SYSTEM THEORY BASED MODELING AND SIMULATION OF SOA-BASED SOFTWARE SYSTEMS

by

Muthukumar V. Ramaswamy

An Applied Project Presented in Partial Fulfillment

of the Requirements for the Degree

Master of Engineering

ARIZONA STATE UNIVERSITY

May 2008

ABSTRACT

Service oriented architecture (SOA) has drawn increased attention from both academic and industrial communities, and have put forth several standards and solutions. However, we found there is no universally accepted tool or procedure showing the importance of modeling a SOA-based software system based on the important system theory's characteristics known as (i) flat or hierarchical composition, (ii) feed forward or feedback message flow, and (iii),sequential or parallel processing. In this work, we use system theory and in particular the Discrete event system specification (DEVS) formalism to create SOA domain-specific models for a publish/subscribe SOA service composition. We then modeled an example prototype scale SOA application in the DEVSJAVA simulation environment to study the above characteristics using our DEVS-SOA models. We also conducted simulation experiments and showed the correctness of the DEVS-SOA models against the prototype SOA system from which the simulation models were devised.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Problem Definition

There is increasing demand in software systems area for more functionality, distributed processing, reuse and integration. Due to this, service oriented architecture has drawn more attention from both academic and industrial communities. Leading vendors and standard organizations have put forth various standards and solutions; however there is no universally accepted tool or approach for modeling a service-oriented architecture (SOA) based on the following system characteristics [ZD63, Wym93, ZPK00]:

(i) flat or hierarchical composition
(ii) feed forward or feedback type message flow
(iii) sequential or parallel processing

Modeling and Simulation (M&S) techniques can provide scientific basis for decision making at all stages of a software system life cycle. For example, in evaluating various design alternatives at the early architectural design stage of a system, to measuring the systems operation effectiveness for various possible operational changes. To conduct meaningful simulation studies, the models developed should represent the source system with adequate accuracy. SOA-based systems deliver composite applications by meeting the increasing demands for scalability and reusability. Therefore, when modeling SOA-based systems, it is imperative that a modeler should also create scalable and reusable models.

A general and straightforward way to achieve scalability and reusability is, by abstracting the domain specific characteristics and the general system characteristics into separate layers, and cataloging the system models based on this separation. Domain specific details for a SOA-based system includes the implementation logic of participating component services, the composition type they follow, and SOA service specific details like message, operations, transport etc, in order to fulfill their functional requirements. Similarly, the general system characteristics can be derived from the systems view of a SOA-based system as listed above. This naturally leads us in creating scalable and reusable SOA domain specific system models based on domain neutral model formalisms, as found in [SSG04], for semiconductor supply chain domain.

Systems theory formalisms, like for example Discrete Event Systems (DEVS) [Zeigler, et al., 2000], has well laid out foundation around the three system properties. In this project, we first attempt to create SOA domain specific publish/subscribe composition models in DEVS formalism based on the three system characteristics, implement the DEVS-SOA models in DEVSJAVA [DEVS04] M&S environment, use the implemented models to model a prototype SOA system having the three system properties, perform model validation and simulate the model for various simple scenarios. Besides, the simulation study has also led us understand the system properties based on which the models were created, play an important role in performance metrics calculation and can provide inputs for deriving optimal architectural design.

1.2 Report Organization

The remaining portions of this report have been organized in five chapters. Chapter 2 details the relevant background information on system modeling requirements, DEVS formalism and its support for modeling, DEVS constraints, SOA, modeling SOA using DEVS formalism, and related works found in our literature review. Chapter 3 describes the DEVS-SOA models we developed in the context of publish/subscribe SOA composition, and implementation details of the DEVS-SOA models in the DEVSJAVA [DEVS04] M&S environment. Chapter 4 includes details of an example prototype system developed, modeling and simulation of the prototype system using our DEVS-SOA models. In Chapter 5 we discuss our results and possible future extensions to this work within the publish/subscribe composition scheme as well as to other SOA service composition schemes.

2. BACKGROUND

2.1 System Modeling Requirements

A system can generally be expressed by its structure and behavior using a formal notation, and system theory primarily deals with these two aspects. [Zeigler, et al., 2000] has details on classifying the specification with respect to the amount of detail one has, and can express the system formally. By viewing a system as components arranged in a hierarchical or a flattened topology, and be able to predict outputs from each component based on their state, or states and inputs, we are attempting to study and model the system as interacting I/O system components, which can be expressed at flat and hierarchical coupled component levels of specification. It is essential that, when choosing a modeling formalism to express the interacting components, the larger system they form by composition, must also be expressed as an equivalent basic model in that same formalism.

Besides, a modeler may represent the system based on two other system characteristics known as sequential or parallel processing and feed forward or feedback. At I/O system level, a sequential processing component can only accept and process one input at a time, and at a coupled component level it means there could be only one active processing component while the other components are idle. Similarly, parallel processing of a component at an I/O system level means, the component is capable of accepting simultaneous input events, but would process one event at a time, and at the coupled component level it means that, there could be more than one active component at a time. A system may have a feed forward only control flow, where inputs enter and leave the system at some point without revisiting any of the system components. In other cases, the system can also have feedback control flow to maintain some desired the control flow logic or be stable. Either way, it has significant impacts on the overall system behavior, because the inputs either leave the system quicker, or don't leave the system or be in the system for prolonged duration. Therefore, these essential system characteristics shall naturally form basic requirements for a modeler and the chosen formalism should be able to support them.

2.2 DEVS Formalism Support

[Zeigler, et al., 2000] provides DEVS formalism for both atomic and coupled models and further classifies them into classic and parallel [CZ94] models. Closure under coupling is proved for these formalisms and hence the hierarchical model construction to represent system of systems is feasible. Parallel DEVS formalism has the confluent function to ensure avoiding the collision caused by the simultaneous events, and the input processor buffer to store the inputs arriving, when the model is already processing an input.

2.3 DEVS Formalism Constraint

DEVS formalism supports zero processing time for a model component. Hence, it has constraints on the feedback flow to avoid deadlock. A DEVS model component cannot have a direct feedback to itself and can only have a feedback to itself through another model component. This constraint does not have impacts in modeling a SOA-based system because; a service can only call an operation within itself through a network component it is connected. Based on the support DEVS formalism has for flat and/or hierarchical, sequential or parallel model construction (as discussed in Section 2.2) and the constraint on feedback control flow, we understand that DEVS formalism can be used to model a SOA system based on the three system properties we are interested, provided if we were able to specify the component services at the I/O system level, and their interactions at the coupled component level.

2.4 Service Oriented Architecture

Service oriented architecture (SOA) is an architectural style that attempts to solve the issues of software reuse, distributed computing and integration. It primarily supports the server side processing that is needed. In SOA, services are building blocks. They use service message to communicate with other services, irrespective of the platform they are developed and executing on. They can be composed to form larger systems. From a simple service to a larger system they form, they all comply with the service computing principles [Tho06]. The three party model of service provider, service broker or repository and service consumer together form a SOA-based system implementation. Web service is a SOA implementation based on HTTP and HTTPS transport layers.

2.5 SOA Services

The building blocks in service oriented architecture are services, which are able to send and/or receive messages, in a pre-defined format, to fulfill a unit of work [Tho06]. The discovery of a service can be enabled through a service broker. The discovery process can be of dynamic type occurring during runtime, or a static type occurring during design time. Either way, the discovery is necessary for a SOA system composition in an ad hoc services network environment. In this work we have assumed the network of services is not ad hoc, and we know the participating service’s interface details during the design time. Hence, we excluded modeling of the service broker component.

The services can be atomic or of composite nature. SOA services have interface(s) that are well defined, discoverable and can be loosely coupled with other services. Interface definitions are also called as service contract. It consists of service endpoint definitions which provide information about the network address, protocol to access, service operation, and input and output message formats. A service can also be called as a collection of endpoints.

Communication between SOA services essentially happens through a network medium, and does not happen directly by object or function calls, as found in a conventional component based system that executes on a single computer. SOA service must have a valid network address for other services to communicate. There are circumstances where a service could also communicate with itself by calling one of its operations. In that case, the request is first directed to the network infrastructure, the address is resolved in the network and then the request reaches back to the invoked service operation.

2.6 Service Composition types

Services are required to transmit messages with other services to fulfill business logic. The message exchange with other services is done by following certain message exchange or service interaction patterns. These patterns are templates that could be simple messaging patterns like “Request-Response”, “Fire and Forget”, to a more complex business process. More detailed information on each of them can be found in [Tho06]. By following these patterns, services can be combined to form a composite service. In this work we came across *publish/subscribe*, *co-ordination*, *orchestration* and *choreography* composition schemes. We found publish/subscribe composition scheme can possibly involve all the basic messaging patterns, it is less complex than other composition schemes by not having the control flow and the co-ordination logic, and a good starting point for our study in modeling SOA-based systems. Hence publish/subscribe was chosen to study in detail.

Publish/subscribe type communication is mainly to address the asynchronous communication need between distributed software applications. It achieves its maximum benefits by decoupling the participant applications with respect to space, time and synchronization [EFGK03]. It has evolved from traditional RPC style communication to current transport independent, web service based communication. The advantages are transport and platform independence due to SOAP and XML message use, robust message filtering and quality of service (QoS) needs like reliability, transactions are now easily defined in the message specifications itself [Tho06].

There are two major specifications for web services based on publish/subscribe system. They are WS-Eventing [W306] and WS-Notification [OASIS04]. The WS-

Notification has three other subsets of specifications as WS-Base Notification, WS-Brokered Notification and WS-Topics. [HG06] has done extensive comparison between WS-Eventing and WS-Notification specifications and has found that, there are several common features among both specifications at the component level functionalities; however, they differ by the message formats they support and are incompatible with each other. This project analyzed the two specifications, used the component level functional similarities for the publisher service, subscriber service, and the publish/subscribe broker service to derive a possible publish/subscribe system architecture. Based on that system architecture; we developed DEVS-SOA models and a prototype system to validate the model.

2.7 Modeling SOA and Related Works

Detailed analysis on workflow patterns using Colored Petri-Net (CPN) to derive evaluation criteria for leading vendor solutions and SOA standards can be found in [Rus06]. Formal description of the workflows has been given in the form of CPN models. Activities in a workflow has been modeled as transitions, the before and after processing state of an activity are modeled as input and output places, and the control flow of each case in the workflow has been represented through tokens of different colors. The authors have identified different workflow patterns, their interrelationships and their supportability by leading vendor solutions and various SOA standards. [WTT06] presents SOA architecture classification (SOAC) schemes for SOA-based applications using properties like application structure, ability to change composition at runtime, fault tolerance and system engineering support in the application life cycle. [DPM06] finds that π-calculus is better suited for expressing service interactions in a workflow, than Petri-Net. The author's conclusion is based on the aspects of modeling difficulties Petri-Net pose for connections between many potential interaction partner services for an interaction, their combinations based on control flow decisions, and its static nature. Nevertheless, we were able to see earlier works using Petri-Net, and process algebra; they only address certain aspects of modeling SOA-based systems. To the best of our knowledge, there is no universally accepted formal specification for SOA systems, based on the three system properties we discussed earlier, and extend their benefits using M&S tools for analysis.

3. DEVS MODELING OF SOA SYSTEM

A service oriented architecture system consists of services satisfying service computing principles [Tho06], and composition scheme. The composition can be either one or combinations of publish/subscribe, coordination, orchestration and choreography. In this project we discuss in detail the modeling of a SOA system based on *publish/subscribe composition*. It contains service components performing publishing and/or subscribing activities using a *publish/subscribe broker coupled component*. The publish/subscribe broker coupled component contains the *notification service* for routing publications received from publisher services to *subscriber services,* and the subscription service to enable subscriber services to register their *subscriptions*. The subscription service also provides the subscribers list to notification service upon request, for publication delivery purpose. We model the notification and the subscription services as variants of the SOA service model. Service communication is enabled by the network infrastructure to which the services are connected and by the service message itself. Hence, we have also included service message model and the network model in our modeling exercise of the SOA system. In this chapter, we describe DEVS modeling of SOA services that defines their operation by using HTTP transport layer, and publish/subscribe based composition scheme. Next, we discuss an implementation of the DEVS-SOA models in the DEVSJAVA [DEVS04] modeling environment.

3.1 SOA Service Message

Services receive and/or produce service transport messages. Their format depends on the network transport layer the service endpoint(s) have binding to.

However, they encapsulate the same SOAP message format irrespective of the transport; a service operation's endpoint binding is based on. For example, a service operation might belong to an endpoint that could support either one of the common transport protocols like HTTP(S), TCP, MSMQ, SMTP, FTP etc., but the structure of the SOAP message is transport independent and their data contents might vary as defined by the service message. In this project we model a service transport message as HTTP based, and provide a placeholder to contain the SOAP message to or from a service.

3.2 Atomic Models For Publish/Subscribe Composition

Structural and behavioral aspects are very important when modeling a system or a system component. The level of details a modeler would be interested is another key factor in the modeling exercise. At the level of input / output conceptual view of a SOA service, we see both the structure and behavioral aspects of the service are defined by its service operations. The structural components are the list of service operations, and the pre-defined message structure they support processing as inputs and/or for producing outputs. The service behavior mainly corresponds to the logic embedded in an operation and its messaging type. For example, a service operation could be of either request/response or solicit request/response or notify request or one-way request type. Figure 3.1 represents conceptual I/O models of SOA service software components (S1, S2 and S3), with different service operations O1, O2, O3, and O4 and their operation types;

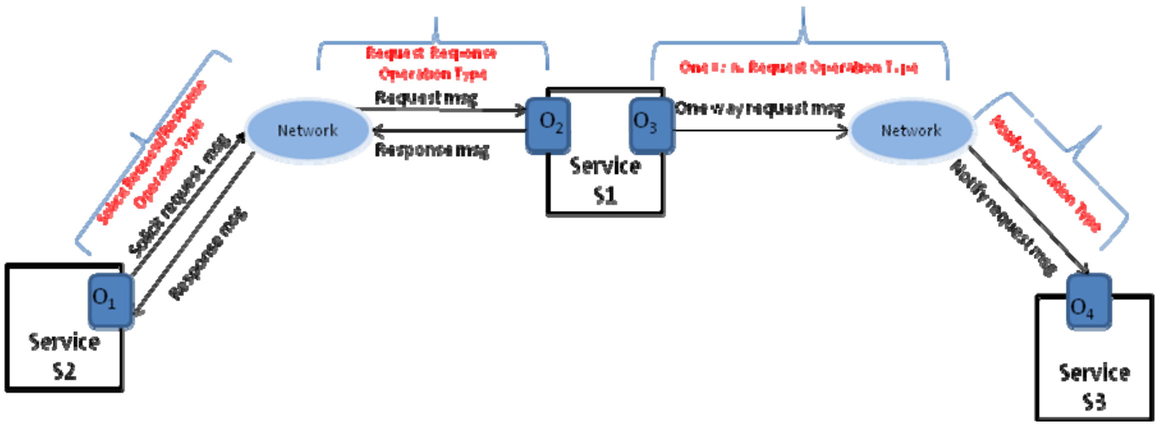


Figure 3.1 Conceptual models of SOA service software components with different operation types.

A SOA service is always connected to a network infrastructure and remains idle until it receives some service requests. The requests can arrive for a service operation at any discrete time points. The service then becomes busy processing the received message and may send output in the predefined format as a result of the service operation to the network it is connected. All messaging routing tasks are done by the network to the required destination. Any message received while it is busy, may be lost. However, if the service has ability to process simultaneous messages concurrently, or can add to a queue for later processing, the service can ensure processing all the request messages it receives. The service resumes its “Wait” or idle state after processing all the messages. Therefore, when analyzing at I/O system level, we find that a SOA service can be modeled using a Parallel DEVS atomic model [Zeigler, et al., 2000] with input processor and queue, thus having the ability to accept and process simultaneous inputs.

3.2.1 SOA Service Model

The model specification for a SOA service using parallel atomic DEVS formalism is given as below;

$DEVS_{service} = (X_{in}, Y_{out}, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, t_a)$,

where

Input port set IPorts = {“in”}

Output port set OPorts = {“out”}

Input message set $X_{request}$ = {valid service request messages, invalid message}

= service request message set = $\{X_{i1}, X_{i2}, X_{i3}, \ldots, X_{ij}\}$

$i \in$ service operation set; j = no of service message received

Output message set $Y_{response}$ = {service response messages}

=service response message set = $\{Y_{i1}, Y_{i2}, Y_{i3} \ldots . Y_{ij}\}$

Input events = $X_{in} = \{(p, v) \mid p \in \text{IPorts}, v \in X_{request}\}$

Output events = $Y_{out} = \{(p, v) \mid p \in \text{OPorts}, v \in Y_{response}\}$

Phase = {“passive”, “servicing”}

“passive” phase represents the idle state of the service, and “servicing” phase represents the busy state of the service when it is processing the inputs it received .

σ = service operation processing time set = $\{\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_i\}$

Sequential states set S = {“passive”, “servicing”} $\times \sigma \times X_{in}$

δ_{ext} = {“servicing”, σ_i, X_{in}} if phase = passive

δ_{int} = {“servicing”,σ_i, X'_{in}} if phase = servicing and queue length > 0 and the inputs are placed in the queue and removed from the queue to process, in the same order they were originally received and $X'_{in} \subset X_{in}$

= {“passive”, ∞} if phase = servicing and queue length = 0

$\delta_{con} = \delta_{int}$ is applied first and then the δ_{ext}

$\lambda = \lambda$(“passive”, ∞) =$\varnothing$, no output is produced

=λ(“servicing”,σ_i, (X_{ij}, “in”)) = (Y_{ij}, “out”)

where $X_{ij} \neq$ invalid request message for operation i

$t_a = t_a$(“servicing”,σ_i, (X_{ij}, “in”)) = σ_i , and $\sigma_i > 0$

= t_a (“passive”, ∞) = ∞

In a publish/subscribe composition, a SOA service can also subscribe for certain events with the subscription service. Therefore, it has an initial phase called “subscribing” and we have assumed the corresponding sigma value as zero. The resulting output function generates a subscribe message to the subscription service (see Section 3.2.4), and the model then goes into passive state. In our work, we assumed that the subscribe service has no subscription timeout, and only subscribes initially. The corresponding model components for a subscriber service that differ from the SOA service model are given as below;

Output message set $Y_{response}$ = {service response messages, subscribe message}

Service response message set = $\{Y_{i1}, Y_{i2}, Y_{i3} \ldots Y_{ij}\}$

Subscribe message = Y_s

Phase = {“subscribing”, “passive”, “servicing”} "subscribing" phase is when a subscriber service prepares the subscription message and sends it to the subscription service.

Sequential states set S = {"subscribing", "passive", "servicing"} $\times \sigma \times X_{in}$

δ_{ext} = {"subscribing", 0} at simulation time zero

= {"servicing", σ_i, X_{in}} if phase = passive

δ_{int} = {"passive", ∞} if phase=subscribing

= {"passive", ∞} if phase = servicing and queue length = 0

= {"servicing", σ_i, X'_{in}} if phase = servicing and queue length > 0 and the inputs are placed in the queue and removed from the queue to process, in the same order they were originally received and $X'_{in} \subset X_{in}$

$\delta_{con} = \delta_{int}$ is applied first and then the δ_{ext}

$\lambda = \lambda$("subscribing", 0) = (Y_s, "out") subscribe message

= λ("passive", ∞) =$\emptyset$, no output is produced

= λ("servicing",σ_i, (X_{ij}, "in")) = (Y_{ij}, "out"),

Where $X_{ij} \neq$ invalid request message for operation i

$t_a = t_a$("servicing",σ_i, (X_{ij}, "in")) = σ_I , and $\sigma_i > 0$

= t_a ("passive", ∞) = ∞

= t_a ("subscribing", 0) = 0

3.2.2 Notification Service Model

SOA services may publish messages, which are basically a specific message type already registered with the notification service as publications. In our work we assume that all publisher and subscriber services know what publications are available, and hence we excluded modeling the publication management functionality within the notification service. Notification service communicates with subscription service (see Section 3.2.4) to get the list of subscriber services, only after it receives a notification from any of the publisher service. It waits for the subscription service to respond with the subscriptions list. Once it is received, the notification service then notifies all those subscribers in the subscriptions list with the notification service message. Notification service also adds all the notifications to its input buffer in the order it was received, while it is waiting for the subscription service to respond with the subscriptions list. All these service communication happen through the network infrastructure. The notification service is modeled as a parallel DEVS atomic model, with input processor and queue; the specification is given as following;

DEVS $_{\text{Notify service}}$ = $\left(X_{in}, Y_{out}, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, t_a \right)$,

Input port set IPorts = {“in”}

Output port set OPorts = {“out”}

Input message set $X_{request}$ = {notification messages, subscriptions list messages}

Output message set $Y_{response}$ = {notification messages, getSubscriptionList message}

Input events = $X_{in} = \{(p, v) \mid p \in \text{IPorts}, v \in X_{request}\}$

Output events = $Y_{out} = \{(p, v) \mid p \in \text{OPorts}, v \in Y_{response}\}$

Phase = {“passive”, “gettingSubscribers”,”waitingForSubscriptions”, “servicing”}

“passive” phase represents the idle state of the service, “gettingSubscribers” phase is when it sends the request to subscription service for getting the subscriptions list for the notification it received, “waitingForSubscriptions” phase is when it waits for the subscriptions list from the subscription service, and “servicing” represents the busy state of the service when it is processing a received notification and needs to send out notification for subscriber services present in the subscriptions list.

σ = notify operation processing time

Sequential states set = S

S = {“passive”, “gettingSubscribers”,”waitingForSubscriptions”, “servicing”} $\times \sigma \times X_{in}$

δ_{ext} = {“ gettingSubscribers”, 0, X_{in}}

If phase = passive and v = notification message

= {“servicing”,σ, X_{in}}

if phase = waitingForSubscriptions and v = subscriptions list message

δ_{int} = {“waitingForSubscriptions”, ∞} if phase = gettingSubscribers

= {“passive”, ∞} if phase is servicing and queue length = 0

= {“gettingSubscribers”, 0, X'_{in}}

If phase is servicing and queue length > 0 and v = notification messages added and removed from the queue in the order they were received and $X'_{in} \subset X_{in}$

$\delta_{con} = \delta_{int}$ is applied first and then the δ_{ext}

$\lambda = \lambda(\text{“passive”}, \infty) = \varnothing$, no output is produced

$= \lambda(\text{“waitingForSubscriptions”}, \infty) = \varnothing$

$= \lambda(\text{“gettingSubscribers”}, 0, X_i) = Y_{out} = (Y_s, \text{“out”})$

$Y_s \equiv$ getSubscriptionList message

$= \lambda(\text{“servicing”}, \sigma, (X_s, \text{“in”}), X_n) = Y_{out} = \{ (Y_1, \text{“out”}), (Y_2, \text{“out”}), \ldots (Y_n, \text{“out”})\}$

$X_s \equiv$ subscriptions list message with n subscriptions.

X_n is the notification message it originally received, added to the queue in the order it was received and then removed from the queue for processing.

$Y_n \equiv$ outgoing notification message to the n^{th} subscriber service

$t_a = t_a(\text{“passive”}, \infty) = \infty$

$= t_a(\text{“gettingSubscribers”}, 0, X_{in}) = 0$ where v = notification message

$= t_a(\text{“waitingForSubscriptions”}, \infty) = \infty$

$= t_a(\text{“servicing”}, \sigma, (X_s, \text{”in”})) = \sigma$ and $\sigma > 0$

3.2.3 Network Model

Network infrastructure has an important role in any SOA system. Its primary role is to route the service messages to the correct destination. As described in Section 3.1, the service messages are transport based, hence the network infrastructure shall be able to route service messages of all transports. In this work, we have assumed the transport is HTTP based. A service communicates to any other service or to itself, only through a network infrastructure. The routing is done by inspecting the service message contents for source and destination. The following Figure 3.2 shows how a service communication is made to another service in an intra-network type environment. Similarly a service can also communicate with other services located in different networks. In this case, we use same the network model as another component called global network, which we assume to have connection with all other network components, and has the extra HTTP message routing ability to any network component it has connection with. The following Figure 3.3 shows the inter-network type of service communication.

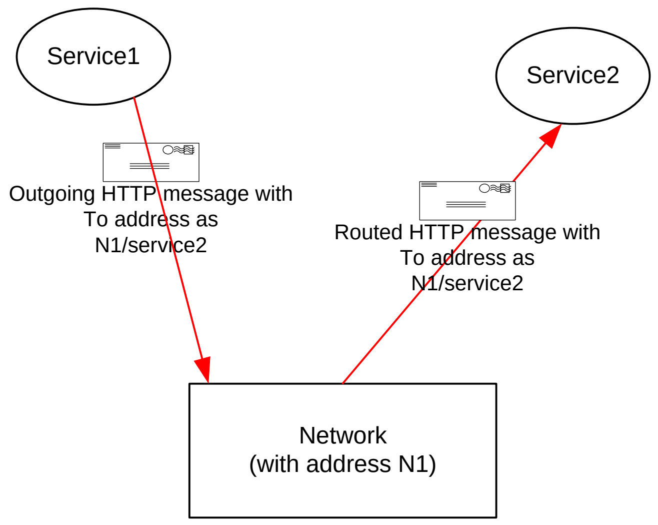

Service1
Service2
Outgoing HTTP message with To address as N1/service2
Routed HTTP message with To address as N1/service2
Network (with address N1)

Figure 3.2 Intra-network type service communications

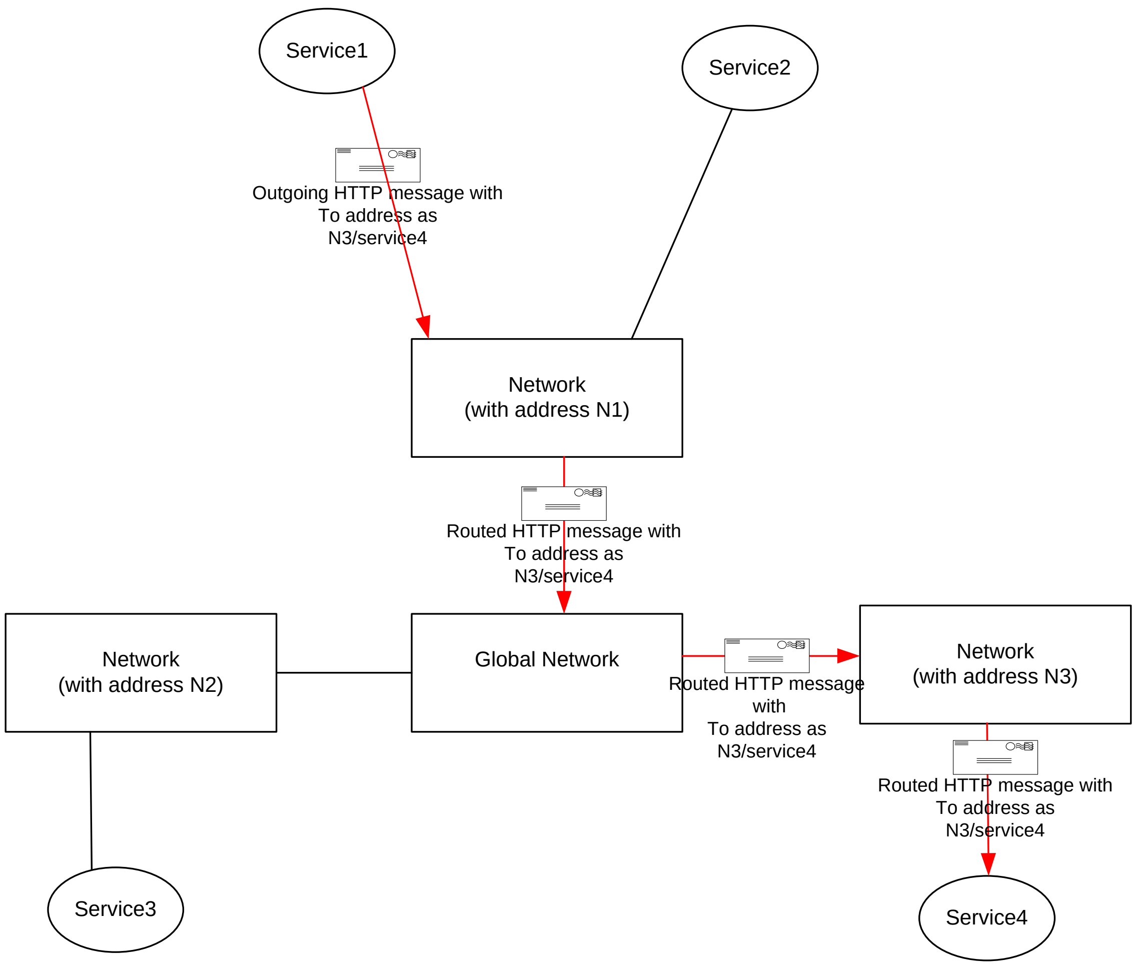

Service1
Service2
Outgoing HTTP message with To address as N3/service4
Network (with address N1)
Routed HTTP message with To address as N3/service4
Network (with address N2)
Global Network
Routed HTTP message with To address as N3/service4
Network (with address N3)
Routed HTTP message with To address as N3/service4
Service3
Service4

Figure 3.3 Inter-network type service communications

We have modeled the network component as parallel DEVS processor model with input buffer.

$$DEVS_{Network} = (X_{in}, Y_{out}, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, t_a),$$

where

Input port set IPorts = {“in”}

Output port set OPorts = {“out”}

Input message set X = {service messages}

Output message set Y = {service messages}

Input events = $X_{in} = \{(p, v) \mid p \in IPorts, v \in X\}$

Output events = $Y_{out} = \{(p, v) \mid p \in OPorts, v \in Y\}$

Phase = {“passive”, “servicing”}

$\sigma = 0$

Sequential states set S = {“passive”, “servicing”} × σ × X_{in}

δ_{ext} = {“servicing”, 0, X_{in}} if phase = passive and multiple messages arriving at the same time

δ_{int} = {“passive”, ∞} if phase = servicing and queue length = 0

= {“servicing”, σ, X'_{in}} if phase = servicing, and queue length > 0 and v = service messages added and removed from the queue in the order they were received and $X'_{in} \subset X_{in}$

δ_{con} = δ_{int} is applied first and then the δ_{ext}

$\lambda = \lambda$(“passive”, ∞) =$\emptyset$, no output is produced

= λ(“servicing”, 0, X_{in}) = Y_{out}

$t_a = t_a$ (“servicing”, 0, X_{in}) = 0

= t_a (“passive”, ∞) = ∞

3.2.4 Subscription Service Model

Subscription service is part of the publish/subscribe broker coupled components. It keeps track of all subscription details. A subscription consists of the subscriber service address, and the event it has subscribed. Services subscribe to a particular event using the subscription service. Notification service (as discussed earlier in Section 3.2.2) queries the subscription service for subscribers, when it has an event to notify. Based on the subscriber list provided by the subscription service, the notification service then notifies the event details to services in the subscriber list. The model specification for a subscription service is same as that of a SOA service model, except, that the inputs and outputs are specific to the subscription. The inputs are of “getSubscriptionList” message from notification service, and the “subscribe” message from subscribing services. The output message is of type “subscribers list” message. We have modeled the subscription service component as parallel DEVS processor model with input buffer.

DEVS $_{\text{Subscription service}}$ = (X_{in}, Y_{out}, S, δ_{ext}, δ_{int}, δ_{con}, λ, t_a),

where

Input port set IPorts = {“in”}

Output port set OPorts = {“out”}

Input message set $X_{request}$ = {getSubscriptionList messages, subscribe messages}

getSubscriptionList messages = {X_{gSL1}, X_{gSL2}, X_{gSL3}….X_{gSLn}}

subscribe messages = {X_{s1}, X_{s2}, X_{s3}….X_{sn}}

n $\in$ number of events available to subscribe

Output message set $Y_{response}$ = {subscriptions list messages}

subscriptions list messages = {Y_1, Y_2, Y_3….Y_n}

Input events = X_{in} = {(p, v) | p$\in$ IPorts, v$\in$ $X_{request}$}

Output events = Y_{out} = {(p, v) | p$\in$ OPorts, v$\in$ $Y_{response}$}

Phase = {“passive”, “servicing”}

$\sigma = 0$

Sequential states set S = {“passive”, “servicing”} × σ × X_{in}

δ_{ext} = {“servicing”, 0, X_{in}} if phase = passive

δ_{int} = {“passive”, ∞} if phase = servicing and queue length = 0

= {“servicing”, 0, X'_{in}} if phase = servicing, and queue length > 0 and v = service messages added and removed from the queue in the order they were received and $X'_{in} \subset X_{in}$

δ_{con} = δ_{int} is applied first and then the δ_{ext}

λ = λ(“passive”, ∞) =$\varnothing$, no output is produced

= λ(“servicing”, 0, (X_{gSLn}, “in”)) = (Y_n, “out”)

= λ(“servicing”, 0, (X_{sn}, “in”)) =$\varnothing$, no output is produced for the subscribe request.

t_a = t_a (“servicing”, 0, X_{in}) = 0

= t_a (“passive”, ∞) = ∞

3.3 Publish/Subscribe Broker Service Coupled Model

A publish/subscribe broker is a composite service by itself, consisting of notification service, subscription service and both connected to a network infrastructure. The below Figure 3.4 shows the UML component model of a publish/subscribe broker service software, having its inner service software components and its relationship with publisher services and subscriber services software.

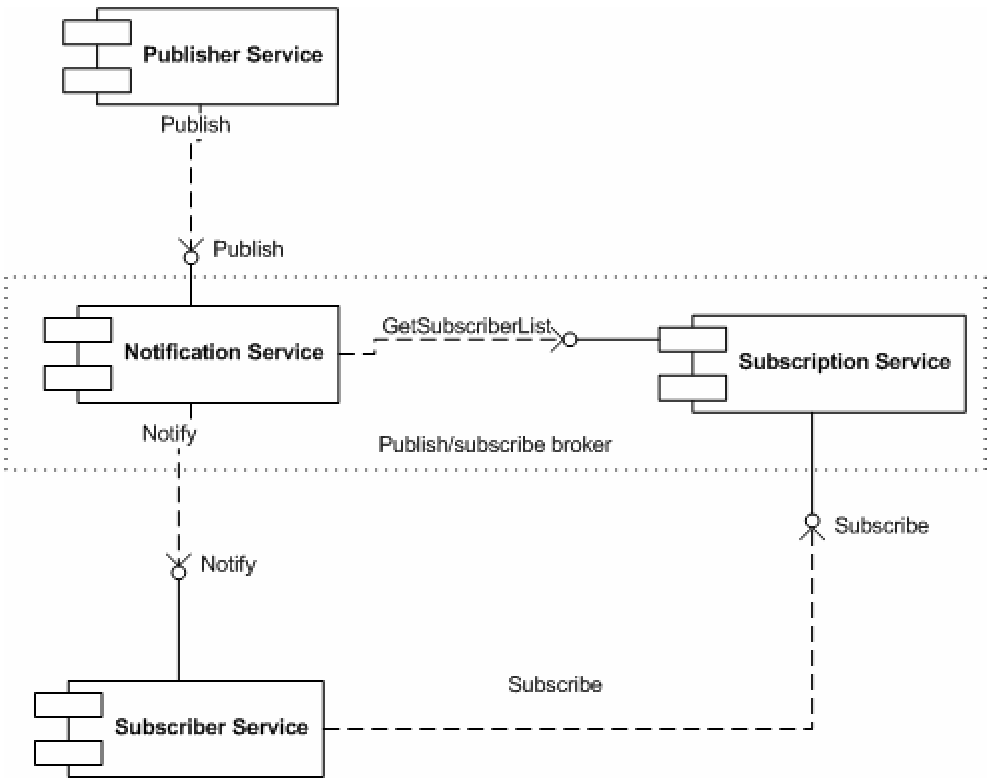


Figure 3.4 UML Component model of a publish/subscribe broker service and its relationship with publisher and subscriber services.

In the Figure 3.4, the network infrastructure between any two services communication is not shown as they represent the hardware interface that actually enables the communication at runtime. Figure 3.5 provides the network graph of a publish/subscribe broker composite service and outlines the message types involved, and their flow between notification service, network and subscription service components.

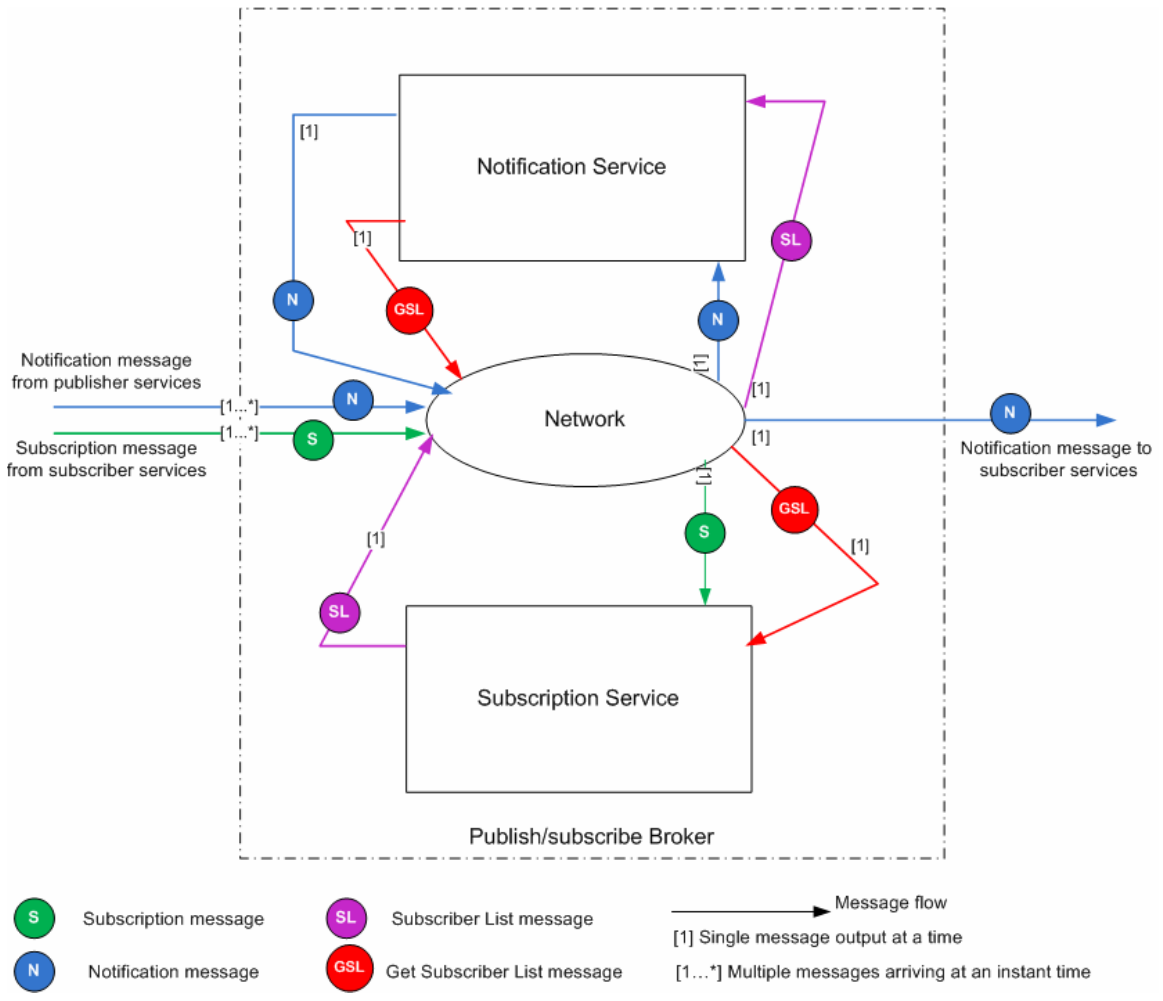

Notification Service
[1]
[1]
SL
N
GSL
N
[1]
[1]
Notification message from publisher services
[1...*]
N
Network
N
Notification message to subscriber services
[1...*]
S
Subscription message from subscriber services
[1]
[1]
GSL
S
[1]
[1]
SL
Subscription Service
Publish/subscribe Broker
S
Subscription message
SL
Subscriber List message
Message flow
[1] Single message output at a time
N
Notification message
GSL
Get Subscriber List message
[1…*] Multiple messages arriving at an instant time

Figure 3.5 Publish/subscribe broker, its components, message types involved and their flow.

The publish/subscribe broker service is modeled as parallel DEVS coupled model and its specification is given as following;

$DEVS_{pub/sub} = (X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC)$,

InPorts = {“in”}

OutPorts = {“out”}

X_{in} = {“notification message”, “subscribe message”}

Y_{out} = {“notification message”}

Input events = $X = \{(p, v) \mid p \in \text{IPorts}, v \in X_{in}\}$

Output events = $Y = \{(p, v) \mid p \in \text{OPorts}, v \in Y_{out}\}$

Component set D = {“Notification Service”, “Network”, “Subscription Service”}

$M_{\text{Notify service}}$ = Notification Service

M_{Network} = Network

$M_{\text{Subscription service}}$ = Subscription Service

EIC = {((“Broker”, “in”), (“Network”, “in”))}

EOC = {((“Network”, “in”), (“Broker”, “in”))}

IC = {((“Network”, “out”), (“Notification Service”, “in”)) ,

((“Notification Service”, “out”), (“Network”, “in)),

((“Network”, “out”), (“Subscription Service”, “in”)),

((“Subscription Service”, “out”), (“Network”, “in”)) }

3.4 Implementing DEVS-SOA Models in DEVSJAVA

DEVSJAVA [DEVS04] is an object oriented, scalable, and flexible DEVS modeling and simulation environment. It has graphic utilities to visualize the model execution. DEVS-SOA models were implemented in this environment and packaged as GenWS API. In this work, we have only considered a HTTP based service message transport and the service models were developed in the context of publish/subscribe broker based composition. The models have the “ServiceClock” property to use the atomic simulator clock time at each atomic component level for any output analysis purpose. The following class diagrams in Figures 3.6, 3.7 and 3.8 summarize the DEVS-SOA model components and their associations.

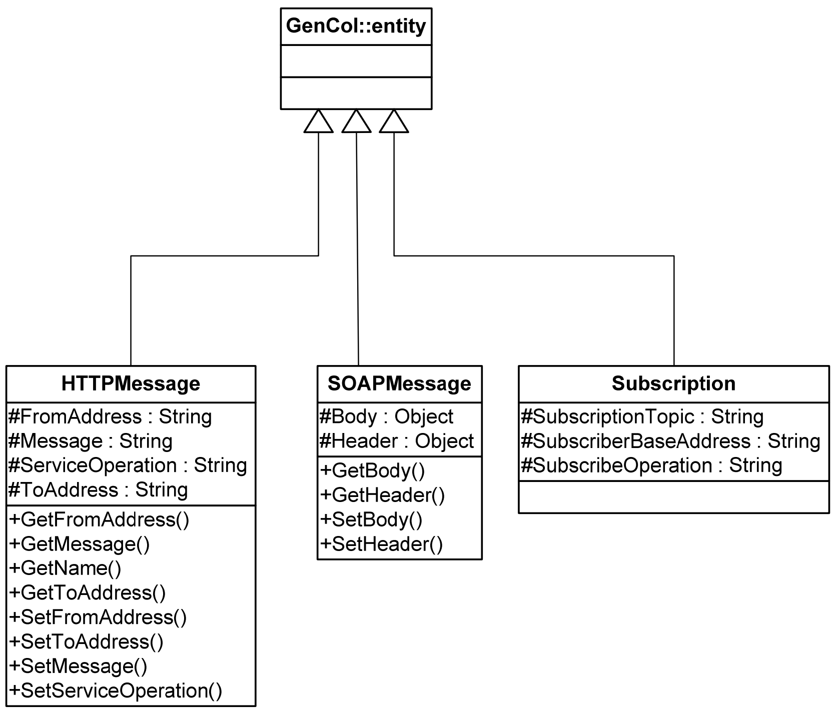


Figure 3.6 Class diagram of the DEVS-SOA Service messages.

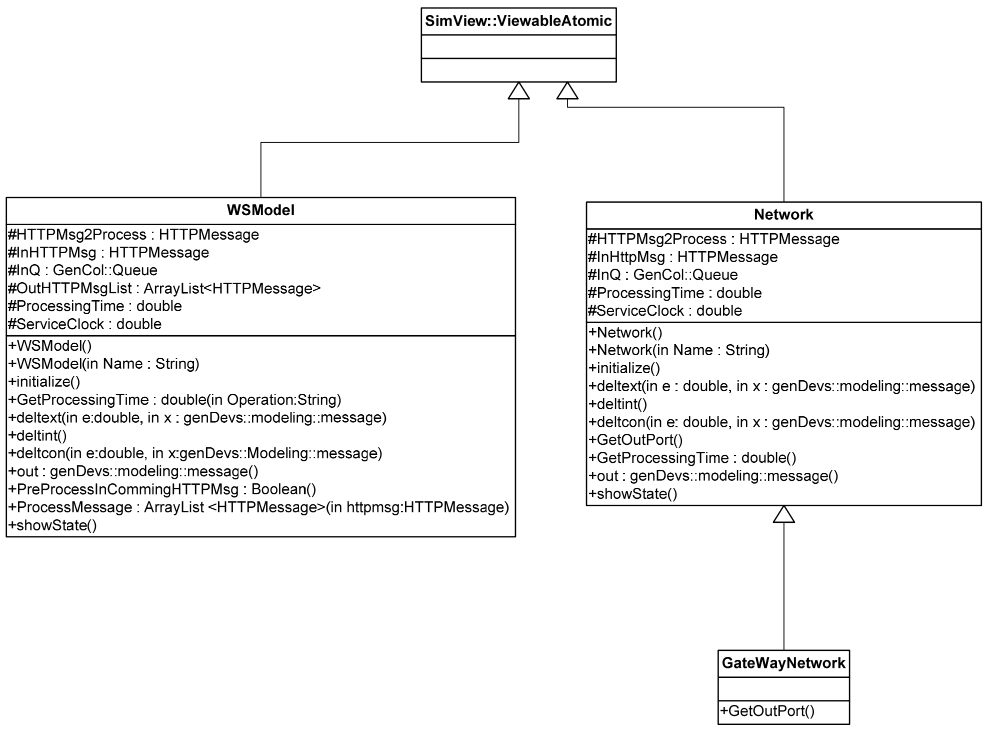

SimView::ViewableAtomic
WSModel
#HTTPMsg2Process : HTTPMessage
#InHTTPMsg : HTTPMessage
#InQ : GenCol::Queue
#OutHTTPMsgList : ArrayList<HTTPMessage>
#ProcessingTime : double
#ServiceClock : double
+WSModel()
+WSModel(in Name : String)
+initialize()
+GetProcessingTime : double(in Operation:String)
+deltext(in e:double, in x : genDevs::modeling::message)
+deltint()
+deltcon(in e:double, in x:genDevs::Modeling::message)
+out : genDevs::modeling::message()
+PreProcessInCommingHTTPMsg : Boolean()
+ProcessMessage : ArrayList <HTTPMessage>(in httpmsg:HTTPMessage)
+showState()
Network
#HTTPMsg2Process : HTTPMessage
#InHttpMsg : HTTPMessage
#InQ : GenCol::Queue
#ProcessingTime : double
#ServiceClock : double
+Network()
+Network(in Name : String)
+initialize()
+deltext(in e : double, in x : genDevs::modeling::message)
+deltint()
+deltcon(in e: double, in x : genDevs::modeling::message)
+GetOutPort()
+GetProcessingTime : double()
+out : genDevs::modeling::message()
+showState()
GateWayNetwork
+GetOutPort()

Figure 3.7 Class diagram of DEVS-SOA service and Network models

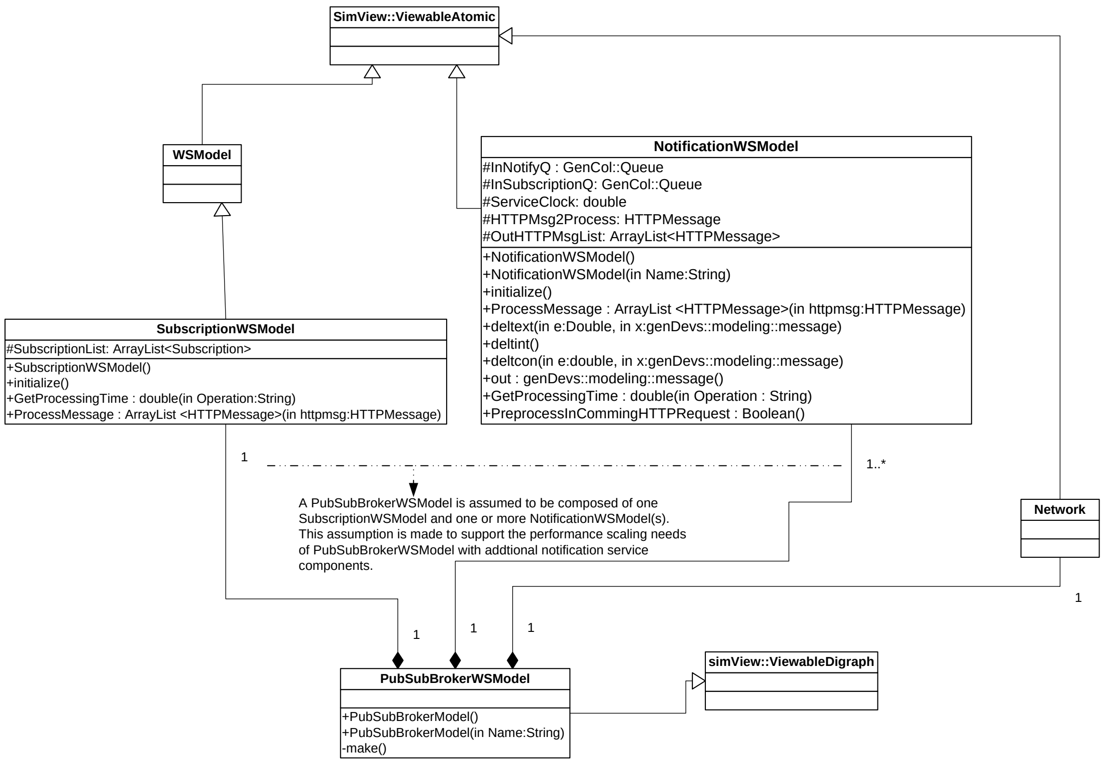

SimView::ViewableAtomic
WSModel
NotificationWSModel
#InNotifyQ : GenCol::Queue
#InSubscriptionQ: GenCol::Queue
#ServiceClock: double
#HTTPMsg2Process: HTTPMessage
#OutHTTPMsgList: ArrayList<HTTPMessage>
+NotificationWSModel()
+NotificationWSModel(in Name:String)
+initialize()
+ProcessMessage : ArrayList <HTTPMessage>(in httpmsg:HTTPMessage)
+deltext(in e:Double, in x:genDevs::modeling::message)
+deltint()
+deltcon(in e:double, in x:genDevs::modeling::message)
+out : genDevs::modeling::message()
+GetProcessingTime : double(in Operation : String)
+PreprocessInCommingHTTPRequest : Boolean()
SubscriptionWSModel
#SubscriptionList: ArrayList<Subscription>
+SubscriptionWSModel()
+initialize()
+GetProcessingTime : double(in Operation:String)
+ProcessMessage : ArrayList <HTTPMessage>(in httpmsg:HTTPMessage)
1
1..*
A PubSubBrokerWSModel is assumed to be composed of one SubscriptionWSModel and one or more NotificationWSModel(s). This assumption is made to support the performance scaling needs of PubSubBrokerWSModel with addtional notification service components.
Network
1
1
1
1
PubSubBrokerWSModel
+PubSubBrokerModel()
+PubSubBrokerModel(in Name:String)
-make()
simView::ViewableDigraph

Figure 3.8 Class diagram of DEVS-SOA publish/subscribe models with their associations

4. MODELING AND SIMULATION OF A PUBLISH/SUBSCRIBE SOA SYSTEM IN DEVSJAVA

4.1 Order Processing System

For our modeling purpose, we considered a typical order processing system with minimum required functionality as an example system. To process incoming orders, a publish/subscribe composition was developed at a prototype scale using Microsoft .Net 3.5 WCF framework. The requirements for the prototype system development were;

a) The system shall be composed of multiple SOA services.
b) The services in the system shall interact using a publish/subscribe broker.
c) The system composition shall represent the three system characteristics listed below;
    i. Hierarchical and flattened service topology.
    ii. Sequential or parallel processing.
    iii. Feed forward or feedback message flow.

The system developed met all the above stated requirements with sequential message processing capability. Figure 4.1 shows a conceptual view of the system components and the communication between them.

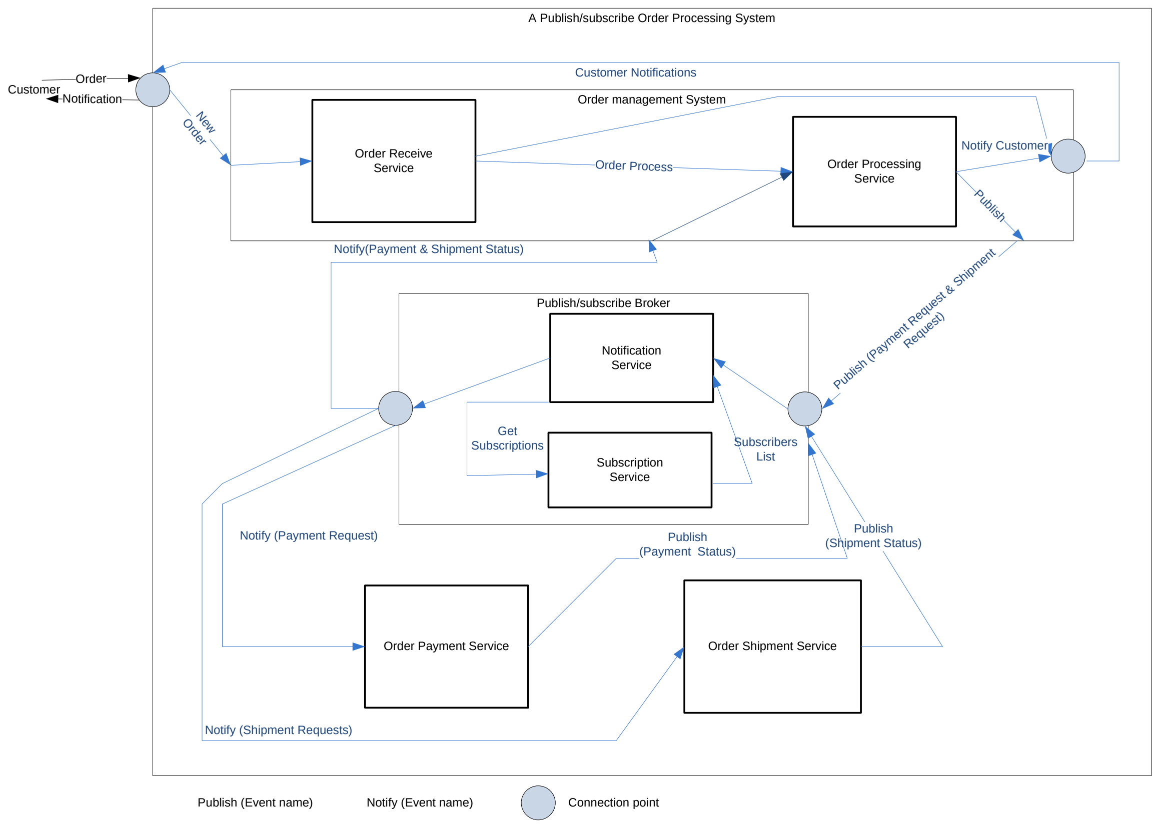


Figure 4.1 Conceptual view of a publish/subscribe order processing system

Each service in the system has the ability to record the processing time for every message it receives. The sequence of service interactions to complete processing one order transaction is provided as in the following Figure 4.2;

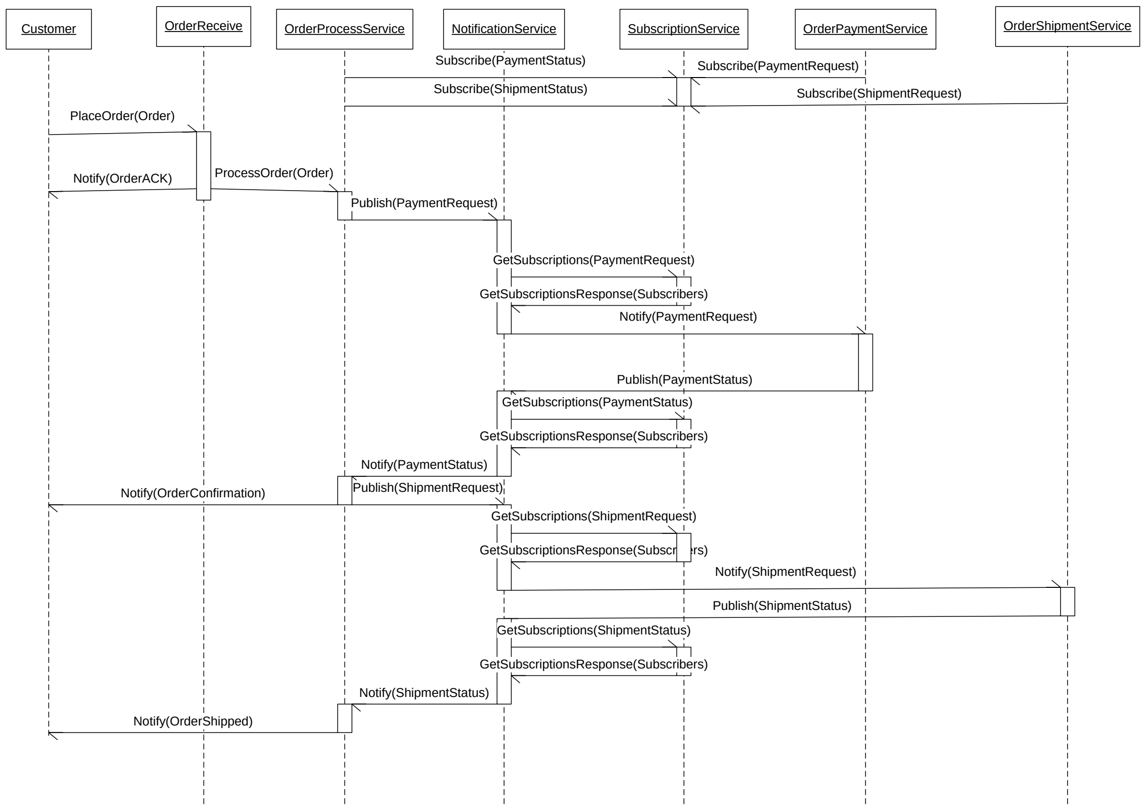


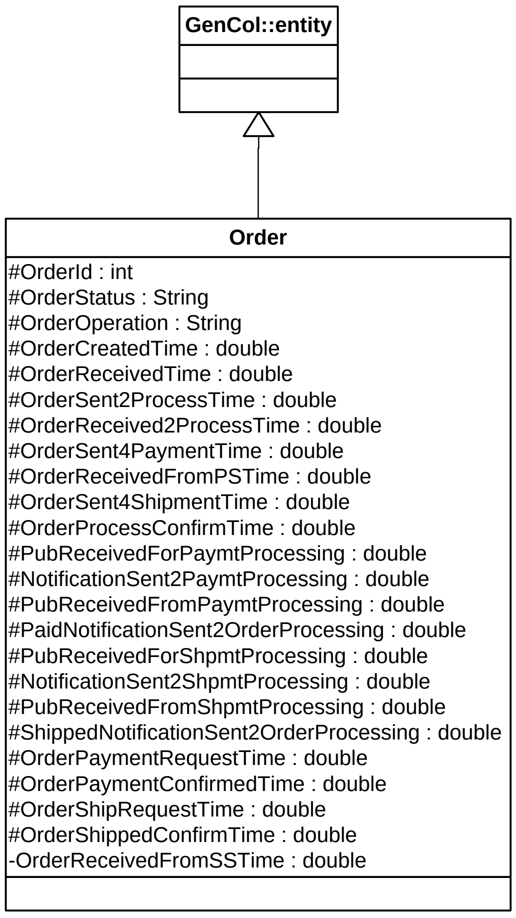
Figure 4.2 Order processing sequence

4.2 Modeling Purpose

The modeling exercise purposes were to represent the system components at the I/O system level and the whole system at hierarchical coupled component level, simulate the system for different transaction arrival rates, observe the effects of the three system properties in the performance of individual services and at the prototype system level and finally derive inputs for service configuration, and make topology changes to scale up system performance.

4.3 Model

The system model was developed in the DEVSJAVA environment using the DEVS-SOA models described in Chapter 3. The component services were modeled at the I/O system level with ability to select the processing time as sigma values, from the underlying Input model (see Section 4.4) for each input message processing. The input messages were actually HTTP message, encapsulating the SOAP message, which in turn had the Order object serialized in the SOAP message's Body (See SOAP message class in Figure 3.6) property. Figure 4.3 shows the class diagram of the Order class.



GenCol::entity
Order
#OrderId : int
#OrderStatus : String
#OrderOperation : String
#OrderCreatedTime : double
#OrderReceivedTime : double
#OrderSent2ProcessTime : double
#OrderReceived2ProcessTime : double
#OrderSent4PaymentTime : double
#OrderReceivedFromPSTime : double
#OrderSent4ShipmentTime : double
#OrderProcessConfirmTime : double
#PubReceivedForPaymtProcessing : double
#NotificationSent2PaymtProcessing : double
#PubReceivedFromPaymtProcessing : double
#PaidNotificationSent2OrderProcessing : double
#PubReceivedForShpmtProcessing : double
#NotificationSent2ShpmtProcessing : double
#PubReceivedFromShpmtProcessing : double
#ShippedNotificationSent2OrderProcessing : double
#OrderPaymentRequestTime : double
#OrderPaymentConfirmedTime : double
#OrderShipRequestTime : double
#OrderShippedConfirmTime : double
-OrderReceivedFromSSTime : double

Figure 4.3 Order class diagram

Each service process the Order objects and update the operation type, status, simulation time at which it received the message, and the simulation time at which it completes processing. The model had similar components services, and topology setup as the prototype system we originally developed (see Section 4.1). The below Figures 4.4 shows the graphical view of the publish/subscribe coupled model in DEVSJAVA environment.

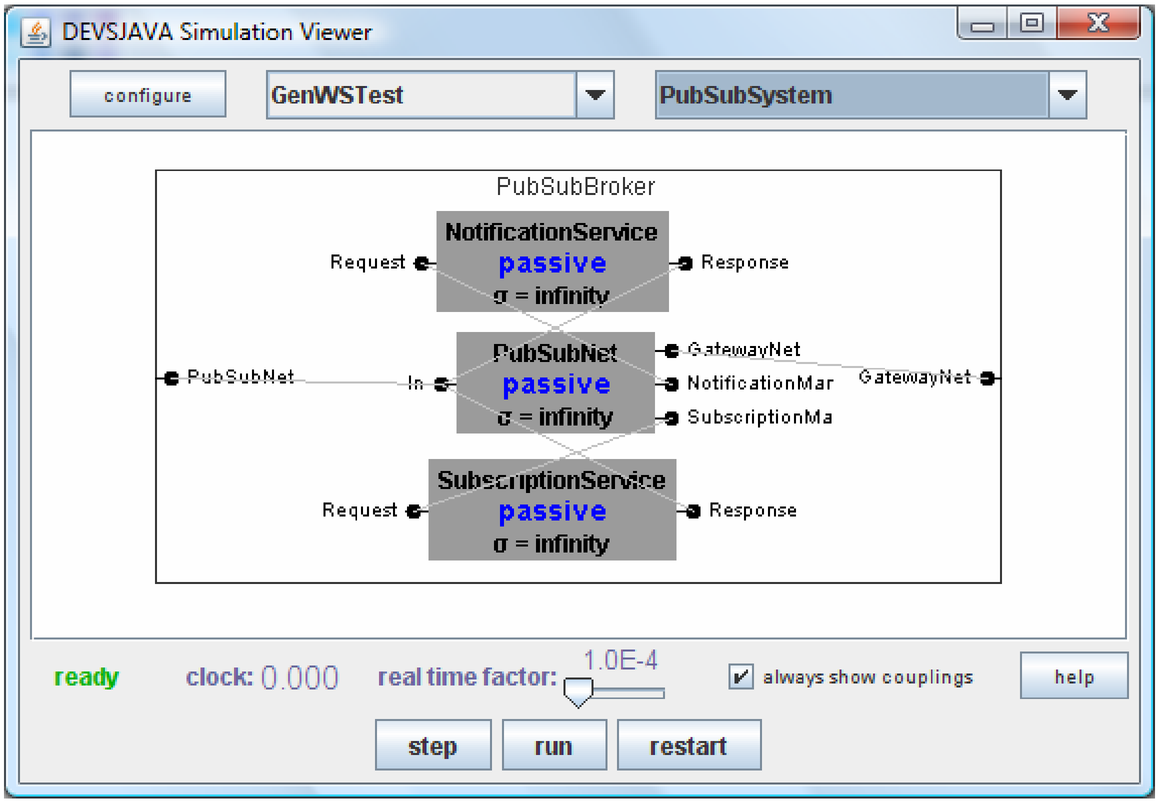


Figure 4.4 Graphical view of publish/subscribe broker coupled model in DEVSJAVA simulation environment

To meet our modeling objectives (see Section 4.2) we developed an experimental frame [ZPK00], which can produce “Order messages” as model inputs at varying inter-arrival time, and receive model outputs to analyze the processing time of each service for those input “Order messages”. The experimental frame is called as “Customers” and is connected to the “APS system model” as shown in the Figure 4.5.

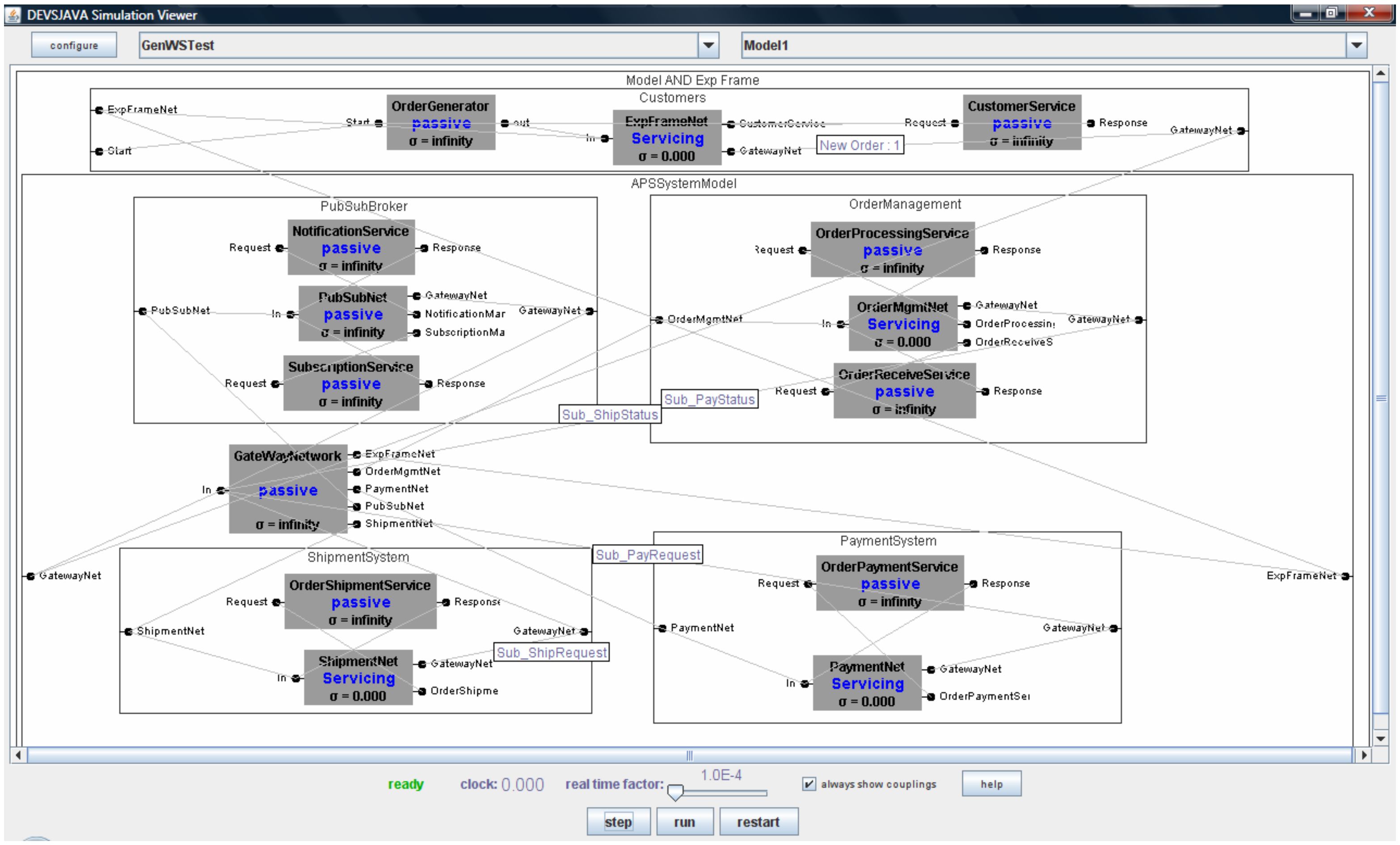


Figure 4.5 **Graphical view of the prototype system model and the experimental frame in DEVSJAVA environment**

4.4 Model Inputs

Model Inputs are the most important driving force for a discrete event model simulation. They can be input event data, input event inter-arrival time, processing time for model phase(s), and output event data to be generated. Selection of the model inputs actually depends on a modeler’s need in representing appropriate detail of the system, the simulation requirements, and the simulation execution environment’s ability to support. For example, a stochastic simulation may be required to model inputs from an underlying distribution derived from the system of interest to introduce randomness, or provide constant inputs to simply observe the system dynamics, or reuse the observed data from the system itself. Selection of appropriate input modeling exercise is very important to obtain reliable and accurate model outputs.

4.4.1 Input Variables

Based on the Order Processing System of our interest outlined in Section 4.1, and the modeling objectives as outlined in Section 4.2, we considered to model the service processing time for each operation type and order inter-arrival time. The following Table 4.1 has Service processing time input variables at each service level.

Service	**Operation**	**Input variables based on the type of message being processed**
Order Receive Service	OrderReceive	OrderReceive
Order Process Service	Notify	PayRequest ShipRequest SendOutFinalConfirmation
OrderPayment Service	Notify	PayRequest
OrderShipment Service	Notify	ShipRequest
Notification Service	Publish	NotifyPayRequest NotifyPaid NotifyShipRequest NotifyShipped

Table 4.1 Input variables summary for participating services in composition

The order inter arrival time for simulation in steady state was considered as 0.5 transactions per second, and variable input inter-arrival cases with 1, 2, 3, 4 and 5 transactions per second.

From the system, the processing time variables as in Table 4.1 were sampled and found to have dependency among each individual variable values. That is, as order id increases the processing time variable value also increases. This is due to the fact that we executed the prototype system in a single computer, and all the services were sharing the same database for processing time updates. Hence, the stochastic input modeling was not performed, and we decided to use the raw data sampled [BN02] out of the 8000 order processed, as model inputs for the ten service processing time variables (see Table 4.1).

4.4.2 Inputs Constants and Other Assumptions

The prototype system of our interest was run on a single computer, and the subscription service was defined as simple database lookup operations, which we were not interested in initial system observations. Hence, we assumed the processing time for all network components and the subscription service to provide the subscription list, as zero processing time (see Section 3.2.4).

The inputs are assumed to be stationary, hence the arrival rates as discussed in Section 4.4.1 are to be constant in our simulation experiments. We noted that, by assuming a constant arrival rate we induce a discrete time simulation rather than a discrete event simulation. The system model is capable of processing non stationary inputs, however considering the prototype system analysis nature and the resource constraints in this study, for this initial simulation runs, we decided to have this constant model input rate.

4.5 Model Verification

Initial base case scenarios were executed to verify model correctness from the model outputs. The base case scenarios were assumed with a known constant value of 1, and known probable values in between 2 and 4 simulation time steps for each service processing time. The following Figures 4.6 and 4.7 show the timing diagram plot for these two base cases and how the model verification was done.

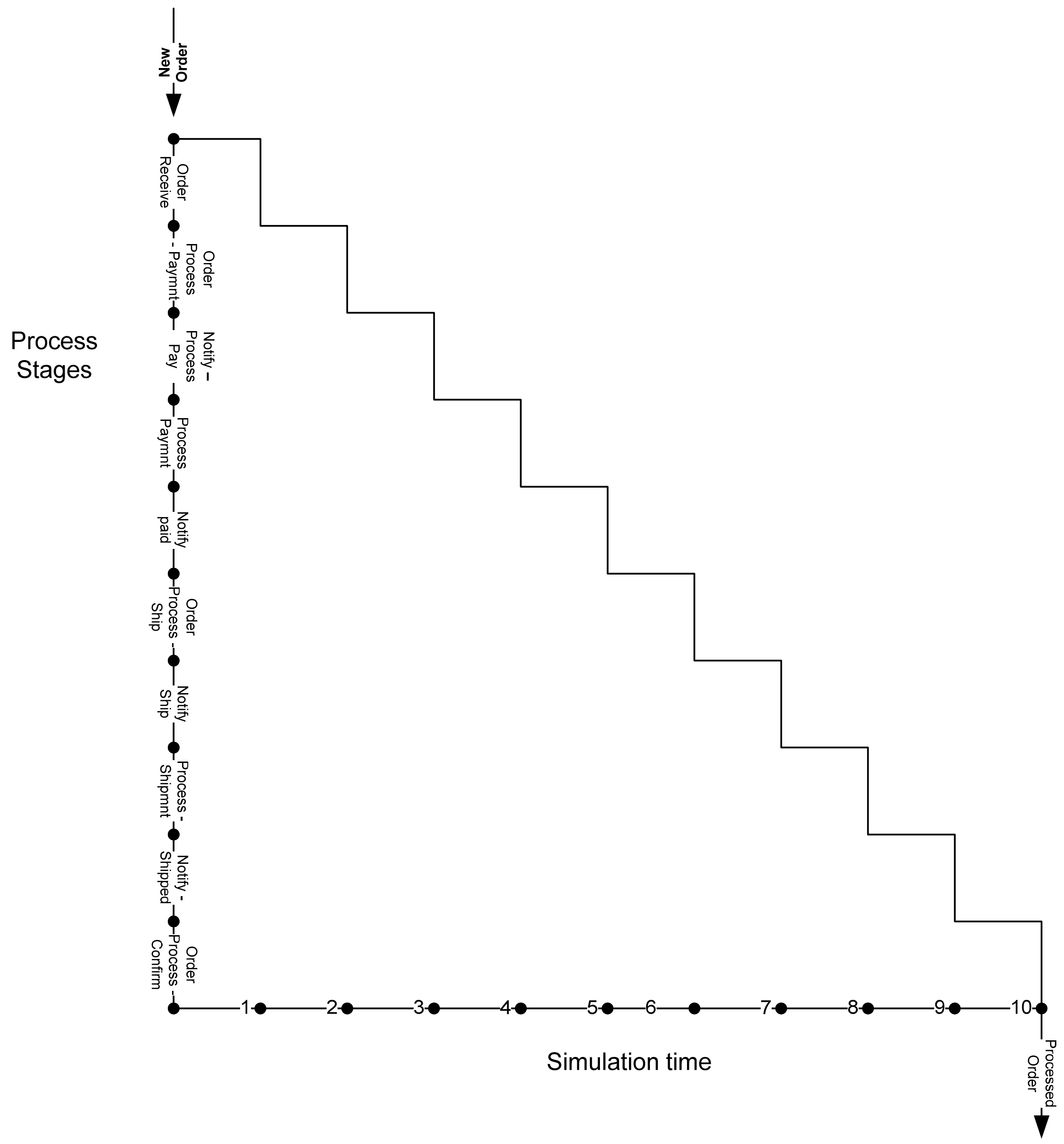


Figure 4.6 Timing diagram for base case verification with known constant processing time value

In the base case verification with known processing time, each processing time input variable was assigned with value of one simulation time unit and the expected and observed total processing time were compared and found to have the same value of 10 simulation time steps. By this we ensured that our model's processing logic (APS System Model shown in Figure 4.5) is correct for known discrete processing time values.

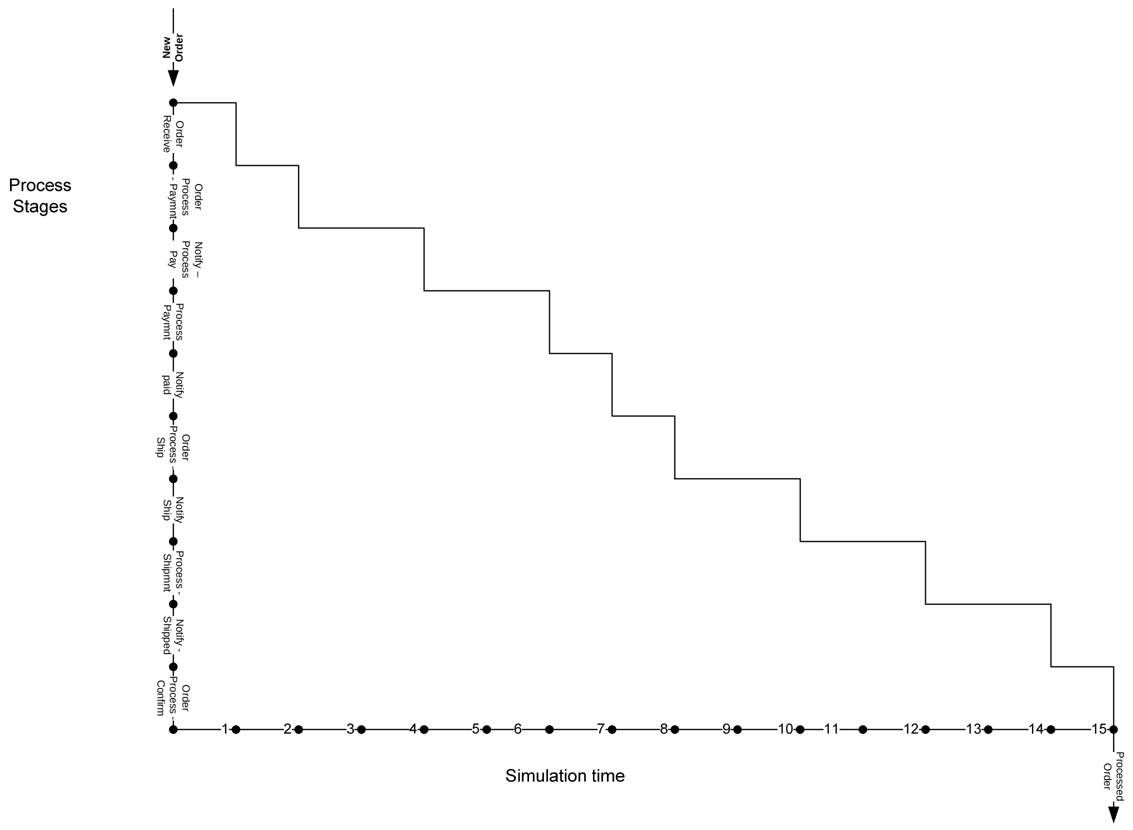


Figure 4.7 Timing diagram for base case verification using known probable values

In the base case verification with known probable processing time values, each processing time input variable was assigned with probability value between one and two simulation time steps. The expected and observed processing time for each input processing time value were compared and found to have the value between one and two simulation time steps. The total processing time calculation was also found to be correct, based on the observed individual service processing time. By this we ensured our model’s processing logic is correct for stochastic input processing time values.

4.6 Model Validation

As discussed in Section 4.4.1, we provided the model with processing time values observed for 8000 transactions, and executed the simulation under steady state condition. The model and the system outputs were exactly same for all 8000 transactions. The below plot in Figure 4.8 shows the system and the model transactions time are same for each order in the first 100 transactions

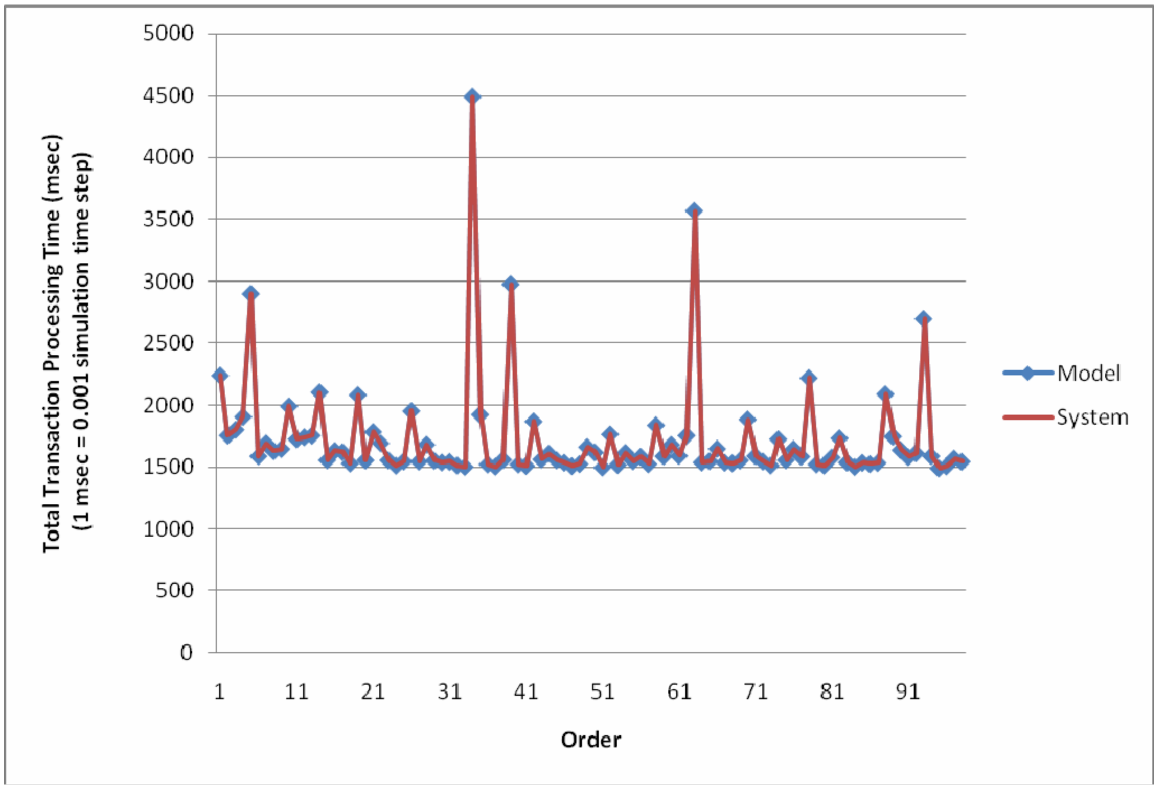


Figure 4.8 Comparison of system and model total transaction time processing per order

4.7 Simulation and Output Analysis

The input arrival rate was varied from 1 to 5 transactions per simulation time step and the service performance time output variable values were collected during separate simulation runs. We were able to calculate metrics as listed in the below table;

Metrics	Calculation
Service Throughput (Transactions/ Simulation time steps)	= Number of Transactions processed / Total elapsed simulation time units
Response Time (simulation time steps)	= Average processing simulation time units of the service model
Queue Length (or average no. of transactions in the system and waiting to be processed)	Based on the discrete time simulation we had Queue length = Input arrival rate * Average response time
Service Utilization (%)	=Total service simulation time units / Total system observation simulation time
Service Demand (simulation time steps)	= No of visits at that service by the same transaction * Average service time per visit

Table 4.2 Performance metrics for DEVS-SOA models.

From the analysis, we see the effect of the feedback message flow in both the order processing and notification service, and eventually affecting the system response time as well as the system queue length. We assumed the network components have zero processing time. There will be significant effect in the system's overall performance if we would have taken any value greater than zero. The model had parallel processing capability during the simulation run, however to achieve increased performance, we still need to add with multiple processors model for the services like order receive, order process and the notification services. See Appendix A, for the throughput, response time, system model queue length, service utilization, and service demand analysis results.

5. Conclusion and Future Work

This work has attempted in modeling a SOA-based system, based on the three system characteristics and created DEVS-SOA models for a publish/subscribe service composition. We were able to model an example prototype system having all of the three system properties. We see that publish/subscribe composition is suitable for systems with sequential or parallel processing, hierarchical and/or flattened topology, feed forward or feedback control flow characteristics. Additionally, we have analyzed the example prototype system performance using our DEVS-SOA models in DEVSJAVA simulation environment.

The DEVS-SOA models we developed is suitable for any type of service operation, however they can be further customized for a specific operation type, for example, notify request only. The notification service can be extended to handle the registration of publication types . Similarly the subscription service can also be extended to handle subscription timeouts, and content based subscription. During our literature review, we came across dynamic DEVS or DSDEVS [Barros95] formalism that supports changes in the model structure, and by preserving the model's closure property. We believe an extension of our DEVS-SOA in DSDEVS formalism can support change in the model structure during simulation. Addition of service instances as processors, based on a coupled component's queue length can increase the processing capability for that coupled component. We believe after analyzing other composition schemes like co-ordination, orchestration, and choreography, our DEVS-SOA service models can be extended to support them.

REFERENCES

[CZ94] Alex ChengHen Chow, Bernard P. Zeigler, “Parallel DEVS: A Parallel, Hierachical, Modular Modeling Formalism”, Proceedings of 1994 Winter Simulation Conference, 1994.

[Barros95] Fernando J. Barros, “Dynamic Structure Discrete Event System Specification: A New Formalism For Dynamic Structure Modeling And Simulation”, Proceedings of the 1995 Winter Simulation Conference, 1995.

[BN02] B Biller, BL Nelson, “Answers To The Top Ten Input Modeling Questions”, Proceedings of the 2002 Winter Simulation Conference, 2002

[DEVS04] ACIMS,DEVSJAVASoftware, http://www.acims.arizona.edu/SOFTWARE/software.shtml#DEVSJAVA, January 2004, accessed date 04/30/2008.

[DPM06] Decker, G., Puhlmann, F., Weske, M.,"Formalizing Service Interactions.", Proceedings of the 4th International Conference on Business Process Management (BPM 2006), http://bpt.hpi.uni-potsdam.de/pub/Public/MathiasWeske/bpm2006-interaction-short.pdf , accessed date 05/10/2008.

[EFGK03] “The Many Faces of Publish/ Subscribe”, Patrick TH. Eugster, Pascal A. Felber, Rachid Guerraoui, Anne-Marie Kermarrec. ACM Computing Surveys, Vol 35, No 2, 2003 June.

[HG06] “A Comparative Study of Web Services-based Event Notification Specifications”, Yi Huang and Dennis Gannon, 2006, Proc. of the Int. Conference on Parallel Processing Workshop, IEEE.

[Oasis04] WS-Base Notification V1.3 & WS-BrokeredNotification V1.3 specification, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn, 10/1/2004, Accessed Dt 02/16/2008.

[Rus06] N. Russell, Arthur.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns: A Revised View., 2006, http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2006/BPM-06-22.pdf, Accessed Dt 05/25/2008

[SSG04] Ranjit K Singh, Hessam S. Sarjoughian, Gary W. Godding, “Design of Scalable Simulation Models for Semiconductor Manufacturing Processes”, http://www.modelingandsimulationorg/issue12/Singh.html, accessed date 02/03/2008.

[Tho06] Thomas Erl, “Service-Oriented Architecture Concepts, Technology and Design”, Prentice Hall, June 2006.

[W306] http://www.w3.org/Submission/WS-Eventing/, 03/15/2006, WS-Eventing.

[WTT06] W.T.Tsai, Chun Fan, Y. Chen, Ramond Paul, Jen-Yao Chung, “Architecture Classification for SOA-Based Applications”, Proceedings of IEEE International Symposium on Object and Component Oriented Real-time Distributed Computing, 2006.

[Wym93] Wymore, A. W., Model-Based Systems Engineering: An Introduction to the Mathematical Theory of Discrete Systems and to the Tricotyledon Theory of System Design, CRC Press, 1993

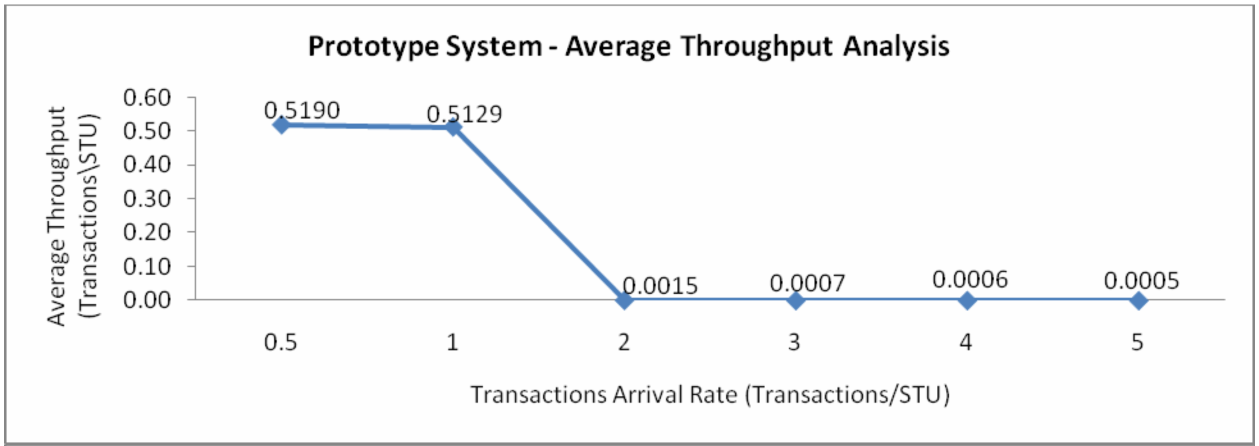
[ZD63] Zadeh, L.A., and Desoer C. A., Linear System Theory, McGraw-Hill, New York, 1963.

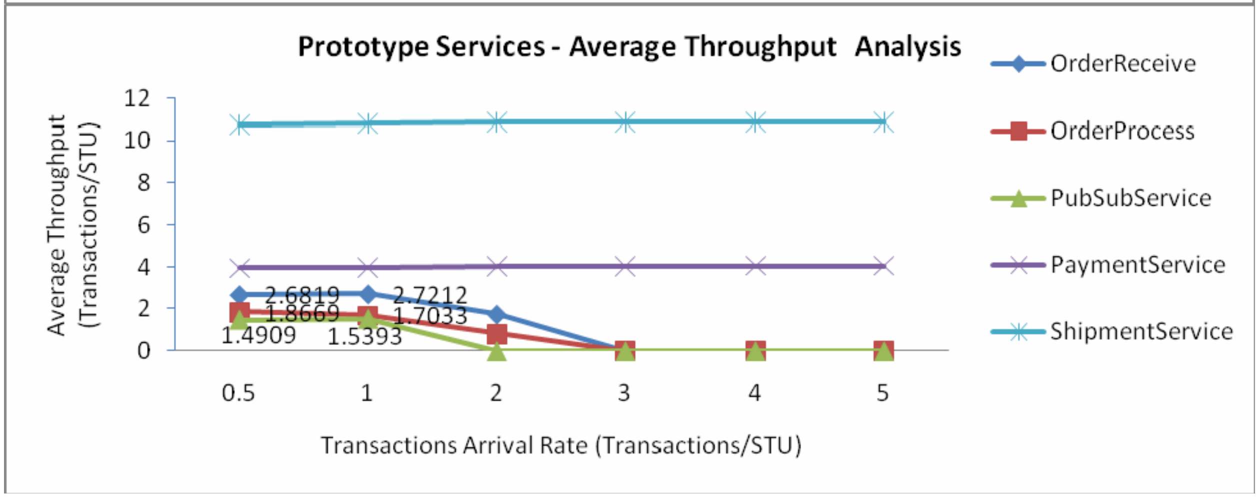
[ZPK00] Bernard P. Zeigler, Herbert Praehofer, Tag Gon Kim. “Theory of Modeling and Simulation”, 2000, Academic Press.

Appendix A

In all graphs, the STU = simulation time units.

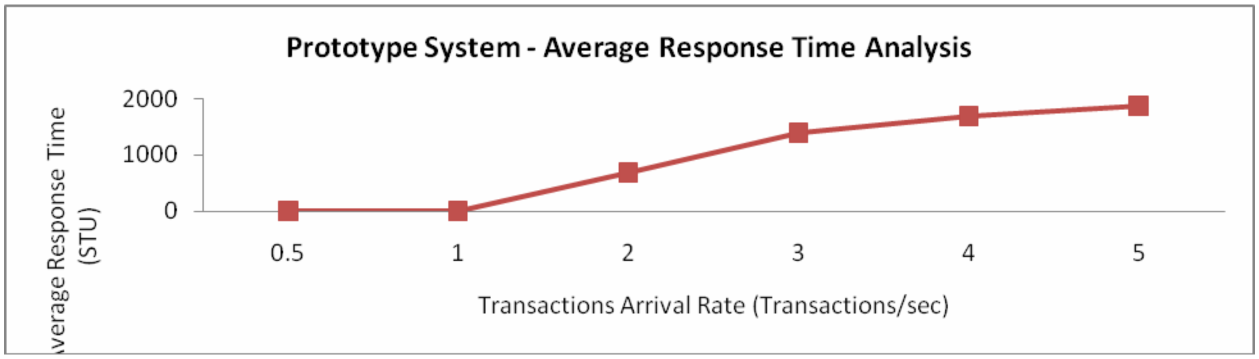
Throughput analysis

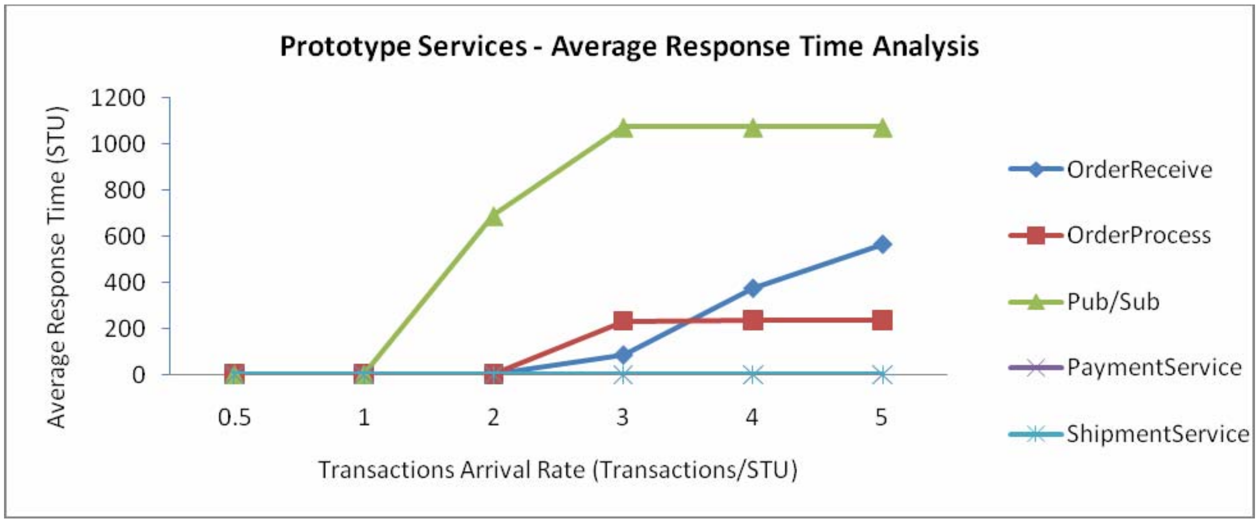




The throughput analysis shows publish/subscribe broker service, order process service and the order receive service requires more processing capability for increased throughput need. The publish/subscribe broker service is having the least throughput capacity because of the effect of more feedbacks it receives in the transaction flow, than the order process service.

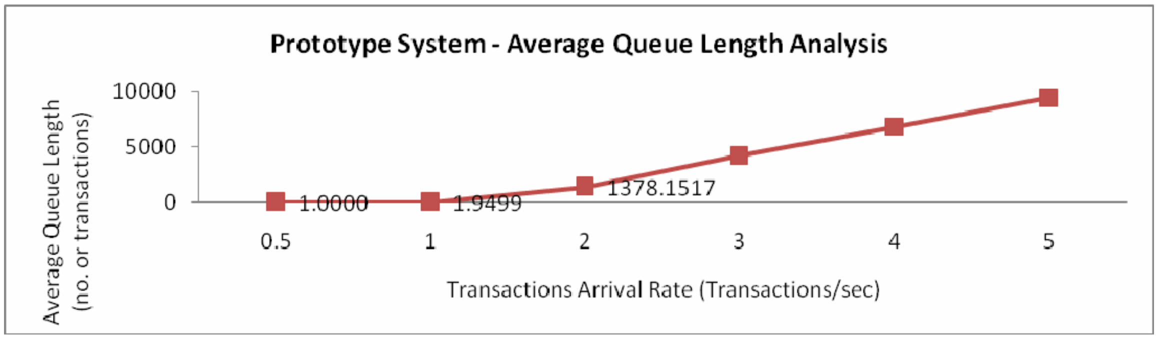
Response time analysis





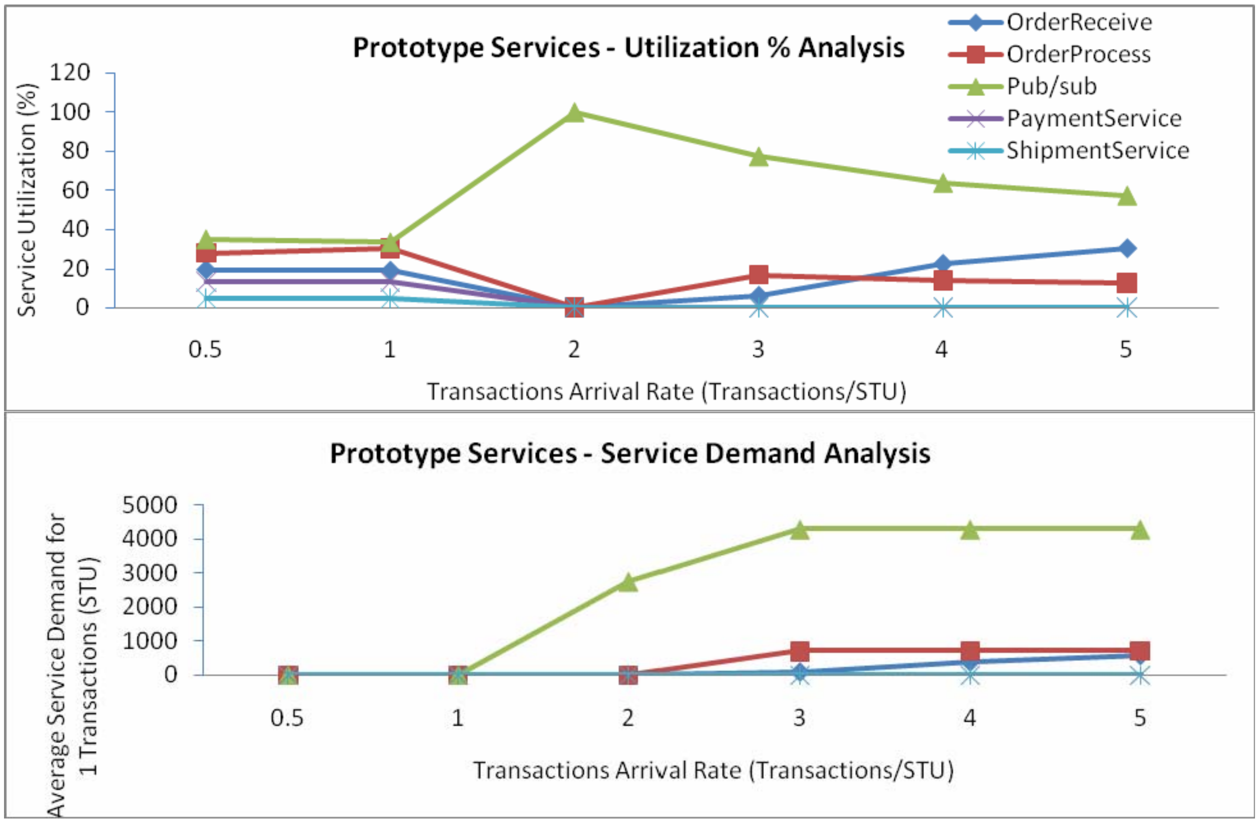
The response time for publish/subscribe, order process, and order receive need to be reduced by increasing the processing capability. However, the graph shows that any increase in both publish/subscribe service and order process service will not show benefits, unless the order receive service has sufficient processing power by not increasing its queue and cause bottleneck.

System model queue length analysis



The queue length analysis shows the average number of transactions still in the model system and waiting to be processed. The queue length increases considerably even with a slight change in the transaction arrival rate.

Service utilization and service demand analysis



Service utilization % graph shows that, the order receive service should process more number of orders, in order to increase the utilization % of any other service in the system. The service demand analysis shows that, the number of feedback to a service and the amount of time an order is waiting in a service queue increases the service demand. Here, the publish/subscribe, order process and the order receive needs more processing capacity to reduce the service demand.