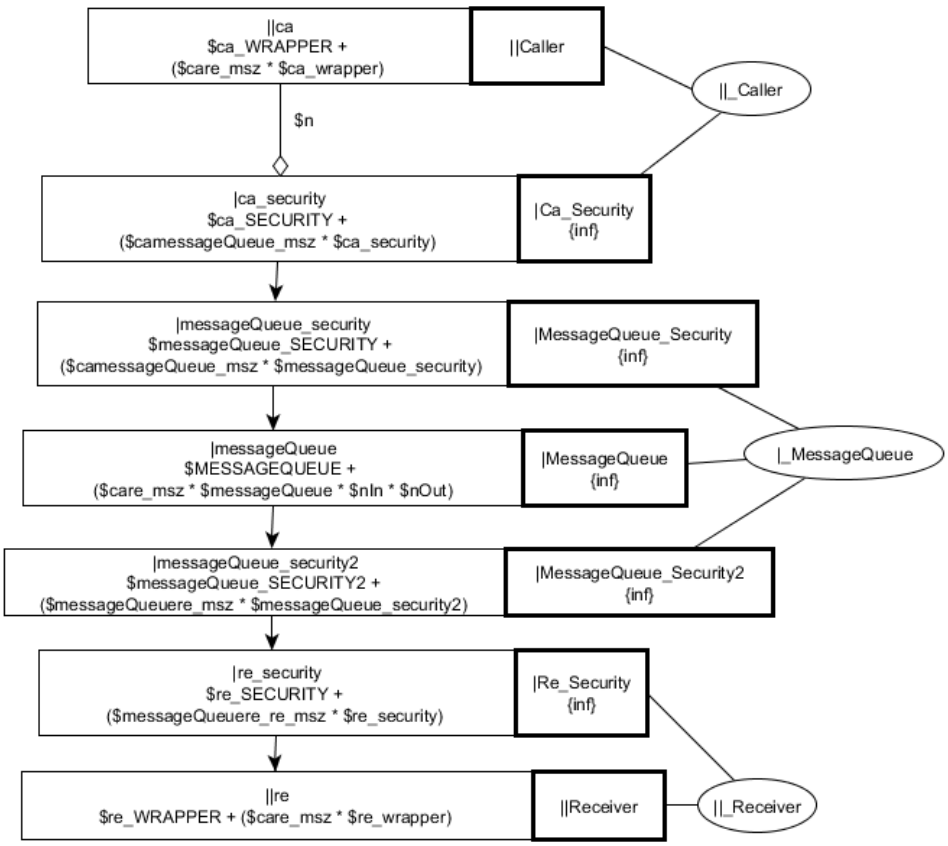


Figure 4.20 Security feature is composed to SMM_MqSynch

4.3.9 Composing Multiple Features to the DoBM Model

Feature compositions may be complicated. An application may require its middleware to use many features together. Also, a feature may cause a task to make a new invocation. The following example covers these cases.

Listing 4.14 Composing multiple features to the DoBM model

```
FCD 2
in  SMM_DoBM at ca re
    put Security -t SMR
    && Compression -d caller
    && Compression -t SMR -d receiver
    && Session -t SMR -h self
out auto
```

```
in  auto  at ca ?
    put NameService -t SMR -d call -h self
get SMM_DoBM_ManyFeatures
```

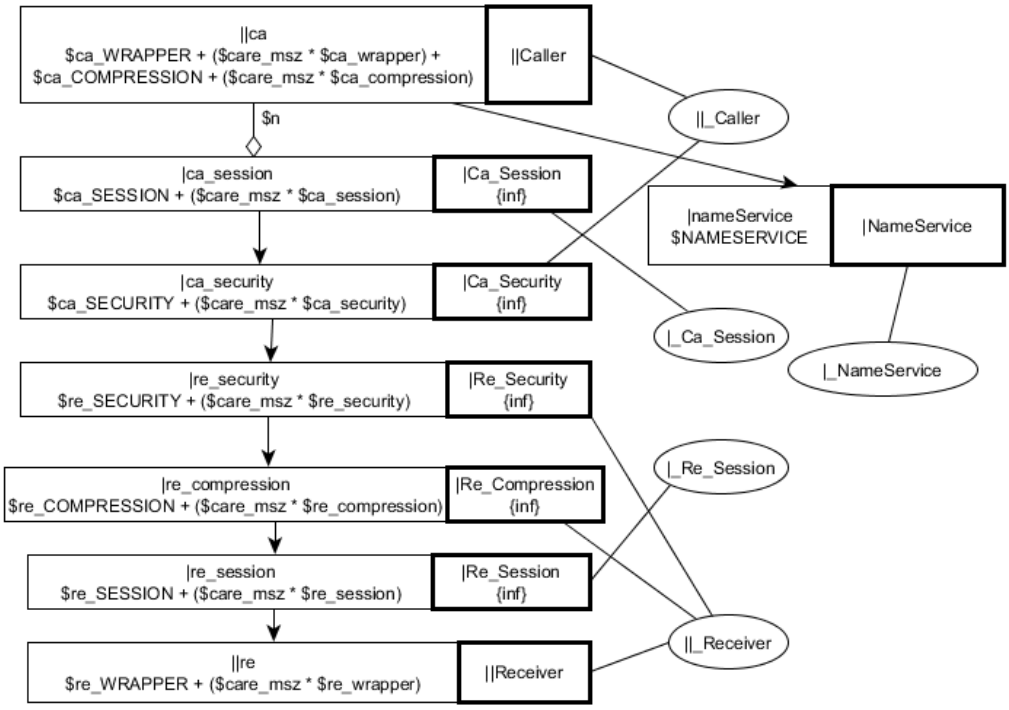


Figure 4.21 Many features composed to the DoBM model

The FCD of Listing 4.14 contains 2 in-out blocks (indicated by the number 2 after the keyword 'FCD'). The first in-out block directs to compose Security, Compression and Session features to the only call (ca>re) of SMM_DoBM. A second in-out block is used for directing the composition of NameService feature, because it is an unbound feature that requires external calls to be added to the model. The keyword 'auto' directs the composer to auto-generate the name of the intermediate model and use it for the next in-out block. The final specialized middleware model (i.e., SMM_DoBM_ManyFeatures) is presented in Figure 4.21.

4.3.10 Composing Multiple Features to the MBM model

An example of many features composition to MBM is presented below. A message queue sits between the communication of caller and receiver, and therefore there are two calls instead of one for every request. This makes the MBM composition descriptor little different than those of DoBM or CBM. Features need to be composed separately at both sides of the message queue.

In the following example, features are composed in two in-out blocks. In the first in-out block, Security and Session features are composed between caller and MessageQueue. In the following in-out block, Compression, Security and Session features are composed between MessageQueue and the receiver.

Listing 4.15 Composing multiple features to the MBM model

```
FCD 2
in  SMM_MqAsynch at ca messageQueue
    put Security -t SMR  -c unblock
    && Session
out auto
in  auto at messageQueue re
    put Compression
    && Session
    && Security -t SMR -c unblock
out SMM_MqAsynch_ManyFeatures
```

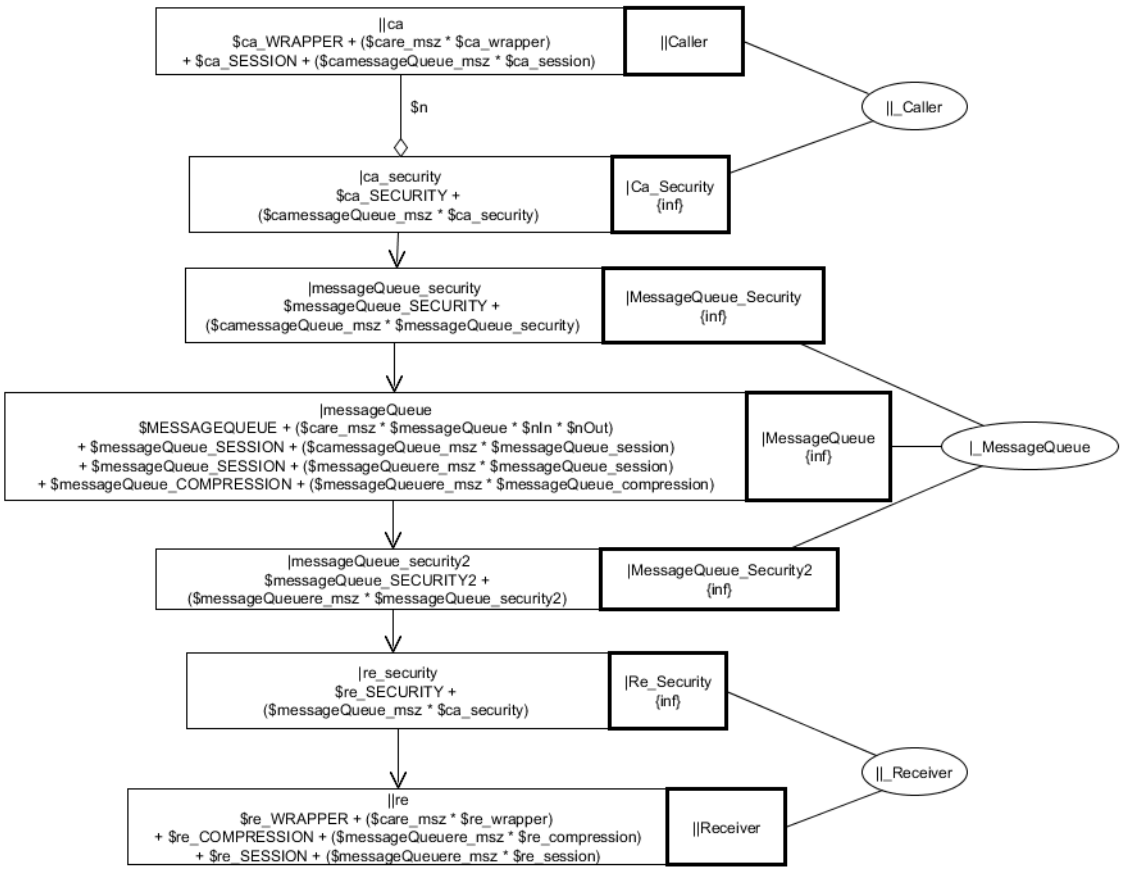


Figure 4.22 Many features composed to the MBM model

The use of the option '–c unblock' makes every composing call asynchronous. The variable names of overheads may become complicated, but they are not carried to the next composition. After building the Specialized Middleware Models, the next step in MMLQ is to calibrate these models with values. Chapter 5 will assume that the different calibrated values of the features are already given and show how these models can be composed to a Base Application Model to obtain application models with middleware model composed into it. The description of the calibration process is postponed till Chapter 6 (Experimental Analysis).

5 Chapter: Middleware Composition

Chapter 4 described how features can be composed to Base Middleware Models (BMM) to produce Specialized Middleware Models (SMM). This chapter presents a process to compose the SMMs with a Base Application Model (BAM) (i.e., a software performance model without any middleware) to produce the final result, a Specialized Application Model (SAM) (i.e., a middleware-aware application model). This composition process is directed by a Middleware Composition Descriptor (MCD), similar in concept to the feature composition descriptor (FCD) introduced in Chapter 4.

5.1 Middleware Composition Descriptor (MCD)

Whereas an FCD builds up an appropriate middleware model for one or more calls, an MCD describes which middleware models will be applied to which calls in the base application model, and how they will be applied.

The template of an MCD is presented in Listing 5.1. A formal description of MCD in extended BNF form is presented in Appendix A

Listing 5.1 Template of Middleware Composition Descriptor (MCD)

```
MCD $[nInOutBlock]?
in $BaseApplModel  put $SpecializedMM
   at ($SrcEntity $DstEntity)+
  [but ($SrcEntityBt  $DstEntityBt)+ ]*
   // use "&&" followed by entity pairs to specify more
calls
   // in "at" and "but" commands
out $SpecializedApplModel
/* repeat the in-out block as many times as needed */
```

The template begins with the keyword 'MCD', indicating it is a middleware composition descriptor, followed by `$nInOutBlock` which is an integer variable giving the number of 'in-out' blocks to follow. If a value for `$nInOutBlock` is not given then its default value 1 is taken by the composition process. In the template, a superscript question sign ($^{?}$) after a variable means zero or one such variable needs to be specified, a superscript plus sign ($^{+}$) after a variable indicates that one or more such variable must be specified and a superscript asterisk sign (*) after a variable indicates that zero or more such variable must be specified. Optional parts of the MCD are shown in square brackets in the template.

Next begins the first in-out block of the MCD. An MCD may contain one or more in-out blocks. Each in-out block specifies an SMM and a set of calls in a base application model, with which that middleware (i.e., SMM) is to be composed. If an application uses *n* different middleware platforms or feature combinations, then its MCD will require at least *n* in-out blocks.

An in-out block starts with the keyword 'in' followed by the name of the base application model (`$BaseApplModel`), then the 'put' keyword, followed by the specialized middleware model name (`$SpecializedMM`). Next, lines starting with 'at' and 'but' keywords are used to specify a set of calls, called a `CallList`, where the specialized middleware model will be composed. Command *'at'* is used to include calls into a `CallList` and 'but' is used to explicitly exclude calls from a `CallList`. An MCD must contain an 'at' command while it may or may not contain a 'but' command.

The 'at' command is followed by one or more entity pairs `($SrcEntity, $DstEntity)` separated by `"&&"`. An entity pair can be one of:

- Two entries that identify a single call from the entries `$SrcEntity` to `$DstEntity`

- Two tasks that identify all calls from the tasks `$SrcEntity` to `$DstEntity`
- Two hosts that identify all calls from the hosts `$SrcEntity` to `$DstEntity`
- `$SrcEntity` and `$DstEntity` can be any combination of entry, task and host that identifies all calls from `$SrcEntity` to `$DstEntity`

Furthermore, an asterisk (*) can be used as `$SrcEntity` or `$DstEntity` to represent all the entries of a caller or receiver.

The 'but' command is optional and if used, it must be in conjunction with an 'at' command. Command 'but' is used to exclude some of the calls that are already included in the `CallList` by a previous 'at' line. This is useful, for instance, when a modeler wants to compose a particular middleware with all calls of a BAM, but a few. The 'but' line has the same format as the 'at' line.

Next, a line starting with the 'out' keyword indicates the end of the in-out block. It is followed by the name of the composed output model (`$SpecializedApplModel`). If the modeler needs to compose another middleware to the BAM, another in-out block will be used to describe it. Every in-out block but the last can set the output model to `'auto'`, as in a FCD, to avoid giving names to the intermediate models.

Example:

Consider the base application model `Ecommerce` presented in Figure 5.1, taken from Li et al. [61] with slight modifications. In this model, user requests go to a web server and then to an image server and an application server. The application server receives two kinds of requests: browse and order. These requests complete their operations after getting replies from a database server and a file server.

The MCD shown in Listing 5.2 is used to specify that in this model all calls but the ones from StoreApp to ShoppingCart are composed with middleware model `mw1`, and the calls from StoreApp to ShoppingCart are composed with middleware model `mw2`.

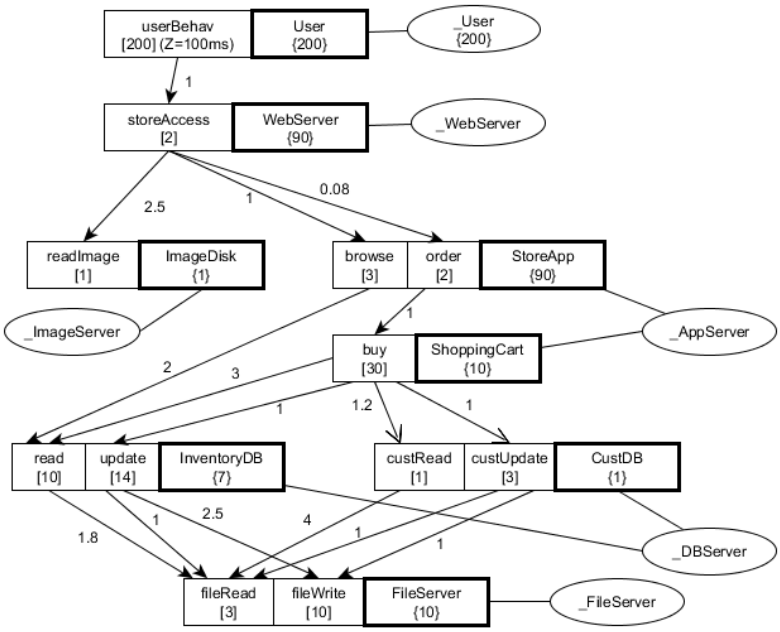


Figure 5.1 Base Application Model (BAM) of an E-Commerce System [61]

Listing 5.2 A sample middleware composition descriptor

```
MCD 2
in ECommerce  put mw1
   at * *
   but StoreApp ShoppingCart
out auto
in auto put mw2
   at StoreApp ShoppingCart
out Ecommerce_Composed
```

Due to the use of the 'auto' keywords, it was not required to provide any name to the intermediate model produced by the first in-out block. After the second in-out block, the desired specialized application model Ecommerce_Composed is produced.

5.2 Examples of Call Specification in the MCD

The 'at' and 'but' commands can be used to specify any groups of calls (CallList) in a given model. The following table presents a few examples of call specifications on the Ecommerce model of Figure 5.1.

Table 5-1 MCD call specification examples

Command	Calls addressed
at * *	All calls in the model
at _User *	All calls from all entries of all the tasks deployed to the host _User
at * _FileServer	All calls coming from any source to any tasks deployed to the host _FileServer
at WebServer StoreApp	All calls going from WebServer task to StoreApp task
at browse read	The call from browse entry to read entry
at browse read **&&** buy read	The calls from browse entry to read entry, and from buy entry to read entry
at * * **but** _AppServer _DBServer	All calls but the ones going from _AppServer host to _DBServer host

at * * **but** CustDB FileServer **&&** InventoryDB FileServer	All calls but the ones going from CustDB task to FileServer task, and from InventoryDB task to FileServer task

5.3 Middleware Composition Process

The middleware composition process has a structure similar to the feature composition process described in Section 4.2.3. It also uses a parser, a composition controller and a middleware composer (Figure 5.2). The parser parses a Middleware Composition Descriptor (MCD) and for each in-out block, it builds a CallList to which the specified middleware model is going to be composed. After parsing an MCD completely, the parser sends a list of Base Application Model (BAM) names, CallLists and their corresponding SMM names to the composition controller. The composition controller then passes the current BAM, its corresponding CallList and SMM to the middleware composer. The middleware composer uses the Middleware Composition Algorithm (described next) to compose the given middleware to the CallList. A composed model is returned to the composition controller. The composition controller uses this composed model as the next BAM until all middleware models are composed.

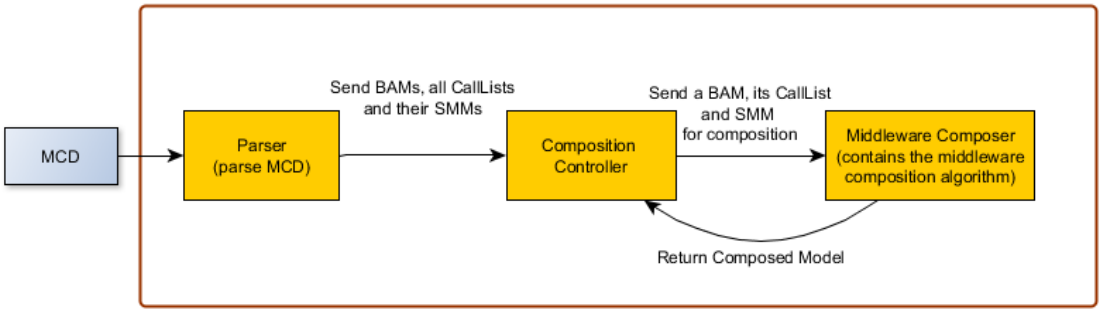


Figure 5.2 The Middleware Composition Process

5.4 Middleware Composition Algorithm

The middleware composition algorithm, presented below, composes a specialized middleware model to a number of calls of the base application model while keeping the number of newly added tasks to the minimum. This is desirable because smaller models are easier to understand by the modelers and quicker to solve by the solver [40].

The algorithm composes the property-modifying realizations first (Step 1), and then the structure-modifying realizations (Steps 2 to 4). Composing a property-modifying realization does not add any new task or host and therefore the composition algorithm only updates the demands of the relevant entries. But, in case of a structure-modifying realization, the algorithm examines the feature tasks, their deployments, their relations with the base application model and composes the model in such a manner so that it avoids adding any unnecessary task that does not affect the performance prediction. At first, feature tasks are added to the base application model (Step 2 and Step 3), then calls are added to those tasks (Step 4) and finally the middleware-aware application model, known as SAM (Specialized Application Model) is returned.

The basic principles of the composition algorithm are as follows:

- If a feature task is deployed to its caller's (or receiver's) host, then create a single feature task with an entry to handle each call going out from (or coming into) any task deployed to that host (that is, for each call that requires the feature).
- If a feature task is deployed to its own host, then create a single task for all calls in the system that use that feature. Thus if a name server were added, there would be one for the entire system. In most cases, each call that uses the feature will use a

Listing 5.3 Middleware Composition Algorithm

```
ComposeMiddleware
(Model BAM, ArrayList<Call> CallList, Model SMM): Model{
   Model SAM := BAM /* SAM is the output model */

   /* The following variable is needed to update the names of the newly
added tasks and entries */
   Char appenT := "A"

   /* All the following changes take place in SAM */
   /* Compose PMR */
   /* Step:1 Update caller and receiver demands with PMR */
   For each Call c in CallList
      If SMM.caller.demand is not already added
         SMM.caller.demand is added to c.caller.demand
      End If
      If SMM.receiver.demand is not already added
         SMM.receiver.demand is added to c.receiver.demand
      End If
   End For

   /* Compose SMR */
   /* Step 2: Add feature tasks deployed to caller / receiver */
   From CallList get the arraylist of participating hosts HostList
   For each Host h in HostList
      CallsOut := all calls going out from Host h that needs a MW
      FeatureTasks := all feature tasks that are to be deployed to h
```

```
      For each Task ft in FeatureTasks
         If ft's destination = Caller
            Create a task t with n entries where n is the number of calls
            going out from host h
            Deploy t to h
            appenT := updateNames(t,appenT)
         Else If ft's destination = Receiver
            Create a task t with n entries where n is the number of calls
            coming into host h
            Deploy t to h
            appenT := updateNames(t,appenT)
         End If
      End For
   End For

   /* Step 3: Add feature tasks deployed to personal hosts (to their
own hosts) */
   FeatureTasks = all feature tasks with personal host
   For each Task ft in FeatureTasks
      If a personal host for ft does not exist in SAM
           add a personal host ft.host
      End If
      Create a task t with n entries where n is the number of calls in
      CallList that use that feature
      Deploy t to ft.host
      appenT := updateNames(t,appenT)
   End For
```

```
   /* Step 4: Add calls to feature tasks */
   For each call c in CallList
      caller := c.caller
      Sort the feature realization models of c by ca, re then by feature-order
      For each feature realization model fm to be composed to c
         If(fm.type = SMR)
           Find a feature task ft in SAM that models fm &&
             matches with the deployment of f mentioned in MCD
           Find the next uncalled entry fe in ft
           Add a c.type call from caller to fe
           caller := fe
         End If
      End For
      /* add a call from the last feature task to the receiver */
      Add a c.type call from caller to c.receiver
   End For
   Return SAM
}


/* Task t is passed by reference */
updateNames(Task t, String appenT): String{
  Append "_" + appenT to t's name
  Append "_" +_appenT to all entries of t's
  Int appenE : = 1;
  For each Entry e in t
    Append appenE to e's name
    appenE++
  end For
```

```
  if appenT = 'Z'
     appenT := 'A'
  else
    Move appentT to the next Char
  Return appenT
}
```

different entry of this task.

- Host, task and call multiplicities of the base application model are retained in the composed model.

In Listing 5.3, a 'participating host' refers to a host in the given BAM that contains a task which makes a call that is contained in the CallList. A feature task having a 'personal host' refers to a task that is not deployed to any host of the given BAM. Colors have been added to some loops in the algorithm to help the reader easily identifying the end of the blocks.

5.5 Examples of Middleware Composition

To illustrate the process of middleware composition, three base application models are introduced. These applications are invented by the author for the purpose of explaining the compositions. Then some specialized middleware models to be composed with these base application models are presented. All models have `$nUser` number of users and all other tasks have infinite threads. Various combinations of the models will be composed to explain the middleware composition algorithm.

5.5.1 Base Application Models

The first base application model “BAM1_ChessMaster” represents a distributed chess application. The chess application has a chess server (ChessS) where the entry ‘master’ is a computer program containing a chess engine that represents a master chess player. Every ‘master’ operation requires to access the database for 20 times. The think time of “user” is 10,000ms. Service demands of entries are given in round parentheses in milliseconds. Both calls in this model are blocking.

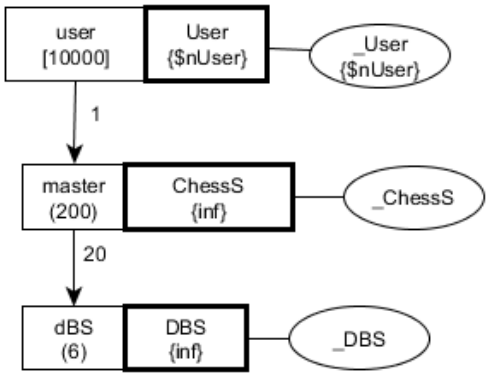


Figure 5.3 Base Application Model "BAM1_ChessMaster"

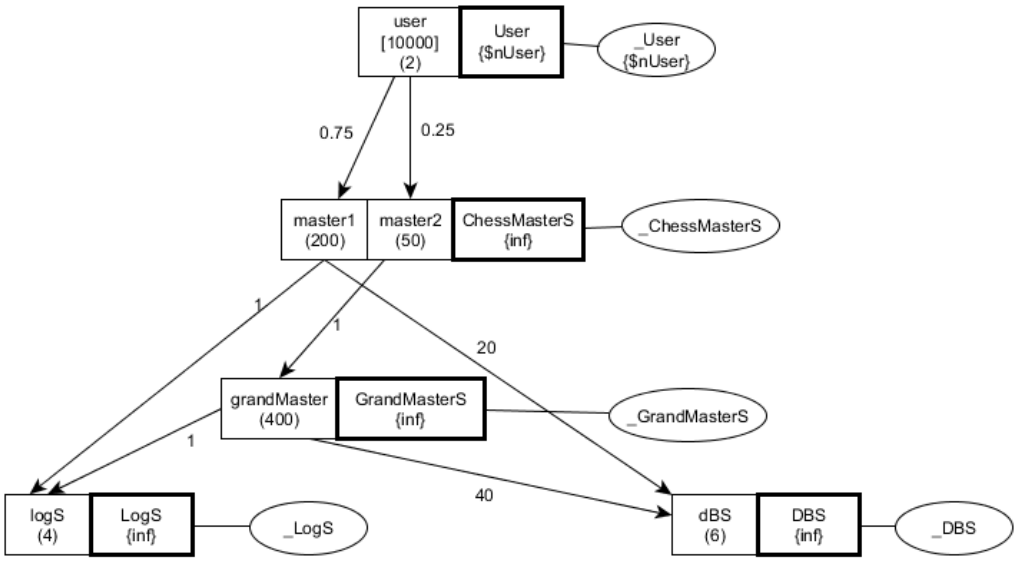


Figure 5.4 Base Application Model "BAM2_ChessGrandMaster"

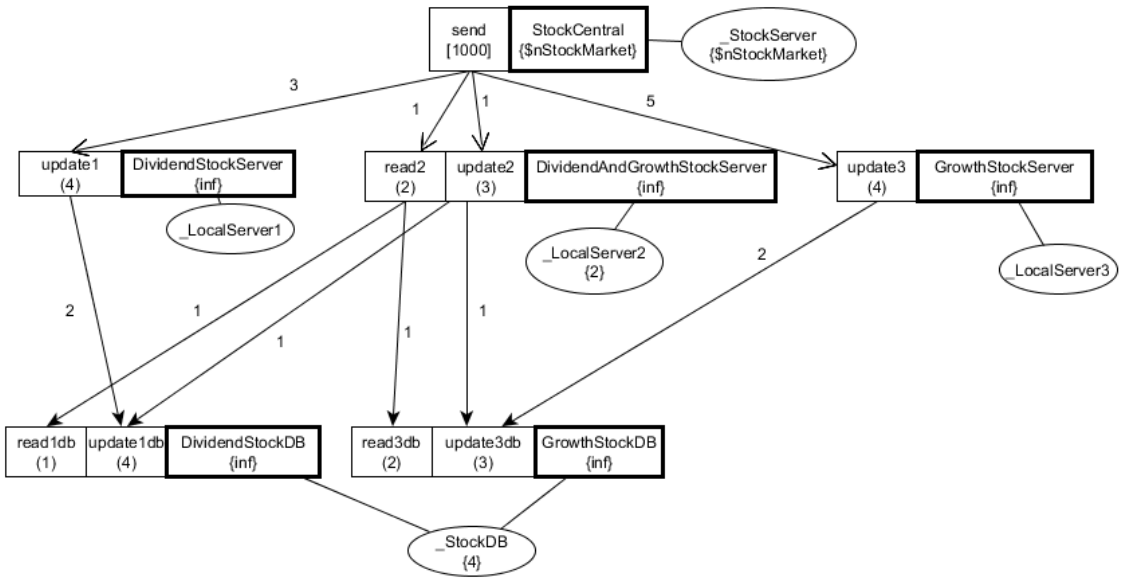


Figure 5.5 Base Application Model "BAM3_Stock"

Suppose that the chess master application (BAM1_ChessMaster) has gained popularity. The users can beat the master easily and they want a more challenging opponent. The makers of the chess master thus have planned to improve the application by introducing a grand master engine. After doing a quick evaluation of the chess board using a tiny script, sometimes the client chess application will call 'master2' that does a little re-evaluation of the board and eventually calls the 'grandMaster'. The grandMaster operation computes more moves and thus has a higher service demand and it needs to access the database more frequently. This version also has a Log Server task "LogS", that logs the analyses made by the masters and the grandMaster.

The third base application model "BAM3_Stock" (Figure 5.5) models a stock market system. The prices of stocks are delivered from a central server "StockCentral". Local servers are of three kinds: 'Growth stock server', 'Dividend stock server', and their mix known as 'Dividend and Growth stock server'. Local server 1 (DividendServer) and Local server 3

(GrowthServer) update themselves regularly, while Local server 2 (DividendAndGrowthServer) is updated less frequently. While all the local servers update themselves with the new prices of the stocks, sometimes Local server 2 is used by the StockCentral to double check current prices through read operations. The stock prices are finally stored through the tasks "DividendStockDB" and "GrowthStockDB" to the shared host "_StockDB". The host mulplicity of _LocalServer2 and _StockDB are 2 and 4 respectively. All calls made from the host _StockCentral are asynchronous, while the next level calls are synchronous.

5.5.2 Specialized Middleware Models

Eight different specialized middleware models are presented below from the three middleware categories. The first three models are for remote method invocation (Distributed Object Based Middleware such as RMI), next three models (SMM 4, 5 and 6) are for Spring (Container Based Middleware), SMM7 models ActiveMQ (Container Based Middleware) and finally SMM8 is purely a Network model. The composition of these middleware models to the base application models will show the different possible ways middleware platforms can be composed.

At this stage of the composition, the middleware models are already calibrated. But, since we have postponed the discussion of calibration till the next chapter, the models will be presented here with some estimated demand values. Some of these values (e.g, for Wrapper, Security) are taken from the calibration shown in Chapter 6, and the others are educated guesses. The exact values of service demands of the features are not important for the purpose of this illustration. The important matter here is how middleware models are composed to obtain specialized middleware models.

Table 5-2 Assumed overheads of the features

Feature	**General overhead formula**	**Assumed value [ms]**
Wrapper (RMI/DoBM)	$WRAPPER + $msz * $wrapper	0.4
Compression (RMI/DoBM)	$COMPRESSION + $msz * $compression	0.8
Compression (RMI/DoBM) with a dedicated compression server	$COMPRESSION + $msz * $compression	0.2
Transaction (RMI/DoBM)	$TRANSACTION + $msz * $transaction	0.7
Security (RMI/DoBM)	$SECURITY + $msz * $security	0.6
Container (Spring)	$CONTAINER + $msz * $container	0.5
Transaction (Spring)	$TRANSACTION + $msz * $transaction	0.77
Security (Spring)	$SECURITY + $msz * $security	0.67
MessageQueue (ActiveMQ)	$MESSAGEQUEUE + $msz * $messageQueue * $nIn * $nOut	0.3
Security (ActiveMQ)	$SECURITY + $msz * $security	0.63
Network	$NETWORK	100

The features that will be used in the specialized middleware models are: Wrapper, Compression, Transaction, Security, Container, MessageQueue and Network. For the purpose of simplicity, for a particular feature the same operation overhead is assumed across all base application models. The overheads are presented in Table 5-2.

RMI Middleware Models:

SMM 1 to 3 are RMI middleware models. Their features are as follows:

- SMM1_RMI_Compression: RMI + Compression as SMR
- SMM2_RMI_CompressionShared: RMI + Compression as SMR Shared Host
- SMM3_RMI_Transaction: RMI + Transaction as SMR

Their specialized middleware models (both un-calibrated and calibrated) are presented in Figure 5.6 to Figure 5.8. The calibrated models will be composed with the base application models.

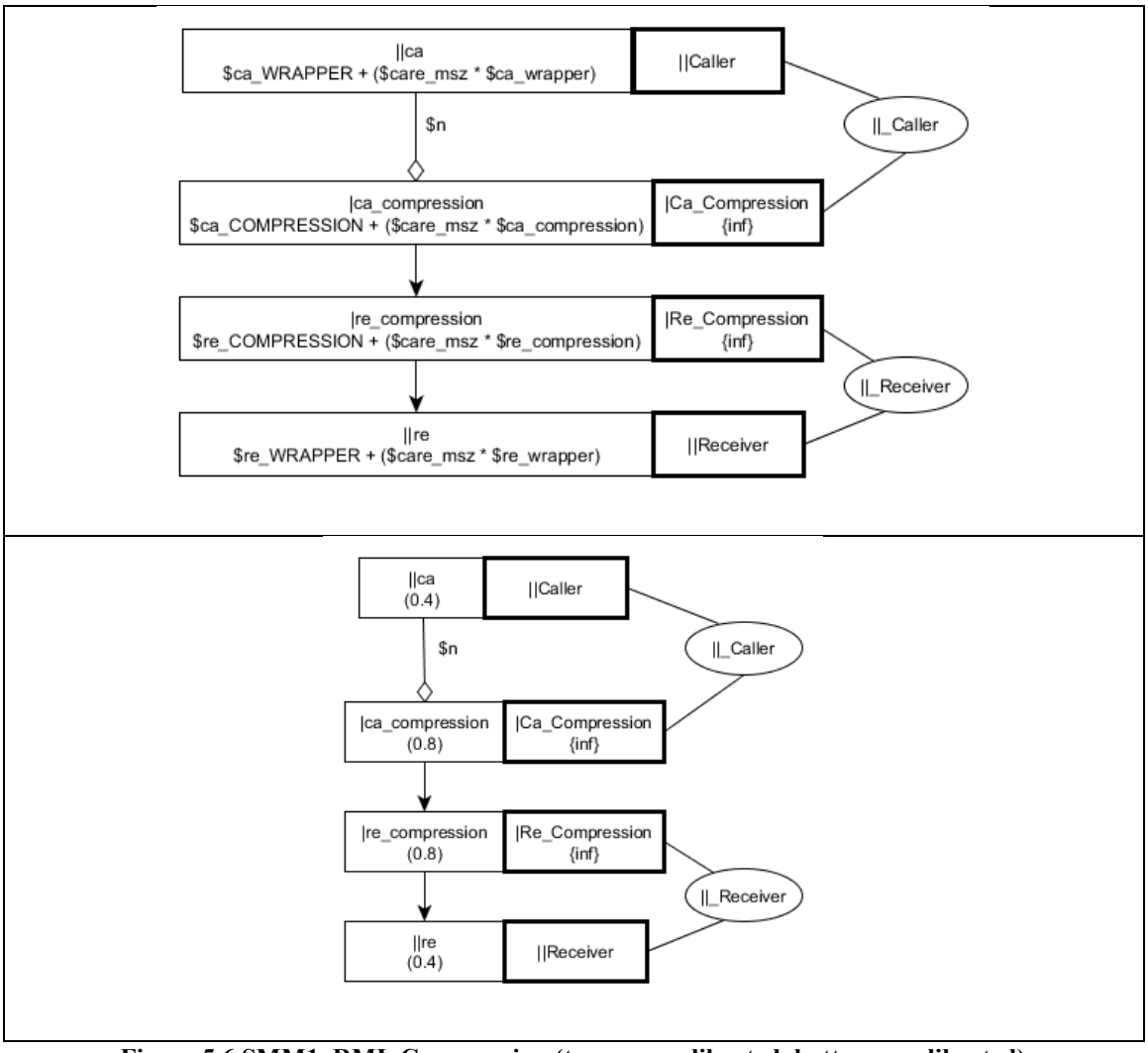


Figure 5.6 SMM1_RMI_Compression (top = un-calibrated, bottom = calibrated)

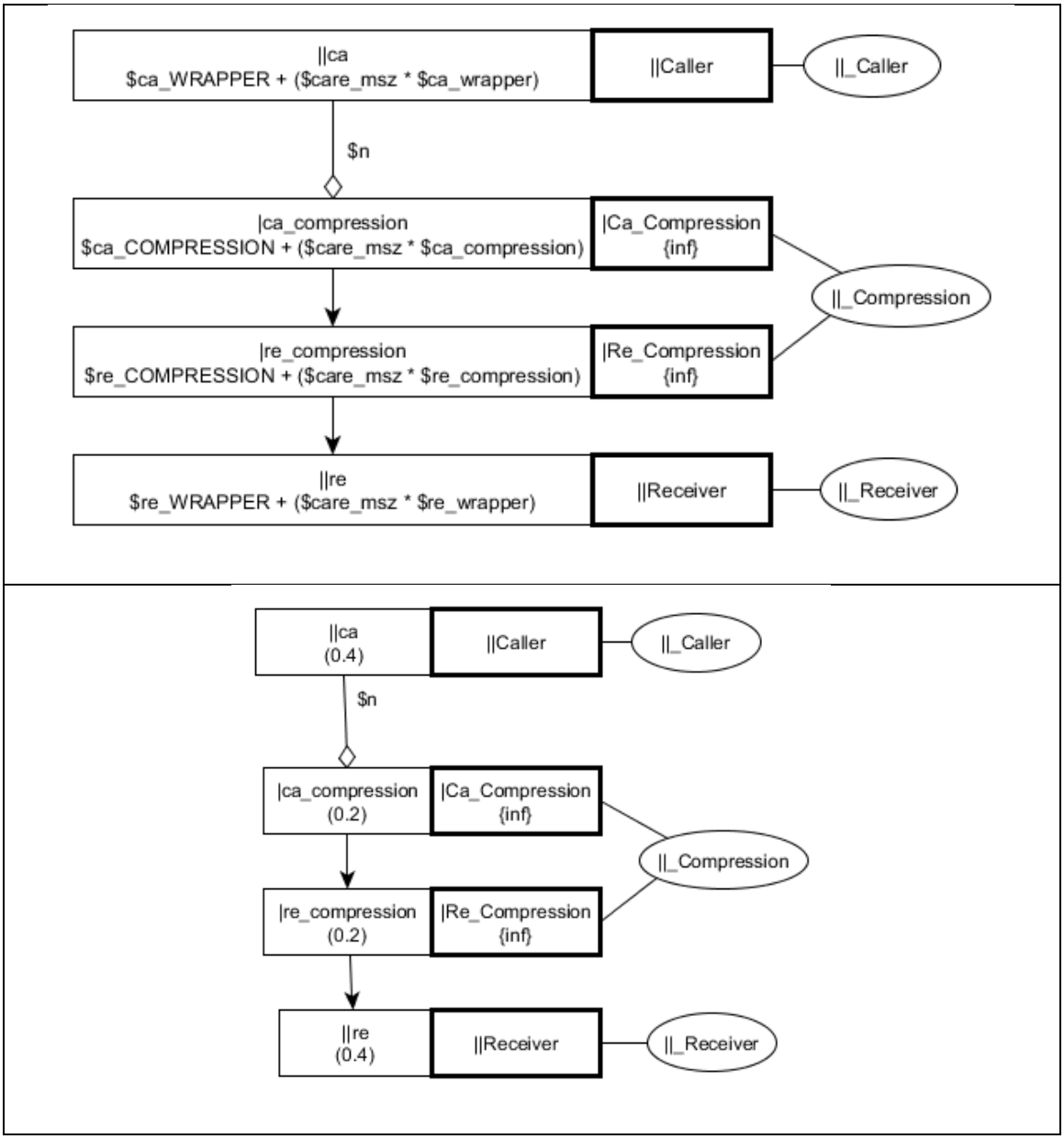


Figure 5.7 SMM2_RMI_CompressionShared (top = un-calibrated, bottom = calibrated)

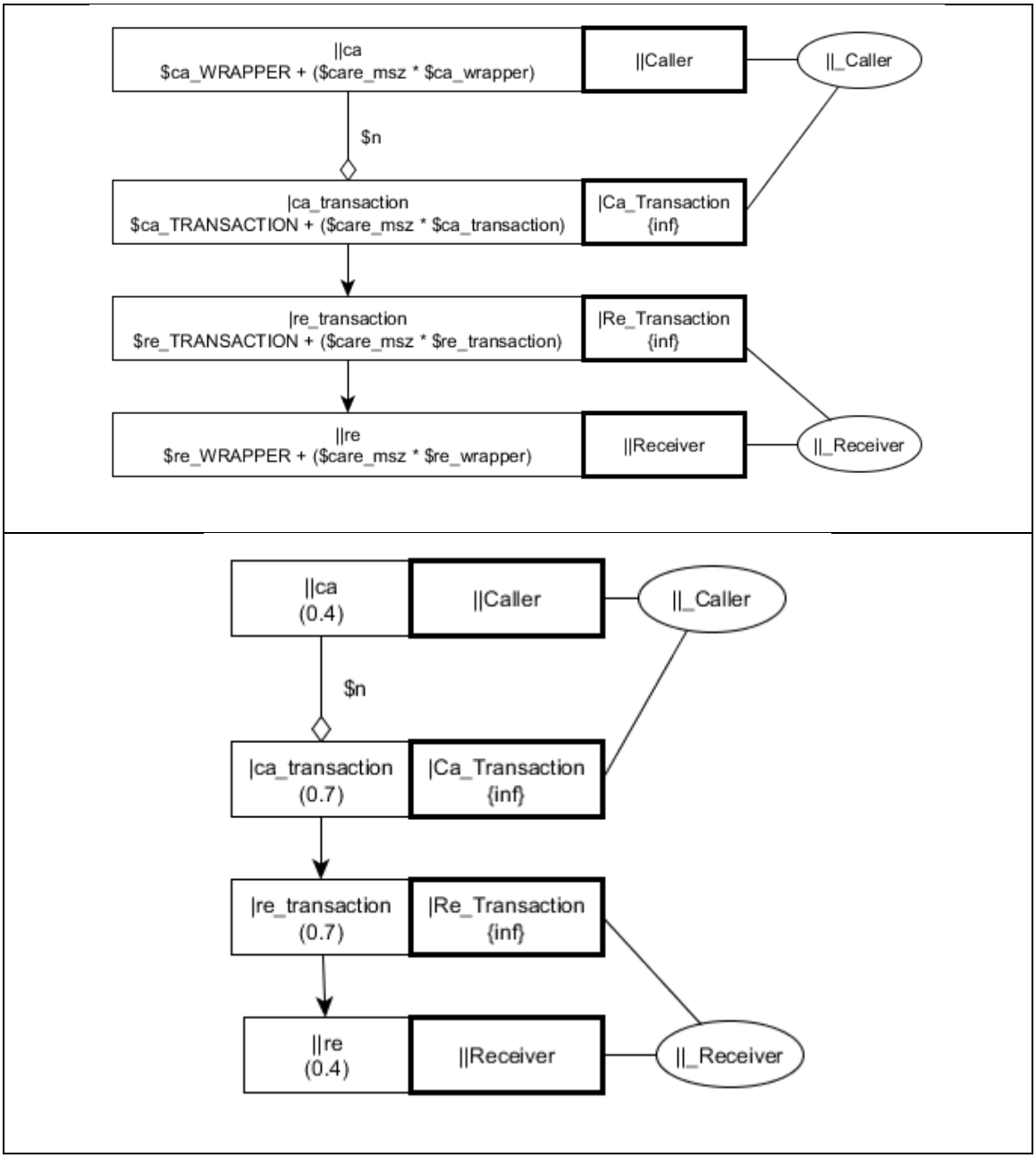


Figure 5.8 SMM3_RMI_Transaction (top = un-calibrated, bottom = calibrated)

Spring (Container Based Middleware) Models:

SMM 4, 5 and 6 are specialized Spring (CBM) models. Their features are as follows:

- SMM4_Spring_Security: CBM + Security as PMR
- SMM5_Spring_Transaction: CBM + Transaction (in caller as SMR, in receiver as PMR)
- SMM6_Spring_Spring: Spring as PMR in both ends of EMM

Their specialized middleware models (both un-calibrated and calibrated) are presented in Figure 5.9 to Figure 5.11.

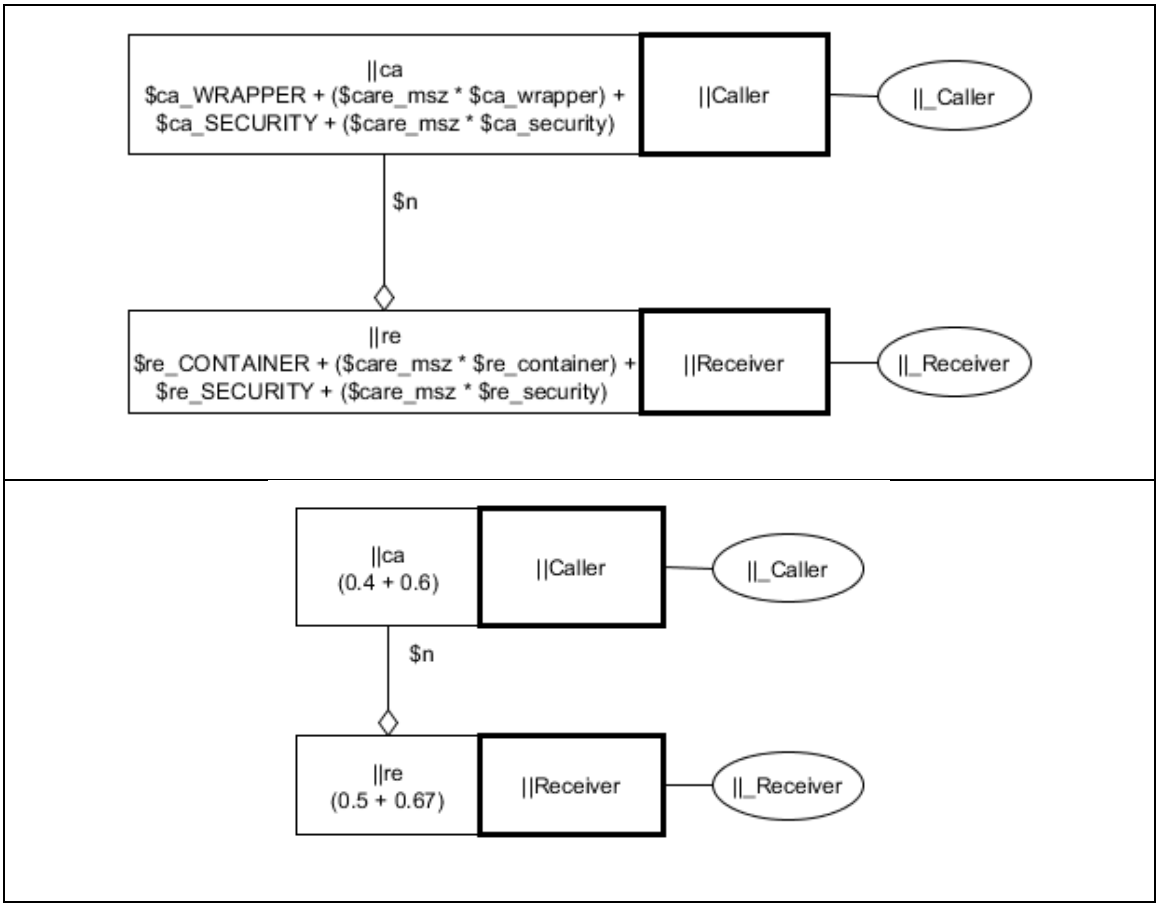


Figure 5.9 SMM4_Spring_Security (top = un-calibrated, bottom = calibrated)

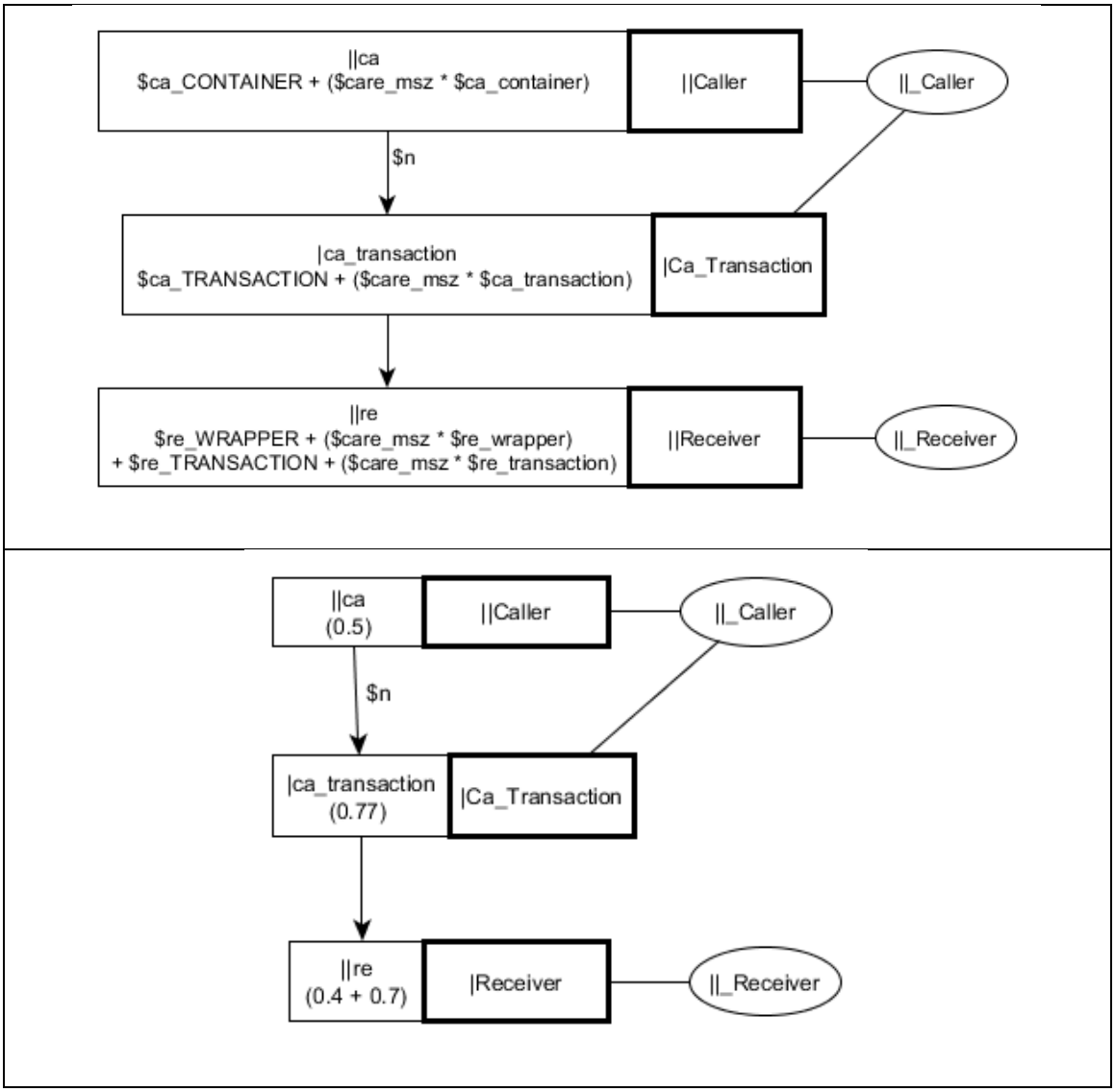


Figure 5.10 SMM5_Spring_Transaction (top = un-calibrated, bottom = calibrated)

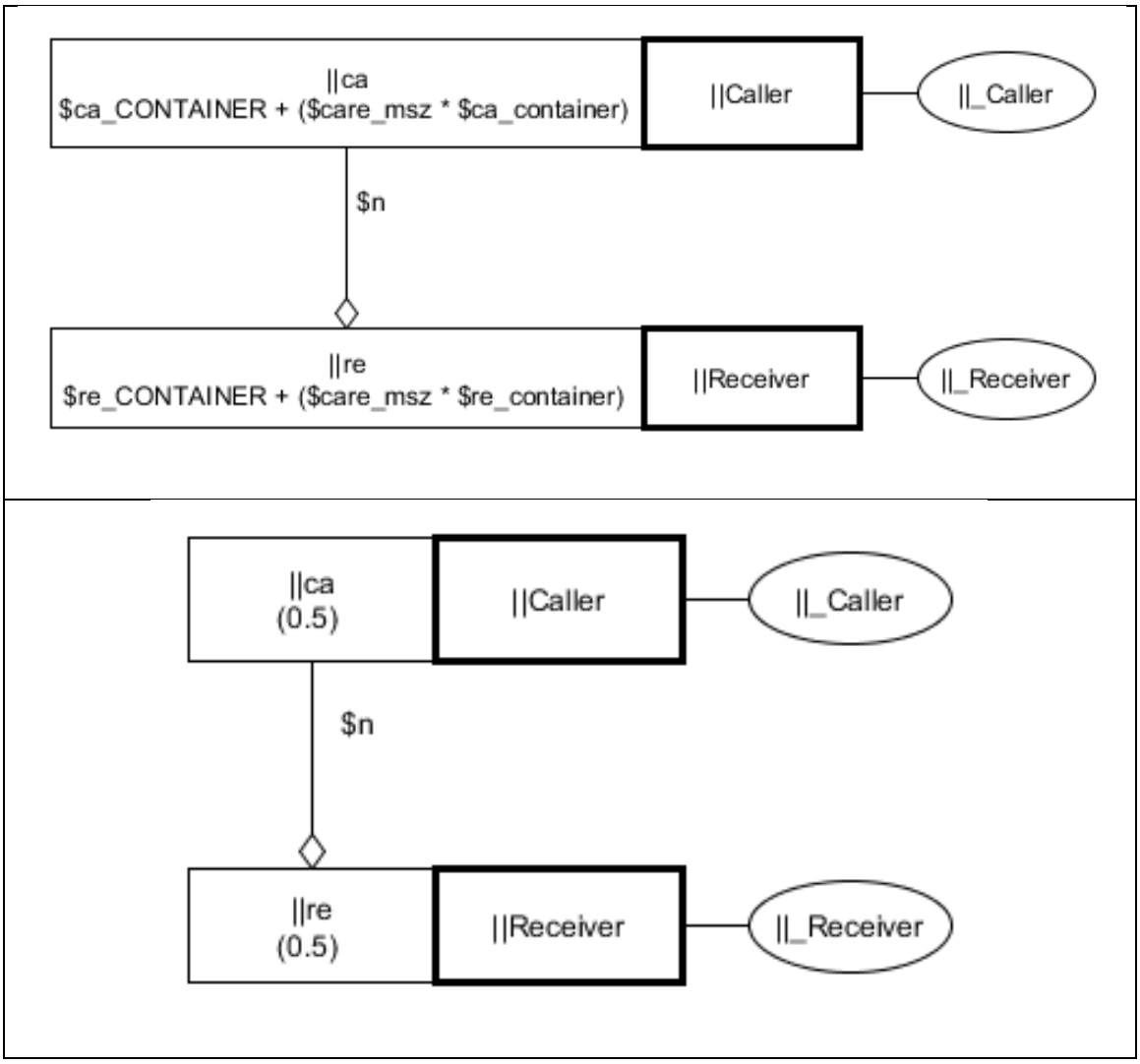


Figure 5.11 SMM6_Spring_Spring (top = un-calibrated, bottom = calibrated)

ActiveMQ (Message Based Middleware) Model:

SMM 7 is a specialized ActiveMQ (MBM) model. The features are as follows:

- SMM7_ActiveMQ_Security: MBM + Security as SMR

The specialized middleware model is presented below.

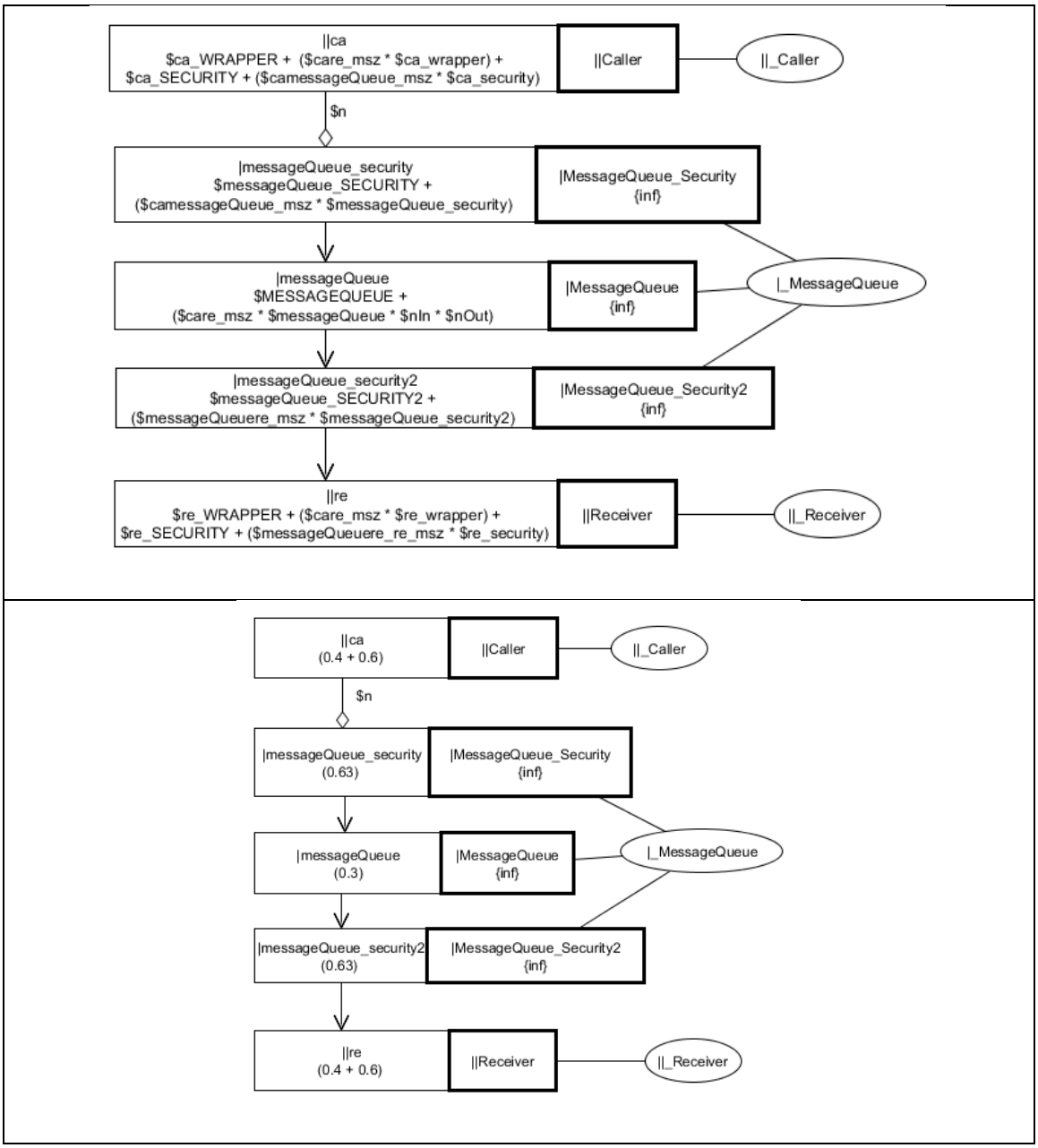


Figure 5.12 SMM7_ActiveMQ_Security (top = un-calibrated, bottom = calibrated)

Network Latency Model:

SMM 8 is the network model. For simplicity, we assume the network latency do not change across applications.

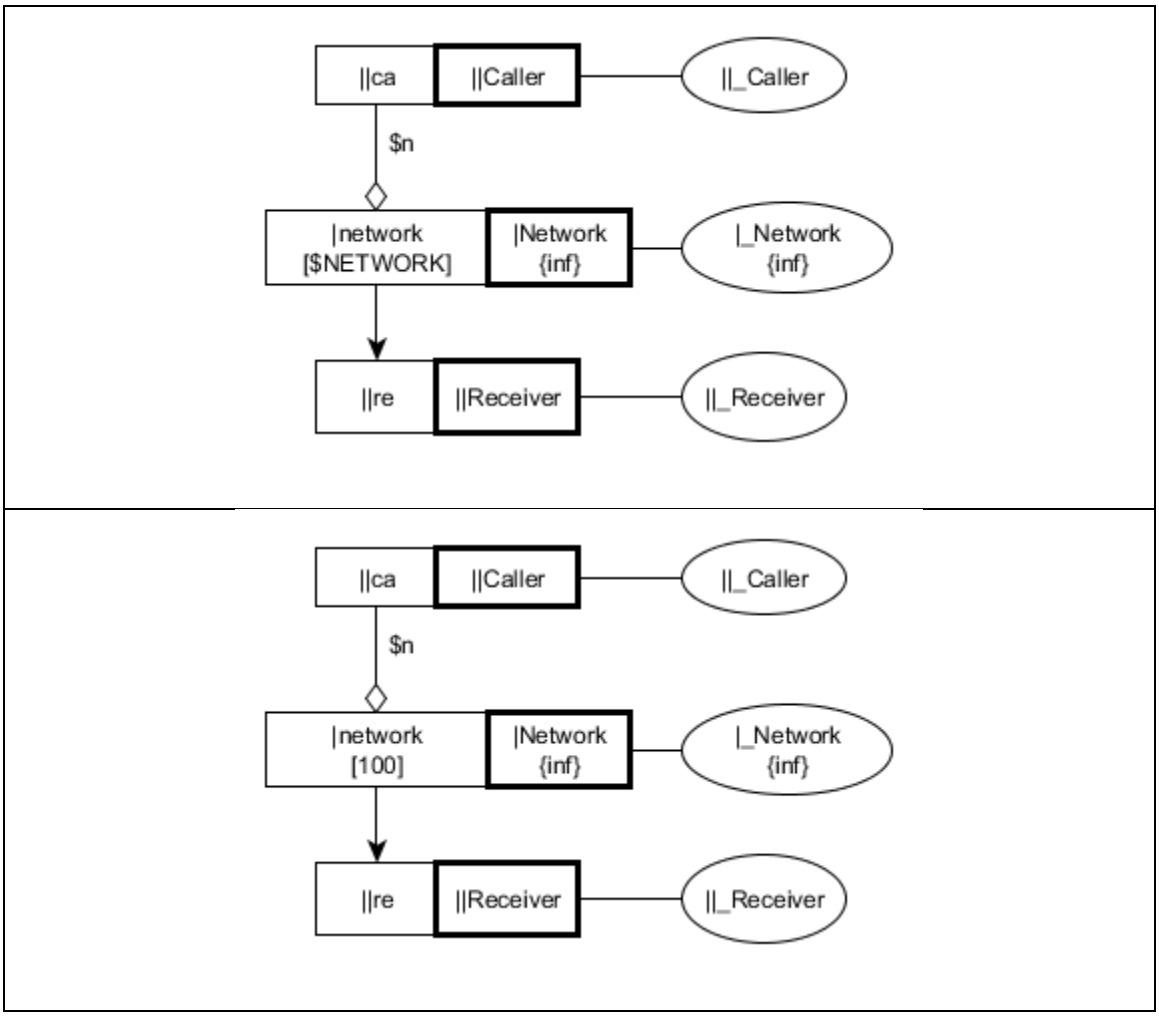


Figure 5.13 SMM8_Network (top = un-calibrated, bottom = calibrated)

5.5.3 Specialized Application Models

In this section, a number of specialized application models are created by composing the base application models with the specialized middleware models presented in the last two subsections. The goal is to demonstrate by examples how the composition algorithm composes middleware of different kinds and deployments.

Table 5-3 Combinations of Basic Application Models and Middleware Models

No.	Input BAM	SMM(s)	Output SAM
1	BAM1_ChessMaster	SMM1_RMI_Compression, SMM8_Network	SAM1
2	BAM1_ChessMaster	SMM1_RMI_Compression, SMM2_RMI_CompressionShared, SMM8_Network	SAM2
3	BAM2_ChessGrandMaster	SMM4_Spring_Security, SMM5_Spring_Transaction, SMM6_Spring_Spring, SMM8_Network	SAM3
4	BAM3_Stock	SMM7_ActiveMQ_Security, SMM3_RMI_Transaction, SMM8_Network	SAM4

The compositions shown in Table 5-3 are presented below. Note that all compositions use the Network Model to model network latency.

SAM1

The makers of ChessMaster (BAM1) wants to use RMI with the Compression feature in the application, as the application will exchange heavy data. To get a performance model

of this application, the specialized middleware model SMM1_RMI_Compression is composed to produce specialized application model SAM1. The MCD and the SAM1 are presented in Listing 5.4 and Figure 5.14 respectively.

Listing 5.4 MCD to obtain SAM1 (ChessMaster with Compression)

```
MCD 2
in BAM1_ChessMaster  put SMM1_RMI_Compression
   at * *
out auto
in auto put SMM8_Network
   at _User _ChessS
out SAM1
```

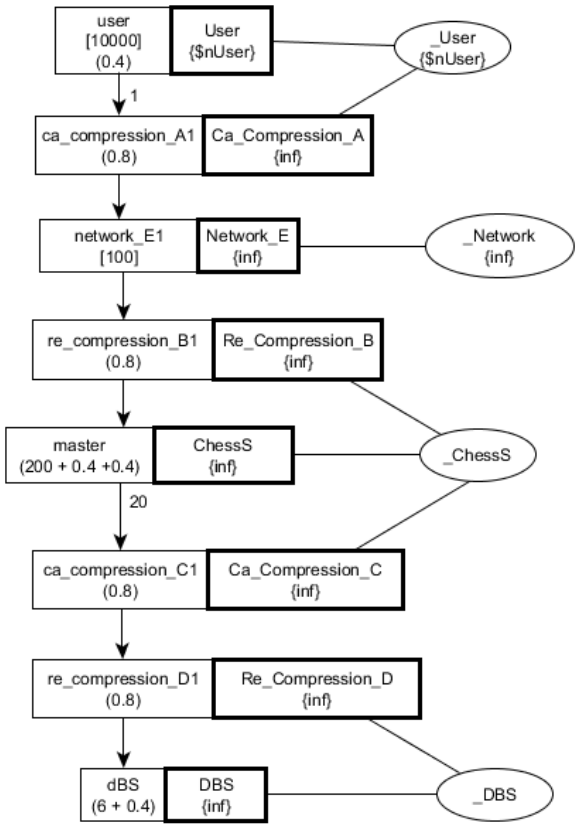


Figure 5.14 ChessMaster model with RMI + Compression gives SAM1

To carry out the composition, two feature tasks are added in BAM1_ChessMaster and connected between both the existing calls, then the feature tasks are deployed to the caller/receiver hosts. The mean number of calls for the previously existing calls remain same. All the newly added calls have mean number of calls 1. The Network model is finally added between the only direct call (i.e., ca_Compression_A1 to re_Compression_B1) between _Network and _ChessS.

SAM2

After running the SAM1 model, the makers of the Chess_Master decided that since 'master' and 'dBS' communicate heavily with each other, a dedicated compression server might be justified. So, they build a model where between 'user' and 'master' the compression feature remains same, but between 'master' and 'dBS' the compression feature has a dedicated host. Note that this composition reuses SMM1. Listing 5.5 MCD to obtain SAM2 (ChessMaster with CompressionShared)Listing 5.5 is the MCD used to obtain the specialized application model SAM2 (Figure 5.15).

Listing 5.5 MCD to obtain SAM2 (ChessMaster with CompressionShared)

```
MCD 3
in BAM1_ChessMaster  put SMM1_RMI_Compression
   at user master
out auto
in auto put SMM2_RMI_CompressionShared
   at master dBS
out auto
in auto put SMM8_Network
   at _User _ChessS
out SAM2
```

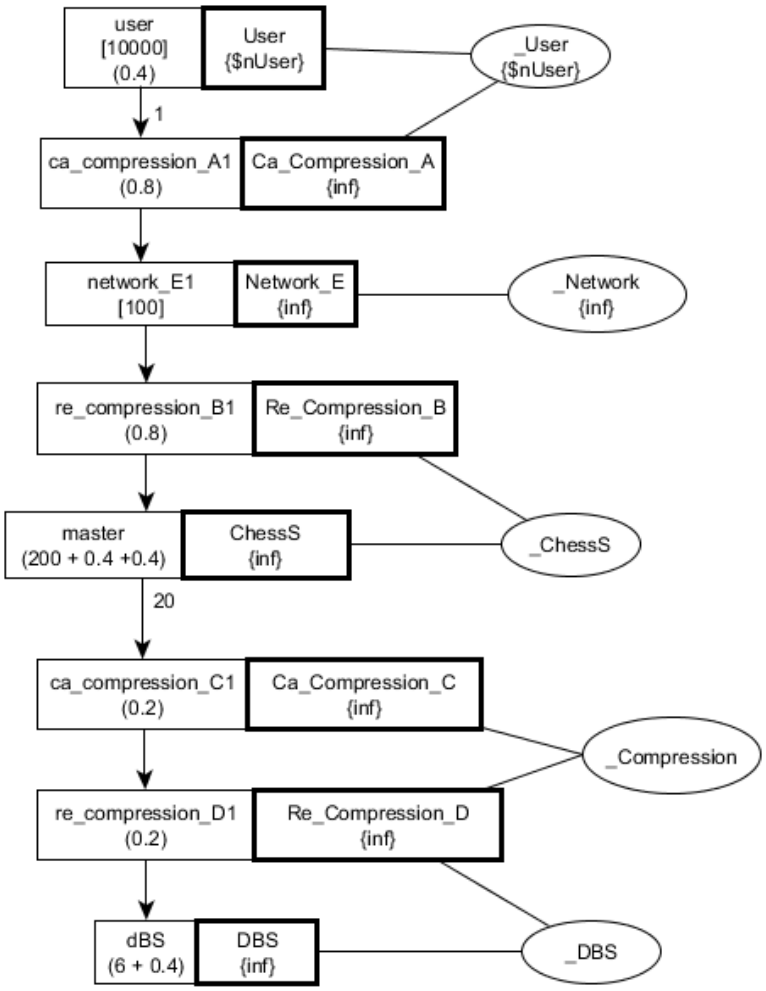


Figure 5.15 ChessMaster model with RMI + CompressionShared gives SAM2

SAM3

In planning the improved application ChessGrandMaster (BAM2_ChessGrandMaster), the makers want to create a sound web application, and chooses the Spring framework for its robust features. They want user communications to be secured and a transaction service to be used for the database and log server accesses. The MCD and the resulting specialized application model (SAM3) are presented in Listing 5.6 and Figure 5.16 respectively.

Listing 5.6 MCD to obtain SAM3 (ChessGrandMaster with Spring)

```
MCD 4
in BAM3_ChessGrandMaster  put SMM4_Spring_Security
   at User ChessMasterS
out auto
in auto put SMM5_Spring_Transaction
   at * LogS
   && * DBS
out auto
in auto put SMM6_Spring_Spring
   at master2 grandMaster
out auto
in auto put SMM8_Network
    at _User _ChessMasterS
out SAM3
```

This example demonstrates the strength of the MCD. For the first in-out block, calls are specified using task names, second in-out block uses wild cards, third in-out block uses entry names and the final in-out block uses host names.

From Figure 5.15, we can see that the transaction features at the callers are modeled as separate tasks (e.g., Ca_Transaction_A and Ca_Transaction_B) and deployed to callers' hosts, while the receiver's transaction feature is modeled as a single task (Re_Transaction_A) having an entry for each call and deployed to its dedicated host. Such a deployment might represent a web application with a dedicated server for managing transaction services at the receivers.

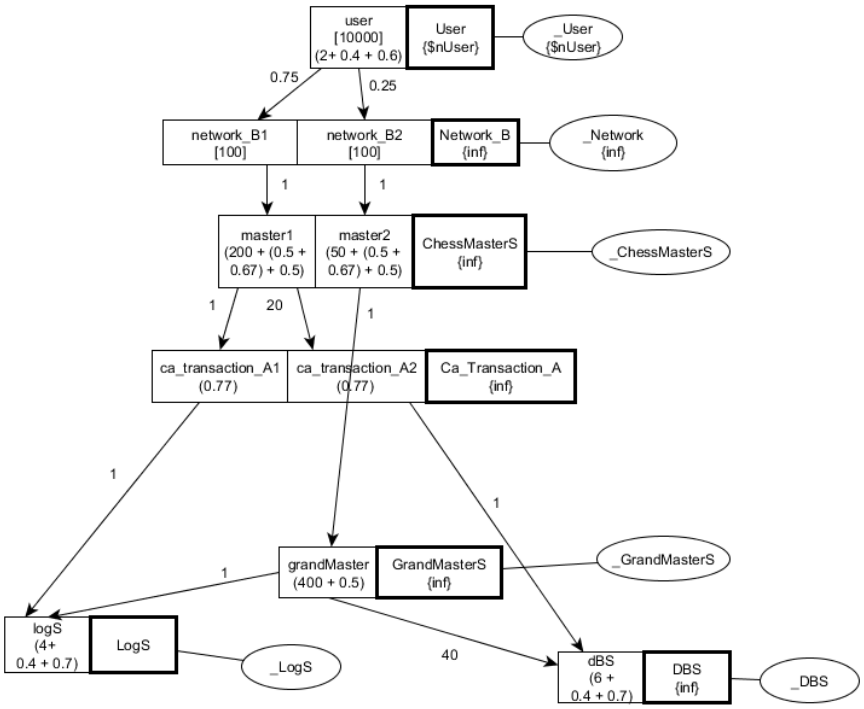


Figure 5.16 ChessGrandMaster application has been composed with Spring and many features

SAM4

This example is about the BAM3_Stock application. Since, the prices found in the StockCentral is distributed to all the three subscribed stock servers, to use a message queue is an obvious choice in this case. The makers of the Stock application wants to know how the application will performs if ActiveMQ with Security feature is used between the StockCentral and the stock servers. The communication in this level is asynchronous. For accessing database, the makers want to use remote method invocation with transaction feature. The MCD and the composed application model are presented in Listing 5.7 and Figure 5.17 respectively.

Listing 5.7 BAM3_Stock is composed with ActiveMQ to obtain SAM4

```
MCD 3
in BAM4_Stock  put SMM7_ActiveMQ_Security
   at StockCentral *
out auto
in auto put SMM3_RMI_Transaction
   at * _StockDB
out auto
in auto put SMM8_Network
    at _User _ChessMasterS
out SAM4
```

This example illustrates the strengths of MCD. The use of host names and asterisks enable to specify many calls with a few words. The composed model (Figure 5.17) shows that the composition algorithm strives to create models as compact as possible, with fewer tasks. Feature tasks are created per host. This is why the three stock servers have three tasks for using the transaction service in their outgoing calls. As the two databases share the same host, a single transaction task with many entries is created for them. If the modeler wants to keep the model even more compact, he/she could use PMR for the features instead of SMR.

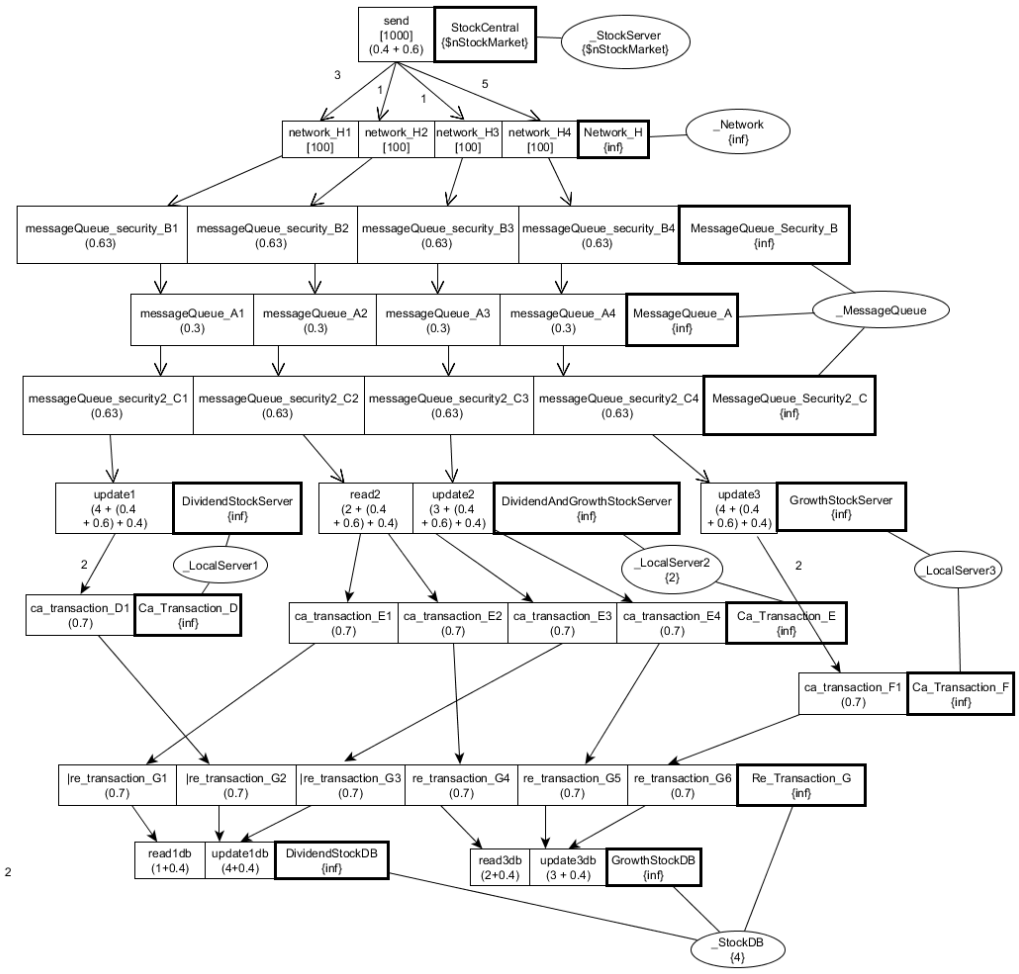


Figure 5.17 The composition of ActiveMQ to the Stock application gives SAM4

To conclude, this chapter describes the process of composing specialized middleware models (SMM) to the base application models (BAM) to obtain specialized application models (SAM) as output. The middleware composition descriptor (MCD) for specifying different combination of calls to which SMMs can be composed is defined. The composition process for carrying out the actual middleware composition, by using the middleware composition algorithm, is described. Finally, various examples of middleware compositions are presented. The next chapter presents experimental results for the calibration and validation of the MMLQ framework, by comparing the predictions obtained by solving the composed LQN models with measured experimental results.

6 Chapter: Experimental Analysis: Calibration and Validation

In this chapter the validity of the MMLQ framework is demonstrated by experimental analysis. The first section describes the measurement testbed. The second section describes the approach taken to calibrate specialized middleware models (i.e., finding the service demands of the middleware operations). The final section describes experiments to validate the middleware models and their calibrations on real distributed systems. This is done by composing the calibrated middleware models to some distributed software models (i.e., base application models) and comparing the model performance predictions with the measurements from the real systems. It is also shown how the MMLQ framework can be used to answer various performance related questions of distributed applications.

6.1 Measurement Testbed Description

The performance of a given software system depends heavily on the platform to which it is deployed. Testbed calibrations are affected by the hardware, software or network configuration of the testbed. Therefore, it is best to calibrate a model of middleware in the environment where the actual application software will be deployed, or in a test environment with a similar hardware and software configuration.

All experiments presented in this chapter are done in the Amazon Web Services (AWS) cloud (i.e., EC2 cloud). All tasks are deployed to the same availability zone of the cloud (in this case *us-west-2b*). All virtual machines are of type *m3.medium* [3]. Any m3.medium instance described in chis chapter has the following configuration:

- Memory: 3.75GB
- Computing units (ECU): 3 units (1 unit = 1.0 – 1.2 Ghz)

- vCPU : 1
- Storage: 4 GB SSD
- Architecture: 64 bit
- Network: moderate (maximum: 400 Mbit/sec, we found the measured average (using *iperf*) to be: 285 Mbit/sec)
- Operating system: Ubuntu 14.04.3 LTS (trusty)
- JDK version 1.8

6.2 The Middleware Calibration Process

Most calibration methods found in literature use the utilization law (U = XS) to find service demand [82]. But, the CPU utilization reported by the operating system is affected by various external factors [75]. In his experiments, the author have found that the CPU utilization reported by Linux's *top* command is not always reliable. The reported utilization value depends on the measurement interval and fluctuates over time. Therefore, a different approach was chosen to find service demands when possible (i.e., for DoBM and CBM). The approach is based on response times [57] determined at different nodes and it is applicable to any middleware that has a client-server architecture. Below, the approach is described for RMI, which is a distributed object based middleware. The same approach can be used to calibrate any distributed object based middleware or container based middleware, since both of these middleware platforms follow a client-server architecture. All the middleware platforms in the following experiments are used with their default configurations.

6.2.1 Calibrating the Wrapper Feature of Java RMI

The author's approach is an elaboration of this simple procedure:

1. Find the response time between two communicating processes for the middleware, communicating with a dummy service. If we subtract the delay of the dummy service, the remaining delay is due to the network latency and the middleware execution;
2. Deduct the round-trip network latency found from a ping command. This assumes one round-trip latency is included in the response time;
3. Divide the middleware overhead equally between the caller and receiver.

The elaboration uses additional processes to better divide the overhead between caller and receiver ends.

Even with the same hardware and set of features, the performance of a middleware may vary across hosts [15]. From our experiments, we have learnt that the performance may vary due to the implementation details of a feature (e.g., different ways to implement a security or compression feature), the version of the software being used (e.g., JVM 1.5 vs JVM 1.8), and whether middleware objects are created for every call or stored in a pool to be used by other calls. Therefore, in order to properly predict the performance of a base application over a middleware, it is proposed to measure a synthetic application that contains the implementation of the middleware as it will be used by the base application. This synthetic application is called a *Middleware Benchmark Application*.

A four-node Middleware Benchmark Application was built, where each node (except the last one) calls its subsequent node. The first node is the requester (i.e., caller), then there are two intermediate nodes (playing the role of both caller and receiver) and then the replier

(i.e., receiver) node. The response times are measured for the calls from the first 3 nodes (a response time cannot be measured at the last node as it does not invoke any other node). The requester node has request & receive operations at one end, the replier node has receive & reply operations at one end, and the intermediate nodes have all 4 kinds of operations. The rationale behind creating a four node application is that a two-node or a three-node system is too little to check if the middleware overhead is consistent across the system. The sequence diagram of Figure 6.1 shows how the four nodes communicate.

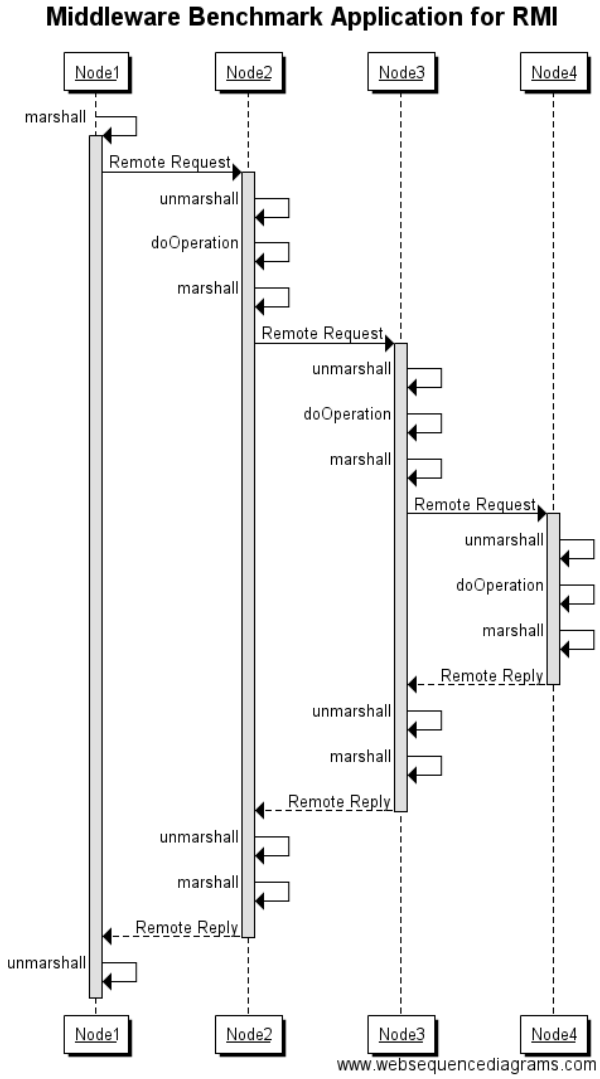


Figure 6.1 Sequence diagram of the Middleware Benchmark Application of RMI

Each node has some responsibility from a middleware perspective. Before sending a request or reply, each node has to marshall the message; similarly upon the reception of any message, each node has to unmarshall it. In terms of middleware overhead, it is expected that the middleware overheads of the request-only node (node 1) and that of the reply-only node (node 4) are about the same, because each of them is doing one marshalling and one unmarshalling operation. Also, it is expected that the middleware overheads of the two request-reply nodes (node 2 and node 3) are about the same and twice as large as the others, because each of them is doing two marshalling and two unmarshalling operations.

A Middleware Benchmark Application for RMI was written in Java (JDK 1.8). The application was deployed to four different m3.medium virtual machines in the AWS cloud. Each request contains no payload except for the request type, while each reply contains a payload of 1 KB. A dummy operation (doOperation) with a duration of 2 milliseconds was introduced to all nodes except the request-only node (node 1). Upon receiving a request, a node performs doOperation and then sends a subsequent request (node 2 and node 3) or replies (node 4).

For the proper calibration of the model, keeping a dummy operation in each node is important, especially in the final reply-only node. This keeps the reply-only node doing some operation, thus blocking the preceding node from overflowing it by message re-sending.

For the purpose of calibration, this middleware benchmark application (BMA-RMI) is run for 1 user, as more users may create contention. This single user submits a service request and waits for a response. Once a cycle of operations is complete, a reply is returned to the user. The user immediately makes the next call, meaning the user think time is set to zero, thus keeping the resources busy. The experiments were run 10 times, each time for 120

seconds. Response times obtained at each node (except the final node) are reported in Table 6-1.

Table 6-1 Measured response times of the Middleware Benchmark Application (RMI)

Run#	R at node 1 [ms]	R at node 2 [ms]	R at node 3 [ms]
1	10.212	6.702	3.337
2	10.108	6.797	3.347
3	10.193	6.845	3.414
4	10.382	6.920	3.410
5	10.144	6.682	3.417
6	10.108	6.682	3.389
7	10.100	6.935	3.430
8	10.233	6.798	3.388
9	10.222	6.810	3.420
10	10.290	6.782	3.408
AVG	10.19902	6.79519	3.39591

Each of these response times contain a network latency and a dummy operation of 2 ms at the replier node. So, to obtain the RMI wrapper overhead, these values need to be deducted from the measured response times. Now, the question is how to find the value of network latency.

The main contributors to network latency are transmission delay and propagation delay [58]. Transmission delay takes place at the communicating computers, so it is right to keep it as an overhead of the communication nodes. But the propagation delay takes place purely in the network and therefore it is not an overhead at the communicating nodes. The propagation delay is independent of the message size, it rather depends on the physical length of the network. Propagation delay between nodes are found by the *ping* command. For the present testbed, the results of the command "*ping destinationNode*" were averaged over 20 times to find the network delay. It was discovered that the network delays between all the

directly communicating nodes (e.g., node1 – node 2, node 2 – node 3 and node 3 – node 4) are 0.6 ms on average.

As mentioned earlier, to obtain the RMI overhead the network latency and dummy operation demand have to be deducted from the measured response time. This gives the summation of the wrapper overheads at the two communicating nodes. This result is divided by two to estimate the overhead at one end of each node. For example, let us consider the first row of the response time measurement table (Table 6.1). The RMI wrapper overhead at Node 3 (in ms) is:

```
(R at node 2 – dummy operation - network latency) / 2
= (3.337 – 2 – 0.6) / 2
= 0.368
```

The measured response times at Node 1 and Node 2 contain the response times of the subsequent nodes. So, in order to obtain RMI wrapper overheads at these nodes, these response times need to be deducted from the measured response time. Therefore, the RMI wrapper overhead at Node 2 (in ms) is:

```
(R at node 2 – dummy op. – net. latency - R at next node) / 2
= (6.702 – 2 – 0.6 – 3.337) / 2
= 0.383
```

The RMI wrapper overhead at Node 1 (in ms) is:

```
= (R at node 1 – dummy op. – net. latency – R at next node) / 2
= (10.212 – 2 – 0.6 – 6.702) / 2
= 0.455
```

The computed RMI Wrapper overhead of all 10 runs are reported in Table 6-2.

Table 6-2 RMI Wrapper feature overheads

Run#	Overheads measured at one end of node 1 [ms]	Overheads measured at one end of node 2 [ms]	Overheads measured at one end of node 3 [ms]
1	0.455	0.383	0.368
2	0.356	0.425	0.373
3	0.374	0.415	0.407
4	0.431	0.455	0.405
5	0.431	0.333	0.408
6	0.413	0.347	0.394
7	0.283	0.453	0.415
8	0.417	0.405	0.394
9	0.406	0.395	0.410
10	0.454	0.387	0.404
AVG	**0.402**	**0.400**	**0.398**
STDDEV	0.0524	0.0402	0.0157
CONF INT (95%)	0.037	0.029	0.011

Averaging over the average values of each node give a Wrapper overhead of 0.4 having 95% confidence interval of ±0.03. In other words, a pair of request & receive or a pair of receive & reply operations cost 0.4 ms. Any edge node (e.g., Node 1, Node 4) has 1 such pair of operations, but any intermediate node (e.g., Node 2, Node 3) has two such pair of operations at its two ends.

The calibrated value will be validated in Section 6.3.

6.2.2 Calibrating Security Feature of Java RMI

The process to calibrate the Security feature of the Java RMI is presented here. The same process can be used to calibrate other middleware features. The implemented Security feature is based on an encryption operation [92].

For this calibration, the previous middleware benchmark application (BMA-RMI) is used again, but with the simple security feature added. The security feature is added by implementing RMI socket factories and by adding encryption/decryption operations to the messages [43], [92]. An encryption operation is added before every request and reply, and a decryption operation is added upon every message reception. Then the response times are measured at node 1, node 2 and node 3. For this experiment, the request and reply message payloads are the same as the previous experiment, and the encryption key length is set to 128 bits. This experiment is run 10 times and response times are measured at the first 3 nodes.

Table 6-3 Response times using RMI with Security

Run#	R at node 1 [ms]	R at node 2 [ms]	R at node 3 [ms]
1	11.98	8.09	4.03
2	12.07	8.03	4.01
3	12.11	7.98	4.07
4	12.01	8.01	4.04
5	11.93	7.93	4.04
6	11.96	7.93	4.03
7	12.03	8.12	4.11
8	12.04	8.04	4.12
9	12.02	7.94	4.05
10	11.98	8.05	4.14
AVG	12.013	8.012	4.064

In order to derive the RMI Security feature overhead, the same approach that was used for RMI Wrapper is followed here. But in this case, apart from deducting network latency, dummy operation and measured response time at the nodes, the RMI Wrapper feature overhead is also deducted, then the result is divided by 2, to get the Security feature overhead at one end. The results of the overhead computations are shown in Table 6-4.

Table 6-4 RMI Security feature overheads

Run#	Overheads measured at one end of node 1 [ms]	Overheads measured at one end of node 2 [ms]	Overheads measured at one end of node 3 [ms]
1	0.245	0.330	0.315
2	0.320	0.310	0.305
3	0.365	0.255	0.335
4	0.300	0.285	0.320
5	0.300	0.245	0.320
6	0.315	0.250	0.315
7	0.255	0.305	0.355
8	0.300	0.260	0.360
9	0.340	0.245	0.325
10	0.265	0.255	0.370
AVG	**0.301**	**0.274**	**0.332**
STDDEV	0.0376	0.0311	0.0221
CONF INT (95%)	0.027	0.022	0.016

The computations of Security feature overhead for all 3 nodes for run number 1 (i.e, row 1 of Table 6.2) are presented below. The Security feature overhead at Node 3 (in ms) is:

```
(R at node 3 – dummy op. – net. latency - wrapper overhead) / 2
= (4.03 – 2 – 0.6 - 0.8) / 2
= 0.315
```

The Security feature overhead at Node 2 (in ms) like this:

```
(R_node2 – dummy op. – net. latency – R_next_node – wrapper) / 2
= (8.09 – 2 – 0.6 – 4.03 - 0.8) / 2
= 0.330
```

The Security feature overhead at Node 1 (in ms) is:

```
(R_node1 – dummy op. – net. latency – R_next_node - wrapper) / 2
= (11.98 – 2 – 0.6 – 8.09 – 0.8) / 2
= 0.245
```

Averaging over the average values of each node (from Table 6.4) gives a Security overhead of 0.302 ms having 95% confidence interval of ±0.022. In other words, a pair of encryption & decryption operations cost 0.302 ms. Any edge node (e.g., Node 1, Node 2) has 1 such pair of operations, and any intermediate node (e.g., Node 2, Node 3) has two such pairs of operations, one pair at each end.

The calibrated value will be validated in Section 6.3.

6.2.3 Effect of Message Size on Middleware Overhead

In Chapter 4 it is mentioned that the overhead caused by a feature is affected by its message size. Here, that claim is verified for the Wrapper feature of Java RMI middleware. The overhead increase due to message size is also shown.

The Middleware Benchmark Application (BMI-RMI) was run for message sizes ranging from 0 KB to 1000 KB, at an interval of 200 KB. The response times (R) were measured at node 1, node 2 and node 3, and the following response times are collected.

Table 6-5 Response times for different message size using RMI

Message size (KiloBytes)	R at node 1 [ms]	R at node 2 [ms]	R at node 3 [ms]
0	9.225	6.142	3.073
1	10.18	6.79	3.39
200	16.56	10.96	5.26
400	23.12	15.46	7.53
600	29.78	19.79	9.71
800	34.32	22.61	11.24
1,000	40.47	26.76	13.15

This data is plotted in the following graph. It is observed that the response time increases as the message size increases.

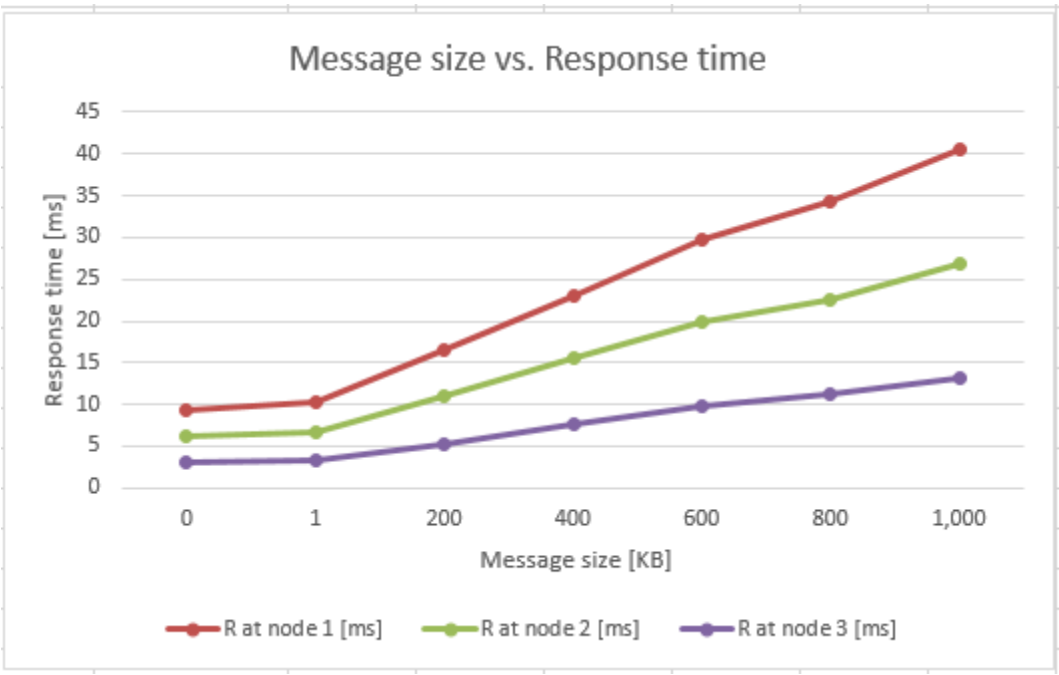


Figure 6.2 Response times for different message size using RMI

The measured wrapper overhead at a node is obtained by following the same approach used for RMI, i.e., deducting response time measured at the subsequent node, network latency (0.6 ms) and the service demand of the imposed operation (2 ms) from the measured response time at the current node, and then dividing it by 2. The results of this calculation are shown in Table 6-6.

Note that the response time for 0 KB message is not zero, which refers to a constant wrapper cost associated to any message to send. Moreover, the response time increment is almost linear to the message size.

Table 6-6 RMI Wrapper overheads for different message size

Message size (KiloBytes)	Overhead measured at node 1 [ms]	Overhead measured at node 2 [ms]	Overhead measured at node 3 [ms]	Average overhead
0	0.2415	0.2345	0.2365	0.24
1	0.395	0.4	0.395	0.40
200	1.5	1.55	1.33	1.46
400	2.53	2.665	2.465	2.55
600	3.695	3.74	3.555	3.66
800	4.555	4.385	4.32	4.42
1,000	5.555	5.505	5.275	5.45

The average overheads for different sizes are also shown in the graph below.

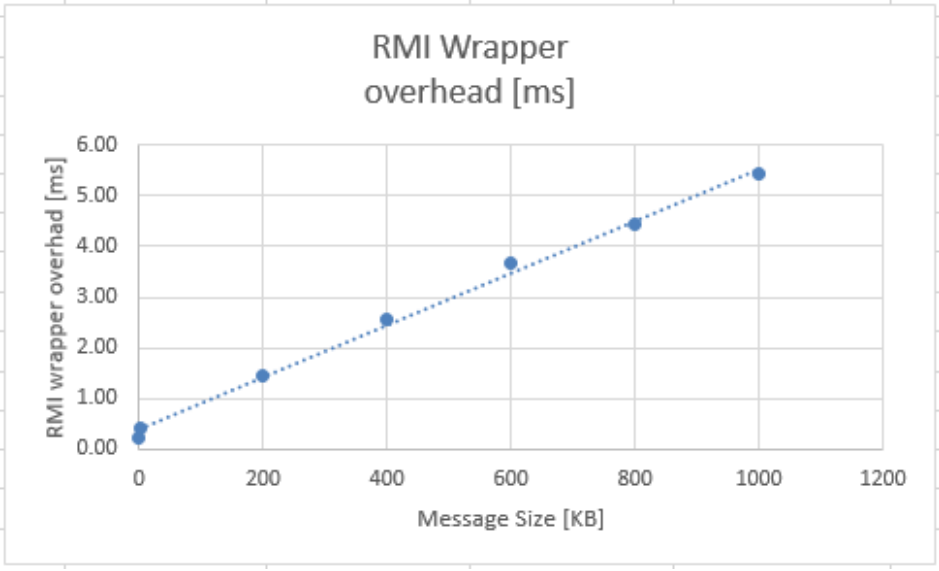


Figure 6.3 RMI wrapper overhead for different message sizes

A least-squares regression analysis [74] was done with message size as the independent variable and y = RMI Wrapper overhead as the dependent variable. The fit had $R^2 = 0.996$, indicating a very good fit. The regression line is:

```
y = 0.39 + 0.005 * (message size)
```

Therefore, for each KB increase in message size, the service demand of an RMI Wrapper operation is expected to increase by 0.005 ms.

6.2.4 Calibrating the JAX-RS (REST)

In this subsection the Container feature of REST is calibrated. For this experiment, REST will be deployed to the Apache Tomcat 7.0.57 container. The API being used is known as Java API for RESTful Web Services or JAX-RS in short [45]. There are several implementations of JAX-RS available. The presented experiment uses the Jersey implementation, which is an open source framework for developing RESTful Web Services in Java and serves as a JAX-RS (JSR 311 & JSR 339) Reference Implementation [46].

REST does not require marashal/unmarshal operations, instead it uses simple request-reply protocol over HTTP. Since REST services are deployed to a container, the transport layer protocol (e.g., TCP / UDP) overhead of REST can be modeled using the container feature. In this way, REST can be modeled as a container based middleware.

Calibration of a container based middleware such as REST is done following the same procedure as used for a distributed object based middleware. A synthetic middleware calibration model, called in this case a Middleware Benchmark Application for REST (MBA-REST) was developed, as was done for RMI (Section 6.2.1). MBA-REST is deployed to the same Amazon Web Services (AWS) cloud virtual machines. A single user having no think time was used to generate workload. Response times were measured at the three requesting nodes and the experiment was run for 10 times. Network latencies and dummy operation overheads were deducted from the observed response times to obtain REST overheads. Such a REST overhead contains the overhead of one request-receive-reply-receive cycle (i.e., request by this node, receive by next node, reply by next node and receive by this node). REST overheads measured at node 1, node 2 and node 3 are found to be 0.482,

0.548 and 0.458 milliseconds. By averaging them we get REST overhead of 0.495 ms having 95% confidence interval of ±0.02. Therefore, any node that performs REST request (or reply) and receive operations at one end, has an overhead of 0.495 ms due to these operations.

6.2.5 Calibrating Security Feature of REST

In JAX-RS (REST), the Security feature can be enabled in many ways [55]. There are many well-known methods such as Basic Auth, Digest Access Authentication, Asymmetric cryptography, OAuth, JSON web tokens etc. The present experiment uses the Basic Auth method.

In Basic Auth, the secured information (such as username, password, credit card information etc.) are encrypted and added to the header of the request message. The server decrypts the encrypted header information and cross checks against its stored information. The present experiment uses Base64 encoding. Since in Basic Auth only the sensitive information are encrypted rather than the whole message, the expected overhead is low.

Basic Auth mechanism was incorporated to all nodes of the MBA-REST application. The measurements followed exactly the same steps as for the REST calibration described in the last subsection.

Response times are measured at the three requesting nodes and REST + Security overheads are derived from them. Such a REST + Security overhead contains the overhead of one request-response cycle (encrypt & request by this node, receive & decrypt by next node, encrypt & reply by next node and decrypt & receive by this node). After deducting dummy operation and network latency overheads, REST + Security overheads at one end of node 1, node 2 and node 3 are 0.606, 0.595 and 0.603 ms, respectively. By subtracting the mean

REST overhead (0.495ms) found in the previous subsection, Security overhead is found to be 0.111, 0.100 and 0.108 at nodes 1, 2 and 3, respectively. The average value is 0.106 ms with a 95% confidence interval of ±0.02 ms. Therefore, any node that performs encrypted request (or reply) and receive operations at one end has a Security feature overhead of 0.106 ms.

Note that the Container overhead of REST (0.495 ms) is higher than that of RMI (0.4 ms). This is because REST messages are wrapped as JSON messages, which are text messages longer than marshalled RMI messages. On the other hand, the security overhead of RMI is higher (0.305 ms) than REST's Basic Auth security overhead (0.106 ms) because RMI encrypts the entire message body, but REST encrypts only the sensitive information contained in the message header.

6.2.6 Calibrating ActiveMQ

Both distributed object based middleware and container based middleware follow a client-server architecture, with the server and the client communicating in a synchronous manner across a network transport (TCP, UDP and so on). But in message based middleware, senders and receivers are decoupled. Messages are sent by one process to a broker, and messages are received from a broker by a different process. As a result of this architectural difference, the process of calibration that were used for distributed object base middleware and container based middleware (i.e., RMI and REST) in the last few subsections, cannot be used for a message based middleware such as ActiveMQ. This is due to the fact that dividing the system response time by three to get the communication overheads at the three communicating nodes (requester, broker and replier) does not work here, as the broker has

very different responsibilities than those of the requester and replier. Therefore, this experiment follows the well-known Utilization Law for calibrating ActiveMQ [82].

A Middleware Benchmark Application for ActiveMQ (MBA-ActiveMQ) containing three nodes is built. The three nodes are: a requester node, a broker node and a replier node. A synchronous request/reply architecture has been used for their implementation, meaning the requester expects to receive a reply for every request sent, and the requester sends a new request only upon receiving a reply of the previous request.

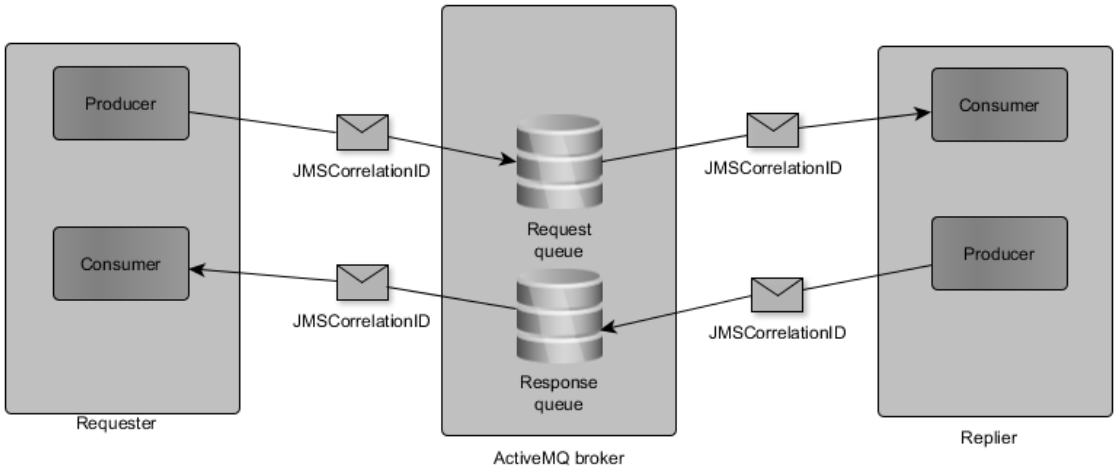


Figure 6.4 Middleware Benchmark Application for ActiveMQ

Both the requester and the replier contain both a producer and a consumer [84]. The communication starts from the requester. At first the requester's producer creates a request in the form of a JMS message and sets a couple of properties of the message, among which the most important two are: the correlation ID and the reply destination. The correlation ID allows the requests to be correlated with replies if there are multiple outstanding requests. The reply destination is where the reply is expected to be delivered. The requester then configures a consumer to listen on the reply destination.

On the other end, the replier receives a request, processes it and sends a reply message using the destination and correlation ID mentioned in the received message property. Upon receiving a reply, the requester needs to properly associate a reply with its request using the message properties. Since, the requester does more operations than a replier, it is expected that the requester's middleware overhead is greater than the replier's middleware overhead.

All the three nodes are deployed to m3.medium EC2 virtual machines in the AWS cloud. The experiment was run for a single user (i.e., requester) with a 10 ms think time. Throughputs and utilizations are measured at the requester, broker and replier nodes. The experiment was run 3 times. The average values of the measurements utilization are shown in Table 6-7. The average measured throughput was 0.085 jobs/ms. The service demands of these nodes are derived using the Utilization Law (U = X * S) and they are reported in the same table. Note that the requester's overhead is much higher than the replier's overhead as expected, due to the extra operations that take place at the requester.

Table 6-7 ActiveMQ middleware overhead

	Requester	Broker	Replier
Utilization (ActiveMQ)	0.016	0.033	0.010
Service demand (ActiveMQ) [ms]	0.188	0.388	0.118

Next, to calibrate ActiveMQ's Security feature, that feature was added to each of the nodes. There are two plug-ins available to securing ActiveMQ calls: *Simple Authentication plug-in* and *JAAS Authentication plug-in*. For the experiments presented in the thesis, the simple authentication plug-in was chosen. The simple authentication plug-in handles credentials

directly in the XML configuration file or in a properties file. Encryption and decryption operations took place at message header before/after every request and reply.

For the calibration of the Simple Authentication with encryption, an experiment was run for a single user (i.e., requester) with a 10 ms think time. Throughputs and utilizations are measured at the requester, broker and replier nodes. The experiment was run again 3 times. The average values of the measurements utilization are presented in Table 6-8. The average measured throughput was 0.082 jobs/ms. The service demands of these nodes are derived using the Utilization Law (U = X*S) and they are reported in the same table. ActiveMQ overheads reported in Table 6.7 are deducted from the (ActiveMQ + Simple Auth) overheads to obtain the Security overhead. Again, it was found that the requester's overhead is much higher than the replier's overhead.

Table 6-8 ActiveMQ Security overhead

	Requester	Broker	Replier
Utilization (ActiveMQ + Simple Auth)	0.022	0.035	0.013
Service demand (ActiveMQ + Simple Auth) [ms]	0.268	0.427	0.159
Service demand (Simple Auth) [ms]	0.080	0.039	0.041

The calibrated values of all 3 middleware platforms are summarized in Table 6-9.

Table 6-9 Calibrated values of all 3 middleware platforms

Middleware	Role	Wrapper [ms]	Security [ms]
RMI	Everybody	0.4	0.305
REST	Everybody	0.495	0.106
ActiveMQ	Caller	0.188	0.08
	Broker	0.388	0.039
	Receiver	0.118	0.041

6.3 An Airline Reservation System (ARS) case study

This section presents an end-to-end case study representing an Airline Reservation System (ARS) to demonstrate the usefulness of the proposed framework when different kinds of middleware are used, including the calibration approach in the previous section.

The ARS is a distributed system written in Java (JDK 1.8) and deployed in the Amazon Web Services (AWS) cloud (i.e., EC2). The initial version has only one class of traffic called 'browse'. A browse operation is generated by a user and goes through a controller to be dispatched to a service layer and eventually to a catalog server. The catalog server replies to the service layer with product information which is sent to controller and then to the user.

An LQN model of the ARS software without middleware was created and calibrated. This is the base application model, to be composed with the middleware models that were calibrated in the previous section. The composed models were used to make predictions about the performance of the ARS software under different middleware and middleware features. The ARS software was then updated with the same middleware platforms, and its performance was compared with the model predicted results.

The base application model for ARS (browse) is presented below.

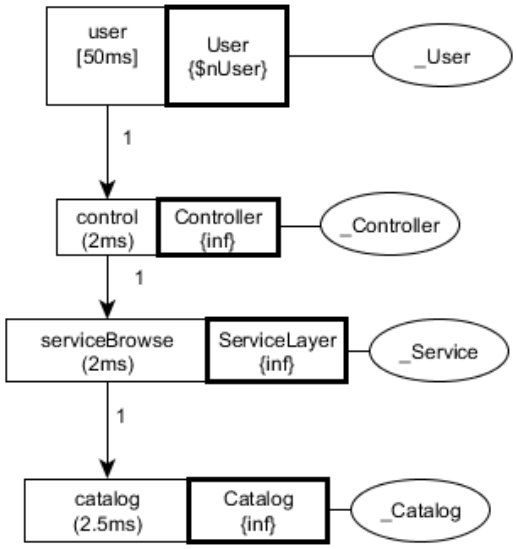


Figure 6.5 Base application model ARS_Browse

The user requests go to a control service. The control service works as a dispatcher and sends the requests to the service layer, which implements the business logic of the system. The serviceBrowse operation of the service layer sends a request to the catalog server, which provides the data layer. In this implementation the data layer uses a plain text file to store information. The service demands of the control, serviceBrowse and catalog operations are 2ms, 2ms and 2.5 ms, respectively. A user has a think time of 50 ms. Each request has zero KB payload and each operation replies to its caller by sending back a message of average length one KB.

All tasks are deployed to the same availability zone of the Amazon cloud (in this case *us-west-2b*) and all virtual machines are of type *m3.medium,* as described in Section 6.1.

6.3.1 Performance Questions

We suppose that the makers of the ARS software want to know how it would perform under different middleware platforms, features and modified functionality. In particular,

they are interested to know the system response time and system throughput of the software for various changes.

The makers of the ARS software want to have two different versions of the software, one for distributed desktop clients and the other for web clients. For the distributed desktop version the preferred middleware is RMI and for web version the preferred middleware is REST. The team then would like to add a purchase operation to the system that will allow users to also purchase airline tickets. The team is interested in knowing the system performance for this added operation. Since purchase requests contain sensitive information (such as a credit card number), the team would like to make the purchase operations secure through security and check how the security feature affects the system performance.

Some vendors claim that the use of message based middleware improves system performance [84] because of its inherent decoupled nature. The team therefore also wants to check whether introducing a JMS broker (such as ActiveMQ) would improve the systems performance.

In summary, the performance analysts need to answer the following questions.

1. What will be the performance (i.e., System response time and System throughput) of ARS (browse) software when RMI is used as middleware?
2. After adding the 'purchase' functionality to the software and making the purchase related operations secured, what will be the system performance?
3. What will be the performance of ARS (browse and purchase) software when REST is used as middleware?

4. Which is the bottleneck server? Will the system performance be improved if ActiveMQ is used to make calls to the bottleneck server?

Along with answering these questions, the calibrated middleware models composed with the application model will also be validated. Changes will be made to both the LQN models and to the ARS Java software, and LQN predicted values will be compared to the measured values for the software system. Through answering question 1 the RMI middleware model is validated, through answering question 2 RMI's Security feature is validated, through question 3 both REST and REST's Security feature will be validated and finally through answering question 4 the ActiveMQ middleware and its Security feature will be validated.

What is an "adequate" prediction? As a guideline, this work follows Das et al. [82] where it is stated that according to Lazowska et al. [60] an error of less than 10% for system throughput and less than 30% for system response time is generally acceptable.

Note that, in reality, if the model's predicted values do not meet the performance requirements, the team would probably not proceed with implementing it.

6.3.2 Validating a Java RMI model

This section considers how the ARS (browse) software will perform if Java RMI is used as middleware. Section 4.3.1 shows a model of DoBM that can be used for Java RMI (repeated in Figure 6.6), and Section 6.2.1 derives a calibration for the parameters, which together give the calibrated SMM in Figure 6.7.

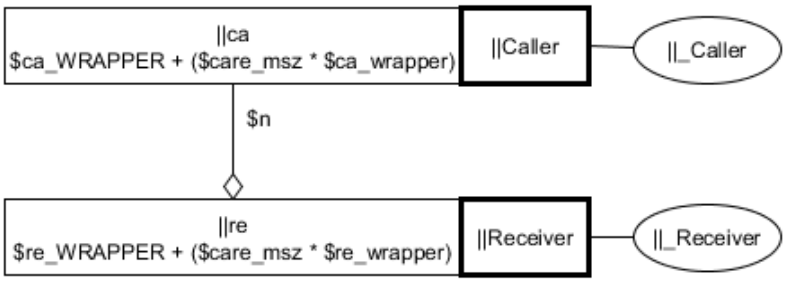


Figure 6.6 Specialized middleware model of RMI (SMM_RMI)

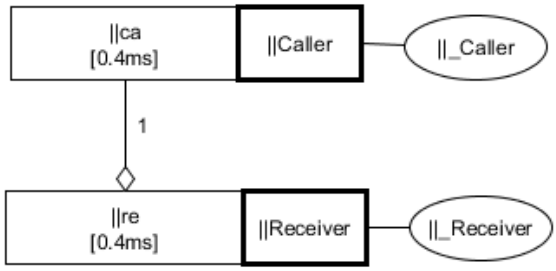


Figure 6.7 Java RMI calibrated specialized middleware model (SMM_RMI)

The round-trip network latency between a pair of communicating tasks in Amazon EC2 was found to be 0.6ms on average. The network latency of the whole system is lumped together and modeled as a single task (a simplification that is supported by LQNS). So, for four tasks the total network latency is 0.6ms * (4-1) = 1.8ms. The network feature realization shown in Chapter 4 is calibrated with this measured value, producing the following network model, that we call 'net1KB'.

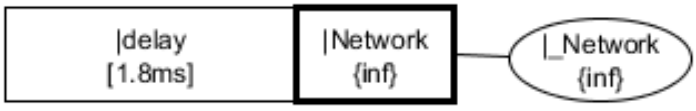


Figure 6.8 Calibrated network model (net1KB)

This model is composed with the base application model ARS_Browse to obtain the intermediate specialized application model. Finally, the network feature is composed to obtain

the desired specialized middleware model. The corresponding middleware composition descriptor and the specialized application model are presented below.

Listing 6.1 Middleware composition descriptor to compose ARS with RMI

```
MCD 2
in ARS_Browse  put SMM_RMI
   at * *
out auto
in auto  put net1KB
   at _User _Controller
out ARS_v1
```

The code above composes SMM_RMI with every call in the model. This means that the user and catalog entries demands are increased by an overhead of 0.4ms each, and the control and serviceBrowse entries by an overhead of 0.8ms each. The network model net1KB is also composed. The resultant specialized application model is shown below.

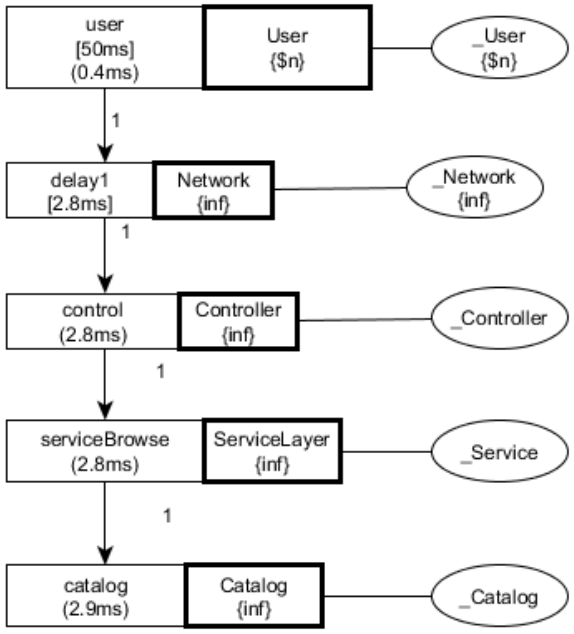


Figure 6.9 ARS composed with RMI (ARS_v1)

The LQN model ARS_v1 was solved for different numbers of users between 1 to 60, to give predictions of the system throughputs and system response times. The ARS software was also run for the same numbers of users and their system throughputs and response times were measured. The predicted and measured values are reported in the following two tables.

Table 6-10 ARS system response time where RMI is the middleware

N_user	R_Meaured[ms]	R_LQN[ms]	Absol. Error in %	U_CatalogServerH
1	10.78	10.75	0.28%	0.048
5	12.68	12.53	1.18%	0.23
10	16.56	15.67	5.37%	0.44
20	29.78	26.5	11.01%	0.76
30	49.76	44.17	11.23%	0.92
40	73.69	67.89	7.87%	0.98
50	101.64	95.49	6.05%	0.99
60	132.06	124.18	5.97%	1
Average			**6.12%**	

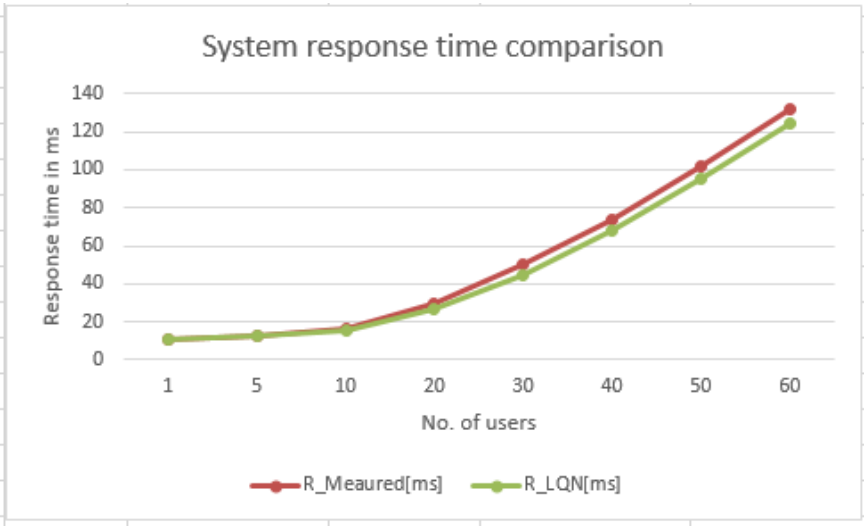


Figure 6.10 ARS system response time where RMI is the middleware

Table 6-11 ARS system throughput where RMI is the middleware

N_user	X_Meaured[ms]	X_LQN[ms]	Absol. Error in %	U_CatalogServerH
1	0.01623	0.0165	1.66%	0.048
5	0.0805	0.08	0.62%	0.24
10	0.148	0.152	2.70%	0.45
20	0.246	0.261	6.10%	0.76
30	0.2943	0.319	8.39%	0.92
40	0.3048	0.339	11.22%	0.98
50	0.305	0.344	12.79%	0.99
60	0.3036	0.344	13.31%	1
Average			**7.10%**	

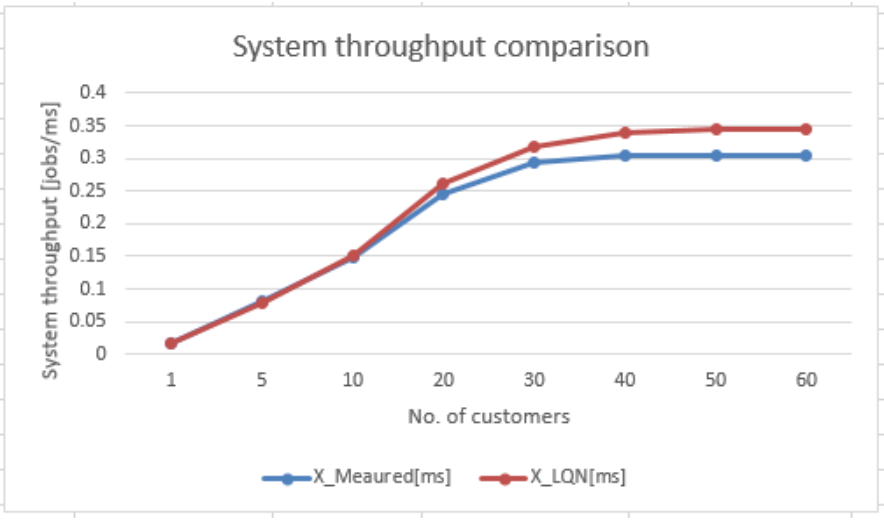


Figure 6.11 ARS system throughput where RMI is the middleware

From the data above, several things are observed. The System throughput increases with the number of users until the system reaches saturation at 40 users. At this point the CatalogServer host is the bottleneck. The absolute error of throughput prediction is less than 7% till 20 users, then the absolute error increases but still less than 14% at 60 users when CatalogServer host is fully utilized (100%). The average absolute error in throughput prediction is 7.10%. When the number of users is low (between 1 to 10 users) the system response time increase slowly. But as the number of users increases, so does the rate of increase in the system utilization and system response time. Between 40 to 60 users, the

system response time increases almost by 30 ms for every 10 users. The average absolute error in the prediction of system response time is 6.12%.

6.3.3 Validating a RMI Security Feature Model

For this experiment the ARS software is updated to handle a purchase operation. Purchase requests from a user go through the same two servers: control server and service layer and then they go to a purchase server. New classes are added to the Java program to reflect these changes. The updated LQN base application model of ARS software is shown below. This model is called 'ARS_Browse_Purchase' (shown in Figure 6.12).

For this experiment, the system will still use RMI as middleware, but the purchase operations are secured through a Security feature. Therefore, any request that goes from 'user' to 'catalog' uses the same RMI specialized middleware model that was used in the previous subsection, but the requests going from 'user' to 'purchase' use secured RMI calls. For the purchase operation, the Security feature needs to be added to the RMI middleware model. In chapter 4, the specialized middleware model of Wrapper + Security features for DoBM was presented. Figure 4.9 presents the model where the Security feature was composed to both caller and receiver tasks. This model is repeated in Figure 6.13 and it is called SMM_RMI_Security.

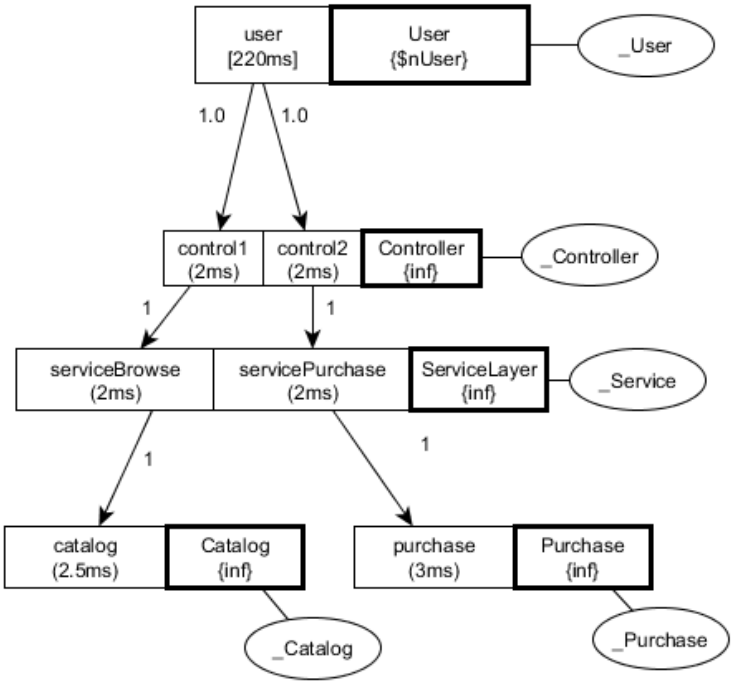


Figure 6.12 LQN Model of ARS_Browse_Purchase

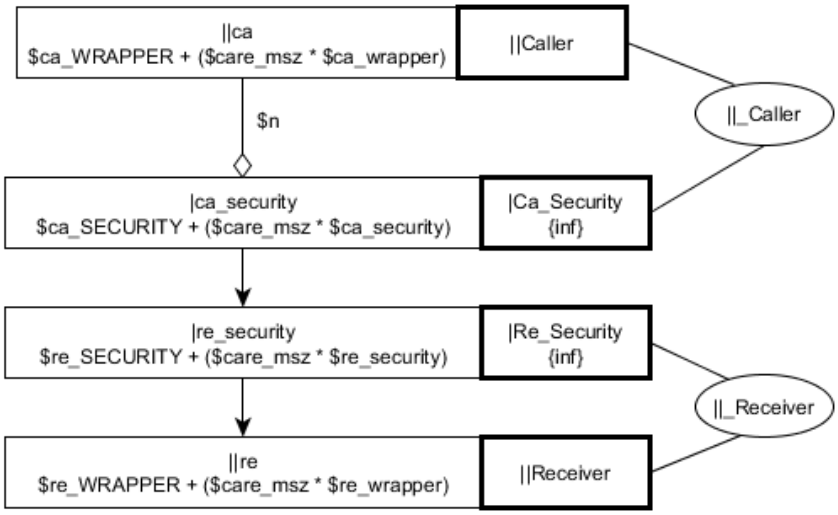


Figure 6.13 Java RMI middleware model with security feature (SMM_RMI_Security)

The calibration of the Security feature of RMI was described in section 6.2.2. It was found that for our measurement testbed, the service demand of the security operation is 0.305ms. This value is used to obtain the calibrated specialized middleware model in Figure 6.14.

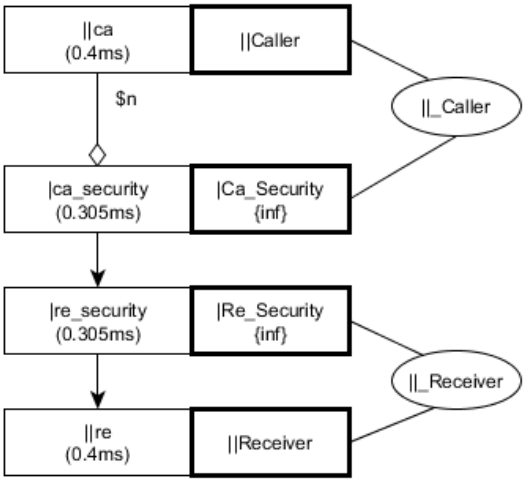


Figure 6.14 Java RMI + Security middleware model with calibrated values

The FCD in Listing 6.2 is used to compose RMI and RMI + Security specialized middleware models to ARS_Browse_Purchase. The two specialized middleware models to compose are SMM_RMI and SMM_RMI_Security.

Listing 6.2 Middleware composition descriptor to compose ARS with RMI and Security

```
MCD 3
in ARS_Browse_Purchase  put SMM_RMI
   at user control1
   && control1 serviceBrowse
   && serviceBrowse catalog
out auto
in auto  put SMM_RMI_Security
   at user control2
   && control2 servicePurchase
   && servicePurchase purchase
out auto
in auto  put net1KB
   at _User _Controller
out ARS_v2
```

The composed specialized application model is Figure 6.15.

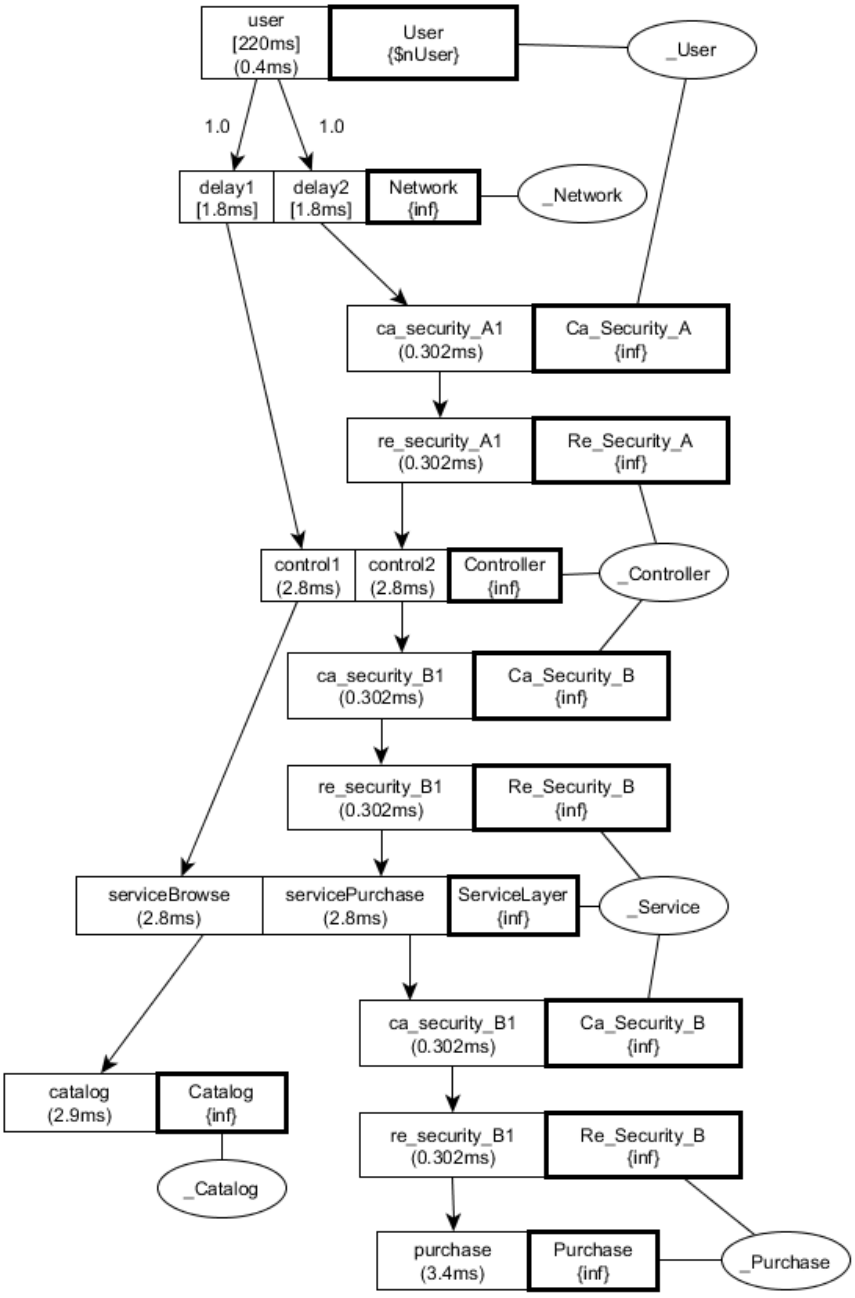


Figure 6.15 ARS composed with RMI and Security (ARS_v2)

This model was solved for different numbers of users from 1 to 40, to get the system response times and throughputs. The ARS software was also run in the EC2 cloud and its

performance was measured. The predicted and actual performance metrics are presented in the figures and tables below.

It is observed that the introduction of the purchase class has moved the bottleneck from the Catalog server host to the Controller server host. The utilization of the new bottleneck server is also reported in the following table.

Table 6-12 System response time of ARS having RMI middleware with Security feature

N_user	R_Meaured[ms]	R_LQN[ms]	Absol. Error in %	U_ControllerServerH
1	12.94	12.96	0.15%	0.026
5	13.45	14.07	4.61%	0.132
10	15.67	15.97	1.91%	0.262
20	24.34	22.93	5.79%	0.508
30	33.22	30.3	8.79%	0.715
40	52.55	46.53	11.46%	0.866
Average			**5.45%**	

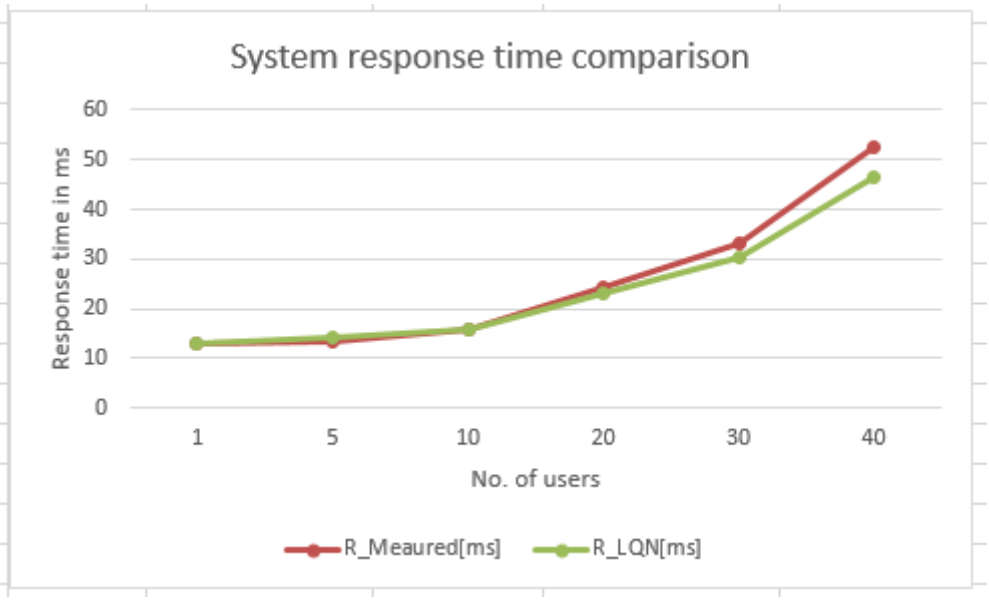


Figure 6.16 System response time of ARS having RMI middleware with Security feature

Table 6-13 System throughput of ARS having RMI middleware with security feature

N_user	X_Meaured[ms]	X_LQN[ms]	Absol. Error in %	U_ControllerServerH
1	0.0043	0.0043	0.00%	0.026
5	0.0224	0.0214	4.46%	0.132
10	0.0438	0.0424	3.20%	0.262
20	0.0792	0.0823	3.91%	0.508
30	0.1152	0.119	3.30%	0.715
40	0.138	0.15	8.70%	0.866
Average			**2.02%**	

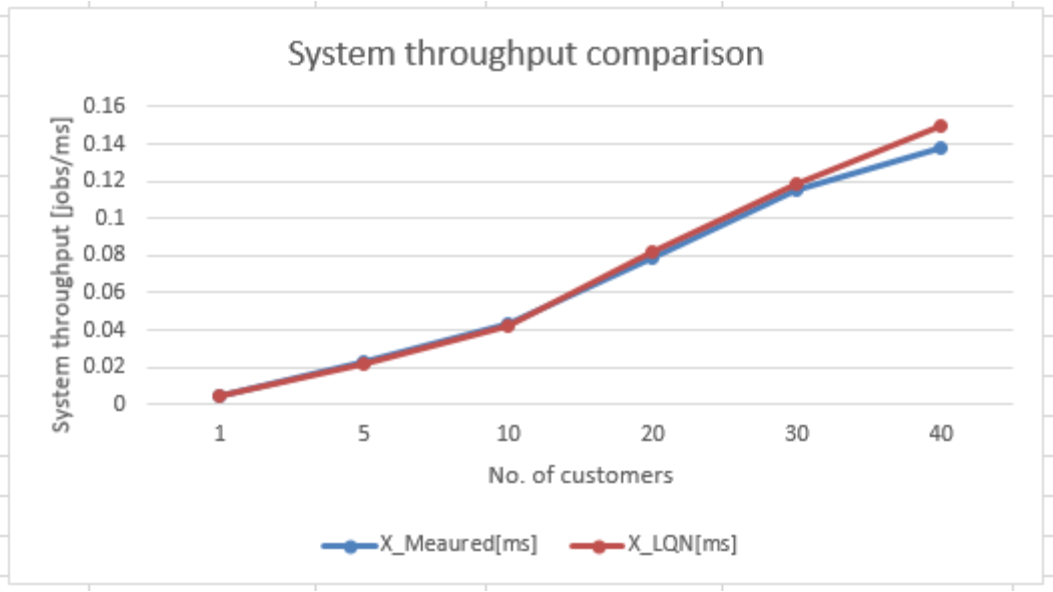


Figure 6.17 System throughput of ARS having RMI middleware with security feature

The system response time increases with the number of users, as expected. The measured response times are slightly higher than the predicted response times, which implies the model is optimistic. Yet, the average absolute error in response time is pretty low, less than 6% for response time and around 2% for throughput. The error is low when the bottleneck server (controller server) is less utilized and the error increases as the system workload increases towards saturation. Note, that the software experiments were run for up to 40 users, because after that the RMI server stopped accepting new user threads and started to

drop new requests. The reason for this behavior might be the number of thread limits of RMI server. Investigating the reason for such behavior could be a possible future work.

6.3.4 Validating a REST model

For this experiment a web version of the ARS (browse and purchase) software was developed. The software developers have decided to use REST (to be deployed to an Apache Tomcat 7.0.57 container) as their desired middleware for this version. To predict its performance, the ARS_browse_purchase base application model is composed with REST and REST + Security middleware models for the browse and purchase requests, respectively. The specialized middleware models for CBM and CBM + Security are presented in Chapter 4 (Figure 4.16 and Figure 4.17). In Sections 6.2.4 and 6.2.5 it was found that the service demands for the REST's Container and Security features are 0.495 ms and 0.106 ms, respectively. The calibrated specialized middleware models for REST and REST + Security are presented below.

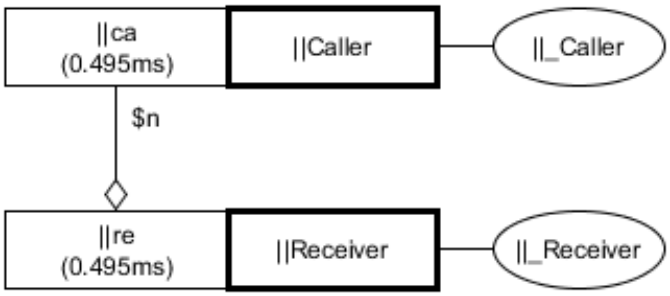


Figure 6.18 REST middleware model (SMM_REST)

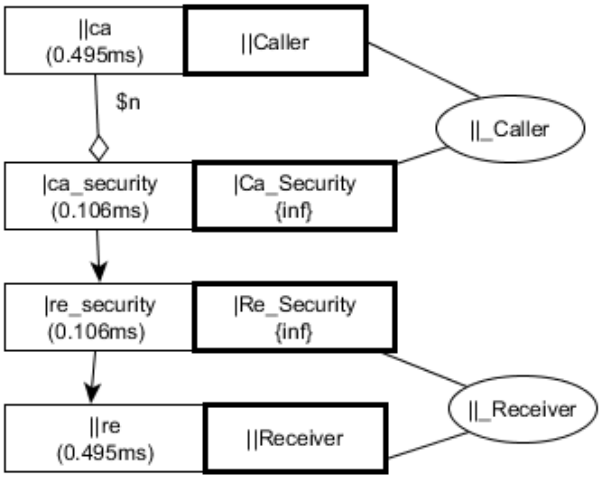


Figure 6.19 Security feature is composed to SMM_REST producing SMM_REST_Security

These specialized middleware models SMM_REST and SMM_REST_Security are composed to the base application model ARS_Browse_Purchase's browse and purchase path respectively. A network latency feature task is also composed. The composition is very similar to the one that we used for RMI Wrapper and Security feature compositions. The Middleware Composition Descriptor for these compositions are presented below.

Listing 6.3 Middleware composition descriptor to compose ARS with REST and Security

```
MCD 3
in ARS_Browse_Purchase  put SMM_REST
   at user control1
   && control1 serviceBrowse
   && serviceBrowse catalog
out auto
in auto  put SMM_REST_Security
   at user control2
   && control2 servicePurchase
   && servicePurchase purchase
out auto
in auto  put net1KB
```

```
    at User Control
out ARS_v3
```

This middleware composition descriptor produces the specialized application model shown in Figure 6.20.

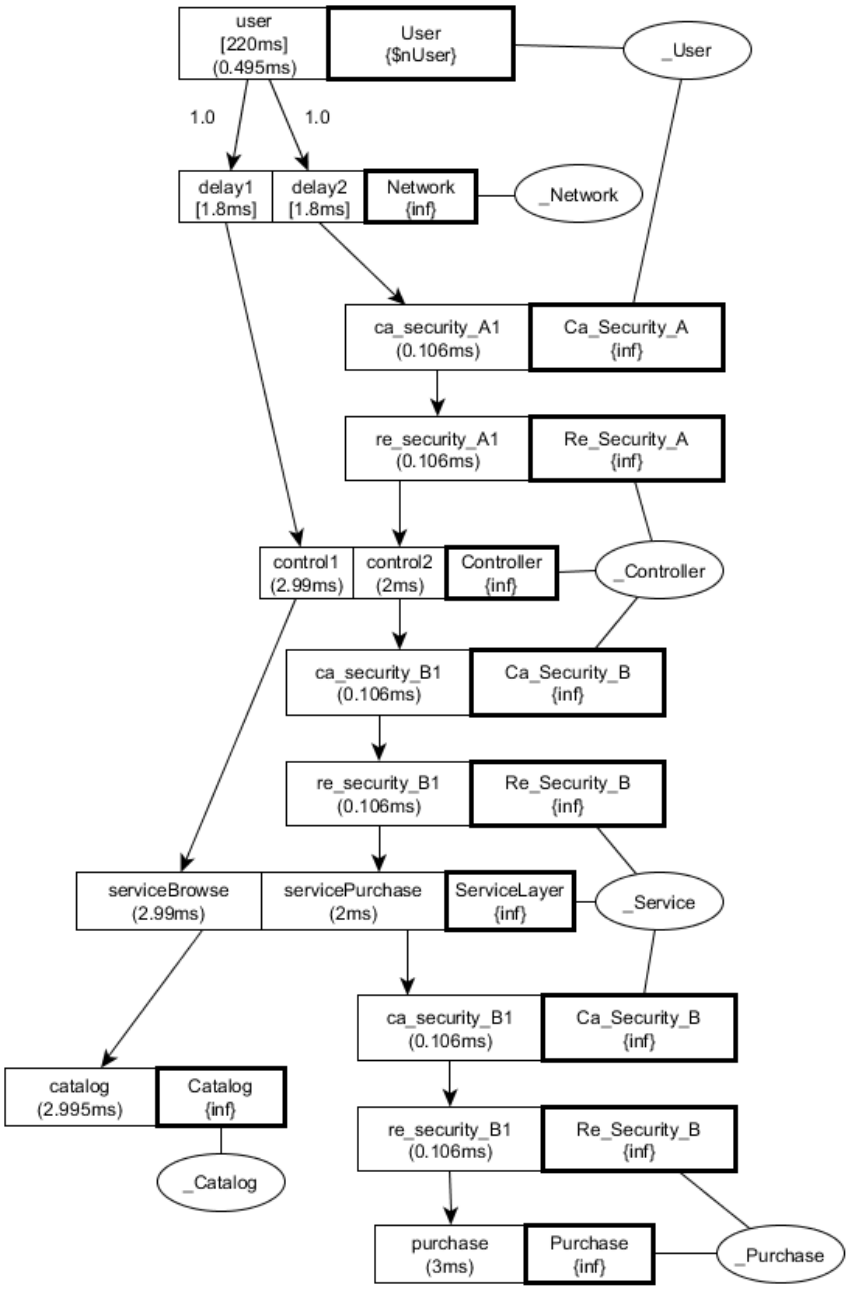


Figure 6.20 ARS composed with REST and Security (ARS_v3)

The composed specialized application model ARS_v3 was run for different number of users from 1 to 45 to obtain predictions about system response time and system throughput. The ARS (browse and purchase) software was also modified to use REST as middleware, with Security added to the purchase operations. The updated ARS software was run with same number of users and performance metrics were measured. The predicted and measured system response times and system throughputs are presented in Table 6.12 and Table 6.13.

Table 6-14 Response times of ARS_v3

N_user	R_Meaured[ms]	R_LQN[ms]	Absol. Error in %	Utilization of Controller Server Host
1	13.8	13.44	2.61%	0.026
5	15.12	14.79	2.18%	0.132
10	18.07	17.04	5.70%	0.262
20	27.73	25.72	7.25%	0.504
30	36.57	34.7	5.11%	0.721
40	62.7	56.91	9.23%	0.875
45	85.51	77.1	9.84%	0.914
Average			**5.35%**	

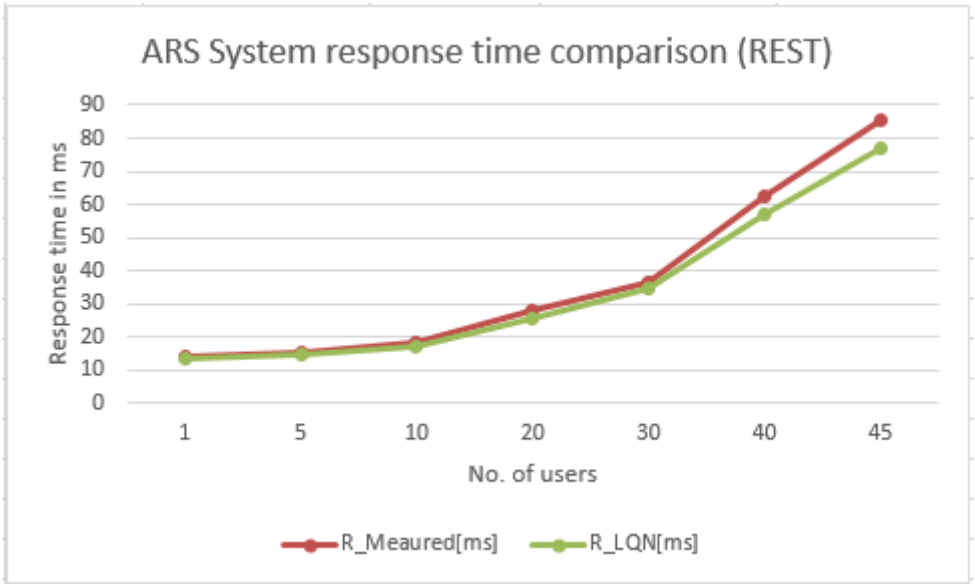


Figure 6.21 Response times of ARS_v3

Table 6-15 System throughputs of ARS_v3

N_user	X_Meaured[ms]	X_LQN[ms]	Absol. Error in %
1	0.00419	0.00428	2.15%
5	0.0213	0.0213	0.00%
10	0.0416	0.0422	1.44%
20	0.078	0.0814	4.36%
30	0.114	0.118	3.51%
40	0.14	0.144	2.86%
45	0.147	0.151	2.72%
Average			**2.43%**

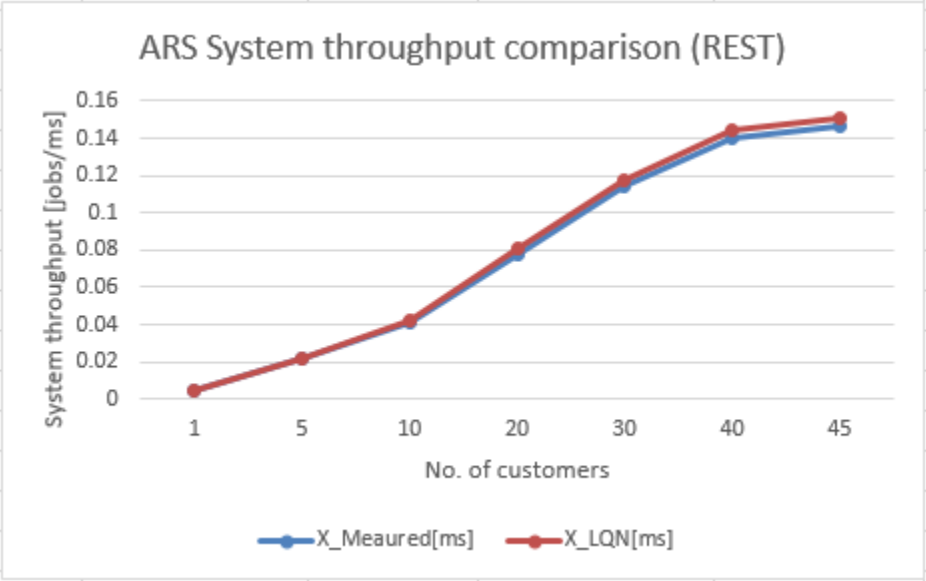


Figure 6.22 System throughputs of ARS_v3

The prediction error for REST is low. For response time measurement, the prediction error is about 5% and for throughput it is less than 2.5%. The system response time increases with the number of users. The system response time of REST-based software is higher than that of the RMI-based software. For example for 40 users the system response time for RMI-based and REST-based ARS software are 52.55 and 62.7 respectively. The LQN solver was unable to run above 45 users as the bottleneck server (in this case the controller) was fully saturated.

6.3.5 Validating an ActiveMQ model

This section shows the impact of introducing ActiveMQ message based middleware to the ARS software. Message based middleware are known for their performance, scalability and security. It is expected that if a client-server system is replaced by a message based system then the performance of the system should improve because of the highly scalable broker architecture [84]. In the following experiment, this expectation is verified and along with that the proposed ActiveMQ model is validated.

In the previous experiment (Section 6.3.4) the ARS software was run with REST middleware and a Security feature. It was observed that the controller server was the bottleneck in that case. Therefore, in this experiment the goal is to achieve better performance by reducing the workload of the controller. This is done by introducing an ActiveMQ broker between the user and the controller. Requests for browse operations go from user to the broker's queue, from where it is picked up by the controller. Upon generating a response, the controller replies to the broker and the broker forwards it to the user. Requests for purchase operation follow the same steps with the addition of the Security feature, which is implemented using the Simple Authentication plug-in mentioned in Section 6.2.6.

The specialized middleware models for MessageQueue and MessageQueue with Security are presented in Section 4.3.8 (Figure 4.18 and Figure 4.20), so they are not repeated here. The values used to calibrate these two features are presented in Section 6.2.6. The middleware composition descriptor for the message based middleware composition is given below.

Listing 6.4 Middleware composition descriptor to compose ARS with REST, ActiveMQ and Security

```
MCD 5
in ARS_Browse_Purchase  put SMM_REST
   at control1 serviceBrowse
   && serviceBrowse catalog
out auto
in auto  put SMM_REST_Security
   at control2 servicePurchase
   && servicePurchase purchase
out auto
in auto  put SMM_Wrapper_MessageQueue
   at user control1
out auto
in auto  put SMM_Wrapper_MessageQueue_Auth
   at user control2
out auto
in auto put net1KB_4hops
   at User Control
out ARS_v4
```

The addition of Message Queue causes to add one more hop to the network of the software. This needs a recalibration of the Network model. We call this recalibrated model as net1KB_4hops. The delay of the Network feature task is set to 0.6 * 4 = 2.4 ms. The resultant composed model is shown in Figure 6.23.

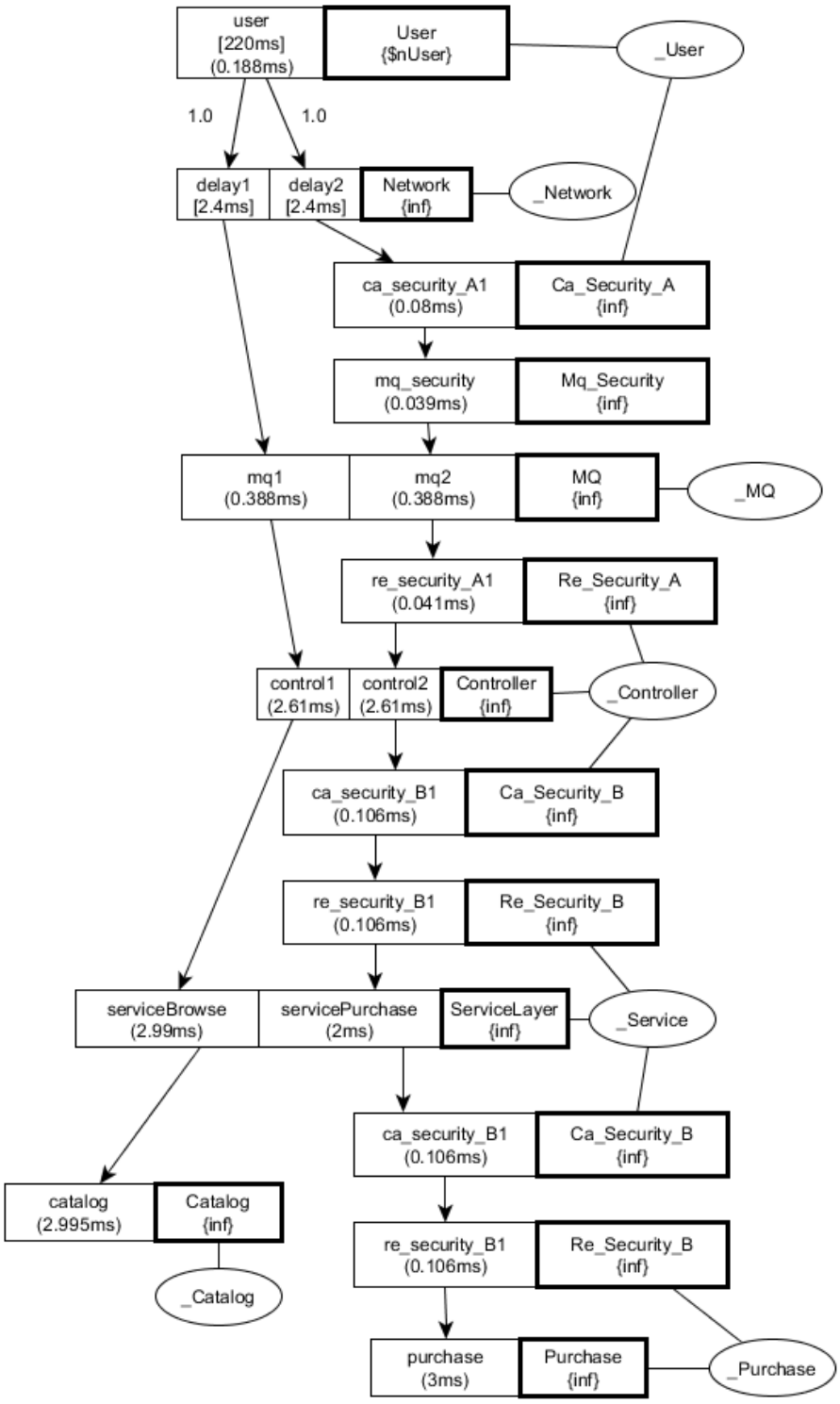


Figure 6.23 ARS_v4 where REST and ActiveMQ are used together

The model solution shows that the system response time and system throughput are better than the previous system where ActiveMQ was not used. The ARS software was also enhanced with the ActiveMQ middleware, deployed to the EC2 cloud and its performance was measured for various numbers of users. The model-predicted and measured system response times and system throughputs are presented below.

Table 6-16 System response times of ARS_v4

N_user	R_Meaured[ms]	R_LQN[ms]	Absol. Error in %	Utilization of Controller Host	Utilization of Broker Host
1	12.515	13.13	4.91%	0.023	0.00333
5	13.77	14.24	3.41%	0.115	0.0166
10	16.96	16.19	4.54%	0.227	0.0328
20	26.21	24.25	7.48%	0.439	0.0635
30	38.76	37.01	4.51%	0.623	0.0906
40	49.51	42.53	14.10%	0.805	0.118
45	61.45	52.79	14.09%	0.868	0.128
50	82.41	68.74	16.59%	0.91	0.134
Average			**8.70%**		

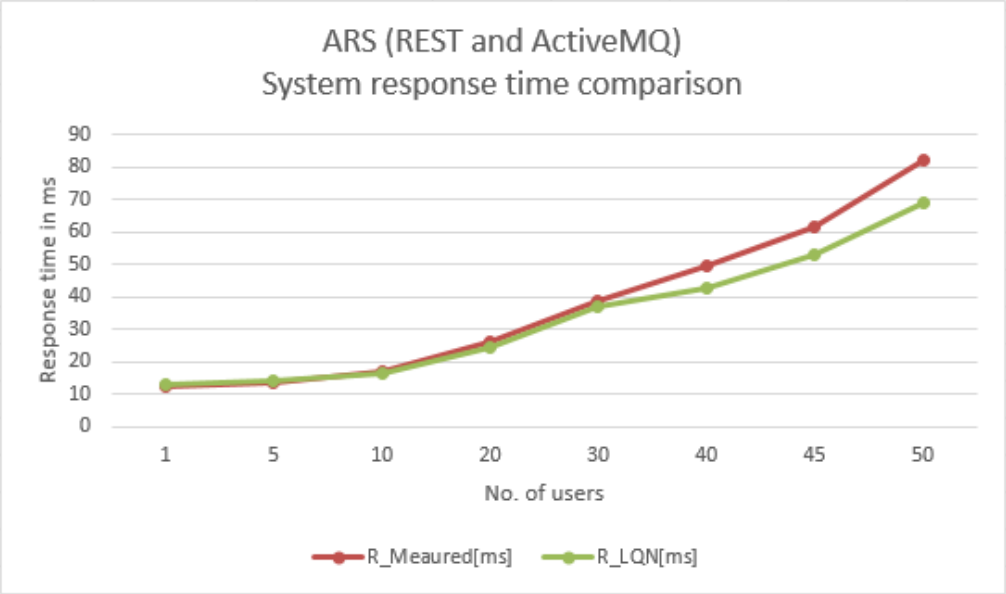


Figure 6.24 System response times of ARS_v4

Table 6-17 System throughputs of ARS_v4

N_User	X_Meaured[ms]	X_LQN[ms]	Absol. Error in %
1	0.004223	0.00429	1.59%
5	0.0208	0.0213	2.40%
10	0.0414	0.0423	2.17%
20	0.0789	0.0819	3.80%
30	0.114	0.117	2.63%
40	0.142	0.152	7.04%
45	0.14805	0.165	11.45%
50	0.154	0.173	12.34%
Average			**5.43%**

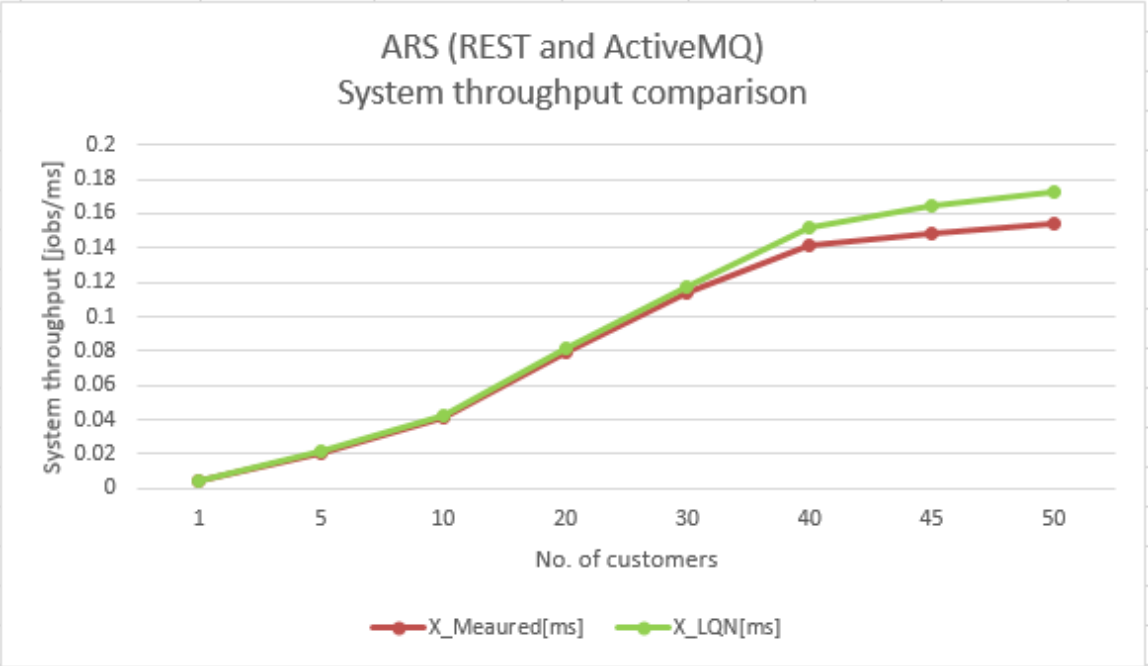


Figure 6.25 System throughputs of ARS_v4

Besides the measured values presented in the above tables, we can obtain the utilization of the servers by applying Little's law [60]. An interesting difference between this system (ARS_v4) and the previous system (ARS_v3) is that this one has a better performance than the previous one. For a single user, the system response times for ARS_v3 and ARS_v4 are 13.8 ms and 12.515 ms, respectively, which is a 9.42% improvement. For ARS_v3, the

system could sustain a maximum of 45 users with 85.51 ms response time and 91.4% of utilization of the bottleneck (i.e., Controller host), whereas for ARS_v4, 91% utilization of the bottleneck is reached at 50 users (11% more users) with a response time of 82.41 ms. ARS_v4 also has a slightly higher (< 1%) throughput than ARS_v3. So, the experiments show that the use of ActiveMQ middleware improves the system performance.

The figure below shows the modeled system response times of the ARS (browse and purchase) system with 3 different combinations of middleware. What is important is that the models predict the performances always towards the right direction.

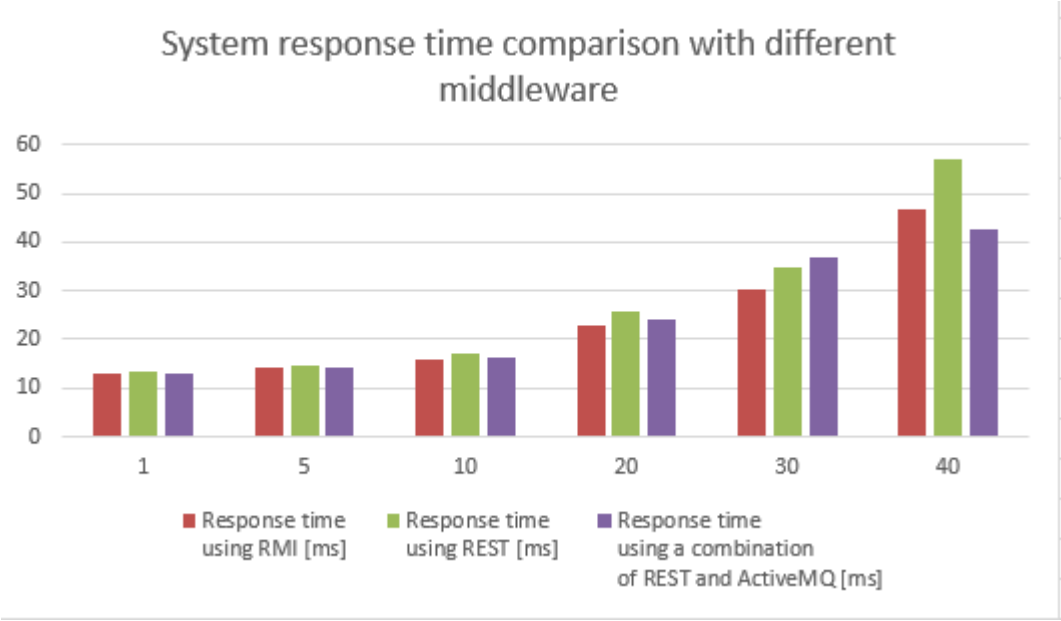


Figure 6.26 Comparison of middleware performance

From Figure 6.26, it is observed that at light load, the distributed desktop client version (using RMI) of the ARS software almost always outperforms the web version built using either REST or REST + ActiveMQ. However, at heavy load ActiveMQ performs so well that it outperforms RMI. For the web version of the ARS software, a combination of using REST and ActiveMQ is almost always better than using REST alone. Using the proposed

framework, model and the calibration process, a performance engineer can make predictions about the software performance before fully implementing the actual software.

7 Chapter: Conclusions

This chapter summarizes the achievements of the thesis and discusses the limitations and directions for future work.

7.1 Achievements

The thesis presents a systematic, flexible framework, called Middleware Modeling for Layered Queues (MMLQ), which supports model completions for a wide range of middleware for software performance models.

MMLQ can be used to build performance models for many middleware platforms and their services. Given a middleware-unaware application model (known as Base Application Model or BAM), MMLQ can compose middleware models into BAM in order to obtain a middleware-aware application model (known as Specialized Application Model or SAM). Following are the achievements of this thesis:

- Modeling multiple middleware platforms. MMLQ was applied to three kinds of middleware platforms: Distributed Object Based Middleware (DoBM), Container Based Middleware (CBM) and Message Based Middleware (MBM). All middleware models start with a special base middleware model known as the Empty Middleware Model (EMM), which is then customized according to the requirements of the middleware and destination application.
- Modeling middleware variability. Middleware platforms offer many similar services, yet they also provide services that are particular to that middleware platform. The commonalities and variabilities between the middleware platforms were studied and identified. The variations of the middleware configurations and services were modeled using a unified feature model.

- Proposing flexible realizations of features within the models of the middleware. The MMLQ describes two different ways to model a feature: Property-Modifying Realization (PMR) and Structure-Modifying Realization (SMR). This is a useful property of the framework that gives control to the performance modelers to keep their models either compact or unabridged. PMRs keep the model size small and are especially useful when it is not essential to model the concurrency of a feature. On the other hand, SMRs model features as a separate thread, and allow to model concurrency at the cost of larger model sizes.
- Developing a feature composition process that specializes EMM by composing feature realizations to it. A feature composition process was developed for this purpose. The feature composition process involves the following:
    - Identifying feature composition properties. These are the properties of a feature that need to be specified for performing the composition. Some examples are: is a feature PMR or SMR, what is the destination call, what is the destination host, etc.
    - Defining the Feature Composition Descriptor (FCD). This text file is an input to the Middleware Composition Engine (MCE). FCD directs the composition engine about the required feature compositions. FCD is a powerful succinct way to direct the MCE to compose many features. A formal description of FCD in extended BNF form is presented in the thesis.
    - Developing a feature composition algorithm. The MCE contains the feature composition algorithm that composes features to the base middleware models

to obtain specialized middleware models. Examples that illustrate how the algorithm works are given for the three kinds of middleware platforms considered in the thesis.

- Developing a middleware composition process. This process composes one or more specialized middleware models with the base application model in order to obtain a middleware-aware application model. The middleware composition process involves the following:
    - A way to address a large number of calls in a model. A middleware model may be needed for all parts of an application, or just for some hosts, some tasks or even some entries. The addressing process described in the thesis can handle all these cases in a succinct way.
    - Defining the Middleware Composition Descriptor (MCD). The MCD is an input to the Middleware Composition Engine (MCE), which directs the composition engine about the required middleware compositions. A formal description of the MCD in extended BNF form is presented in the thesis.
    - Developing a middleware composition algorithm. The MCE contains the middleware composition algorithm that composes specialized middleware models with BAM to obtain SAM. The algorithm follows an optimized way of composition so that a minimum number of middleware tasks are added to the model. Examples are shown for different middleware platforms to demonstrate the strength of the MCD and how the composition algorithm works.
- Describing two processes for middleware model calibration. Examples of three different kinds of middleware (DoBM, CBM and MBM) and their Security features were

calibrated. The thesis describes two calibration processes: a) response time based approach for calibrating DoBM and CBM models; and b) more traditional utilization law based approach for calibrating MBM. The calibrated models were composed to a base application model to obtain the specialized application models. The performance metrics obtained by executing the specialized application models were compared with the measurements obtained from a real system. The average absolute error for system throughput was 4.25% and for system response time was 6.40%.

In order to accomplish the above mentioned achievements, the following practical work was done.

- Building a synthetic application to emulate an Airline Reservation System (ARS). The application was deployed to the Amazon Web Services (AWS) cloud and measurements (e.g., system response time, system throughput, utilization) were taken.
- Experiments with four ARS versions using three different middleware platforms. ARS_v1 uses RMI as middleware; ARS_v2 also uses RMI but has security features for the 'purchase' operation; ARS_v3 uses REST (Jersey on Apache Tomcat) for 'browse' operation and Jersey's BasicAuth Security feature for 'purchase' operation; ARS_v4 extends ARS_v3 by using ActiveMQ for the calls between the users and the Controller service.
- Building a synthetic Middleware Benchmark Application (MBA) for calibration. The MBA uses the middleware platforms with the same configurations as ARS. The three versions of the MBA are: MBA-RMI (to calibrate Java RMI), MBA-REST (to calibrate Jersey implementation of REST) and MBA-ActiveMQ (to calibrate ActiveMQ). All these versions has two sub-versions: with and without Security features.

- Partially implementing the Model Composition Engine (MCE) tool. The MCE has two modules: a) the Parser that follows the described BNF to parse FCD and MCD files; and b) the Composer that composes middleware and application models. At present, the MCE does not have a fully implemented parser. It takes source and destination models from keyboard and performs the composition.

7.2 Limitations

The author has identified the following limitations of the thesis.

- The proposed approach has been built around only one type of performance models known as the Layered Queuing Network (LQN) models.
- The thesis focused on modeling three kinds of middleware platforms. There are other kinds of middleware platforms (e.g., multimedia middleware, peer-to-peer systems) that were not considered.
- The thesis did not show features models for all possible services and configurations of the addressed middleware platforms, but it selected the best-known services and configurations. Chapter 3 describes how the features were obtained. Using that as a guideline, a performance modeler can enhance the feature model based on his / her requirements.
- The thesis did not talk about traceability in model compositions. However, if the temporary composed models generated at the end of each in-out block are stored following any naming scheme, one can easily backtrack to that version of the model by its name.

7.3 Future work

The thesis can be extended in many ways.

- More middleware platforms can be covered using the same approach.

- This thesis describes different features found in various middleware platforms and how they were derived. However, a systematic approach for identifying features would be interesting. A more formal description of how and where to look for features would increase the usability of the proposed framework.
- A library of calibrated middleware models would be useful, particularly in the cloud environment where hundreds of virtual machines with same capacity can be quickly created from images. Once a middleware model is calibrated for one type of VM, that same model can be used for applications deployed to any VM of the same type with that middleware.
- For computing feature overheads, MMLQ uses message size and feature operation overhead as parameters for the service demand formulas. More such parameters could be identified, maybe not in general but specific to some middleware.
- The thesis describes two different calibration techniques in Chapter 6. A more generic calibration process can be defined, with definite steps and requirements.
- The thesis uses a simple way of modeling network latency using infinite tasks and ignoring message sizes. The Network model could be improved by including parameters such as message size, distance etc.

Appendices

Appendix A - Backus-Naur Form (BNF) for the descriptors

In this appendix the Backus-Naur form of the feature composition descriptor and the middleware composition descriptor are presented.

A.1 BNF for Feature Composition Descriptor (FCD)

```
<feature-composition-descriptor> ::= <FCD> <many-in-out-blocks>

<FCD> ::= "FCD" <opt-number-of-in-out-blocks> <EOL>
<opt-number-of-in-out-blocks> ::= <integer> | ""

<many-in-out-blocks> ::= <in-out-block> <many-in-out-blocks> | ""
<in-out-block> ::= <in-part> <put-part> <out-part>

<in-part> ::= "in" <base-middleware-model> "at" <caller-entry>
<receiver-entry> <EOL>
<base-middleware-model> ::= <string>
<caller-entry> ::= <string>
<receiver-entry> ::= <string> | "?"

<put-part> ::= "put" <put-body>
<put-body> ::= <feature-name> <opt-composition-property> <EOL>
<join>
<feature-name> ::= <string>
```

```
<opt-composition-property> ::= <property-switch> <property-value>
<opt-composition-property> | ""
<property-switch> ::= "-"<character>
<property-value> ::= "PMR" | "SMR" | "caller" | "receiver" |
"both" | "call" | "self" | "bound" | "shared" | "block" | "un-
block"  | "FIFO" | "PPR" | "HOL" | <real> | <integer>
<character> ::= "t" | "d" | "h" | "c" | "n" | "m" | "M" | "j" |
"o"
join ::= "&&" <put-body> | ""

<out-part> ::= "out" <out-body>
<out-body> ::= "auto"  <EOL> | <output-model-name> <EOL>
<output-model-name> ::= <string>
<string> ::= <text>
```

A.2 BNF for Middleware Composition Descriptor (MCD)

```
<middleware-composition-descriptor> ::= <MCD> <many-in-out-
blocks>
<MCD> ::= "MCD" <opt-number-of-in-out-blocks> <EOL>
<opt-number-of-in-out-blocks> ::= <integer> | ""

<many-in-out-blocks> ::= <in-out-block> <many-in-out-blocks> | ""
<in-out-block> ::= <in-part> <at-part> <opt-but-part> <out-part>
```

<in-part> ::= "in" <base-application-model> "put" <specialized-middleware-model> <EOL>

<base-middleware-model> ::= <string>

<specialized-middleware-model> ::= <string>

<at-part> ::= "at" <at-body>

<at-body> ::= <caller-entity> <receiver-entity> <EOL> <join-at>

<caller-entity> ::= <string>

<receiver-entity> ::= <string>

<join-at> ::= “&&” <at-body> | ""

<opt-but-part> ::= “” | "but" <but-body>

<but-body> ::= <caller-entity> <receiver-entity> <EOL> <join-but>

<join-but> ::= “&&” <but-body> | ""

<out-part> ::= “out” <out-body>

<out-body> ::= "auto" <EOL> | <output-model-name> <EOL>

<output-model-name> ::= <string>

<string> ::= <text>

References

[1] Alhaj, M., “Automatic Derivation of Performance Models in the Context of Model-Driven SOA”, Doctoral dissertation, Carleton University, Ottawa, Canada, 2013.

[2] Alhaj, M., Petriu, D., “Using Aspects for Platform-Independent to Platform-Dependent Model Transformations”, International Journal of Electrical and Computer Systems, 1, 1, pp.35-48, 2012.

[3] Amazon EC2 Instance Types, https://aws.amazon.com/ec2/instance-types/, last accessed: July, 2017.

[4] An Introduction to Tomcat Servlet Interactions, https://www.mulesoft.com/tcat/tomcat-servlet, last accessed: July, 2017.

[5] Apache ActiveMQ Homepage, http://activemq.apache.org, last accessed: July, 2017.

[6] Apache Struts Homepage, https://struts.apache.org, last accessed: July, 2017.

[7] Balsamo, S., Marco, A. D., Inverardi, P., Simeoni, M., "Model-Based Performance Prediction in Software Development: A Survey", IEEE Transactions on Software Engineering, vol.30, no. 5, pp. 295-310, May 2004.

[8] Batory, D., “Feature models, grammars, and propositional formulas”, Proceedings of the 9th international conference on Software Product Lines, pp. 7–20, Berlin and Heidelberg: Springer-Verlag, 2005.

[9] Becker, S., “Coupled model transformations”, Proc. 7th Int. Workshop on Software and Performance, pp. 165-176, 2008.

[10] Becker, S., Koziolek, H., and Reussner, R., “Model-based Performance Prediction with the Palladio Component Model”, in Proceedings of the 6th International Workshop on Software and Performance (WOSP2007), pp.56–67. SIGSOFT Software Engineering Notes, ACM, New York, NY, USA, Feb 2007.

[11] Bellasi, P., Faisal, A., Fornaciari, W. and Serazzi, G., “Queueing network models for performance evaluation of ZigBee based Wireless Sensor Networks”, EPEW 2010: 7th European Performance Engineering Workshop, Bertinoro, Italy, Springer, Sept 23-24, 2010.

[12] Benavides, D., Martín-Arroyo, P. T., Cortés, A. R., “Automated Reasoning on Feature Models”, in Pastor O., Cunha, J. F. e (eds.), CAiSE (pp. 491-503), Springer. ISBN: 3-540-26095-1, 2005.

[13] Beuche, D., Mark, D., “Software product line engineering with feature models", Methods & Tools, Winter, 2006.

[14] Casale, G., Ardagna, D., Artac, M., Barbier, F., Nitto, E. D., Henry, A., Iuhasz, G., Joubert, C., Merseguer, J., Munteanu, V. I., Perez, J. F., Petcu, D., Rossi, M., Sheridan, C., Spais, I., Vladuic, D., “DICE: Quality-Driven Development of Data-Intensive Cloud Applications”, in Gray, J., Chechik, M., Kulkarni, V., Paige, R. F. (eds.), MiSE@ICSE (pp. 78-83), : IEEE Computer Society. ISBN: 978-1-4673-7055-4, 2015.

[15] Chen, S., Liu, Y., Gorton, I., Liu, A., “Performance prediction of component-based applications”, Journal of Systems and Software, 74, pp. 35-43, 2005.

[16] CHILIES project for automated model completions, https://sdqweb.ipd.kit.edu/wiki/Chilies, last accessed: July, 2017.

[17] Compress your RMI message. http://www.javaworld.com/article/2077363/soa/compress-your-data.html, last accessed: July, 2017.

[18] CORBA Specification, Version 3.3, last updated: November 2012.

[19] Cortellessa, V., Marco, A. D., Inverardi, P., "Model-Based Software Performance Analysis", Springer-Verlag Berlin Heidelberg, 2011.

[20] Cortellessa, V., Pierini, P., Rossi, D., "Integrating Software Models and Platform Models for Performance Analysis", IEEE Transactions on Software Engineering, Vol. 33, No. 6, June 2007.

[21] Coulouris, G., Dollimore, J., Kindberg, T. and Blair, G., "Distributed Systems Concepts and Design", Fifth Edition, Pearson Education Inc., 2012.

[22] Czarnecki, K., Eisenecker, U. W., "Generative Programming. Methods, Tools and Applications", Addison-Wesley, 2000.

[23] DICE Project, http://www.dice-h2020.eu, last accessed: July, 2017.

[24] Digest Access Authentication, https://en.wikipedia.org/wiki/Digest_access_authentication, last accessed: July, 2017.

[25] Encryption, Security and SSL, https://crypto.stackexchange.com/questions/1150/encrypting-and-obscuring-data-between-site-user-without-ssl, last accessed: July, 2017.

[26] Enterprise JavaBeans Specification, Version 3.1, February 24, 2011.

[27] Faisal, A., “Unified Approach for Adding Middleware Completions to Software Performance Models”, Doctoral Symposium, MoDELS 2015, Ottawa, ON, 29 Sept 2015.

[28] Faisal, A., Petriu, D. C., Woodside, M., “A Systematic Approach for Composing General Middleware Completions to Performance Models”, in Computer Performance Engineering. Proc. European Performance Engineering Workshop EPEW14, Florence, LNCS vol. 8721, Springer, pp.30-44, Sept. 2014.

[29] Faisal, A., Petriu, D., Woodside, M., “Network Latency Impact on Performance of Software Deployed Across Multiple Clouds”, CASCON 2013, Markham, Ontario, Canada, November 18-20, 2013.

[30] Fielding, R. T., “REST: Architectural Styles and the Design of Network-based Software Architectures”, Doctoral dissertation, University of California, Irvine, 2000.

[31] Franks, G., “Performance Analysis of Distributed Server Systems”, PhD thesis, Carleton University, 2009.

[32] Franks, G., Al-Omari, T., Woodside, C. M., Das, O., Derisavi, S., "Enhanced Modeling and Solution of Layered Queueing Networks", IEEE Trans. on Software Eng. Vol. 35, No. 2, March/April, 2009.

[33] Franks, G., Maly, P., Woodside, M., Petriu, D. C., Hubbard, A., Mroz, M., “Layered Queuing Network Solver and Simulator Manual”, Carleton University, Ottawa, Canada, October 9, 2014.

[34] Gokhale, A. S., Kaul, D., Kogekar, A., Gray, J. G., Gokhale, S. S., "POSAML: A visual modeling language for middleware provisioning", J. Vis. Lang. Comput., 18, pp. 359-377, 2007.

[35] Gómez-Martínez, E., Merseguer, J., "Impact of SOAP Implementations in the Performance of a Web Service-Based Application", in Min, G., Martino, B. D., Yang, L. T., Guo, M., Rünger, G. (eds.), ISPA Workshops (pp. 884-896), : Springer. ISBN: 3-540-49860-5, 2006.

[36] Grassi, V., Mirandola, R., Sabetta, A., "A Model Transformation Approach for the Early Performance and Reliability Analysis of Component-Based Systems", Proceedings of the 9th International Symposium on Component Based Software Engineering (CBSE), LNCS 4063, pp. 270-284, Springer 2006.

[37] Happe, J., Becker, S., Rathfelder, C., Friedrich, H., Reussner, R. H, "Parametric Performance Completions for Model-Driven Performance Prediction", Performance Evaluation, v 16 n. 8 pp. 694-717, 2010.

[38] Happe, J., Friedrich, H., Becker, S., Reussner, R. H. "A pattern-based performance completion for Message-oriented Middleware", Proc. 7th Int. Workshop on Software and Performance, pp. 165-176, 2008.

[39] IBM MQ Homepage, http://www-03.ibm.com/software/products/en/ibm-mq, last accessed: July, 2017.

[40] Islam, F., Petriu, D. C., Woodside, C. M., "Simplifying Layered Queuing Network Models", EPEW 2015: pp. 65-79, 2015.

[41] Java Message Service (JMS) Specification, Version 1.1 (latest), April 12, 2002.

[42] Java Remote Method Invocation (RMI) Specification, https://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmiTOC.html, last updated: 2010.

[43] Java Remote Method Invocations (RMI) - Security Recommendations, http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/rmi_security_recommendations.html, last accessed: July, 2017.

[44] Java Servlet Specification, Version 3.0, last updated: December 2009.

[45] JAX-RS API Homepage, https://github.com/jax-rs, last accessed: July, 2017.

[46] Jersey Homepage, https://jersey.github.io, last accessed: July, 2017.

[47] JSON Web Token, https://jwt.io/introduction, last accessed: July, 2017.

[48] Juric, M. B., Rozman, I., Brumen, B., Colnaric, M., Hericko M., "Comparison of performance of Web services, WS-Security, RMI, and RMI-SSL", Journal of Systems and Software, 79, pp.689-700, 2006.

[49] Juric, M. B., Rozman, I., Hericko, M., "Performance comparison of CORBA and RMI", Information & Software Technology, 42, pp.915-933, 2000.

[50] Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S., "Feature-Oriented Domain Analysis (FODA) Feasibility Study (Report)", Pittsburgh: Software Engineering Institute, Carnegie Mellon University, 1990.

[51] Kapova, L., "Configurable software performance completions through higher-order model transformations", PhD Thesis, Karlsruhe Institute of Technology, 2011.

[52] Kapova, L., Zimmerova, B., Martens, A., Happe, J., and Reussner, R. H., "State dependence in performance evaluation of component-based software systems", in

Proceedings of the 1st Joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW '10), ACM, New York, NY, USA, 2010.

[53] Karatas, A. S., Oguztüzün, "Attribute-based variability in feature models", Requir. Eng., 21, pp. 185-208, 2016.

[54] Kienzle, J., Abed, W. A., Fleurey, F., Jézéquel, J., Klein, J., "Aspect-Oriented Design with Reusable Aspect Models", Transactions on Aspect-Oriented Software Development, vol. 7, pp. 279 – 327, 2010.

[55] Kothagal K., JavaBrains Advanced Tutorial: REST API Authentication Mechanisms, https://javabrains.io/courses/javaee_advjaxrs/lessons/REST-API-Authentication-Mechanisms, last accessed: July, 2017.

[56] Koziolek, H., "Performance evaluation of component-based software systems: A survey", Perform. Eval., 67, pp.634-658, 2010.

[57] Kraft, S., Pacheco-Sanchez, S., Casale, G., Dawson, S., "Estimating service resource consumption from response time measurements", VALUETOOLS '09, Pisa, Italy, 2009.

[58] Kurose, J. F. and Ross, K. W., "Computer Networking: A Top-Down Approach", Publisher: USA: Addison-Wesley Publishing Company, 2009.

[59] Layered queuing network homepage http://www.sce.carleton.ca/rads/lqns/, last accessed: July, 2017.

[60] Lazowska, E. D., Zahorjan, J., Graham, G. S., Sevcik, K. C., "Quantitative system performance - computer system analysis using queueing network models", Prentice Hall, 1984.

[61] Li, J. Z., Woodside, C. M., Chinneck, J. W., Litoiu, M., “CloudOpt: Multi-goal optimization of application deployments across a cloud”, CNSM: pp.1-9, 2011.

[62] Liu, H. H., “Software Performance and Scalability: A Quantitative Approach”, IEEE Computer Society and John Wiley & Sons Inc., Hoboken, New Jersey, 2009.

[63] Mahmoud, Q. H., “Middleware for Communications”, John Wiley & Sons Ltd., West Sussex, England, 2004.

[64] Message compression in RMI calls, http://www.theserverside.com/discussions/thread/9324.html, last accessed: July, 2017.

[65] Microsoft IIS Server Homepage, http://www.iis.net, last accessed: July, 2017.

[66] Microsoft Message Queuing Reference, https://msdn.microsoft.com/en-us/library/ms700112(v=vs.85).aspx, last accessed: July, 2017.

[67] MODAClouds Project, http://www.modaclouds.eu, last accessed: July, 2017.

[68] Model View Controller (MVC) pattern on Wikipedia, https://en.wikipedia.org/wiki/Model-view-controller, last accessed: July, 2017.

[69] Open Authorization (OAuth), https://oauth.net, last accessed: July, 2017.

[70] Oracle Glassfish Server Specification, http://oracle.com/technetwork/middleware/glassfish/overview/index.html, last accessed: July, 2017.

[71] P. Zave, “FAQ Sheet on Feature Interactions”, www.research.att.com/~pamela/faq.html, last accessed: July, 2017.

[72] Pivotal RabbitMQ Specification, https://www.rabbitmq.com/specification.html, last accessed: July, 2017.

[73] RabbitMQ Specification, https://www.rabbitmq.com/specification.html, last accessed: July, 2017.

[74] Regression analysis in MS Excel, http://www.excel-easy.com/examples/regression.html, last accessed: July, 2017.

[75] Robben, M., Interpreting CPU Utilization for Performance Analysis, https://blogs.technet.microsoft.com/winserverperformance/2009/08/06/interpreting-cpu-utilization-for-performance-analysis/, last accessed: July, 2017.

[76] Roeder T., Cornell University, Asymmetric-Key Cryptography, https://www.cs.cornell.edu/courses/cs5430/2013sp/TL04.asymmetric.html, last accessed: July, 2017.

[77] Sachs, K., Kounev, S., Buchmann, A. P., "Performance modeling and analysis of message-oriented event-driven systems", Software and System Modeling, 12, pp.705-729, 2013.

[78] SeaClouds Project, http://www.seaclouds-project.eu, last accessed: July, 2017.

[79] Security in Java using Simple Security, http://docs.oracle.com/cd/E19698-01/816-7609/6mdjrf86t/index.html, last accessed: July, 2017.

[80] Security in Java using Subject Delegation, http://docs.oracle.com/cd/E19698-01/816-7609/6mdjrf86v/index.html, last accessed: July, 2017.

[81] Security in Java, https://docs.oracle.com/javase/8/docs/technotes/guides/jmx/tutorial/security.html, last accessed: July, 2017.

[82] Shoaib, Y., Das, O., "Web Application Performance Modeling Using Layered Queueing Networks", Electr. Notes Theor. Comput. Sci., 275, pp.123-142, 2011.

[83] Smith, C. U., Williams, L. G., “Performance solutions: A practical guide to creating responsive, scalable software”, Boston, MA: Addison-Wesley, 2001.

[84] Snyder B., Bosanac D., Davies, R., “ActiveMQ in Action”, Manning Publications, Greenwich, CT, 2011.

[85] SOAP Specification, https://www.w3.org/TR/soap, last accessed: July, 2017.

[86] Spring Homepage, https://spring.io/docs/reference, last accessed: July, 2017.

[87] Strittmatter, M. and Happe, L. K., “Compositional performance abstractions of software connectors”, in Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE '12). ACM, New York, NY, USA, pp.275-278, 2012.

[88] Strittmatter, M., Happe, L. “Compositional performance abstractions of software connectors”, Proc. 3rd International Conf. on Performance Engineering, pp. 275-278, 2013.

[89] Tanenbaum, A. S., Steen, M. V., “Distributed Systems: Principles and Paradigms” (2nd Edition), Pearson, 2006.

[90] Tomcat Container Specification, http://tomcat.apache.org/tomcat-7.0-doc/, last accessed: July, 2017.

[91] U-QASAR Project, http://www.uqasar.eu, last accessed: July, 2017.

[92] Using Custom Socket Factories with Java Remote Method Invocation (RMI), https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/socketfactory/index.html, last accessed: July, 2017.

[93] Verdickt, T., “Performance Analysis of Distributed Systems Based on Architectural System Models”, PhD Dissertation, Dept. of Information Technology, Universiteit Gent, Belgium, 2007.

[94] Verdickt, T., Dhoedt, B., Gielen, F., & Demeester, P., “Modelling the performance of CORBA using Layered Queueing Networks”, Proc. EUROMICRO, pp. 117-123, 2003.

[95] Weikum, G., Vossen, G., “Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery”, Morgan Kaufmann, ISBN 1-55860-508-8, 2002.

[96] Wescott, B., “Every computer performance book”, CreateSpace Independent Publishing Platform, 2013.

[97] Woodside, M. and Franks, G., “Tutorial Introduction to Layered Modeling of Software Performance”, Oct 9, 2014. Download link: http://www.sce.carleton.ca/rads/lqns/lqn-documentation/, last accessed: July, 2017.

[98] Woodside, M., Petriu, D. B., Siddiqui, K. H., “Performance-related Completions for Software Specifications”, in Proceedings of 24th Int. Conf. on Software Engineering (ICSE 2002), Orlando. May 2002.

[99] Woodside, M., Petriu, D. C., Petriu, D. B., Shen, H., Israr, T., Merseguer, J., “Performance by Unified Model Analysis (PUMA)”, Proc. 5th Int. Workshop on Software and Performance (WOSP 2005), pp.1-12, July, 2005.

[100] Wu, X., “An Approach to Predicting Performance for Component-based Systems”, MASc thesis, July 2003.

[101] Wu, X., Woodside, M., “Performance Modeling for Software Components”, Proc. 4th Int. Workshop on Software and Performance, pp. 290-301, Redwood Shores, Calif., Jan 2004.