THE RUNTIME BEHAVIOR OF COMPOSITE SOAP WEB SERVICES UNDER TRANSIENT LOADS

A Thesis Submitted to the College of

Graduate Studies and Research

In Partial Fulfillment of the Requirements

For the Degree of Master of Science

In the Department of Computer Science

University of Saskatchewan

Saskatoon

By

Yuxuan Meng

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science

University of Saskatchewan

Saskatoon, Saskatchewan

Canada

S7N 5C

ABSTRACT

Services are computational elements that expose functionality in a platform independent manner. They are the basic building blocks of the service-oriented (SO) design/integration paradigm. Composite Web Services (CWS) aggregate multiple Web Services (WSs), which is typically achieved by use of a workflow language. A workflow coordinates services in a manner that is consistent with the desired overall functionality (e.g. business process).

When the atomic and composite services are exposed to various users, the performance and runtime behavior of WSs becomes important. To ensure wide deployment of CWS, the performance issues must be studied.

This research focuses on the performance of atomic and composite SOAP (Simple Object Access Protocol) WSs under transient overloads. This research includes conducting experiments with WSs, studying the runtime behavior, and building simulation models of WSs workflow patterns. Simulation models of different WSs workflow patterns are built to study different situations. Timeout and network latency are added to the model to better simulate real systems. The simulation models are used to predict the runtime behavior of WSs and CWS, as well as to improve the performance with existing, limited resources.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. Ralph Deters for his support and supervision during my M. Sc. program. I greatly appreciate for his help. Also I would like to thank the other members of my thesis committee: Dr. Derek Eager, Dr. Julita Vassileva, and Dr. Chris Zhang. Finally, I would like to thank my parents for their love and support.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ALWKR	Augmented Least Work Remaining
BPEL	Business Process Execution Language
BPWS4J	Business Process Execution Language for Web Services Java™ Run Time
BPEL4WS	Business Process Execution Language for Web Services
CWS	Composite Web Services
EJB	Enterprise Java Bean
HTTP	Hypertext Transfer Protocol
JMS	Java Message Service
LWKR	Least Work Remaining
SJF	Shortest Job First
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
UDDI	Universal Description, Discovery, and Integration
WAN	Wide Area Network
WS	Web Service
WSDL	Web Service Definition Language
XML	Extensible Markup Language

CHAPTER 1

INTRODUCTION

Services are computational elements that expose functionality in a platform independent manner. They can be described, published, discovered, orchestrated and consumed across language, platform and organizational borders. Services are the basic building blocks of the service-oriented (SO) design/integration paradigm [14].

"The service-oriented architecture (SOA) defines how services are deployed and managed, and facilitates the composition of services across disparate pieces of software, whether new or old; departmental, enterprise-wide, or inter-enterprise; mainframe, mid-tier, PC, or mobile device" [30]. Well-defined services in SOA expose functionality in a platform independent manner. With SOA, developers focus on composing Web Services (WSs) instead of dealing with the complexity of incompatible applications on multiple computers, programming languages, and application packages. Developers can combine the existing services in different environments and add if necessary new services to meet the rapidly changing requirements. As a result, SOA increases reuse, lowers overall costs, and improves the ability to rapidly change and evolve IT systems [30].

Composite Web Services (CWS) aggregate multiple WSs, which is typically achieved by use of a workflow language. A workflow coordinates services in a manner that is consistent with the desired overall functionality. In a workflow (an example of a workflow is shown in Figure 1.1.), services work together to fulfill a complex task (e.g. business process).

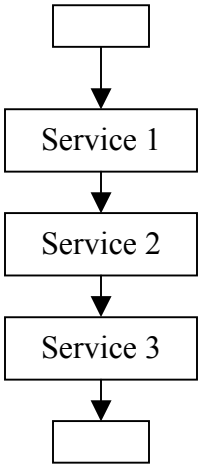

Figure 1.1: A sequential workflow

A CWS itself can be used as a unit in another workflow, and be part of even larger CWS with other atomic services and CWS [14]. In this way, existing services as well as newly built ones from different providers can be aggregated into a CWS, and provide services that meet specified requirements. Workflows facilitate the process of aggregating existing atomic WS and other CWS into new service layers.

When the atomic and composite services are exposed to various users, the performance and runtime behavior of WSs becomes important. To ensure wide deployment of CWS, the performance issues must be studied.

This research focuses on the performance of atomic and composite SOAP (Simple Object Access Protocol [12], [27]) WSs under transient overloads. This research includes conducting experiments with WSs, studying the runtime behavior, and building simulation models of WSs workflow patterns. Simulation models of different WSs workflow patterns are built to study the feature in different situations. Timeout and network latency are added to the model to better

simulate real system. The simulation models are used to predict the runtime behavior of WSs and CWS, as well as to improve the performance with existing, limited resources.

The rest of the thesis is organized as follows. Section two presents the problem definition. Section three presents related work on WSs and performance. Section four provides the outline of experiments and some results. Section five provides the outline of simulation models and the results from simulation runs. Section six presents conclusions and future work.

CHAPTER 2

PROBLEM DEFINITION

This research focuses on the behavior of WSs that are exposed to transient overloads.

For traditional HTTP servers, overload is defined as the point when the demand on at least one of the HTTP server's resources exceeds the capacity of that resource [19].

The typical standard behavior of services under various loads is shown in Figure 2.1 ([14], [15], [22]).

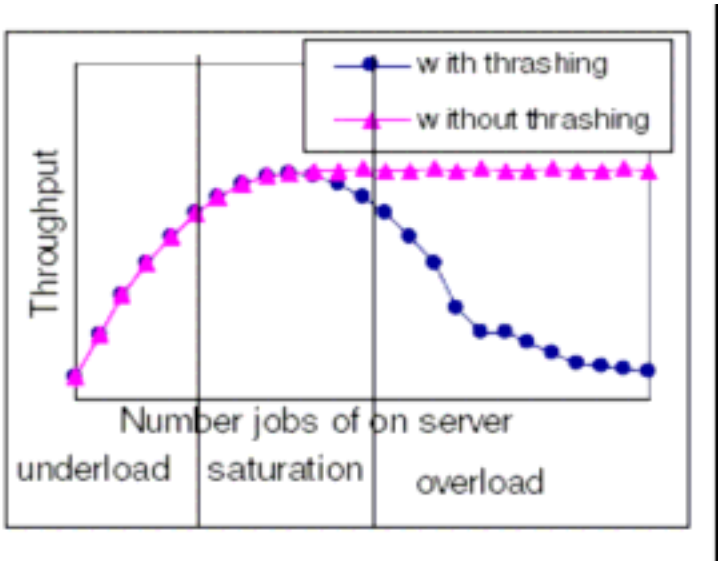

Figure 2.1: Service behavior under various loads [14] [15] [22]

It is not unusual for HTTP servers to experience transient overloads. For example, the traffic increase at the server can lead to a transient period of overload. Transient overload is difficult to

predict, for example, the amount of traffic at a Web site might rise due to an unexpected increase of the site's popularity.

Server overload typically happens when jobs enters the server at a greater rate than the HTTP server can process, which causes the number of jobs at the HTTP server to build up. Soon, the HTTP server reaches the maximum number of connections that it can handle. From the client's perspective, the request for a connection will either never be accepted or will succeed only after several trials. Even when the client's request for a connection does get accepted, the service time may be very long because the request has to share the service with all the other requests at the server [19].

A WS can also experience transient overload. For WSs, the XML processing of the SOAP message can be time-consuming, and the time spent in processing the tasks within the service might not be trivial. When various users frequently invoke the WSs together, it can result in heavy CPU loads, and overload the WSs.

A service that is gradually exposed to an ever-increasing number of service requests experiences three distinct stages, which are "underload", "saturation" and "overload" ([14], [15], [22]). The first phase is the "underload" stage: when handling a load that is below its capacity, the service is not fully utilized. When the number of requests keeps increasing, it leads to an increase of the throughput (number of completed jobs per time unit). As the rate of incoming requests keeps increasing, the server reaches its saturation point (peak load). In the "saturation" stage, the service is fully utilized. As the load continues to increase, the throughput starts to drop and ultimately the "thrashing effect" [22] appears. Thrashing emerges as a result of an overload of physical resources (resource contention) like processor or memory or because of locking (data contention).

A WS can be modeled as a server with a queue according to queuing network modeling [26]. The server's maximum thread number and queue size are limited, that is, the server has limited resources. The service has also timeout limits in its queue. Administrators can use the default value or set the value of these parameters in WSs implementations. When modeling WSs, these parameters can also be defined and initialized to simulate real systems. For a WS that is under transient overload, the jobs arrive faster into the server than the server can handle. The capacity of the server is defined by "service time", and the job rate is defined as "arrival rate". "Arrival rate" can be constant, or follow a certain distribution.

While an atomic WS is a single WS that exposes some functionality, a CWS is more complex, since it can contain atomic WS and other CWS as its components. Those WSs components in CWS are aggregated in certain workflow patterns, such as sequences, loops, and so on. BPEL (business process execution language) is usually used to define workflows, which is an XML-based standard for describing a business process.

CHAPTER 3

RELATED WORK

This chapter reviews the WS and SOA, general behavior of services under load, behavior of atomic WS under loads, and related work of CWS' performance issue.

3.1 WS and SOA

Gartner introduced the Service-Oriented Architecture (SOA) in 1996 as a conceptual framework. (WS is the de facto standard technology for implementing SOA.) SOA defines services as a collection of components. Services have the following features ([20], [30]):

- Services are self contained and modular.
- Services are discoverable and dynamically bound.
- Services are interoperable.
- Services are loosely coupled, reduction of artificial dependencies to their minimum.
- Services have a network-addressable interface.
- Services have coarse-grained interfaces in comparison to finer-grained interfaces of software components and objects.
- Services can be composed.

This research focuses on SOAP based WSs. Therefore, HTTP based WSs ("REST" services) are being ignored.

SOA defines three roles, a Service Consumer, a Service Registry, and Service-Provider. They communicate in the way as shown in Figure 3.1.

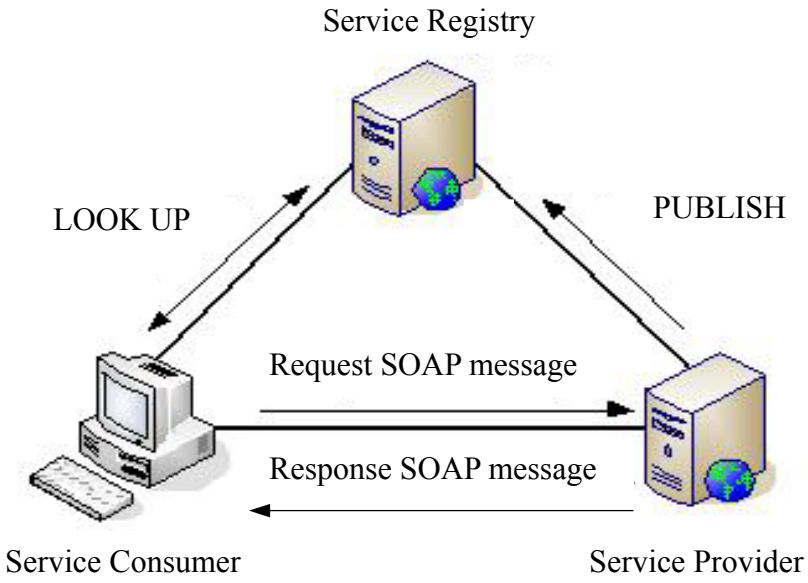

Figure 3.1: The service-oriented architecture

A Service Provider offers an implementation of a WS, and publishes the service in a Service Registry.

The Service Registry acts as a Service Broker, which provides a public listing of registries using UDDI (Universal Description, Discovery, and Integration [37], [38]), so that Service Providers can register their services. Service Brokers within the environment can also replicate their service registries.

The Service Consumer that is also called Service Requester [19] finds a WS, and connects to the service. Service consumers can search for services and get information on how to connect to those services using the Service Providers.

WS technologies include: XML (Extensible Markup Language), SOAP (Simple Object Access Protocol [12], [27]), WSDL (Web Services Description Language [11]), and UDDI (Universal Description, Discovery, and Integration [37], [38]).

The WS and its underlying technologies can be described as shown in the following figure:

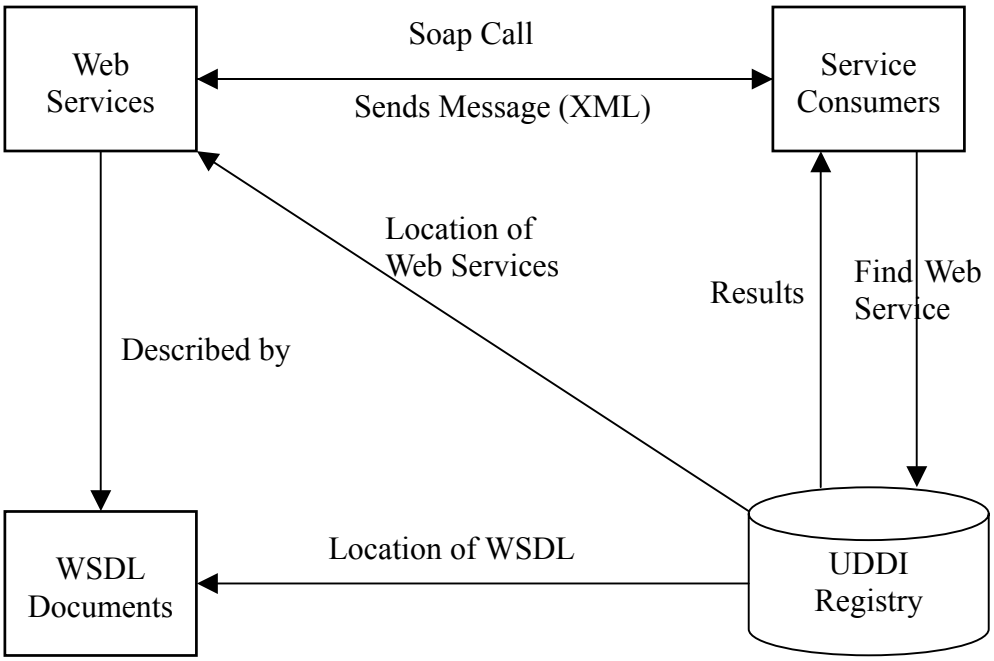

Figure 3.2: Web Services technologies

The uniform data marshaling protocol (SOAP) removes data integration problems, the standard communication infrastructure (WSDL and HTTP) allows incorporating the functions of any services without worrying about their location, and the service query facility (UDDI) further leverages the discovery of desired functions. These features enable rapid service composition to fulfill new functional and nonfunctional (such as reliability [36]) requirements [24].

XML (Extensible Markup Language) is used for data storage. All SOA entities use it as a common language for service description, messaging and service registration, to achieve high interoperability [19];

SOAP (Simple Object Access Protocol [12], [27]) defines the format for registration, searching, and messages in XML (which are sent over HTTP).

Figure 3.3 is a SOAP message example, which uses the HTTP POST request. It includes a SOAP Envelope element and a SOAP Body element.

The SOAP Envelope includes: its "local name"; a "namespace name" of "http://www.w3.org/2003/05/soap-envelope"; none or more information items amongst its "attributes" property; one or two element information items in its "children" property in the following order: an optional Header element; a mandatory Body element [27].

The SOAP body provides a mechanism for transmitting information to a SOAP receiver. In this message, it requests the server to add 25 and 20 (invoke the method “ADD (25,20)”).

```
POST /axis/WebServices/bb1.jws HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/1.2.1
Host: xin.usask.ca:5555
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: ""
Content-Length: 846
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
        <ns1:add soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                 xmlns:ns1="http://xin.usask.ca:5555/axis/WebServices/calculator.jws">
                 <ns1:arg0 href="#id0"/><ns1:arg1 href="#id1"/></ns1:add>
        <multiRef id="id0" soapenc:root="0"
                 soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                 xsi:type="soapenc:int"
                 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
            25 </multiRef>
        <multiRef id="id1" soapenc:root="0"
                 soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                 xsi:type="soapenc:int"
                 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
            20</multiRef>
    </soapenv:Body>
</soapenv:Envelope>
```

Figure 3.3: An SOAP message example.

The WSDL (Web Services Description Language [11]) is an XML document that describes a WS. This description includes all the information needed for invoking service methods from other nodes [19].

A WSDL document uses the following elements in the definition of network services [43]:

- Types– a container for data type definitions that uses some type system (such as XSD).
- Message– a typed definition of the data that are being communicated.
- Operation– a description of an action that is supported by the service.
- Port Type–an abstract set of operations that is supported by one or more endpoints.
- Binding– a protocol and data format specification for a particular port type.
- Port– a single endpoint that is defined as a combination of a binding and a network address.
- Service– a collection of endpoints that are related.

Below is an example that shows the WSDL definition of a simple service. The service provides stock quotes, and supports a single operation called GetLastTradePrice, which is deployed using the SOAP 1.1 protocol over HTTP. This example uses a fixed XML format instead of the SOAP encoding [43].

```
<?xml version="1.0"?>
<definitions name="StockQuote"
targetNamespace="http://example.com/stockquote.wsdl"
                xmlns:tns="http://example.com/stockquote.wsdl"
                xmlns:xsd1="http://example.com/stockquote.xsd"
                xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                xmlns="http://schemas.xmlsoap.org/wsdl/">
    <types>
```

```
    <schema targetNamespace="http://example.com/stockquote.xsd"
            xmlns="http://www.w3.org/2000/10/XMLSchema">
        <element name="TradePriceRequest">
            <complexType>
                <all>
                    <element name="tickerSymbol" type="string"/>
                </all>
            </complexType>
        </element>
        <element name="TradePrice">
            <complexType>
                <all>
                    <element name="price" type="float"/>
                </all>
            </complexType>
        </element>
 </schema>
</types>

<message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
</message>

<message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
</message>

<portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
        <input message="tns:GetLastTradePriceInput"/>
        <output message="tns:GetLastTradePriceOutput"/>
    </operation>
</portType>
```

```
    <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
        <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetLastTradePrice">
            <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>

    <service name="StockQuoteService">
        <documentation>My first service</documentation>
        <port name="StockQuotePort" binding="tns:StockQuoteBinding">
            <soap:address location="http://example.com/stockquote"/>
        </port>
    </service>

</definitions>
```

Figure 3.4: SOAP 1.1 Request/Response via HTTP

The UDDI (Universal Description, Discovery, and Integration) keeps registry listing of companies, WSs offered, and information on how to connect to those WSs ([37], [38]). Each registry entry is an XML document. It is a directory model, providing a yellow page service for WSs. Its entries consist of three parts: the “white pages” that contain information about the WS provider, the “yellow pages” that include industrial categories based on standard taxonomies, and the “green pages” that describe the interface of the service, e.g. by integrating the WSDL file [19].

3.2 Service under Load

3.2.1 Queuing Network Modeling

Queuing network modeling [26] is an approach for computer system modeling in which the computer system is represented as a network of queues, which can be evaluated analytically. A network of queues is a collection of service centers, which represent system resources, and customers, which represent users or transactions. Figure 3.5 [26] shows a queuing service center:

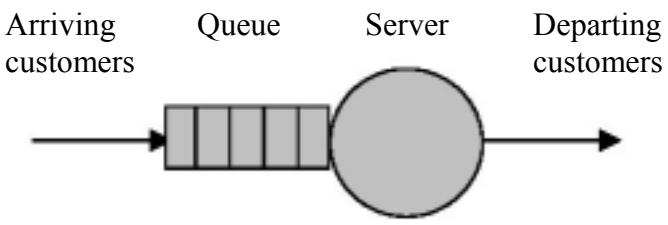

Figure 3.5: A single service center

This model has two parameters. The first one specifies the workload intensity, which in this model is the average rate of customers. The second one specifies the service demand, which is the average service requirement of a customer [26].

In this research, the first parameter is named “inter-arrival time”. For example, if the inter-arrival time is 2 seconds, this means that the customers arrive at the server every 2 seconds. In this example the “arrival rate” is 0.5 customers/second, which is noted as “λ”. The second parameter is named as “service time”, and noted as “S”. For example, the service time is 1.25 seconds for one customer. Their measurement and definitions are discussed in more detail as below:

For an “open system” that has customers arriving and departing, the following quantities can be measured [26]:

- T, the length of time observed.
- A, the number of request arrivals observed.
- C, the number of request completions observed.

From these measurements, the following additional quantities can be defined [26]:

- λ, the arrival rate: λ=A/T. For example, if 8 arrivals are observed during an observation interval of 4 minutes, then the arrival rate is 8/4 = 2 requests/minute.
- X, the throughput: X =C/T. For example, if 8 completions are observed during an observation interval of 4 minutes, then the throughput is 8/4 = 2 requests/minute.

If the system consists of a single resource, the following quantities can also be measured [26]:

- B, the length of time in which the resource was observed to be busy.

Two more defined quantities now are meaningful [26]:

- U, the utilization: U=B/T. For example, if the resource is busy for 2 minutes during a 4 minute observation interval, then the utilization of the resource is 2/4, or 50%.
- S, the average service requirement per request: S=B/C. For example, during an observation interval, if there are 8 completions and the resource is busy for 2 minutes during that interval, then on the average each request requires 2/8 minutes of service.

The Utilization Law can be defined as: U = XS [26].

Queuing network modeling can be viewed as a small subset of the techniques of queuing theory, which is selected and specialized for modeling computer systems. Queuing network models achieve relatively high accuracy at relatively low cost [24].

3.2.2 Bottleneck Resources of HTTP Servers

In a HTTP server, bottleneck resources are defined as the resources that have the biggest impact on the performance of a HTTP server that is under load, and the resources in a HTTP server that experience overload first. The three important bottleneck resources for traditional HTTP servers are the CPU, the disk to memory bandwidth and the server's limited fraction of its ISP's bandwidth [33]. On a web site that consists primarily of static content, a common performance bottleneck is the limited bandwidth that the server gets from its ISP. Therefore, buying more bandwidth is more costly than upgrading memory or CPU. Schroeder et al. [33] assume that the bottleneck resource is the limited bandwidth that the server has purchased from its ISP, and model the limited bandwidth by placing a limitation on the server's uplink.

For traditional HTTP servers with static workloads, CPU load is typically not an issue, and the main bottleneck resource is the limited bandwidth [33]. However, the CPU can be the bottleneck resource of WS. For WS, the XML processing of the SOAP message is time-consuming, and the time cost in calculation within the service can be non-trivial. When various users frequently invoke the WSs together, it can spend much CPU time, cause heavy CPU loads, and make WSs overloaded.

3.2.3 Persistent and Transient Overload

There are two types of overloads, persistent overload and transient overload. Persistent overload is used to describe a situation where the server is run under a fixed load $\rho > 1$ during the whole experiment [26]. The motivation behind experiments with persistent overload is mainly to gain insight into what happens under overload.

The overloaded state is unlikely to persist for too long in practice due to system upgrades. However, there are bursts in Web traffic, even in the case of regular upgrades. Therefore, a popular HTTP server is still likely to experience transient periods of overload [33].

3.3 Atomic WS under Loads

The performance of WSs is similar to that of other general servers or HTTP servers in computer systems. As introduced in the previous part of the thesis, the concepts and conclusions in the issue of servers under loads can usually be applied to this domain.

The WS testing domain is relatively new. Most of the current work focuses on functional testing. Performance related WS testing usually focused on comparing different SOAP implementations, rather than application-specific performance [45]. However, such application-specific performance is very important in situations like service level driven management and QoS-aware WS.

Zhu et al. [45] used the Axis 1.4 package and Java to develop services that performed simple processor bound tasks [3]. A lightweight WS that returns its single argument was used. Zhu et al. [45] got samples of response time distribution from experiments, and average response time and throughput in an incremental requests simulation of open model. The experimental runs revealed that every WS provider exhibits thrashing either due to the costs of parsing XML messages or the resource/data contention issues of its underlying business logic [45].

Zhu et al. ’s work has some limitations. The default implementation of the load-testing suite is still relatively simple. It covers only successful test scenarios and does not produce more interesting stress testing data.

3.4 Composite Services & Loads

3.4.1 Workflows

Services are different from objects or components in traditional Object-Oriented or Component-Based development paradigms. As described in Huang et al. [24], some universally accepted features of services are:

1. Every service is an entity consisting of data, and operations over the data. Both objects and components share this feature, too.

2. Every service has its own processes. This may not be true for either objects or components.

3. Every service can interact with other services. The interaction may have different meaning for objects and components because they may not have stand-alone processes.

As described in Huang et al. [24], researchers may choose to impose some of the following assumptions:

I. That the standard communication infrastructure among services is well established. For example, it can imply reliable, synchronous, ordered communication, etc.

II. That the execution of every service is an atomic operation: it is either an entire success or nothing happens, and is "ACID" transaction.

III. There is no real-time requirement over the execution of services.

IV. The services admit a two-segment ownership. The service vendors own the services, the hardware, and the resources that the services are being executed upon. The application developers own the applications composed by the services, the hardware, and the resources that the applications are being executed upon.

V. The service discovery facility is well established.

The vision of workflows is formed by accepting all assumption I, II, and III, and not affected by assumptions IV and V [24].

The complexity of the model, and hence the verification, is significantly reduced, by accepting assumptions I, II, and III [24]. The interactions among services can be simplified to be atomic method calls. Thus, the business logic of the application is simplified to be a workflow of activities, which is connected by various control flow constructs, such as sequence, branch, loop, and concurrency, where an activity is either a local computation or a method call [24].

Figure 1.1 showed a very simple workflow of three sequential WSs. Here, Figure 3.6 gives another practical example (Figure 1 in [24]), which is a typical workflow of a WS performing online shopping.

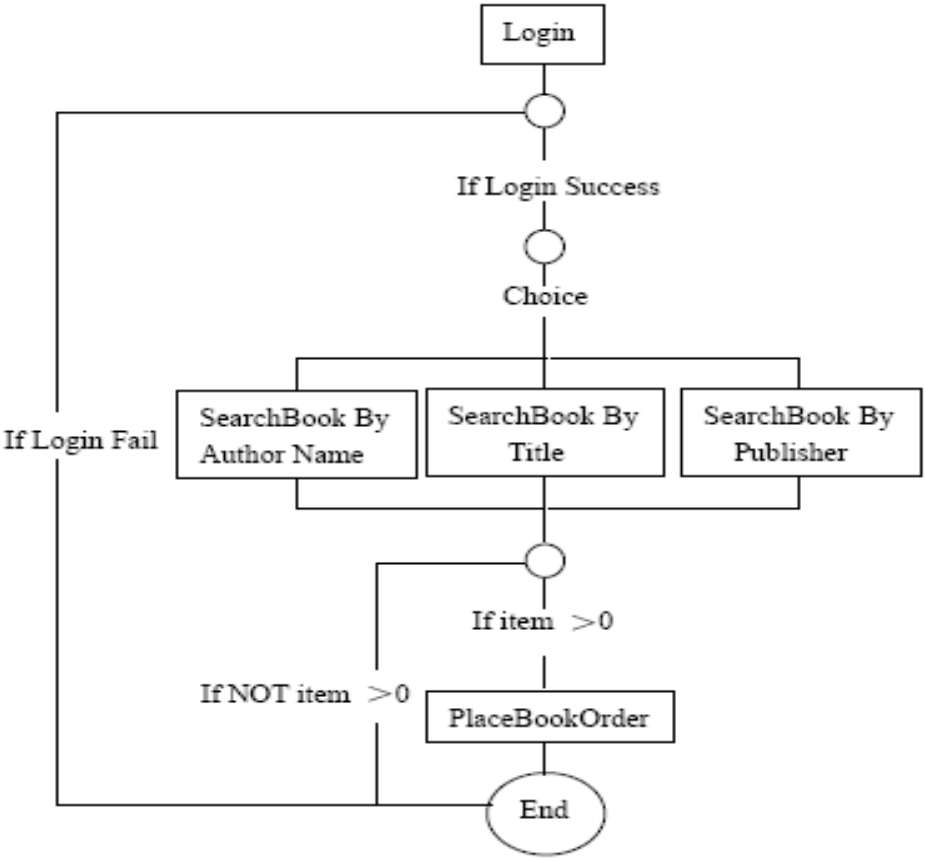

Figure 3.6: Workflow for online shopping [24]

Workflows are designed for concealing the implementation details of the application, and for better understanding the business logic [24]. The introduction of workflows is to help users without specific software engineering training to design and develop WSs. It assumes that common functional activities are already available, and thus users can compose new services by simply connecting together those building blocks. This assumption matches some known industrial best practices, such as Google Map and Amazon Web Services [2].

3.4.2 WS Composition and BPEL

WS composition aims to create a Web process from individual WS. Web processes facilitate expanding the utility of WSs. WS composition can be static or dynamic. In a static composition, the services are predetermined when designing the Web process. In a dynamic composition, the WS is decided at run-time by, for example, the process enactment engine. Dynamic composition needs to find services by searching registries at run-time [10].

WS composition can be represented as a workflow graph that has activities (services) and transition links (control and data)[10]. Data links and control links are used to specify the data flow and control flow respectively among the services. Standard constructs like XOR splits, AND splits, XOR joins, AND joins are used to capture the execution logic. For example, an XOR split can indicate the branching of the control flow in one of the outgoing control links, and an AND split indicates the branching of the control flow in all of the outgoing control links in parallel; an AND join indicates synchronizing on all incoming controls links, while an XOR join indicates waiting on one of the indicated incoming control links [10].

CWS can contain atomic WS and other CWS as components. Those WS components in CWS are aggregated in certain workflow, such as sequences, loops, and so on.

BPEL (business process execution language) [7], [42] is usually used to define workflows. BPEL is an XML-based standard for describing a business process, which defines a notation for specifying a business process behavior based on WS [7]. BPEL files are executed on workflow servers (server executes scripts).

The BPEL basic structure is as following:

<process>

<partners> … </partners>

<variables> … </variables>

<correlationSets> … </correlationSets>

<faultHandler> … <faultHandler>

<compensationHandler> … </compensationHandler>

<eventHandler> … </eventHandler>

(activities)*

</process>

BPEL defines the following activities: receive, reply, invoke, assign, throw, terminate, wait, empty, sequence, switch, while, pick, flow, scope, compensation. These activities have shown to enable easy workflow development. Figure 3.7 shows the activities that are defined in BPEL:

Figure 3.7: The activities in BPEL

Figure 3.8 is a code example of “partnerLink” elements in a BPEL process that invokes a Synchronous Partner:

```
<process>
    <partnerLinks>
        <!-- Interaction between client and this BPEL process-->
        <partnerLink name="ClientPartnerLinkABC"
            partnerLinkType="ClientPartnerLinkType"
            myRole="HelloService"/>

        <!-- Interaction between this BPEL process and the external web service -->
        <partnerLink name="ExternalPartner"
            partnerLinkType="MyPartnerLinkType"
            partnerRole="GreetingServiceProvider"/>
    </partnerLinks>
```

```
    <!-- Receive a request via ClientPartnerLinkABC partnerLink -->
    <receive partnerLink="ClientPartnerLinkABC"
            portType="portTypeA"
            operation="sayHello"
            variable="inputVar">
    </receive>
    . . .
    <!-- Invoke the partner via the ExternalPartner partnerLink -->
    <invoke partnerLink="ExternalPartner"
            portType="GreetingPortType"
            operation="getGreeting"
            inputVariable="inputVar"
            outputVariable="outputVar"/>
    . . .
    <!-- Send a reply via ClientPartnerLinkABC partnerLink -->
    <reply partnerLink="ClientPartnerLinkABC"
            portType="portTypeA"
            operation="sayHello"
            variable="outputVar"/>
</process>
```

Figure 3.8: An example of BPEL process [32]

A BPEL process can define a business protocol role, using the notion of abstract process. For example, in a supply-chain protocol, the buyer and the seller are two different roles; each has its own abstract process. Abstract processes describe public aspects of the business protocol; specifically, they handle only protocol-relevant data. BPEL also provides a method to identify protocol-relevant data as message properties [42].

BPEL can also define an executable business process [42]. “The logic and state of the process determine the nature and sequence of the WSs interactions conducted at each business partner, and thus the interaction protocols” [42].

In summary, BPEL is used to model the behavior of both executable and abstract processes. The scope includes [7]:

- Sequencing of process activities, especially WSs interactions.
- Correlation of messages and process instances.
- Recovery behavior in case of failures and exceptional conditions.
- Bilateral WSs based relationships between process roles.

3.4.3 Workflows Performance

Huang et al. [24] defined different WSs based on the assumptions used for them, and build state-transition models for workflow as well as for other services. No experimental results were given in Huang et al. [24]. However the work gives a guide when selecting model-checking technologies for verifying WSs.

3.4.3.1 Centralized orchestration

Typically, a single coordinator node executes a CWS specification. It receives the client requests, makes the required data transformations and invokes the component WS as defined in the specification. This mode of execution is called “centralized orchestration” [8]. The coordinator node is responsible for coordination of all data and control flow between the components; therefore it becomes a performance bottleneck. All data is transferred between the various components via the coordinator node, instead of being transferred directly from the point of generation to the point of consumption, which leads to unnecessary traffic on the network.

3.4.3.2 Decentralized orchestration

When specifying a composite service using a language like BPEL4WS, the specification can be analyzed using techniques such as program analysis [28], petri-nets [39], etc. The data and control dependences between the components can be analyzed, and the code can be partitioned into smaller components that execute at distributed locations. This mode of execution is called "decentralized orchestration" [8], which has multiple engines, each executing a CWS specification (a portion of the original CWS specification but complete in itself) at distributed locations. The engines communicate directly with each other (instead of through a central coordinator) to transfer data and control when necessary in an asynchronous manner. The decentralized execution brings performance benefits for the following reasons [8]:

- It has no centralized coordinator that is a potential bottleneck.
- Distributing the data reduces network traffic and improves transfer time.
- Distributing the control improves concurrency.
- Asynchronous messaging between engines allows better throughput and graceful degradation [21].

Chafle et al. [8] discussed Centralized Orchestration and Decentralized Orchestration of CWS, and mainly investigate build time and runtime issues related to the "decentralized orchestration" of CWS. They provided a detailed discussion of the servers that participate in decentralized execution: their thread pool design, as well as communication protocols. They experimentally reconfirmed the performance benefits that decentralization provides, and also evaluated two different decentralization schemes, showing that JMS (Java Message Service) [8] is a more efficient communication protocol than HTTP for engine-to-engine communication in a

decentralized setup. They also showed that at very high loads, there is a trade-off between throughput and response time with the two schemes.

Similar to the previous paper [8], Chafle et al. [9] also discussed "centralized orchestration" and "decentralized orchestration" of CWS. Chafle et al. [9] focused on improving performance of CWS over a Wide Area Network (WAN), and they mainly investigated how different topologies generated by decentralized orchestration are affected differently by variations in WAN conditions. CWS with components distributed over a WAN pose performance problems due to changing WAN conditions. Chafle et al. [9] tried to address this problem by proposing an adaptive system based on decentralized orchestration.

A decentralized orchestration runtime consists of multiple flow engines running at distinct nodes. Chafle et al. [9] use the BPWS4J [6] engine to execute the CWS partitions written in BPEL4WS. These partitions interact with the corresponding WSs using SOAP over HTTP.

Chafle et al. [9] described a model for estimating the throughput of a topology. The performance model builds on the basic principles of queuing theory and the characteristics of decentralized CWS, using some monitored parameters. With the model, one can calculate the throughput of all competing topologies at the start of each run, and select the topology with highest throughput, and build an "adaptive" orchestration system that adapts to changes in WAN conditions by varying the communication pattern between the coordinator node and the component WS at runtime in response to fluctuations in available bandwidth. This "adaptive" decentralized orchestration system makes use of the tool developed in Nanda et al. [28] to generate decentralized orchestration topologies for a given CWS. Such an adaptive orchestration system is later proved by experiments to improve performance of CWS.

The system presented in Chafle et al. [9] still has some limitations. For example, if the wide area network conditions deteriorate due to congestion or link failure at the last hop (an edge router), instead of in the Internet backbone, a partition hosted on such a node (that is being served by the affected edge router) could be equally affected in all decentralized topologies. In such situations, all decentralized topologies will show performance degradation. Another limitation of this system is that, this "adaptive" orchestration system relies on a switch component, which can become a single point of failure. Furthermore, the performance model in Chafle et al. [9] makes simplistic assumptions, such as: a separate queue for each link at the application level, same bandwidth for both incoming and outgoing messages, large message size, moderate request rates, and so on, so the model still needs to be improved to represent real WAN behavior.

Dyachuk et al. ([15], [16]) discuss the performance issue of CWS when services are overloaded, and use scheduling of service requests to improve the overall CWS performance in overloads situations. Different scheduling policies are evaluated for the CWS workflow patterns sequence and split-synchronization. Besides, Dyachuk et al. [15] present a heuristic based scheduling policy, named Augmented Least Work Remaining (ALWKR), which extends Least Work Remaining (LWKR [13], [16]) by taking advantage of existing workflow topology information.

Some assumptions used in Dyachuk et al. [15] include: CWS are composed according to the "Sequence and Parallel Split & Synchronization" patterns [40] with a static structure; size of the sub-requests is known a priori or can be estimated ([16], [17]). SJF is the optimal non-preemptive scheduling policy for minimizing average service time if accurate information on job

sizes is available, which is proved by Smith [34]. (SJF is optimal only if there is no preemption possible, or if there are only static sets of jobs.)

Dyachuk et al. [15] use the simulation tool AnyLogic [44] to build models that describe the way WSs and CWS respond to various loads, and run simulations to compare the performance with different ways of request scheduling. The experiments results show that ALWKR improves the performance, but requires placing a proxy in front of each component; SJF scheduling improves performance only for simple structures orchestrating up to three services; while for bigger and more complex structures, SJF does not make any noticeable optimizations; ALWKR outperforms SJF in average by 20-40%.

Dyachuk et al. [15] show promising results of applying scheduling. However, the current work only focused on a very simplified SOA architecture, more complex CWS need be modeled to simulate real SOA implementations well.

3.5 Summary

WSs and SOA are still relatively new research areas. Though a lot of work has been done on the performance issue of traditional HTTP servers, the performance of WSs is not well studied yet. Some studies have been done on the experiment of atomic WS, or the performance comparison of different implementations for CWS.

More work need to be done on the performance of CWS, that is overloaded and under different kinds of load, and on the impact of different workflow patterns. Also, experiments and simulation models can be combined to study the run time behavior of CWS. The experiment results can validate simulation models, and validated simulation models can predict the run time behavior of CWS. The Table 3.1 is a summary of the related work listed in this chapter:

Table 3.1: The summary of related work

Reference paper	Work and contributions	Limitations
Zhu et al. [45]	They used Axis 1.4 package and Java to develop services, got samples of response time distribution from experiments. The experiments revealed that every WS provider exhibits thrashing either due to the costs of parsing XML messages or the resource/data contention issues.	The implementation of the load-testing suite is still relatively simple. It covers only successful test scenario generation and does not produce more interesting stress testing data.
Huang et al. [24]	They defined WS into different visions based on the assumptions used for them, and build state-transition models for workflow as well as for other services. It gives a guide when selecting model-checking technologies for verifying WSs	No experiment results were given.
Chafle et al. [8]	They discussed the servers that participate in decentralized execution: their thread pool design and communication protocols; experimentally reconfirmed the performance benefits that decentralization provides, and showed that JMS is a more efficient communication protocol than HTTP in a decentralized setup.	They focus mainly on decentralized execution. They did not examine different workflow patterns and their influence on the performance of CWS.
Chafle et al. [9]	They discussed improving performance of CWS over a Wide Area Network (WAN); investigated how different topologies generated by decentralized orchestration are affected differently by variations in WAN conditions; and proposed an adaptive system based on decentralized orchestration.	The system depends on the Switch component, which could act as a single point of failure. The performance model makes simplistic assumptions, so the model still needs to be improved to represent real WAN behavior.
Dyachuk et al. ([15], [16])	They discussed the performance issue of CWS when services are overloaded, and evaluated different scheduling policies, aiming to employing scheduling service requests to improve the overall CWS performance. They showed promising experiment and simulation results of applying scheduling.	The work only focused on a very simplified SOA architecture, more complex CWS need be modeled to simulate real SOA implementations well.

CHAPTER 4

EXPERIMENTS

This thesis focuses on studying the performance of atomic WS and CWS, when they are exposed to transient overloads.

In this and the following two chapters, experiments and simulation results are shown.

Different experiments were done in order to study the WS' behavior when slightly overloaded. Experiment platforms, workflow types, and workload types divide different experiments setups. Experiment platforms include Java 6, Axis 1, and Axis 2. Workflow types include atomic WS and CWS. Workload types include constant load, and non-constant load (for example, the interarrival time can be exponentially distributed, or be certain kind of "burst", and so on). A subset of all experiment settings is studied. The following table is a summary of the experiments in this chapter:

Table 4.1: The summary of the experiments

Experiment	Type of workflow	Load	Experiments platforms
Experiment A	An atomic WS	Constant load	Java 6, Axis 1, Axis 2
Experiment B	Two WSs in sequential workflow	Constant load	Axis 1, and Java 6
Experiment C	Three WSs in sequential workflow	Constant load	Axis 1
Experiment D	Atomic WS or CWS in a sequential workflow	Non-constant (e.g., exponentially distributed, "burst")	Axis 1
Experiment E	CWS in a workflow that has loops or switches	Constant and non-constant load	Only implemented in AnyLogic models

The experiment A examines the behavior of an atomic WS that is exposed to transient overloads. The experiment B aims to study the behavior of CWS in short sequential workflows, while the experiment C aims to study the behavior of CWS in longer sequential workflows. The experiments A, B, C focus on constant workload.

As the experiments A, B, C, the experiment D aims to study the behavior of atomic WS and CWS in sequential workflows. The experiments D focus on non-constant workload.

The experiment E aims to study the behavior of CWS in more complex workflows, such as the workflow of loops or switches. Both constant and non-constant load can be studied in experiment E.

4.1 Experiment A

The WS is built with the Apache Tomcat Servlet/JSP container [5]. Apache Tomcat version 5.5 is used. Apache Tomcat is the servlet container that is used in the official Reference Implementation for the Java Servlet and JavaServer Pages technologies. The Java Servlet and JavaServer Pages specifications are developed by Sun under the Java Community Process.

The atomic WS is built on Java SE 6 platform [25], with the NetBeans IDE 5.5 [29]; the same atomic WS is also built on Apache Axis 1.4 [3] and Apache Axis2/Java Version 1.2 [4], with the NetBeans IDE 5.0 [29].

A lab computer is used as server, which receives and processes requests; and an experiment computer is used as client, which sends requests.

The configuration of the computer that serves as the server is as follows:

HP workstation xw6200, Intel(R) Xeon (TM) CPU 3.20 GHz, 2.0 GB of RAM, Windows XP Professional.

The configuration of the computer that serves as the client is as follows:

Intel Pentium III processor, 803MHz, 1.0 GB of RAM, Windows XP Professional.

The server implements a WS, which calculates Fibonacci numbers [35]. Calculating Fibonacci numbers is simple, and it uses small amount of memory, so it is chosen in this research. Fibonacci numbers are defined as follows:

f(n) =1, for n=1 and n=2,

and f(n) = f(n-1)+ f(n-2), for n>2.

Thus, the sequence is 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, and so on.

The well-known recursive algorithms for computing f(n) is used for the WS:

f(n):

if n = 1 or n = 2 then the answer is 1;

if n > 2 then {compute FN1 = f(n - 1);

compute FN2 = f(n - 2);

the answer is FN1 + FN2}.

The recursive algorithm for computing Fibonacci numbers has exponential time complexity [35], and can be used to simulate different job-sizes.

In the experiments, 1000 requests are sent in each experiment with constant interarrival time. Each request includes the parameter n. After receiving requests from the client, the service calculates the value of fibonacci (n) recursively. When the server completes calculating the results, it records the current time as “departure time” (the time when the jobs are completed and departure from the server), and sends this time to the client as the return value. (The return value is not the calculated value of fibonacci (n), but the “departure time”. The server also records the

"departure time". In this way, the time is recorded more accurately, since the client can be very busy when it receives the results, it may not record the correct time. It also eliminates some of the impact from network delay.)

The experiment setting is shown as below (Figure 4.1):

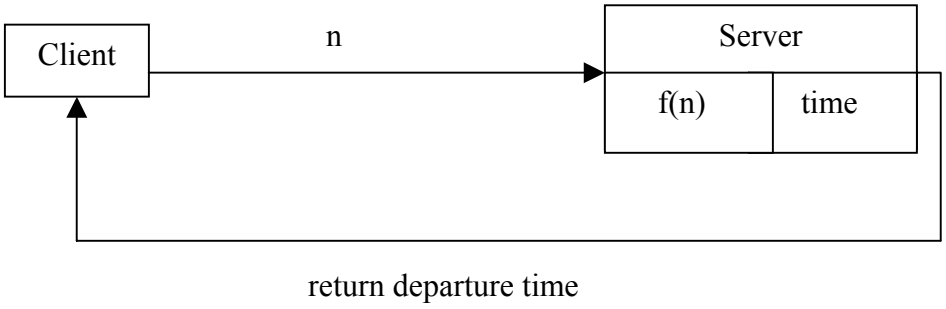

Figure 4.1: The experiment setting of atomic WS

When n is larger, the calculation of its fibonacci (n) is more time-consuming, and requires more processor time. In this research, the parameter n is chosen as 42. The service calculates the value of fibonacci (42). The service time of calculating fibonacci (42) is about 1.9 seconds, so it can simulate some load on the service.

In the experiments, the interarrival time is set slightly smaller than the server's service time; therefore, the server is slightly overloaded.

The atomic WS's experiment's setup is noted as "setup A", which has one client and one server, with constant inter-arrival time. "Experiment A" uses setup A, and is done on different platforms.

4.1.1 Java 6 Platform

The atomic WS is built on the Java SE 6 platform with the NetBeans IDE 5.5. Java SE 6 is the current major release of the Java SE platform, which has full support from NetBeans IDE 5.5.

The client sends 1000 requests in each experiment with constant interarrival time. The server processes requests, calculating the value of fibonacci (42). The interarrival time is slightly smaller than the server's service time; therefore, the server is slightly overloaded. The interarrival time is 1700 milliseconds, and the server's service time should be near 1900 milliseconds or more.

The experiment results are very similar to simulation ones (shown in Section 5.1.1, Figure 5.2), except of "noise" in real systems. The simulation results have almost constant interdeparture time; while the experiment results has fluctuating interdeparture time and "noise".

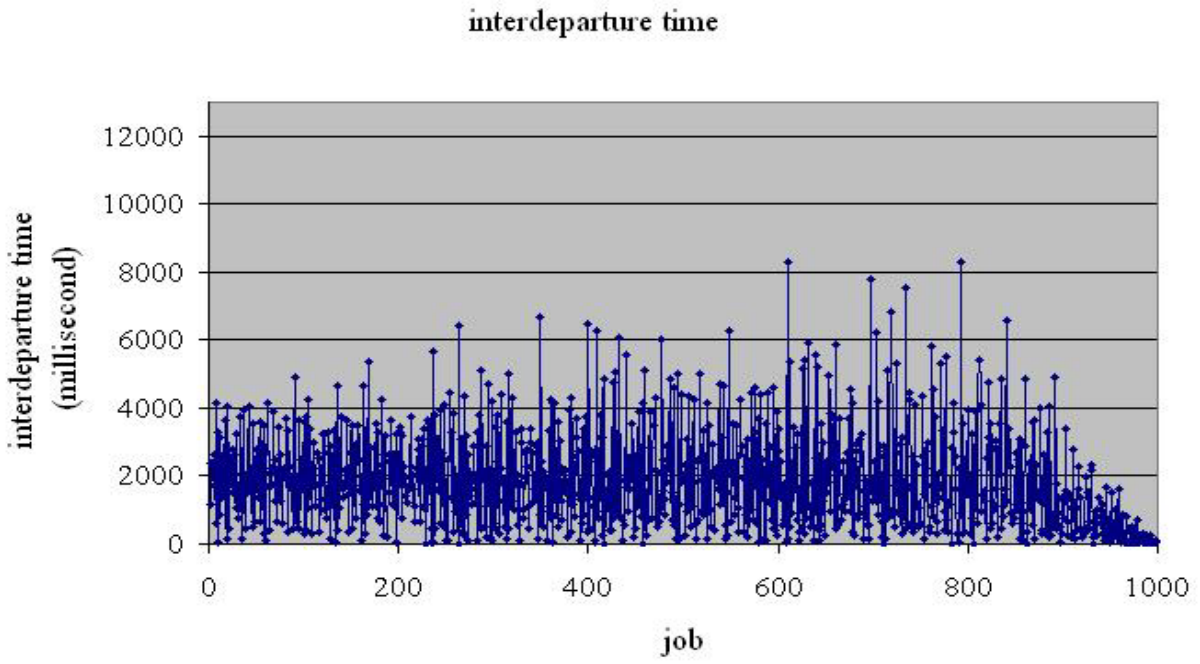

Figure 4.2: Interdeparture time

Moreover, the interdeparture time should be normal distributed, since its distribution fits normal distribution quite well. Its average value is almost constant in each small period, and the

value fits the simulation results. Its variation increases as the “noise” in the system increases, which can be shown in modified models (shown in Section 5.1.2).

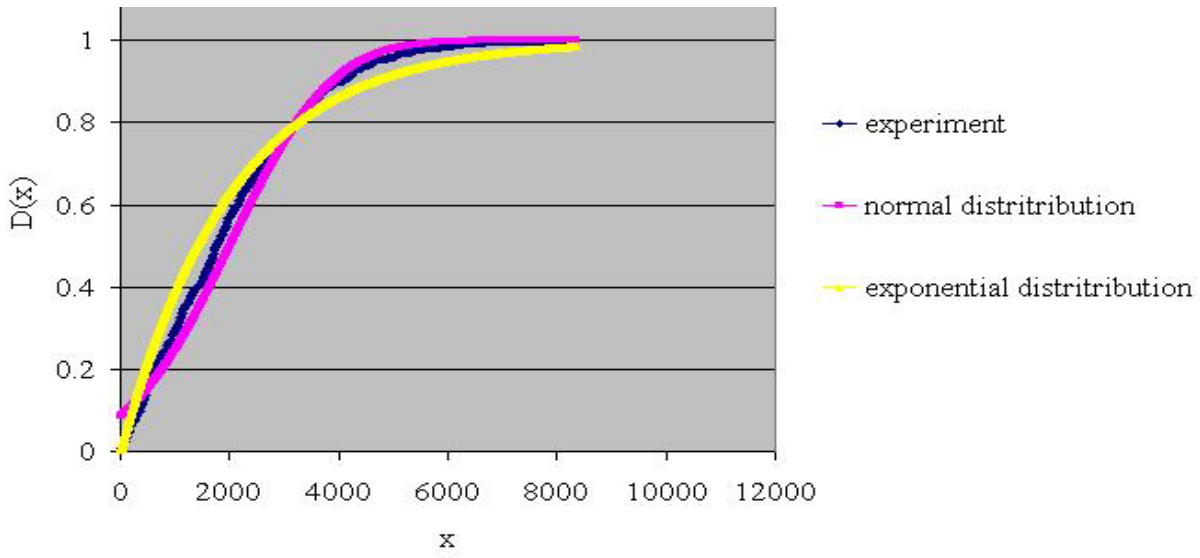

Figure 4.3: Distribution of interdeparture time

In Figure 4.3, the distribution function D (x) of the interdeparture time in experiment is compared to the distribution function of normal distribution and exponential distribution [31]. The distribution function D (x) [31] (also called the cumulative distribution function (CDF) or probability distribution function), describes the probability that a random variable x takes on a value less than or equal to a number x. The distribution function is sometimes also denoted F (x).

The distribution function of the interdeparture time observed in the experiment are shown in the curve, which is compared to the distribution function of normal and exponential distribution, in order to identify the distribution of the result. The x-axis represents the interdeparture time, and the y-axis is its distribution function D (x).

The result shows that the interdeparture time fits the normal distribution well. This conclusion also holds in other results with Axis 1 and Axis2, which can be simulated in modified models

that has simulated “noise” in experiment results (shown in Section 5.1.2).

4.1.2 Axis 1.4 Platform

The atomic WS is built on Apache Axis 1.4. The experiment’s setting and parameter are the same as the experiments with Java SE 6.

The interarrival time is 1700 milliseconds, and the server’s service time should be near 1900 milliseconds or more.

The result is shown in Figure 4.4.

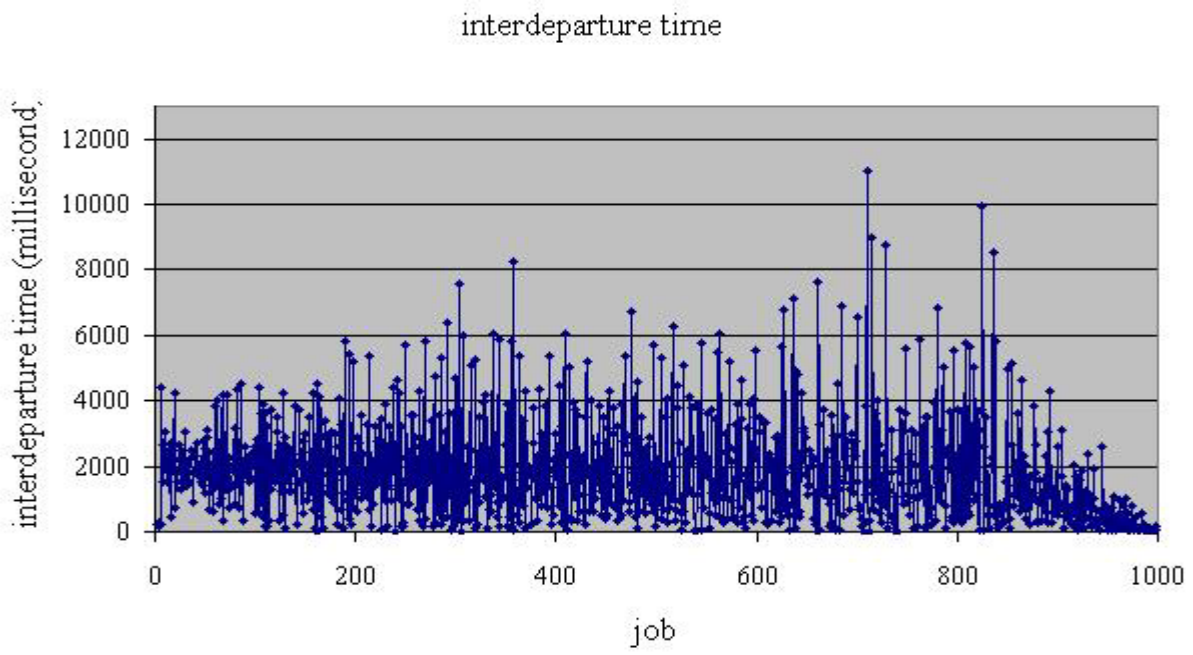

Figure 4.4: 1st Axis 1.4 experiment result

The same experiment is done repeatedly on the same computer. Each experiment was done after rebooting the computer, in order to make sure that the experiment environments are exactly the same. Figure 4.5 and 4.6 shows more results when repeated the same experiments.

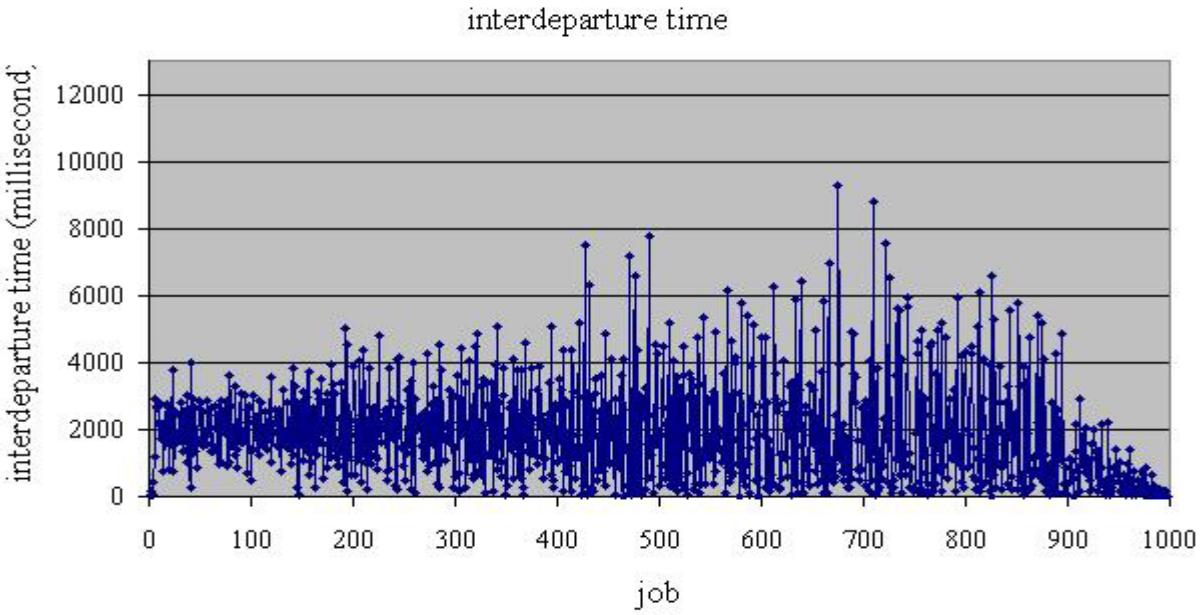

Figure 4.5: 2nd Axis 1.4 experiment result

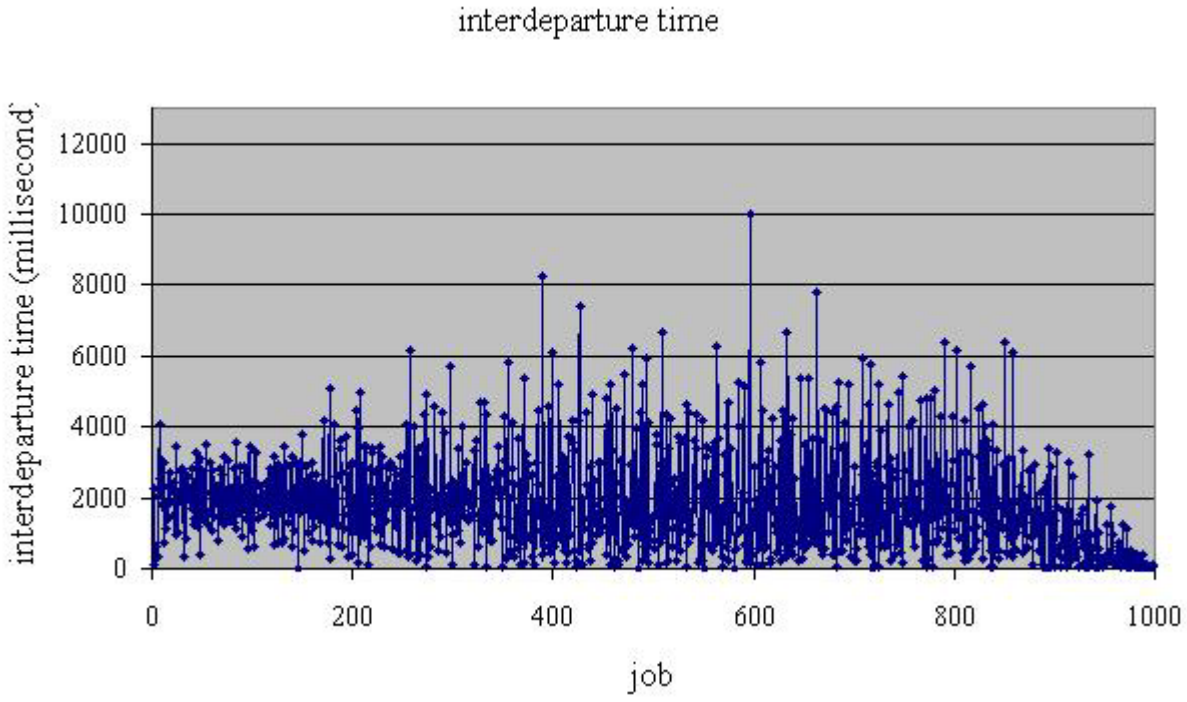

Figure 4.6: 3rd Axis 1.4 experiment result

The three experiments' results show similar distributions to normal distribution:

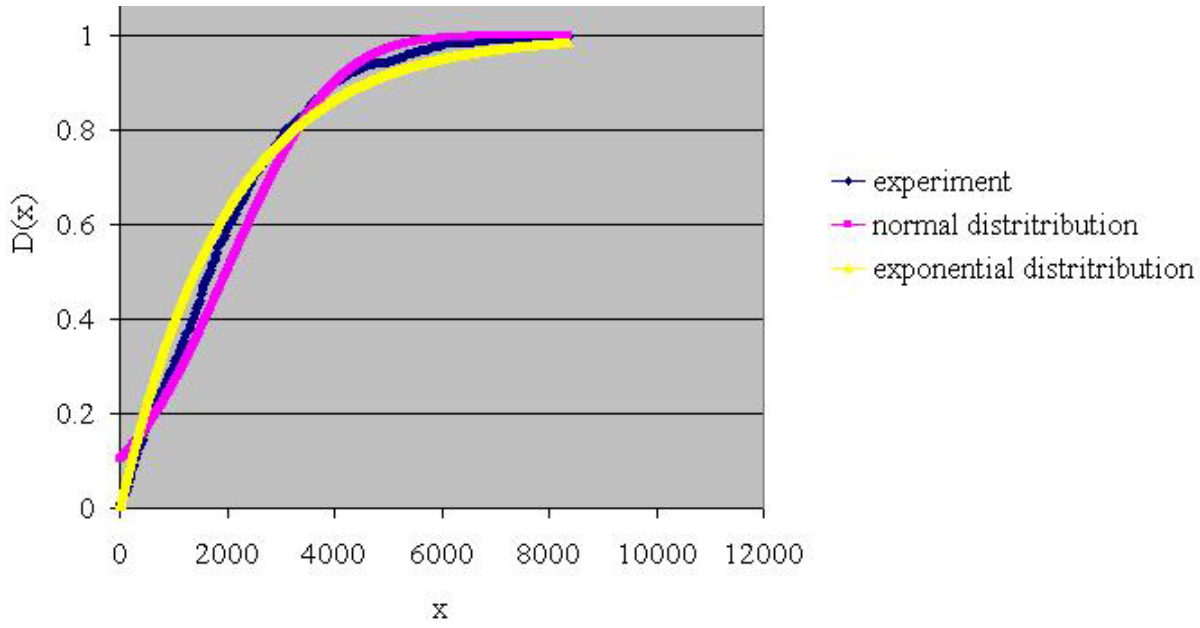

Figure 4.7: Distribution of 1st Axis 1.4 experiment result

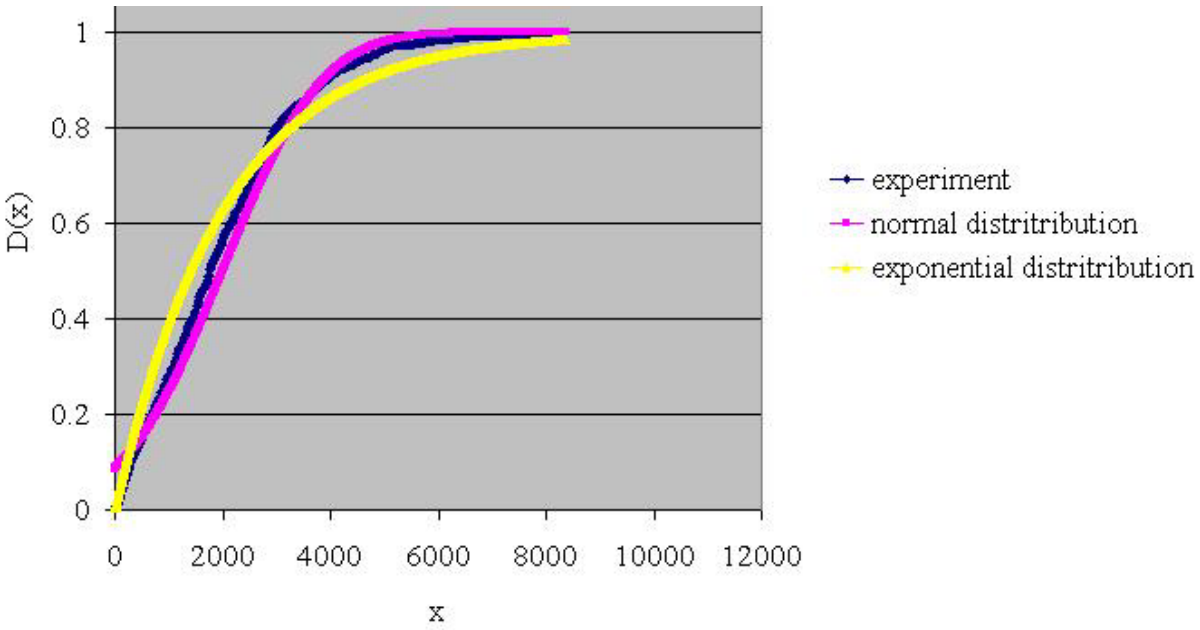

Figure 4.8: Distribution of 2nd Axis 1.4 experiment result

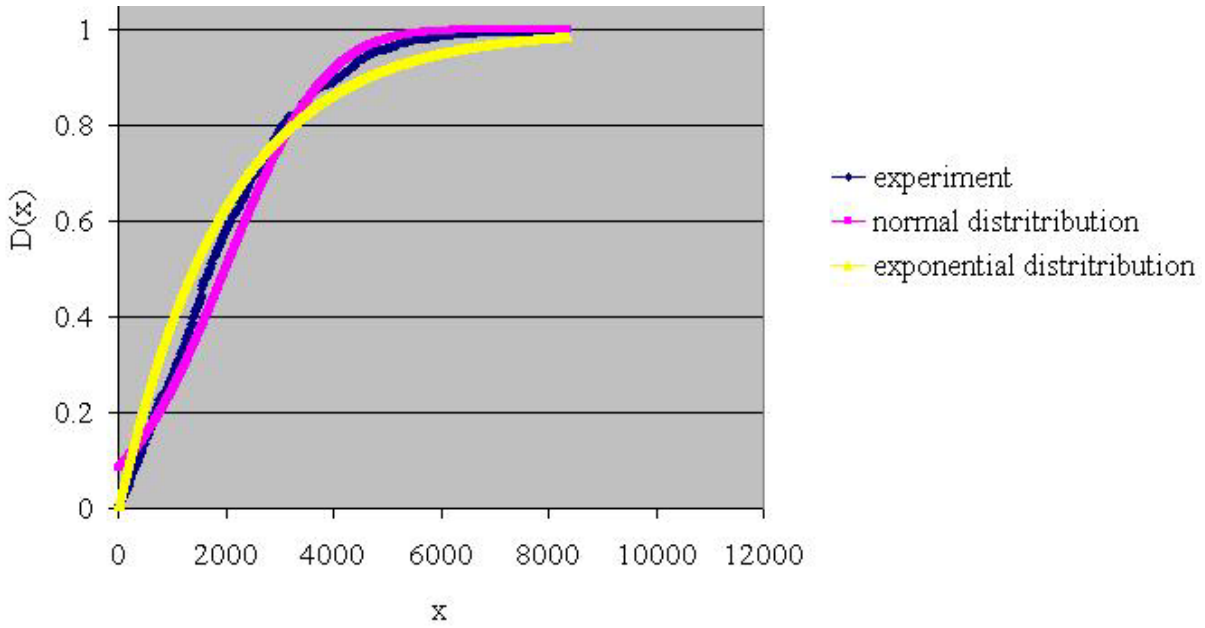

Figure 4.9: Distribution of 3rd Axis 1.4 experiment result

Notice that the incremental garbage collection is used instead of the default one. The default setting of garbage collection in Java is to do it periodically. We changed this way into incremental garbage collection, which reduces garbage collection pauses at the expense of throughput [23]. It reduced periodical noise and reduced about 10% of CPU capacity.

4.1.3 Axis 2 Platform

An atomic WS is built on Apache Axis 2/Java Version 1.2 [4]. The experiment is also using setup A, with a different interarrival time. When running experiments on Axis 2 platform, the loads that is used in the experiments on Java 6 and Axis 1.4 causes timeouts [25], which may indicate that the timeout limit in Axis 2 is stricter. Therefore, in the experiments on Axis 2, larger interarrival time is used, to form lighter load (but still overloaded). That means the interarrival time is just a little smaller than the server's service time. The interarrival time is 1840 milliseconds, and the server's service time should be near 1850 milliseconds or more.

The results are shown in Figure 4.10 and 4.11. They are slightly different from the results in Java 6, but still show the similar runtime behavior feature of overloaded WS. With lighter load, the "noise" shown in Axis 2 seems to be smaller than the results of experiments on Java 6 and Axis 1.4

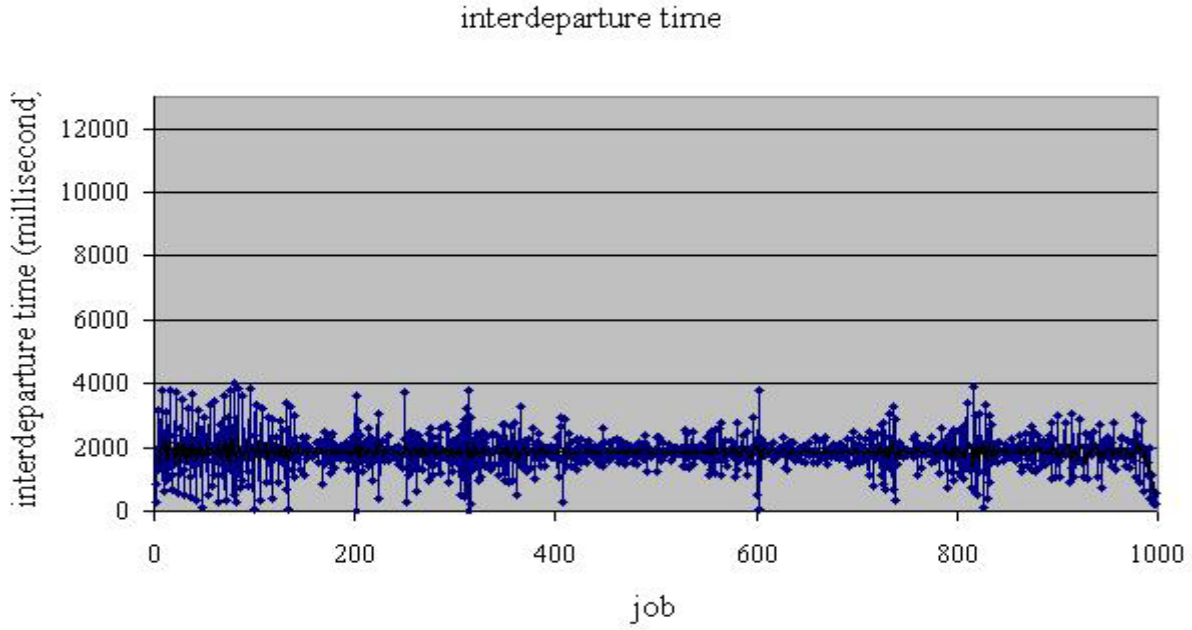

Figure 4.10: 1st Axis 2 experiment result

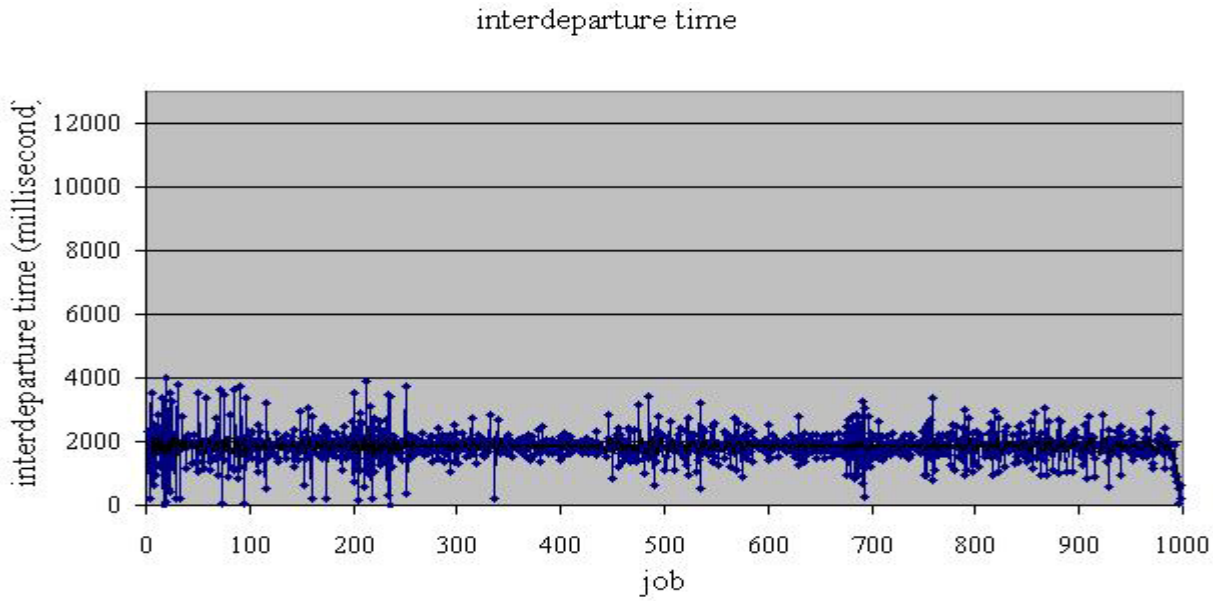

Figure 4.11: 2nd Axis 2 experiment result

Besides, the interdeparture time seems to be normal distributed, and its distribution function fits normal distribution function quite well; the following figures also show obviously that the results are not exponentially distributed:

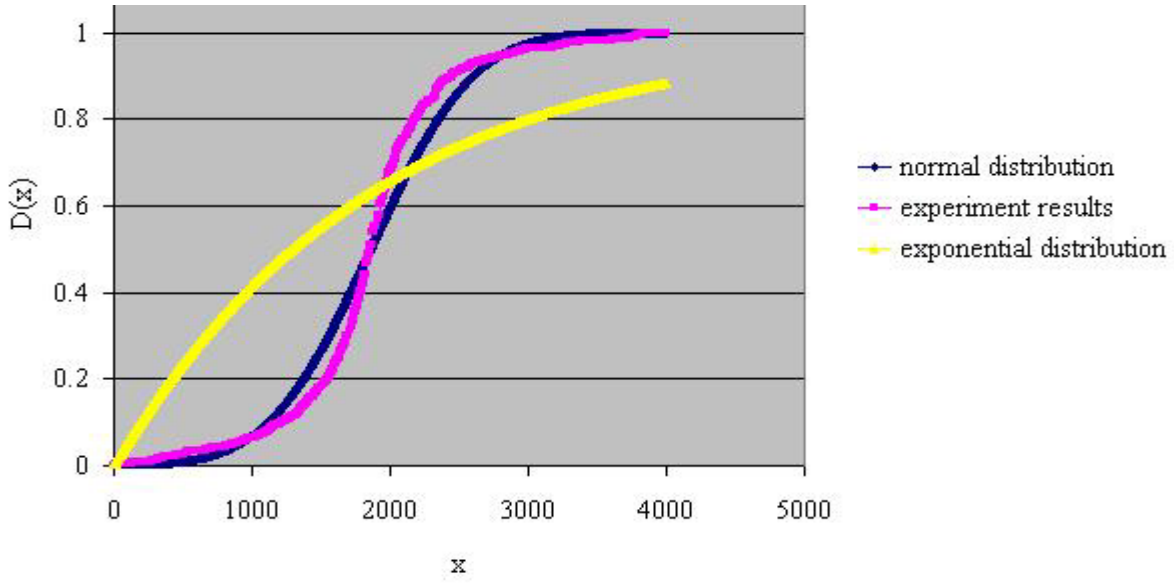

Figure 4.12: Distribution of 1st Axis 2 experiment result

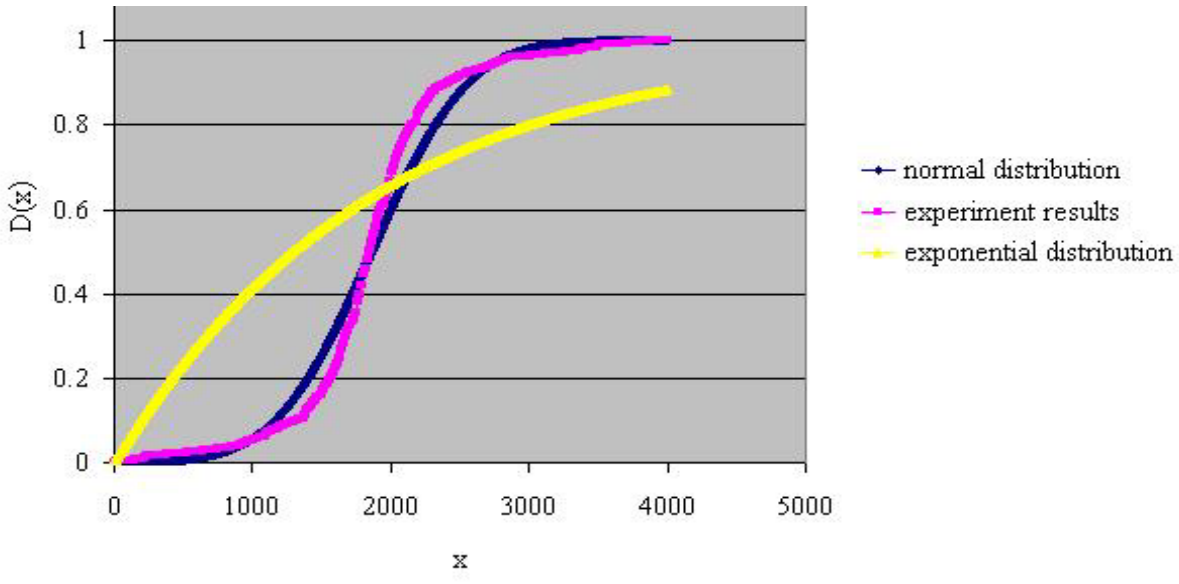

Figure 4.13: Distribution of 2nd Axis 2 experiment result

4.2 CWS's Experiments

4.2.1 Experiment B and Experiment C

4.2.1.1 Experiment B and experiment C on Axis 1

The behavior of CWS in a sequential workflow can be studied by implementing it. However using existing atomic WS and its result, we can also simulate a CWS.

In the previous experiments with atomic WS, the "departure time" was recorded in a file. Next, the file data is copied to Excel to calculate the "interdeparture time". The "interarrival time" is constant.

The "interdeparture time" is saved as "interarrival time" in a file, and the client code is changed to read the "interarrival time" from this file. In this way, the first WS's "interdeparture time" can be as the second WS's "interarrival time", and the second service's "interdeparture time" can be recorded. Thus, the first and second WSs simulate the behavior of CWS in a sequential workflow: the request from the client comes into the first service; when a request is processed by the first service, the first service sends requests to the second service. Such experiment setting of sequential workflow is noted as "setup B", with two sequential WSs and constant load (constant inter-arrival time). Experiment B uses setup B, and is done on the platform of Axis 1.

The sequential workflow is shown as below:

Figure 4.14: A sequential workflow

If using the second service's "interdeparture time" as the third service's "interarrival time", the third service's "interdeparture time" can be recorded, and a workflow with three services can be simulated with three atomic WS. Such experiment setting of sequential workflow is noted as "setup C", with three sequential WSs and constant load (constant inter-arrival time). Experiment C uses setup C, and is done on the platform of Axis 1.

The "interdeparture times" of the first, second, and third WS in a sequential workflow are recorded. Note that experiment B's results are only the "interdeparture times" of the first and second WS, which is included in the results of experiment C.

The experiment results of experiment C with three WSs in sequential workflow are as below:

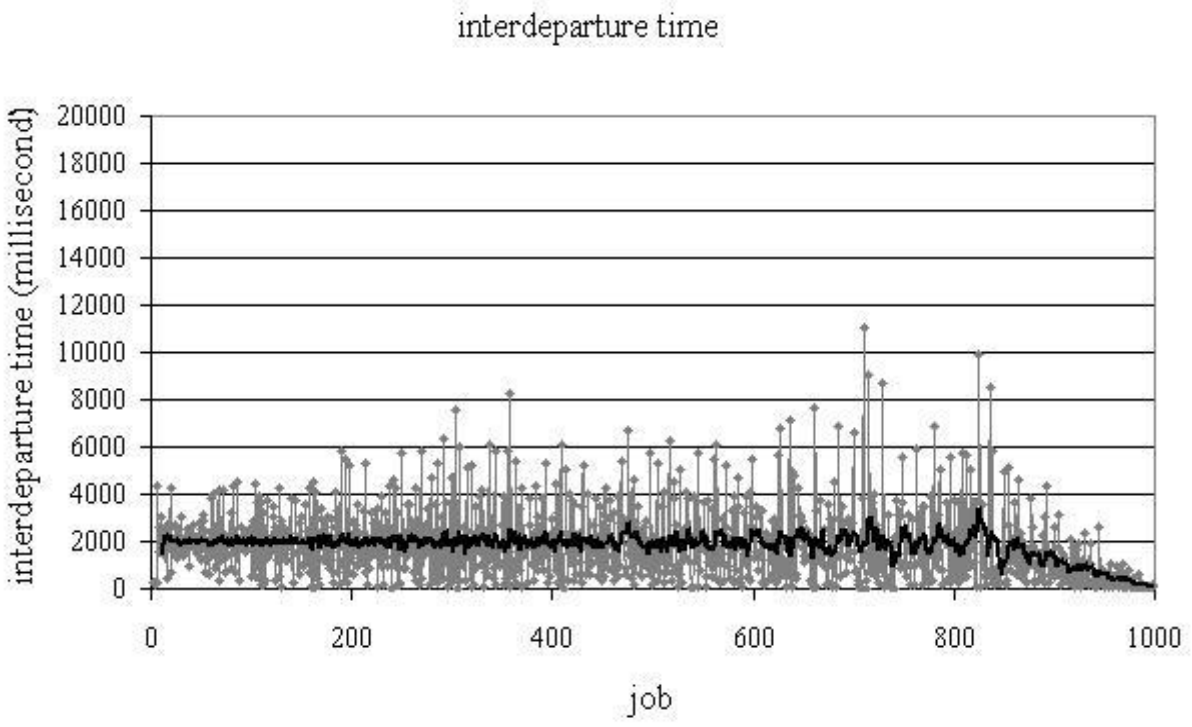

Figure 4.15: First WS's result in a sequential workflow

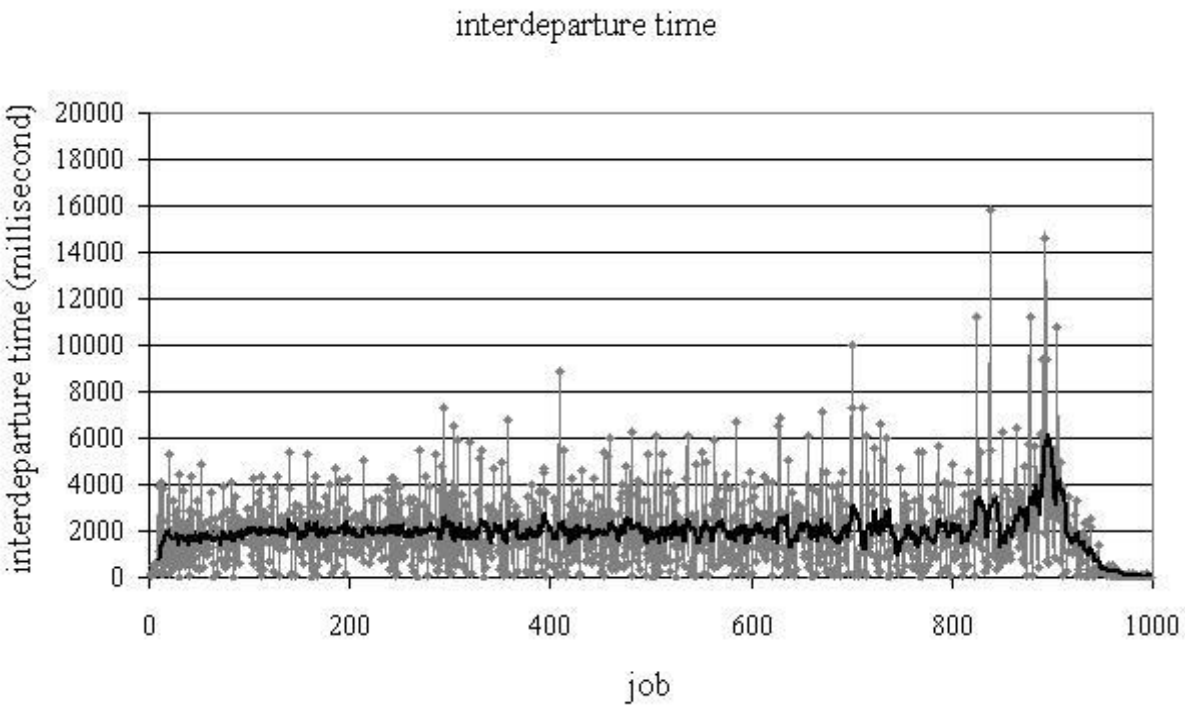

Figure 4.16: Second WS's result in a sequential workflow

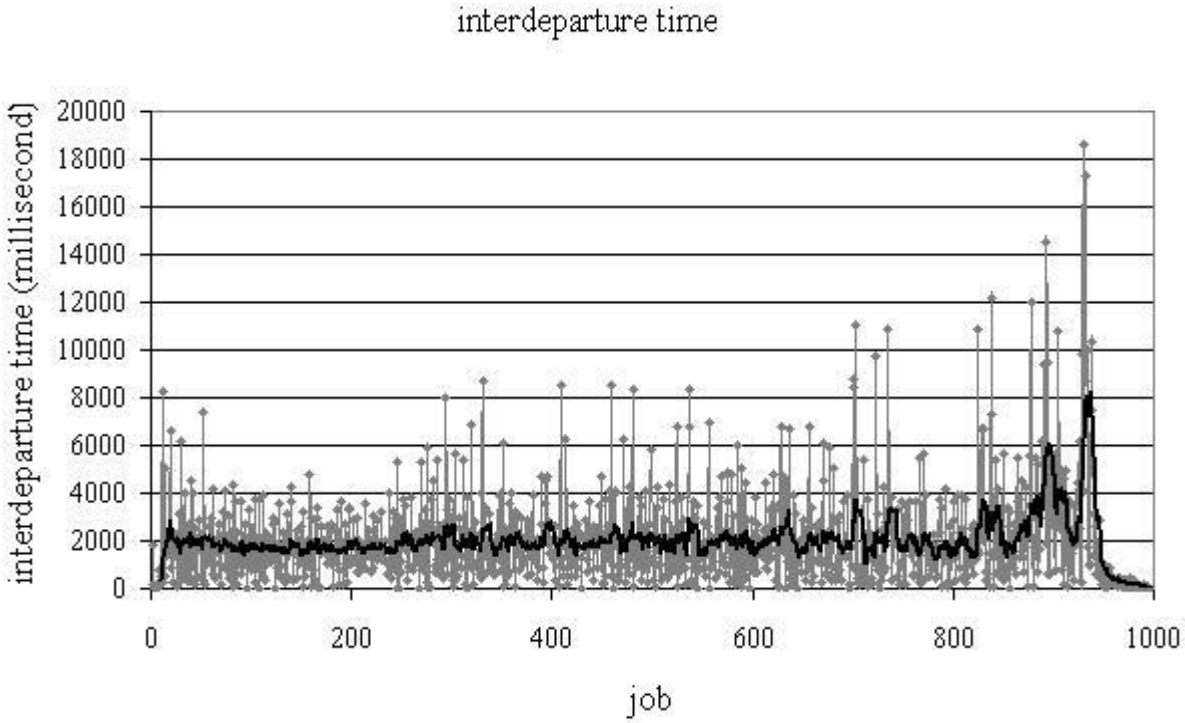

Figure 4.17: Third WS's result in a sequential workflow

The black curve is the trend line of the average value, which can show the features of results obviously, and can be easier compared to the simulation results.

The similar results can be seen in the simulation. The simulation has no “noise”, and can reflect the features obviously. See section 5.2.1.1 for more detail.

4.2.1.2 Experiment B on Java 6

To build more realistic CWS, a workflow server is added. Now the client does not send requests directly to the services; instead, the client sends the requests to the workflow server, and receives results from it, too. The workflow server receives the requests from the client, sends the request to each service in a certain order as defined in the workflow pattern, receives the results from each service, and sends them back to the client.

In the experiment B, there are two services in a sequential workflow. The two services are implemented separately on two computers. A workflow server is also used in the workflow, which receives requests from the client and sends messages to each service. The workflow server is implemented on the third computer. The sequential workflow with a workflow server works as following: The client sends request to the workflow server. The workflow server first sends the request to the first service, gets the result from the first service; then sends a request to the second service, and gets the result from the second service. Finally, the workflow server sends the result from the second service back to the client.

In this research, the experiment B is implemented on Java 6. Three computers are used: two computers are used for implementing the two services in a sequential workflow, noted as S1, S3; a client noted as C2 and a workflow server noted as S2 are both implemented on one computer, since S2 does not do time-consuming job. The S1 and S3 are the same WS, which calculate

Fibonacci numbers “fib (n)”, and return the finish time. Here, the S2 just receives the parameter “n” from C2, sends it to S1 and S3 sequentially, and sends the result from S3 back to C2. The implemented sequential workflow is as following:

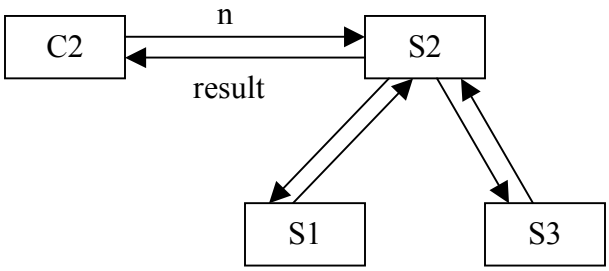

Figure 4.18: The sequential workflow of two services with a workflow server

The configuration of the computer that serves as server (S1) is as follows:

Intel Pentium III processor, 803MHz, 1.0 GB of RAM, Windows XP Professional.

The configuration of the computer that serves as server (S2) and the client (C2) is as follows:

HP workstation xw6200, Intel(R) Xeon (TM) CPU 3.20 GHz, 2.0 GB of RAM, Windows XP Professional.

The configuration of the computer that serves as server (S3) is as follows:

Intel Pentium III processor, 803MHz, 1.0 GB of RAM, Windows XP Professional.

In the experiment, the finish time (departure time of jobs) of S1 and S3 are recorded. The interdeparture times of S1 and S3 are calculated from the results, which shows the run time behavior of the CWS.

There are 1000 requests in the experiments. Fib (42) and fib (44) are used in different experiments. When calculating fib (42), and using 1700 milliseconds as the interarrival time, the result (result 1) of S1 w is as follows:

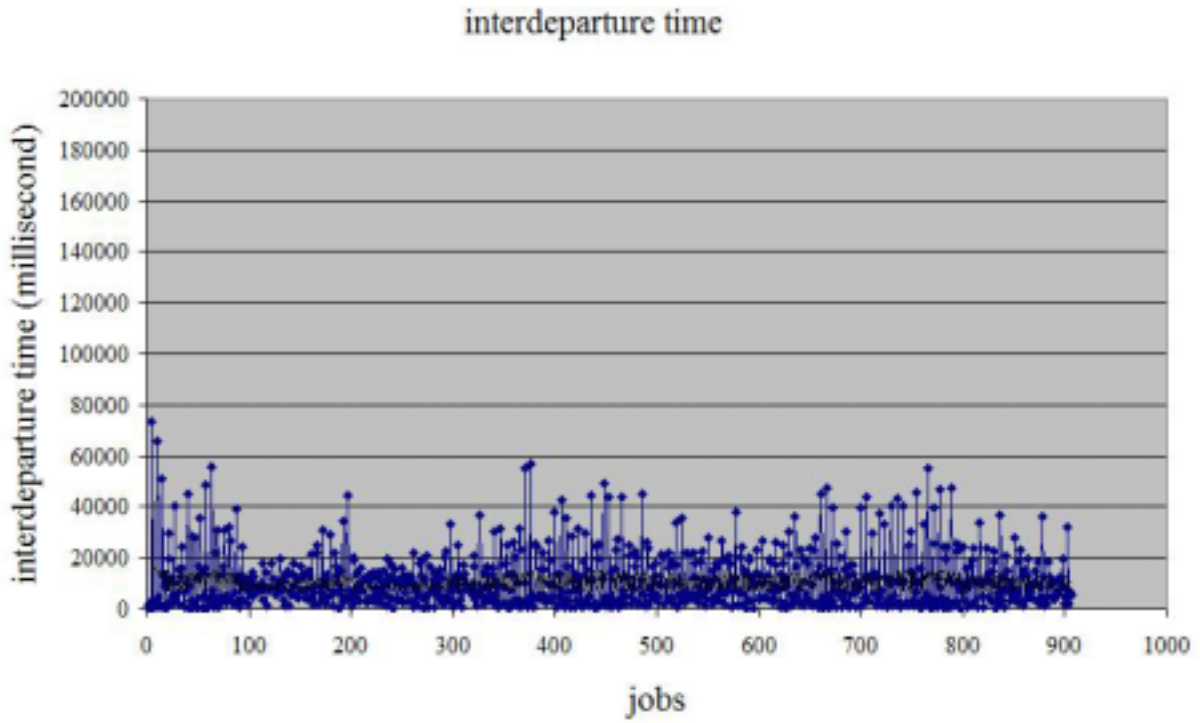

Figure 4.19: S1's result (result 1) with fib (42) and using 1700 milliseconds as the interarrival time

In this experiment, timeouts happened, and only 906 results of interdeparture times were recorded (although 1000 requests were sent in total). The average value of interdeparture times, the average value from the 100^{th} to 800^{th} results, is 10573 milliseconds, which is much larger than the interarrival time (1700 milliseconds).

For calculating fib(42), different interarrival times are tried in different experiments. Here are the results of S1 when using different interarrival times and fib(n):

Table 4.2: Results of S1 when using different interarrival times and fib(n)

Result number	Fib(n)	Interarrival time	Average interdeparture time (100^{th} to 800^{th})	Number of records
Result 1	42	1700	10573	906
Result 2	42	1500	10302	950
Result 3	42	1400	10232	929
Result 4	42	1200	10551	913
Result 5	42	800	10208	989
Result 6	42	400	10383	971
Result 7	42	200	10280	960
Result 8	42	100	10811 (100^{th} to 700^{th})	788
Result 9	42	2000	10601	988
Result 10	42	3000	10089	995
Result 11	44	8000	27136	999
Result 12	44	4000	25789	1001
Result 13	44	2000	25917	1000
Result 14	42	2000	10482	980
Result 15 (repeat experiment 1 after restarting)	42	1700	10966	966

Experiment 1 is done after a few other experiments without restarting the computer, while experiment 15 repeats experiment 1 after restarting. Their results are similar, although the average value of interdeparture time in result 15 is a little larger, and the number of successfully processed jobs in result 15 is a little larger.

Result 12 and 13 may include a couple of requests that are being resent, which makes the number of recorded results a little larger than 999 (999 is the maximum value of the number of interdeparture times recorded, when 1000 requests are sent).

The average interdeparture times are similar when calculating the same fib(n) (n is 42 or 44). The results do not change with the interarrival time obviously.

For fib(42), the average interdeparture times are almost 10200 to 10600 milliseconds, although the interarrival times change from 100 to 3000 milliseconds. Also, the recorded

numbers of results are almost above 900 (only when the interarrival times is 100 milliseconds, the number is 788).

For fib(44), the average interdeparture times are almost 25000 to 27000 milliseconds, although the interarrival times change from 2000 to 8000 milliseconds. Also, the recorded numbers of results are above 900, near to 1000.

In summary, the experiments are run with different interarrival times. The services are overloaded. A few of the requests fail to be processed in most of experiments. The errors exist because timeouts happens. When the interarrival times are larger, the load of the service is lighter. A feature of the results when the workflow server is added in the workflow is: the number of errors (timeouts) does not change with the interarrival times (the load) obviously; instead, the interdeparture times are similar in all the results with the same fib(n). Another feature is that, the results of interdeparture times did not show the decreasing period as in previous results when overloaded.

Also, in the results of this section, the interdeparture times are very large, much larger than in previous results. In previous experiments of atomic WS on both Axis 1 and Java 6 (see Section 4.1.1 and 4.1.2), fib (42) is used, the interarrival time is 1700 milliseconds, and the server's service time should be near 1900 milliseconds or more. Therefore, the service is only slightly overloaded, and the interdeparture time is about 2000 milliseconds in the stable period before it begins to decrease. However, the interdeparture time of fib (42) in this section is much larger than 2000 milliseconds. This is because the service time is larger, and the load of the service is heavier than before. Although the services still calculate fib (42), they are implemented on slower computers, which makes the service time of them larger than before (when implemented on faster computers before, the service time is about 1900 milliseconds). For fib(42), the average

interdeparture times in the results of this section are almost 10200 to 10600 milliseconds, the service time can be concluded as about 5000 to 8000 milliseconds. Comparing the possible service time to the interarrival time used, the load of service in this section's previous experiments is quite heavy. For the previous experiments of CWS without a workflow server, when the load is quite heavy, lots of timeouts happen, and only a few results can be recorded; while in this section with a workflow server added to the CWS, when the load is heavy, more than 900 results are recorded out of 1000 expected results, the timeouts happens but it is not a very serious problem. The conclusion is that, when we add a workflow server to the CWS, the number of timeouts is small and similar when the services are slightly overloaded or under a quite heavy load. A workflow server can help prevent too many timeouts when the load is heavy.

To better study the behavior of CWS with a workflow server, more experiments are done, that calculates fib (39) instead of fib (42). Calculating fib (39) is less time-consuming than calculating fib (42), which makes the service time relatively small even when the services are implemented on slower computers. The services' load is lighter now, but it is still slightly overloaded. Each experiment is run after restarting the computer and the services. When the interarrival time is 1200 milliseconds, 946 results are recorded, and the average value of interdeparture time (from the 1^{st} to 800^{th} results) is 2560 milliseconds. When the interarrival time is 900 milliseconds, 999 results are recorded, and the average value of interdeparture time (from the 1^{st} to 800^{th} results) is 2525 milliseconds. However, when the interarrival time is 600 milliseconds, the load is so heavy that lots of timeouts happen, and only 57 results are recorded (the average of the 57 results is 2737 milliseconds). Therefore, it is concluded that, when interarrival time (the load) changes in a certain range, the number of errors (timeouts) does not change with it obviously. The results also show no obvious decreasing of the curve of

interdeparture time at the end of the experiment. In summary, the new experiments show similar results to the results with heavier load, and show the same feature when CWS has a workflow server.

4.2.2 Experiment D and E

For atomic WSs or WSs in sequential workflow, the workflows load can be constant or non-constant. When the load is non-constant, it can be exponentially distributed, "burst", and so on. Such experiment settings are noted as setup D. Experiment D uses setup D, and can be implemented on the platform of Axis 1, or other platforms.

Besides sequential workflows, there are CWS in other kinds of workflows, such as workflows that have loops, switches, and so on. Such experiment settings are noted as setup E. Experiment E uses setup E, and can be implemented on the platform of Axis 1, or other platforms. In this research, experiment E is only implemented in simulation models with AnyLogic, and the simulation experiments results will be shown in Chapter 5.

4.3 Experiment D on Axis 1.4 Platform

In the previous experiments, the job requests are sent with constant interarrival time. For example, a job request is sent in every 1700 milliseconds. 1700 milliseconds is the interarrival time used in the experiment, which is constant. The constant interarrival time are also used in the simulation model in AnyLogic.

However, in real systems, the job requests rate may not be constant. The load can be exponentially distributed, or following other distributions. The load can also be a kind of "burst". For example, burst may have a period of 10 seconds: in the first 5 seconds, the job requests have a constant rate of 1 job per second; and in the last 5 seconds, there is no job request sent.

Experiment setting with non-constant load for atomic WS or CWS in sequential workflow is noted as “setup D”. Experiment D uses setup D, and can be implemented on the platform of Axis 1, or other platforms. (Constant and non-constant load for CWS in workflow of loops or switches are more complex, which is noted as “setup E”, and studied with simulation experiments.)

We did experiment with exponentially distributed interarrival time on the platform of Axis 1.4. The atomic WS that is used in experiment A is used here. An experiment computer is used as server, which receives and processes requests; and a lab computer is used as client, which sends requests.

The configuration of the computer that serves as server is as follows:

Intel Pentium III processor, 803MHz, 1.0 GB of RAM, Windows XP Professional.

The configuration of the computer that serves as the client is as follows:

HP workstation xw6200, Intel(R) Xeon (TM) CPU 3.20 GHz, 2.0 GB of RAM, Windows XP Professional.

The WS calculates Fibonacci numbers [35]. In this experiments, the client requests for calculating fibonacci (39). The interarrival time is exponential (1.7) seconds (the average value of interarrival time is 1.7 seconds, and it is exponentially distributed). The server is slightly overloaded with such parameters. The server’s service time should be a little more than 1.9 seconds.

Default setting of Axis 1.4 are used in the server, so the maximum number of threads is 150. Totally 1000 requests are sent by the client.

The result is: no timeouts happens since the server is only slightly overloaded. The interdeparture times are exponentially distributed, and their average value is larger than the interarrival time since overloaded. The interdeparture times fit exponential distribution quite well,

which is the same as in simulation results; while “noise” is shown when the interarrival time is constant, and the interdeparture times are normally distributed, instead of constant as in simulation results.

The result of interdeparture times that fit exponential distribution are shown as following:

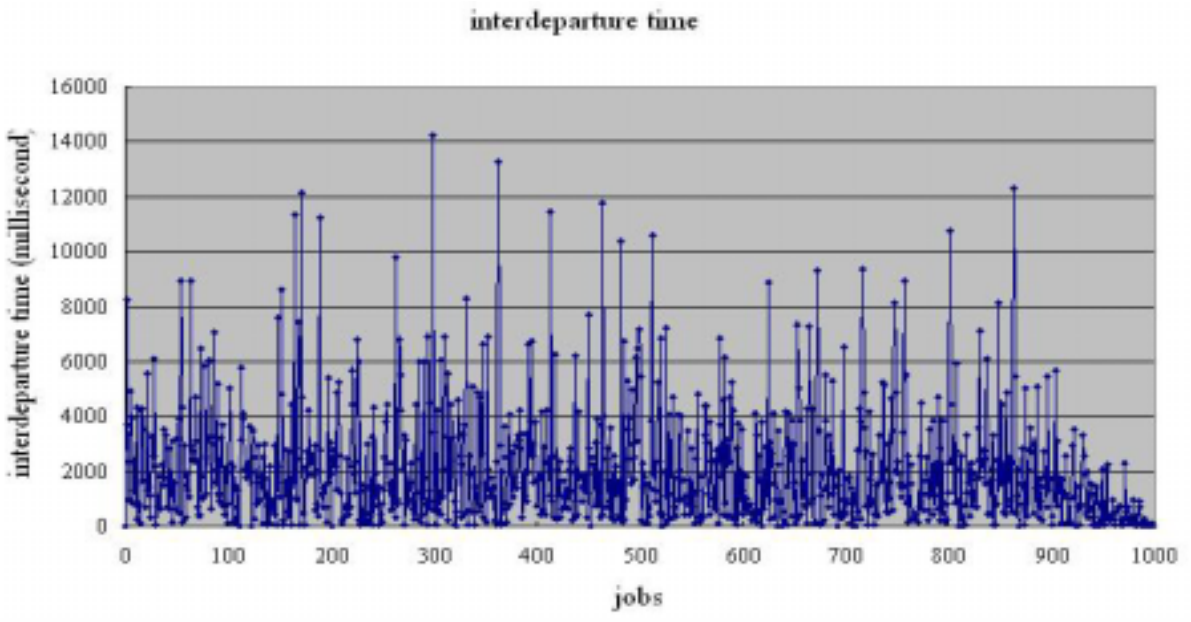

Figure 4.20: Axis 1.4 experiment result with exponentially distributed interarrival times

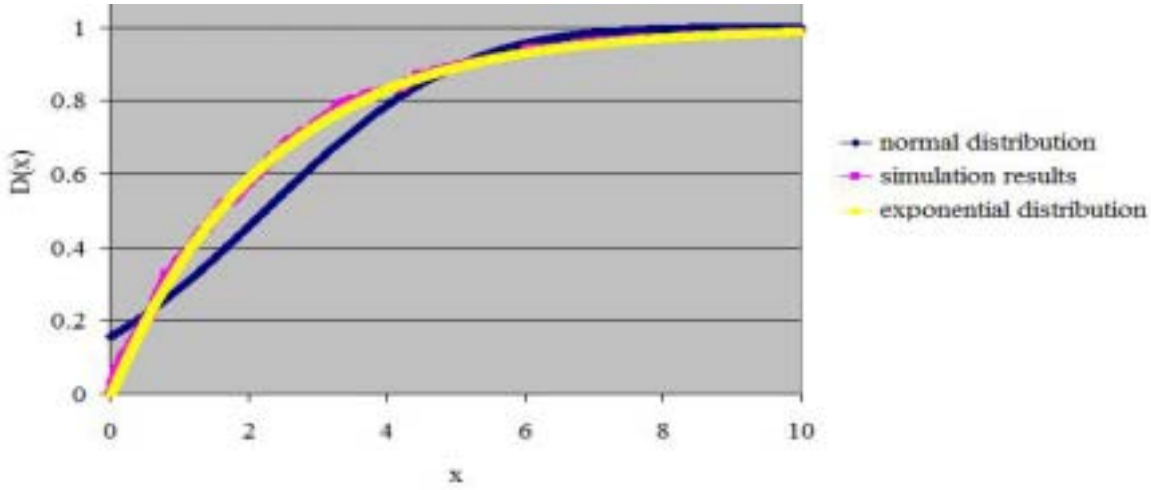

Figure 4.21: Distribution of Axis 1.4 experiment result with exponentially distributed interarrival times

CHAPTER 5

SIMULATION MODEL

In Chapter 4, basic behaviors of WSs were discussed. In order to examine more complex behaviors of WSs, simulation models are used.

Experiment A, B, C, D, E were discussed in Chapter 4. In this chapter, some of those experiments are modeled as basic model (which has no "noise") and modified model (which simulated the "noise" in real system). The following table is a summary of the simulation experiments in this chapter:

Table 5.1: The summary of the simulation experiments

Experiment	If has basic model?	If has modified model?
Experiment A	Yes	Yes
Experiment B	Yes	No
Experiment C	Yes	No
Experiment D	Yes, of atomic WS with exponentially distributed interarrival time	Not needed, simulated results fit experiment results well
Experiment E	Yes, of CWS in workflow that has loops	No

The simulation models of WSs are built in AnyLogic [44], which is software to build simulation models and run simulation experiments.

5.1 Model for Experiment A

5.1.1 Basic Model

In this research, the simulation models of atomic WSs were developed in AnyLogic [44].

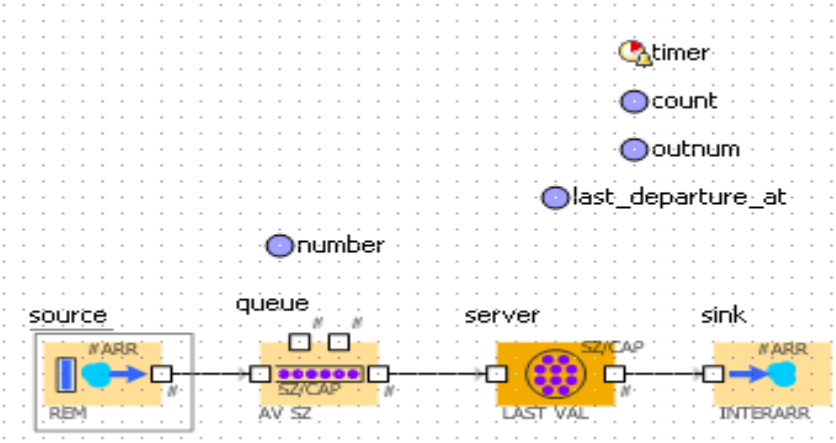

Figure 5.1: AnyLogic model of atomic WS

In the model, the queuing network model is used, and the atomic WS is modeled as a server. The server has a constant service time. A client is used to send requests to the server.

In the model, the client sends 1000 requests in each experiment run with a constant interarrival time. The maximum number of threads in the server is 1000. The CPU time is distributed evenly among all the threads; that is, every thread gets the same share of CPU time.

The interarrival time is slightly smaller than the server's service time, so that the server is slightly overloaded. Here, the interarrival time is 1 second, and the service time is 1.2 seconds.

The simulation experiments run in AnyLogic. The model and its simulation results do not exhibit "noise" found in real systems. Therefore, the simulation results show only the basic behavior of an overloaded WS.

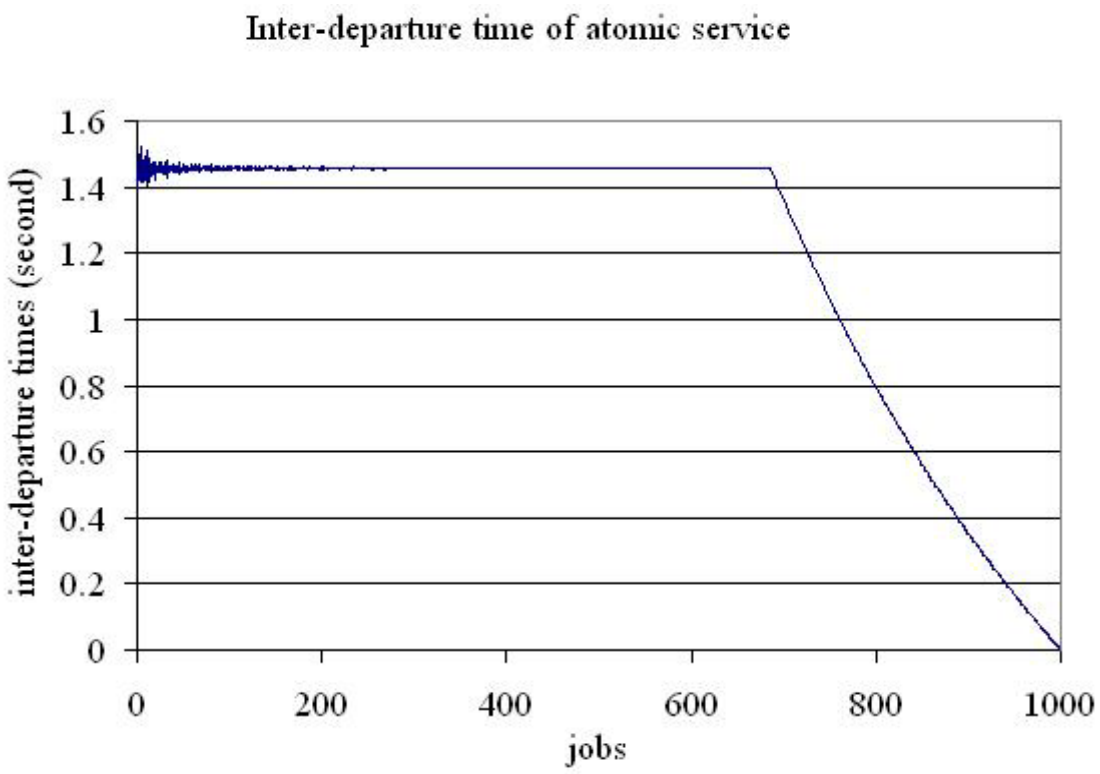

Figure 5.2: Simulated interdeparture time of atomic WS

The interdeparture time fluctuates for a very short period at the beginning of the simulation experiment, which is called a “warm-up” period. Then, the interdeparture time is almost constant for a long period in the simulation experiment, which is called a “stable” period. At the end, the interdeparture time suddenly decreases after a single point, which is called a “decreasing” period.

The interdeparture time suddenly decreases after a single point, because after that point, there is no new request (job) coming into the server. The remaining jobs in the server can enjoy more and more processing resource, and be done faster.

The “stable” period and following “decreasing” period, show important behavior of overloaded WS.

Throughput is another way to show the behavior of a WS:

The throughput can be noted as X: X =C/T. T is the length of time observed. C is the number of request completions observed. For example, if 8 completions are observed during an observation interval of 4 minutes, then the throughput is 8/4 = 2 requests/minute. In the AnyLogic model, a timer is set to get the throughput. The timer has a time interval of 30 seconds, which is used as a unit of length of time for observing the system. Therefore, T is 30 seconds. The throughput shown in the results is the number of request completions observed in every 30 seconds, the throughput can be noted as X: X =C/T=C/30 seconds.

Figure 5.3 shows the throughput of the simulation result. In this figure, the x-axis is the "count times", that is, the number of time intervals. The timer counts the number of request completions in every 30 seconds, and the experiment lasts for about 60*30 senconds=1800 seconds in total, so the maximum "count times" is 60. The y-axis is the "throughput", noted as X: X =C/T.

Figure 5.3 shows that, at the beginning of the experiment, the throughput is almost stable, which shows the same feature as the "interdeparture time" figure does.

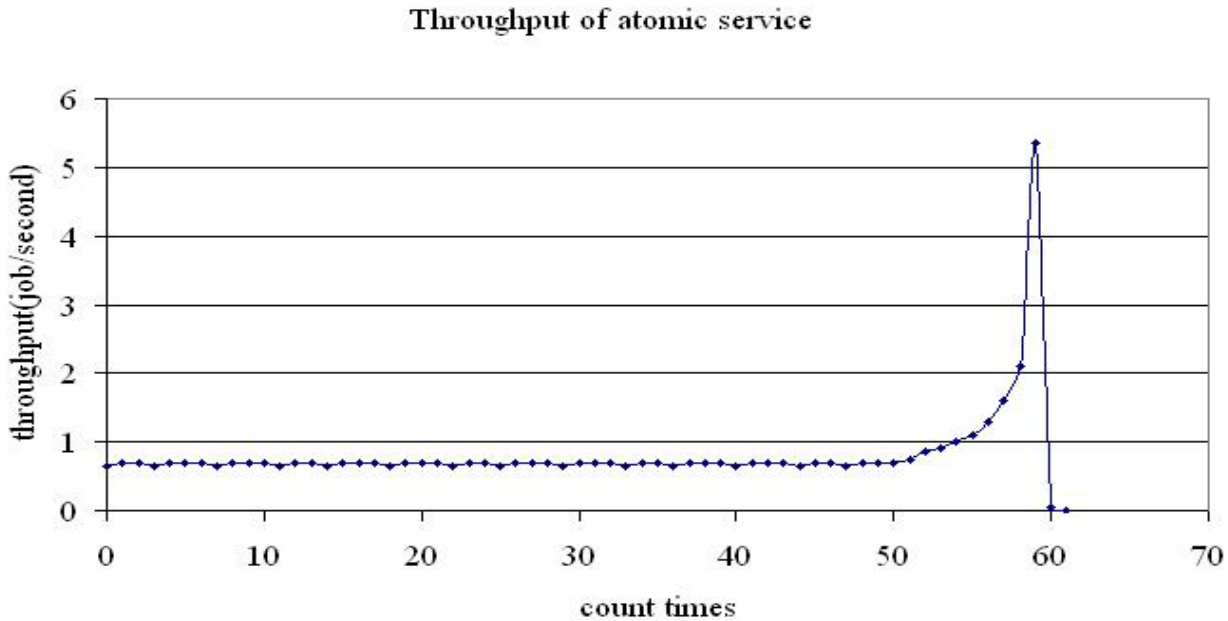

Figure 5.3: Simulated throughput of atomic WS

It can be seen that at the end of the experiment, the throughput becomes higher, which means the jobs are completed fast at the end. This is the same as what is shown in the "interdeparture" time figure.

The simulation experiment is also done under different loads. In the following results, I compared the behavior of two services; the loads of the servers are 120% and 140%. The 120% load server has interarrival time as 1 second and service time as 1.2 seconds, and the 140% load server has interarrival time as 1 second and service time as 1.4 seconds:

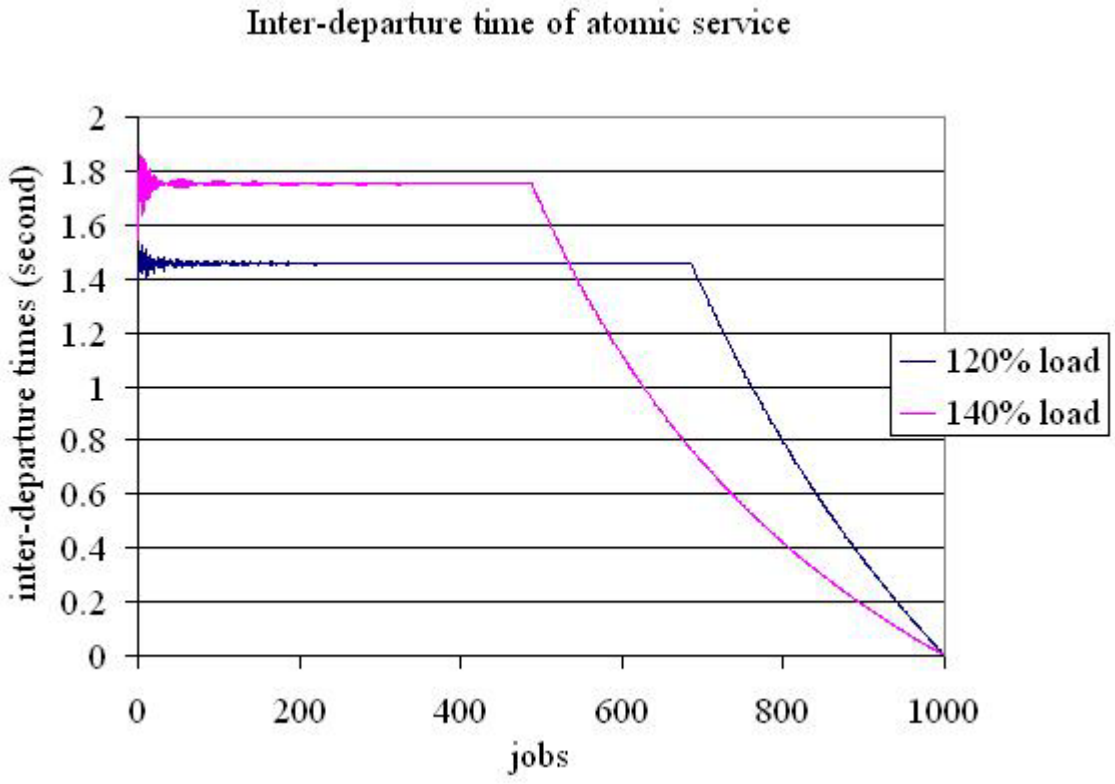

Figure 5.4: Simulated interdeparture time of atomic WS under different loads

The results still have the fluctuating warm-up period, stable period, and fast departure period. Under different loads, the length of stable and the fast departure period is different. Under

heavier loads, in the stable period, the inter-departure time is larger with larger service time. Dyachuk et al. also represented similar simulation results [14].

5.1.2 Modified Model

It is known from the experiment results (in Chapter 4) of the real system that, the system has "noise". Therefore, "noise" is added in the basic model. The interdeparture time in real system experiment results is normal distributed, and the modified model has results that are very similar to experiment results. This model also simulates the system setup of experiment A.

The "noise" in experiment results can be simulated in a modified model. Again the interdeparture time is normally distributed. The average value of interdeparture time is almost constant in each small period, and the value fits the simulation results with the basic model.

Moreover, it is known from simulation results that, the variation of interdeparture time increases as the "noise" in the system increases.

The interarrival time in the model is 1.7 seconds, and the service time is 1.8 seconds. The uniform distributed noise is added into the server, every 2 seconds, and the server's performance is decreased by some "noise".

The function is: every 2 seconds, the server's performance is changed to (1 + uniform(-noise,0)). For example, when the parameter of "noise" is set as 0.04, the "uniform(-noise,0)" is a random variant that is uniform distributed with an average value of (-0.04/2). Every 2 seconds, the "uniform(-noise,0)" will generate a random value, such as –0.017, and the server's performance is changed to (1+(-0.017)), that is 98.3%. This kind of "noise" is used to simulate the real systems; for example, the real systems may execute other jobs in the background, with only 98.3% of the resources processing the WS jobs; at other moment, the number may change.

There are two groups of results; the first one has smaller "noise" than the second one. For the first one, the noise is 0.035; for the second one, the noise is 0.04. The simulation results with modified model are shown as below:

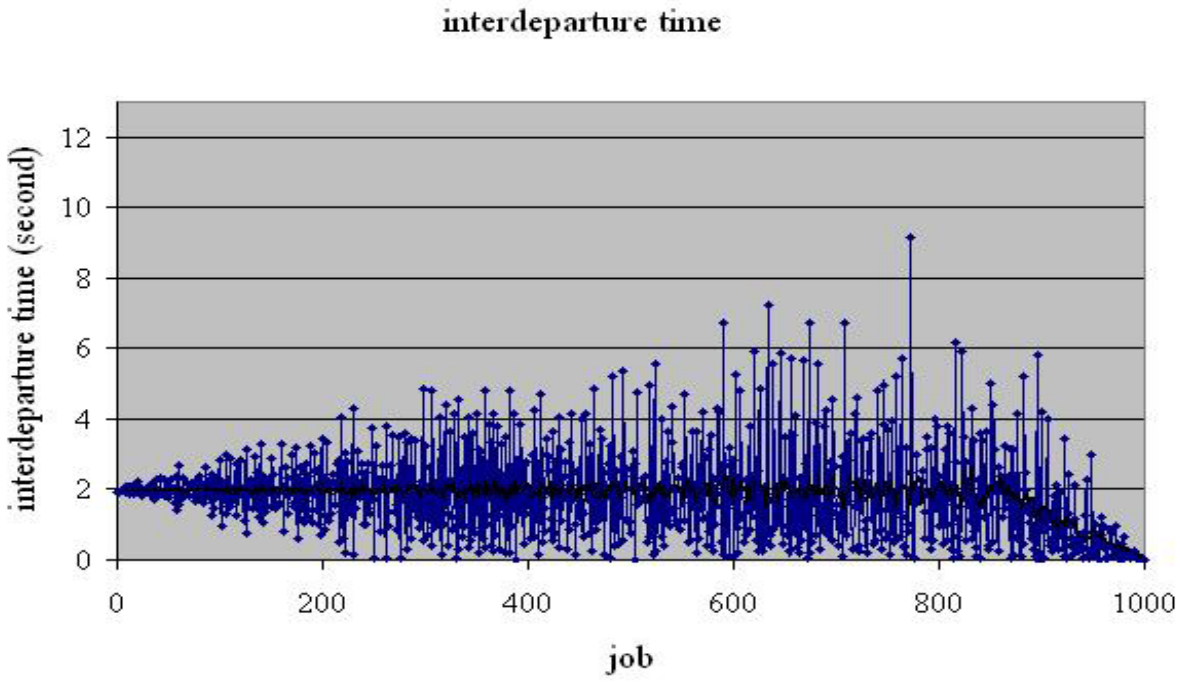

Figure 5.5: Simulated result 1 of atomic WS when noise is 0.035

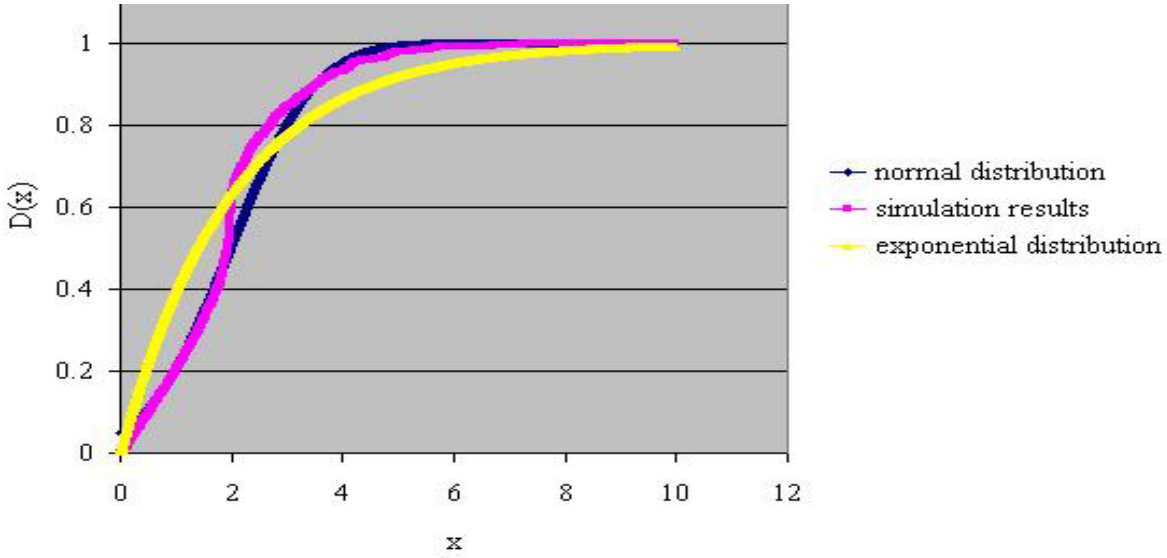

Figure 5.6: Distribution of simulated result 1 of atomic WS when noise is 0.035

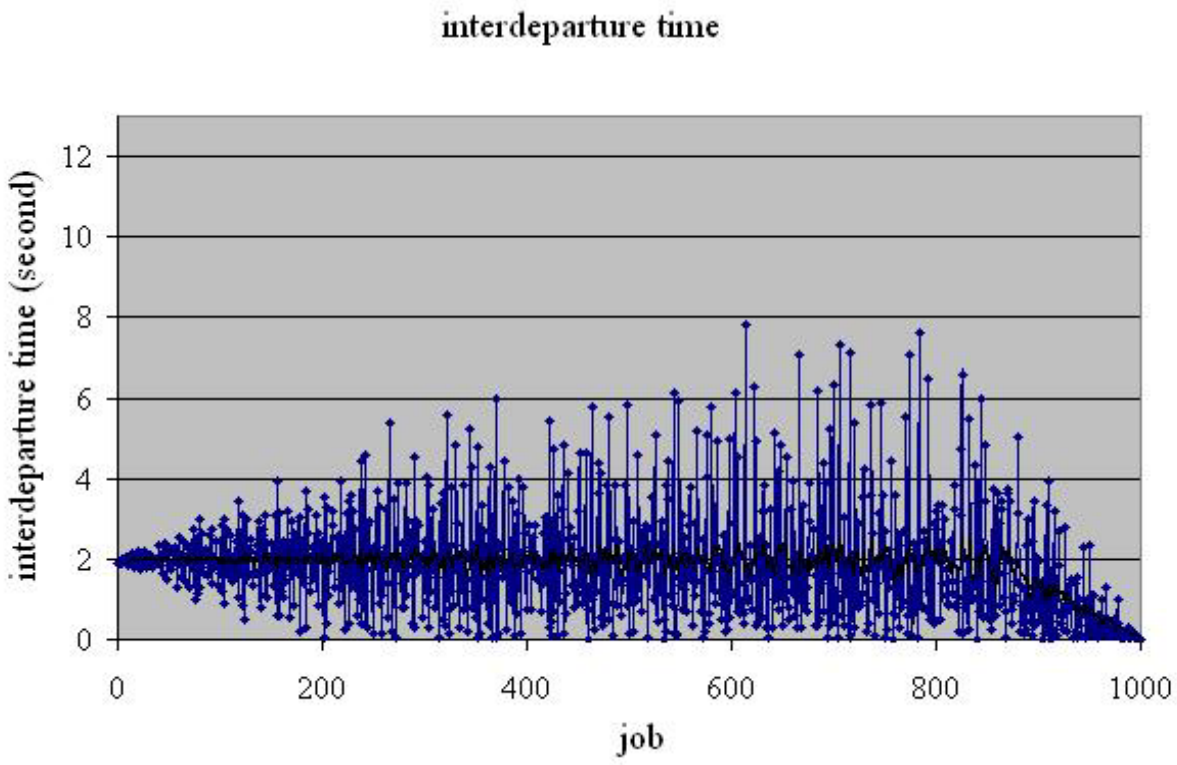

Figure 5.7: Simulated result 1 of atomic WS when noise is 0.04

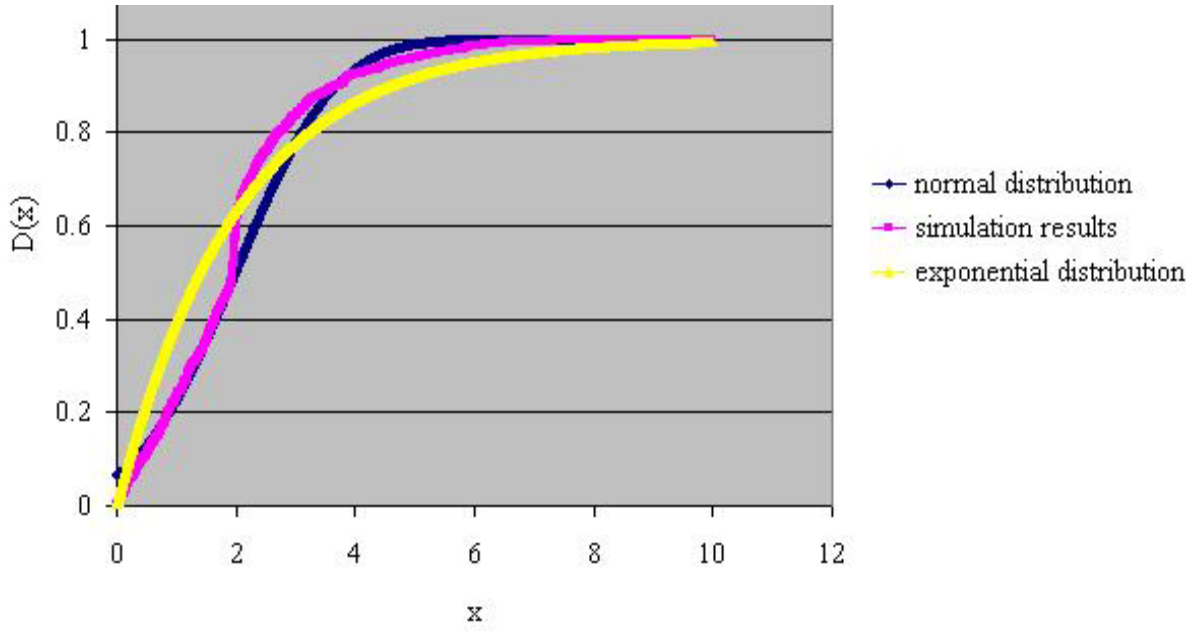

Figure 5.8: Distribution of simulated result 1 of atomic WS when noise is 0.04

Other simulation experiments were performed with the following parameters: the interarrival time in the model is 1.7 seconds, and service time is 1.8 seconds. The uniform distributed noise is added into the server, in each second, the server's performance is added by some "noise". The function is: in every second, server.performance = 1 + uniform(-noise,0); or the performance is 1.

Note that the only difference from the previous simulation experiments is that the noise's time interval is changed from 2 seconds to 1 second. The smaller time interval to add noise, makes the simulation results' distribution curve smoother and more similar to normal distribution curve.

There are also two groups of results. As the previous simulation experiments, the first one has smaller "noise" than the second one. For the first one, the noise is 0.035; for the second one, the noise is 0.04. The simulation results with modified model are shown as below:

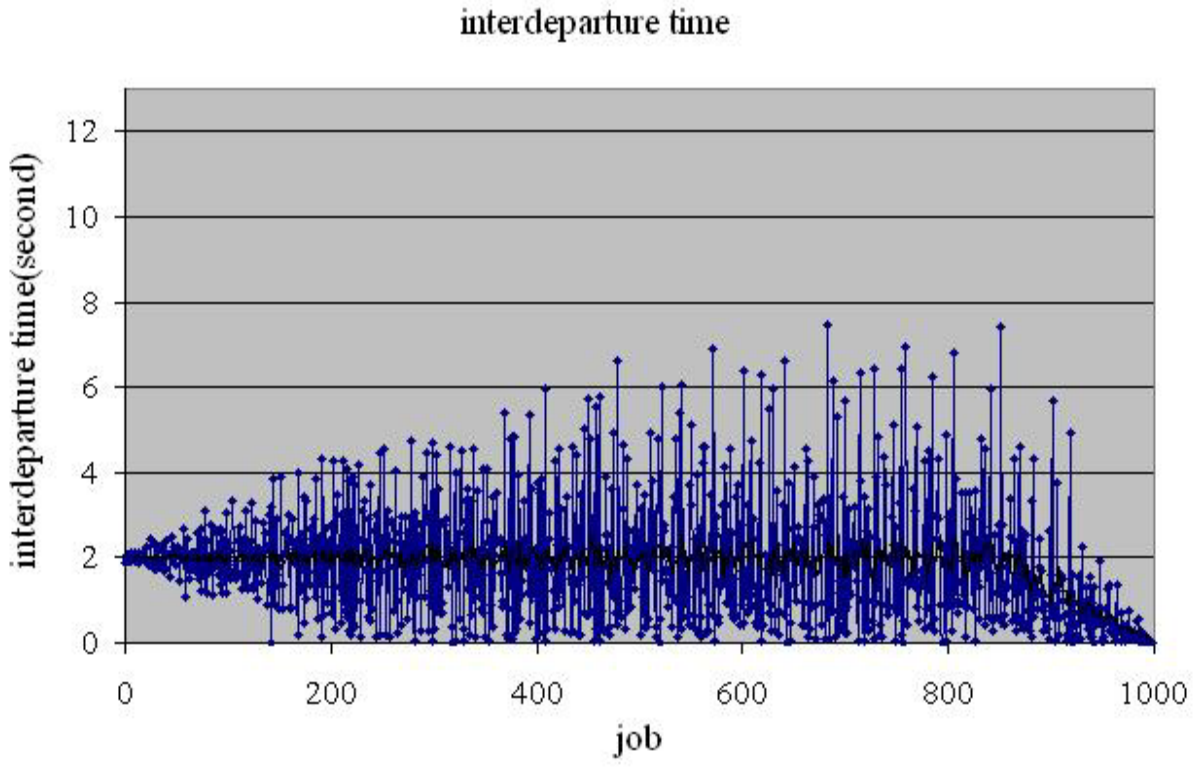

Figure 5.9: Simulated result 2 of atomic WS when noise is 0.035

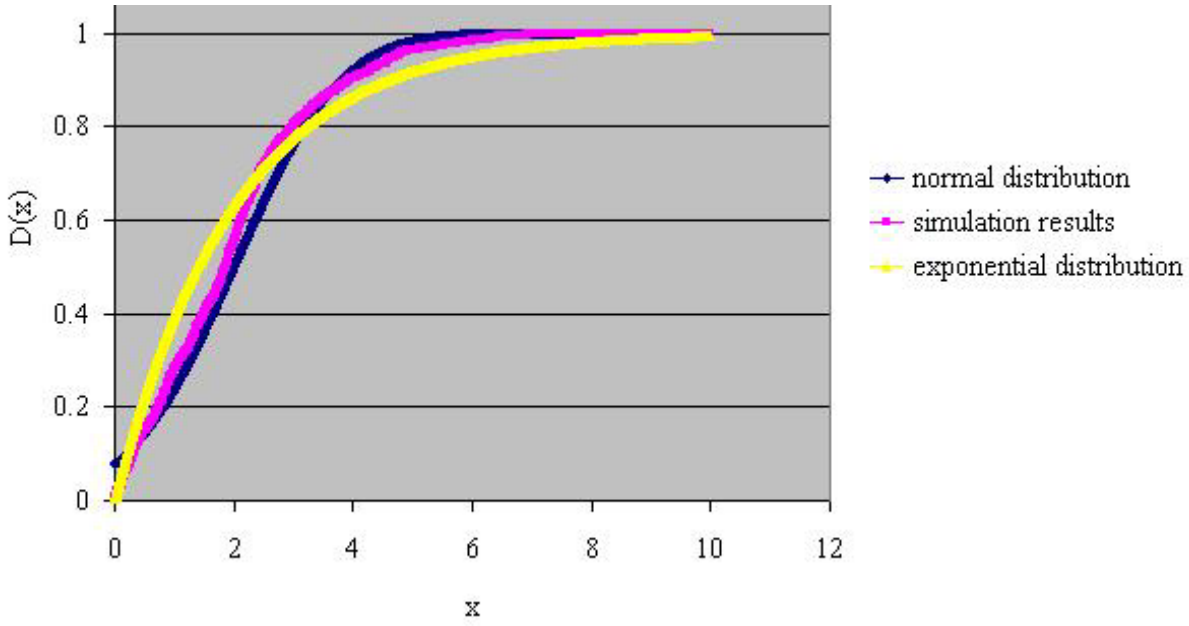

Figure 5.10: Distribution of simulated result 2 of atomic WS when noise is 0.035

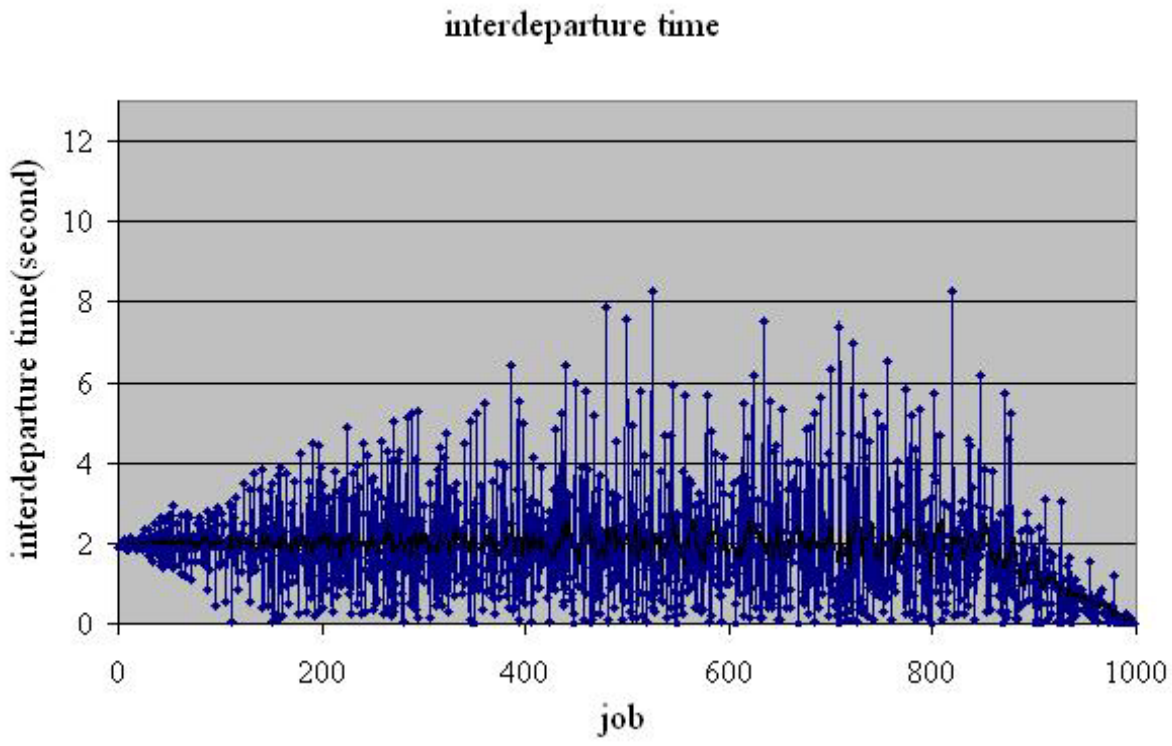

Figure 5.11: Simulated result 2 of atomic WS when noise is 0.04

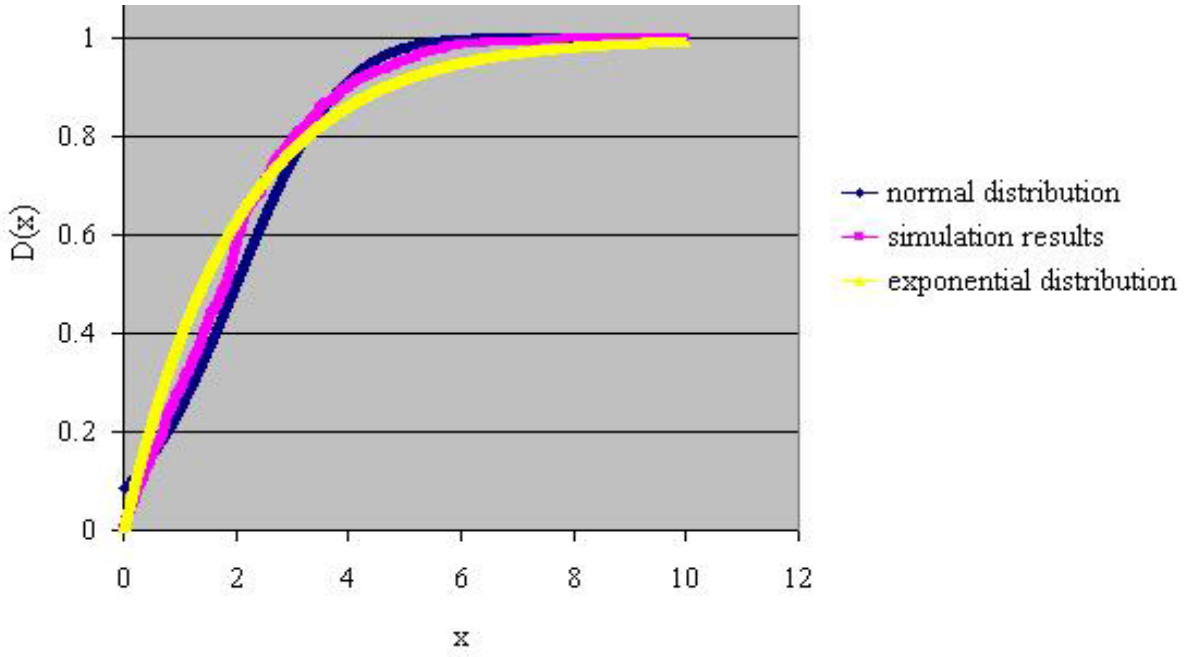

Figure 5.12: Distribution of simulated result 2 of atomic WS when noise is 0.04

5.2 Composite Web Services

5.2.1 CWS's Basic Model

CWS are built by combining multiple WSs in a certain kind of workflow, for example, sequential workflow, loops, and so on.

5.2.1.1 Basic model for experiment B and C

CWS in a sequential workflow are modeled in AnyLogic. These models are simulating the experiment B and C. Here is the model of experiment C with three services in sequential workflow (similarly, model for experiment B with two services in sequential workflow is also built):

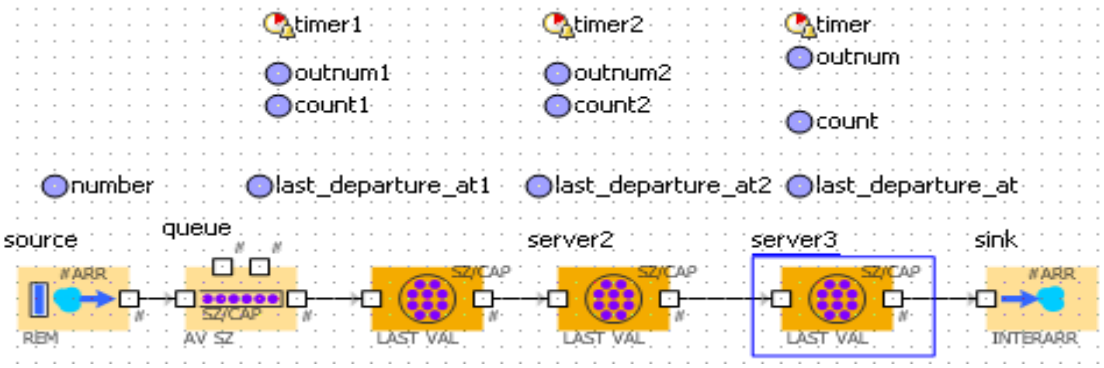

Figure 5.13: Simulation model of three services in sequential workflow in AnyLogic

In this model, the queuing network model is used, and the atomic WS is modeled as a server. The server has constant service time. A client sends requests to the server.

In this basic model, no admission control is considered, that is, the service's maximum number of thread is considered unlimited, and the queue size is considered unlimited. Here, the client sends 1000 requests in each experiment with constant rate (constant interarrival time), and the service's maximum number of thread is set as 1000, which is as large as the maximum number of jobs. Therefore, no jobs will wait in the queue, and the queues in front of the second and third service are omitted.

The model and its simulation results do not have "noise". Therefore, the simulation results show only the basic behavior of overloaded WS. The simulation results are the figures of interdeparture time and requests.

In section 4.2.1.1, the experiment results have been shown, and compared with the simulated results. The experiment results' trend is similar to the simulated results, but the experiment results have "noise" that makes the results fluctuating; while the simulated results can show the features of runtime behavior of WSs obviously, since it has no "noise". Figure 5.14 shows the simulated results of two WSs in a sequential workflow:

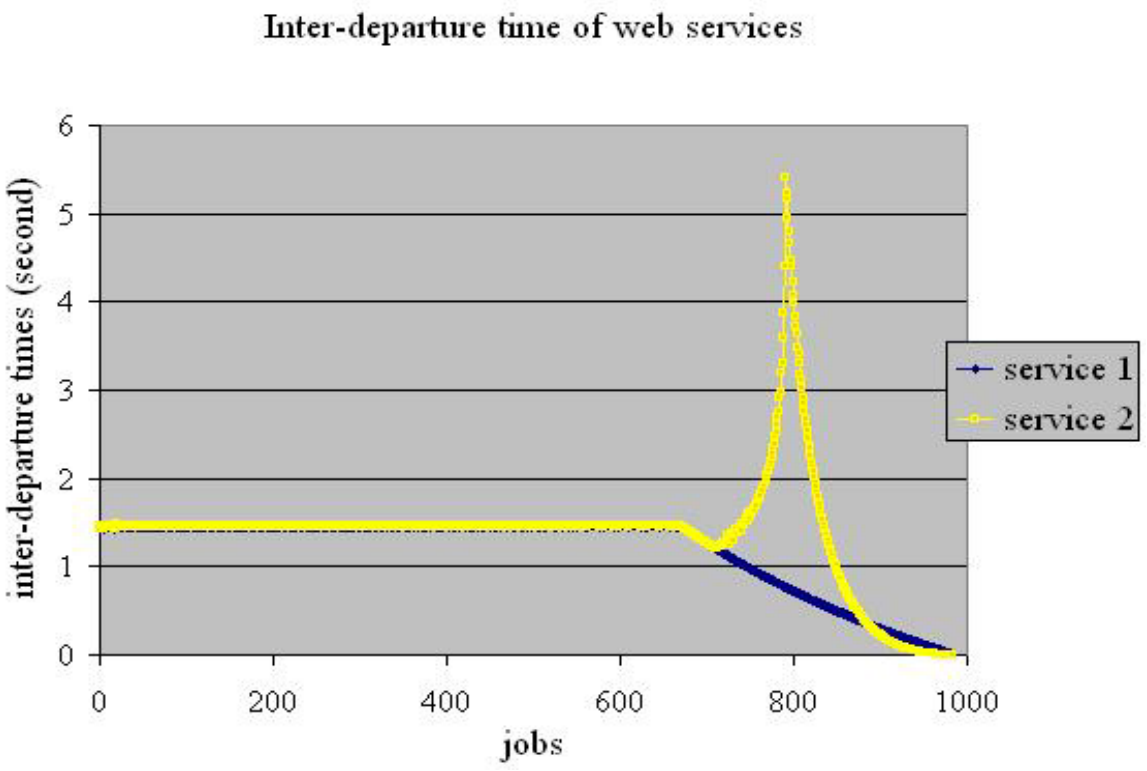

Figure 5.14: Simulated result of two WSs in a sequential workflow

In this model, there are two WSs in a sequential workflow. Every service has the same service time: 1.2 seconds. The interarrival time of the first service is 1 second, which is slightly smaller than the server's service time: 1.2 seconds. Therefore, the first server is slightly overloaded. At first, the first service's interdeparture time is about 1.4 seconds, which is used as the interarrival time of the second service, and is larger than the second service's service time. Therefore, at the beginning, the second service is not overloaded; its interdeparture time is equal to its interarrival time, which is also the first service's interdeparture time. Thus, at the first part of the experiment, the departure rate of the 2nd services is the same as the 1st one, and they are both about 1.4 seconds, which is almost stable.

The departure rate of the first service suddenly becomes faster when no new jobs come to the first service. Then the remaining jobs in the first service are less and less, and each can share

more resource in the first service. The departure rate decreases from about 1.4 seconds to 1.2 seconds/job (1.2 second is the service time), and continues decreasing to 0. This process is also shown in the previous section in the results of atomic WS.

The departure rate of the second service provider differs from the first one, only after the point when the jobs departure from the 1st provider faster than 1.2 seconds/job (1.2 second is the service time). After this point, the second service's interarrival time is smaller than its service time, so it is overloaded. The 2nd provider begins to experience a gradual overload that leads to slowdown in the departure rate (> 1.2) [14].

The departure rate of the second service slows down after the point it becomes overloaded, which is shown in the figure as the increasing departure time of second service. When no new jobs come to the second service, the departure rate start decreasing to 0.

Figure 5.15 shows the simulated results of three WSs in a sequential workflow:

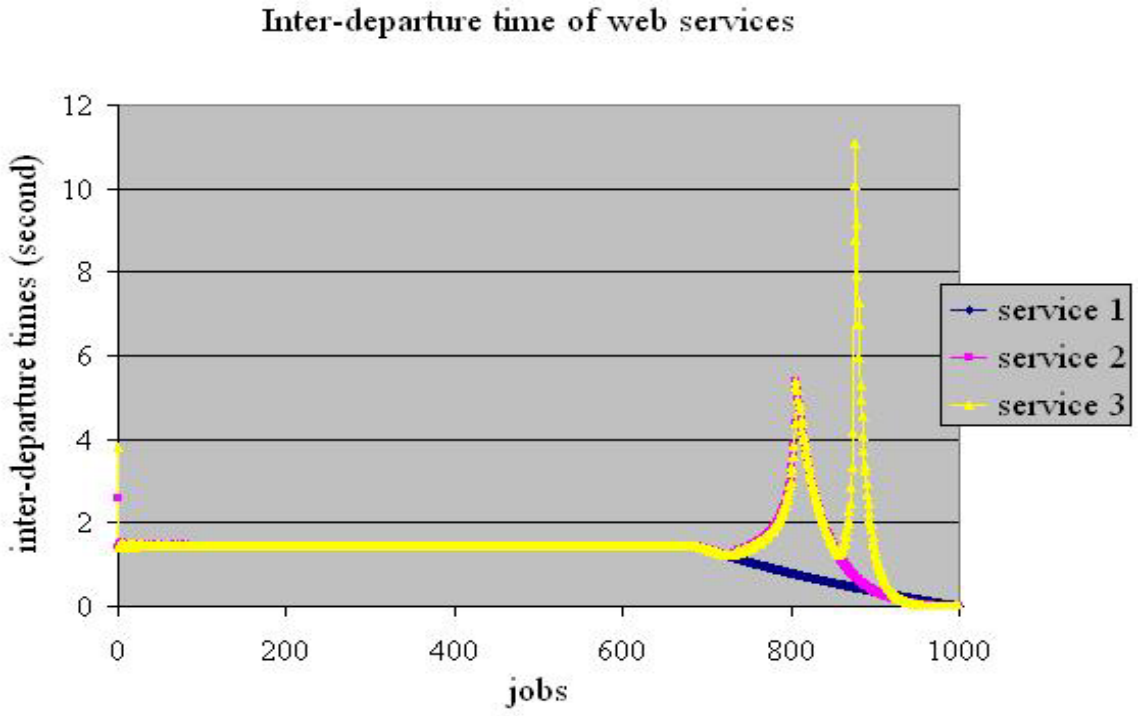

Figure 5.15: Simulated result of three WSs in a sequential workflow

In the simulation experiment of Figure 5.15, every service has the same service time: 1.2 seconds. The interarrival time of the first service is 1 second, which is slightly smaller than the server's service time: 1.2 seconds. The first service and second service has similar behavior as it is shown in the previous result of only two services. The 3rd provider adds an additional spike with the similar reason. When the jobs departure from the second provider faster than 1.2 seconds/job, the third provider begins to experience a sudden overload that leads first to a slowdown in the departure rate (> 1.2) [14].

5.2.1.2 Basic model for experiment D – exponentially distributed interarrival time

The previous basic model uses constant load. The workload can also be non-constant, which is discussed in experiment D. This can be set in the model's parameter.

In this section, simulation models were built in AnyLogic with exponentially distributed interarrival time. The results are the interdeparture time and its distribution (Only the 1^{st} to 350^{th} jobs' results that are in the "stable" period are used in the analyze, the results when the interdeparture time begins to decrease are not used).

The first experiment's setting is as following:

interarrival time=exponential(1), service time=1.2.

The first experiment's result is as following:

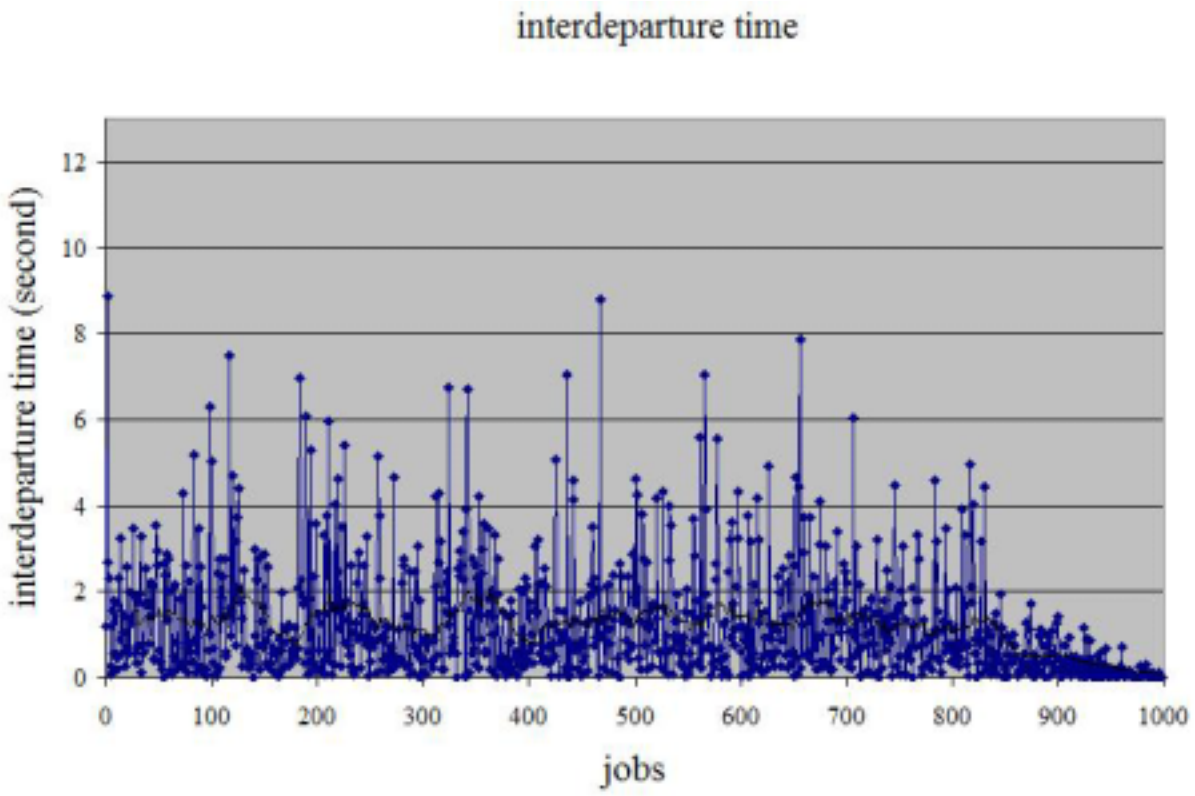

Figure 5.16: The interdeparture time of the first experiment

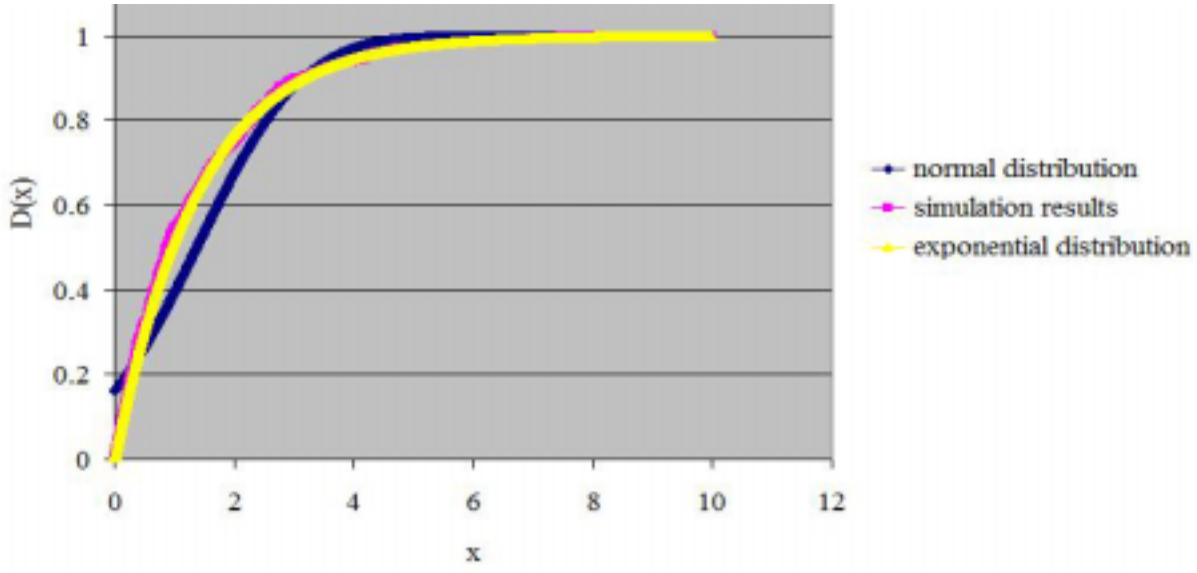

Figure 5.17: Distribution of interdeparture time of the first experiment

The interdeparture time of the first experiment is exponentially distributed, and its average value is 1.371141827.

The second experiment's setting is as following:

interarrival time=exponential(1), service time=1.4.

The second experiment's result is as following:

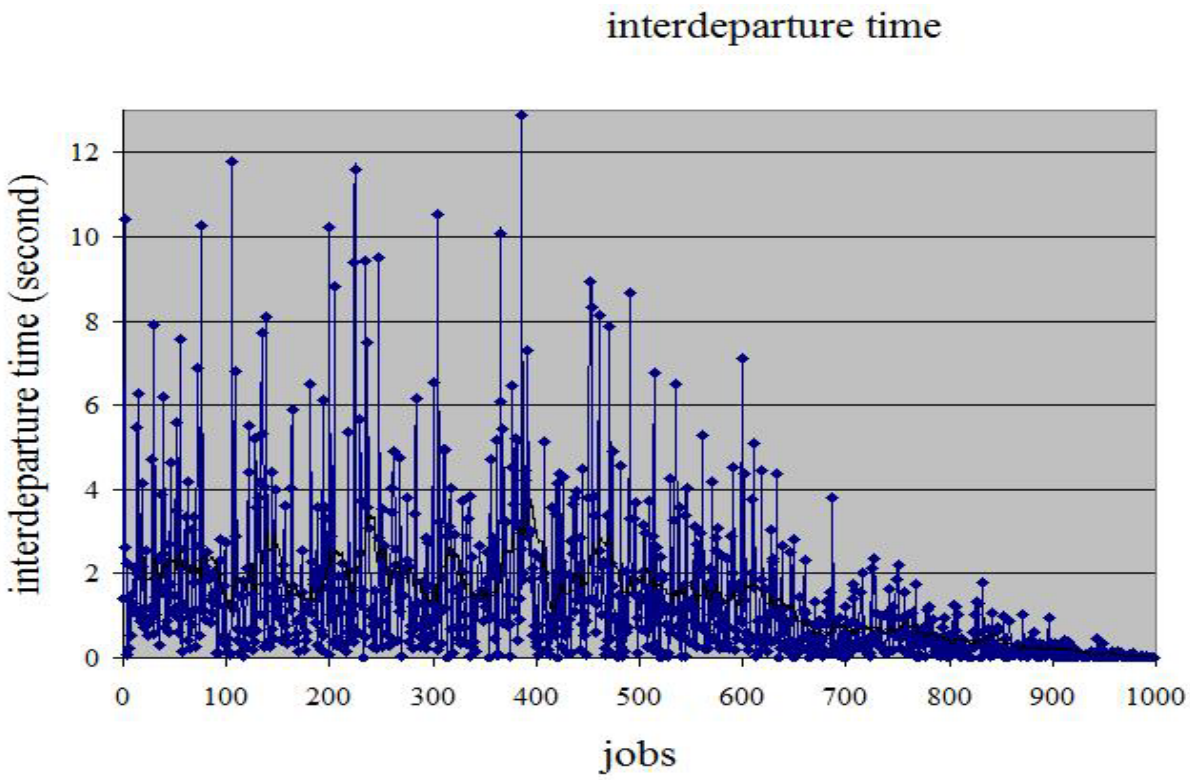

Figure 5.18: The interdeparture time of the second experiment

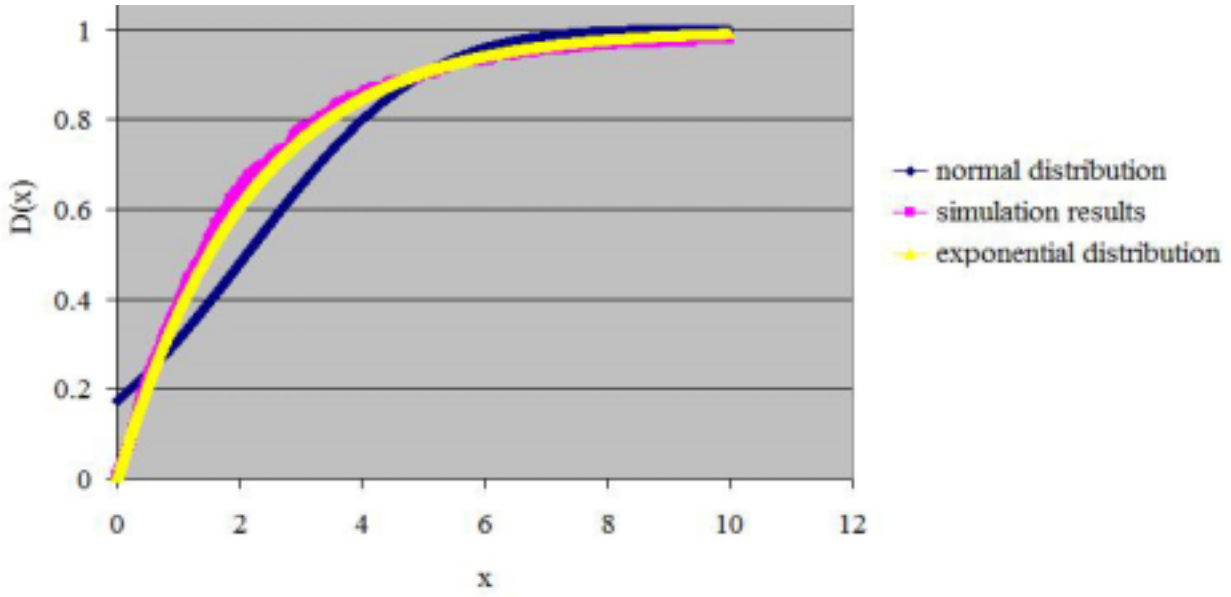

Figure 5.19: Distribution of interdeparture time of the second experiment

The interdeparture time of the second experiment is also exponentially distributed, and its average value is 2.117381528.

In summary, when the interarrival time is exponentially distributed, the interdeparture time is exponentially distributed, too. When the load increases with the service time, the average interdeparture time is larger. The interdeparture time decreases after a time when no new jobs arrives at the server, which is the same as what happens when the interarrival time is constant.

5.2.1.3 Basic model for experiment E –5 services in a sequential workflow with loops

CWS can also be built in other kinds of workflows, for example, loops, split, synchronization, and so on. Such CWS are included in “experiment E”.

In this simulation model, there are 5 services in a sequential workflow with loops. The interdeparture times of 5 services are recorded, as well as the completion time of the workflow. The simulation experiments settings are as following:

Totally 1000 requests are sent to the workflow.

Interarrival time = 1 second.

Service time = 1.2 seconds. (5 services are the same.)

(In service x, if: y>0, then: service x has y loop(s).

int x=1,2,3,4,5.

int y=0,1,2,3.

The simulation experiment x-y means service x has y loop(s), while others have no loops. In detail, such experiments are done:

0: 5 services all have no loops. It is a sequential workflow.

1-1: service 1 has 1 loop, while others have no loops.

1-2: service 1 has 2 loops, while others have no loops.

1-3: service 1 has 3 loops, while others have no loops.

2-1: service 2 has 1 loop, while others have no loops.

2-2: service 2 has 2 loops, while others have no loops.

2-3: service 2 has 3 loops, while others have no loops.

3-1: service 3 has 1 loop, while others have no loops.

3-2: service 3 has 2 loops, while others have no loops.

3-3: service 3 has 3 loops, while others have no loops.

4-1: service 4 has 1 loop, while others have no loops.

4-2: service 4 has 2 loops, while others have no loops.

4-3: service 4 has 3 loops, while others have no loops.

5-1: service 5 has 1 loop, while others have no loops.

5-2: service 5 has 2 loops, while others have no loops.

5-3: service 5 has 3 loops, while others have no loops.

In this research, the "completion time" is recorded to represent the performance of CWS. The "completion time" is the time for all the requests to be completed is called, which is the time from the first request comes into the first service, to the last request departure from the last service.

In the results for the experiment 0, the completion time of the workflow is 1699.17 seconds, which is the smallest, since no loops are in this experiment to have impact on the performance of the workflow. The curves of interdeparture times of the 5 services show the same feature as in the result of sequential workflow of only 2 or 3 services (in section 5.2.1.1), that is, the 2nd, 3rd, 4th, 5th service adds an additional spike to the curve.

Here are the interdeparture times of the 5 services in the experiment 0

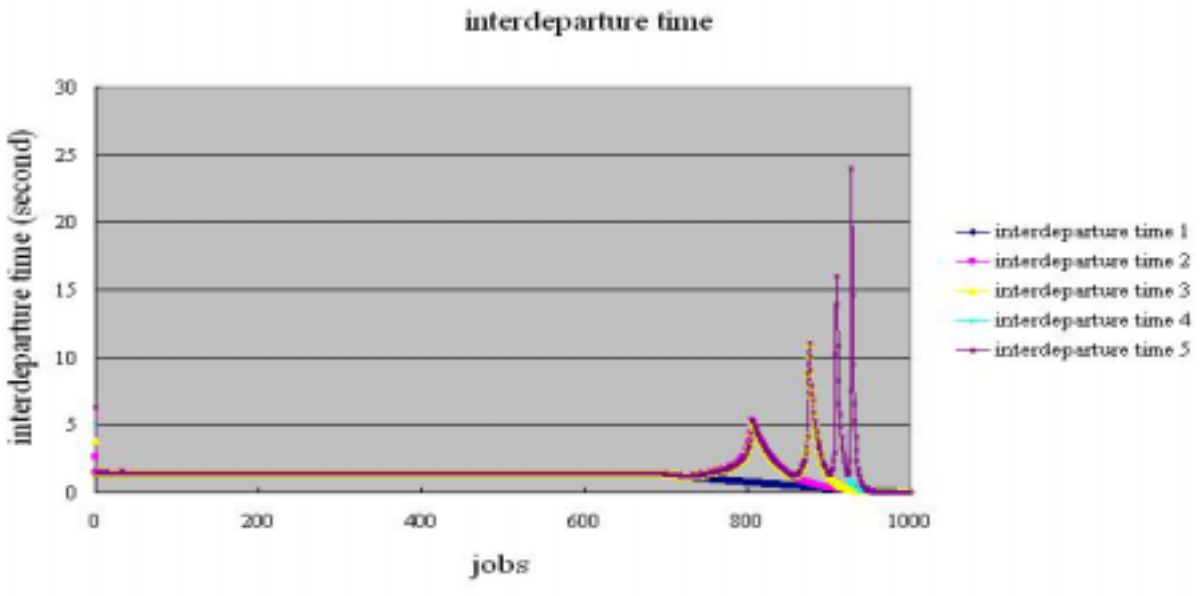

Figure 5.20: The interdeparture times of the 5 services in the experiment 0

The following experiment results are the results of the experiment 1-1, 1-2, 1-3, 2-1, 2-2, 2-3, 3-1, 3-2, 3-3, 4-1, 4-2, 4-3, 5-1, 5-2, 5-3. Those results are different from the result when there is no loop, and they are different from each other.

Here are the interdeparture times of the 5 services in the experiment 1-1:

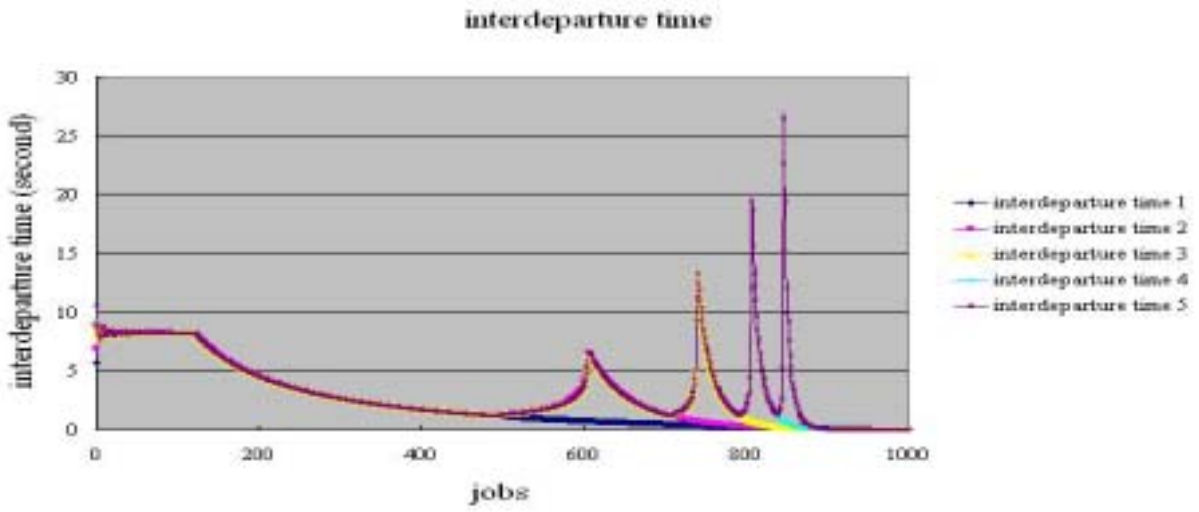

Figure 5.21: The interdeparture times of the 5 services in the experiment 1-1

Here are the interdeparture times of the 5 services in the experiment 1-2:

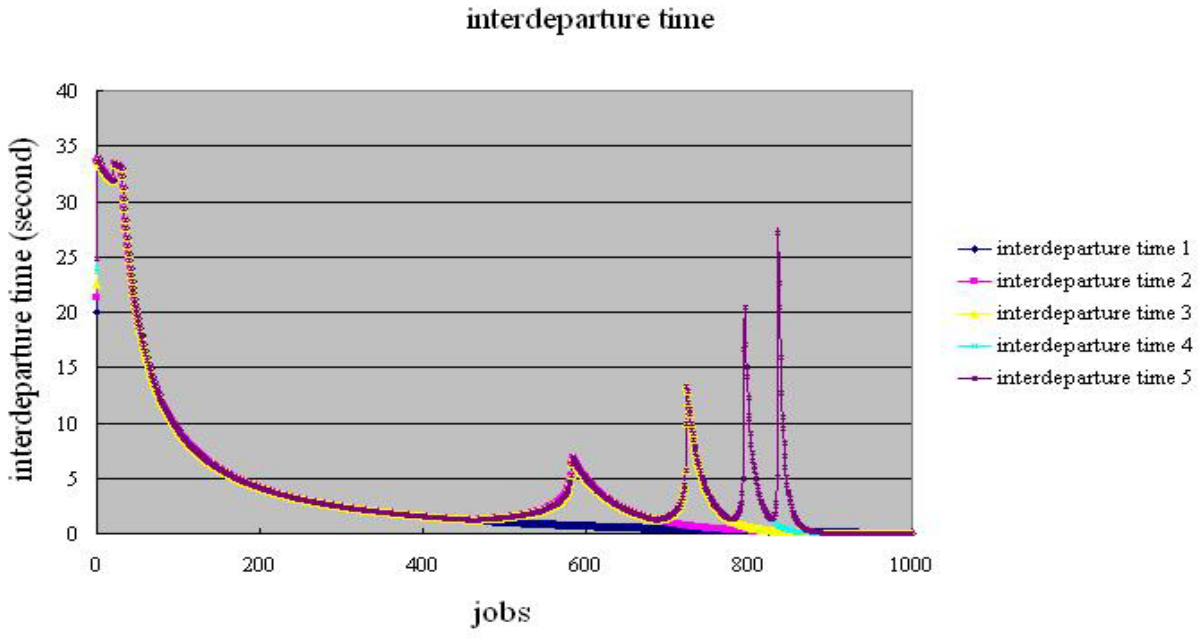

Figure 5.22: The interdeparture times of the 5 services in the experiment 1-2

Here are the interdeparture times of the 5 services in the experiment 1-3:

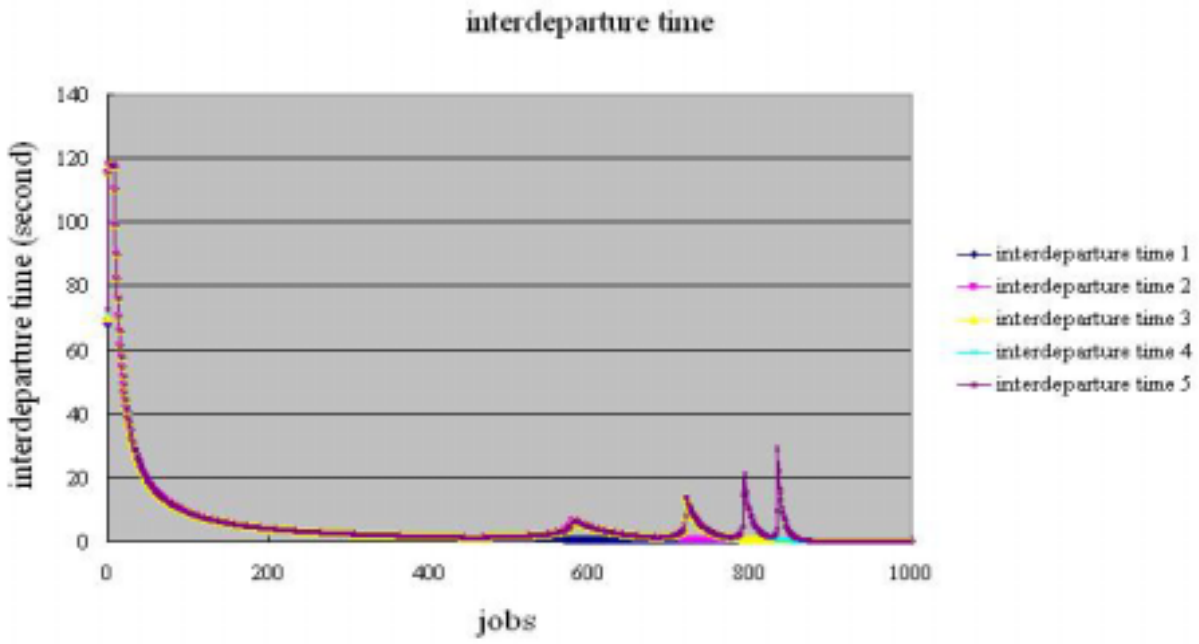

Figure 5.23: The interdeparture times of the 5 services in the experiment 1-3

Here are the interdeparture times of the 5 services in the experiment 2-1:

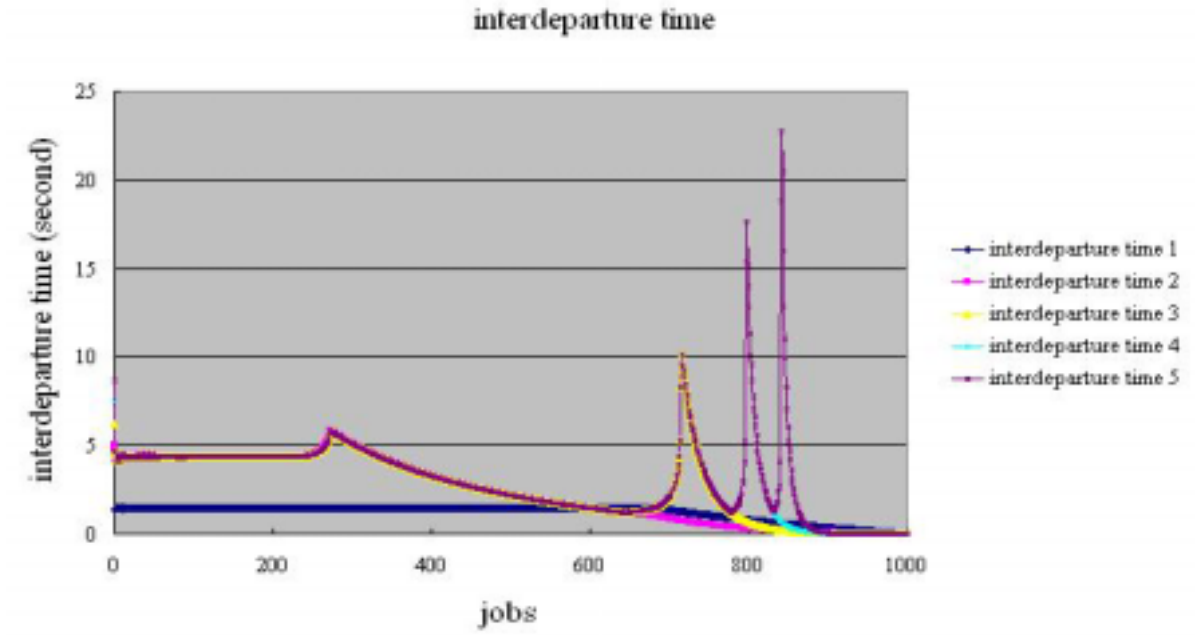

Figure 5.24: The interdeparture times of the 5 services in the experiment 2-1

Here are the interdeparture times of the 5 services in the experiment 2-2:

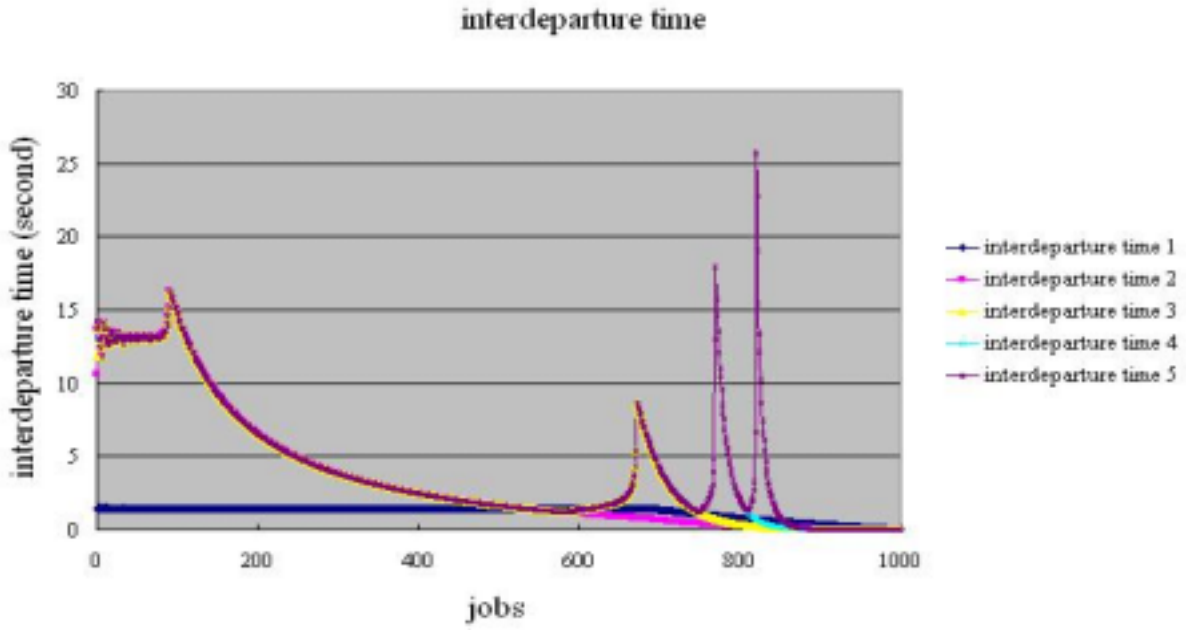

Figure 5.25: The interdeparture times of the 5 services in the experiment 2-2

Here are the interdeparture times of the 5 services in the experiment 2-3:

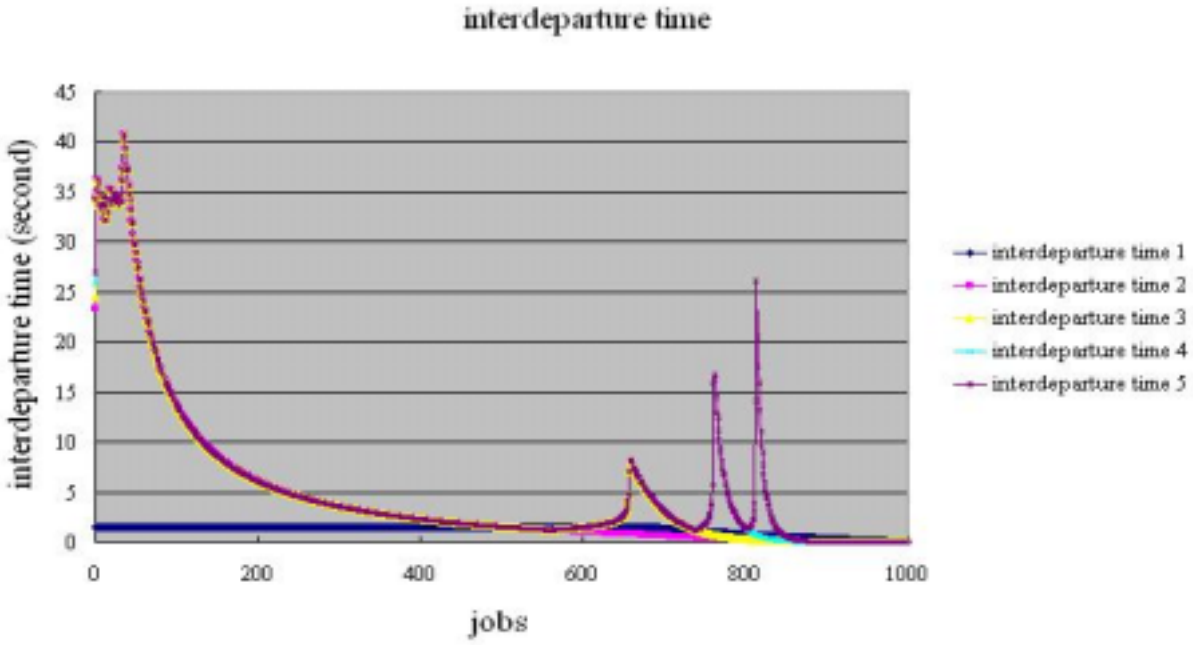

Figure 5.26: The interdeparture times of the 5 services in the experiment 2-3

Here are the interdeparture times of the 5 services in the experiment 3-1:

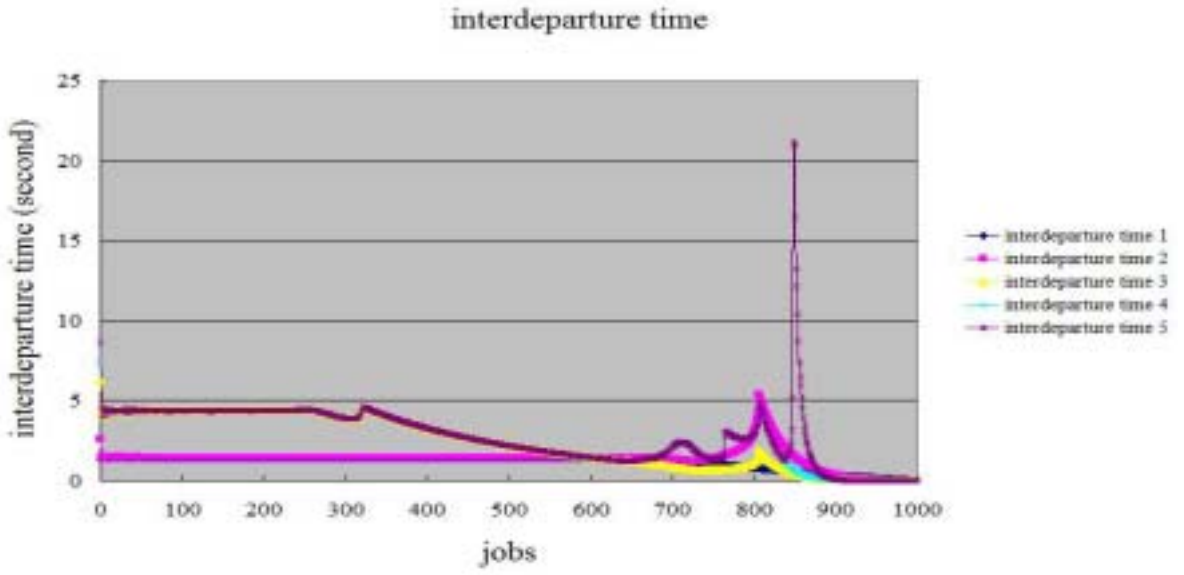

Figure 5.27: The interdeparture times of the 5 services in the experiment 3-1

Here are the interdeparture times of the 5 services in the experiment 3-2:

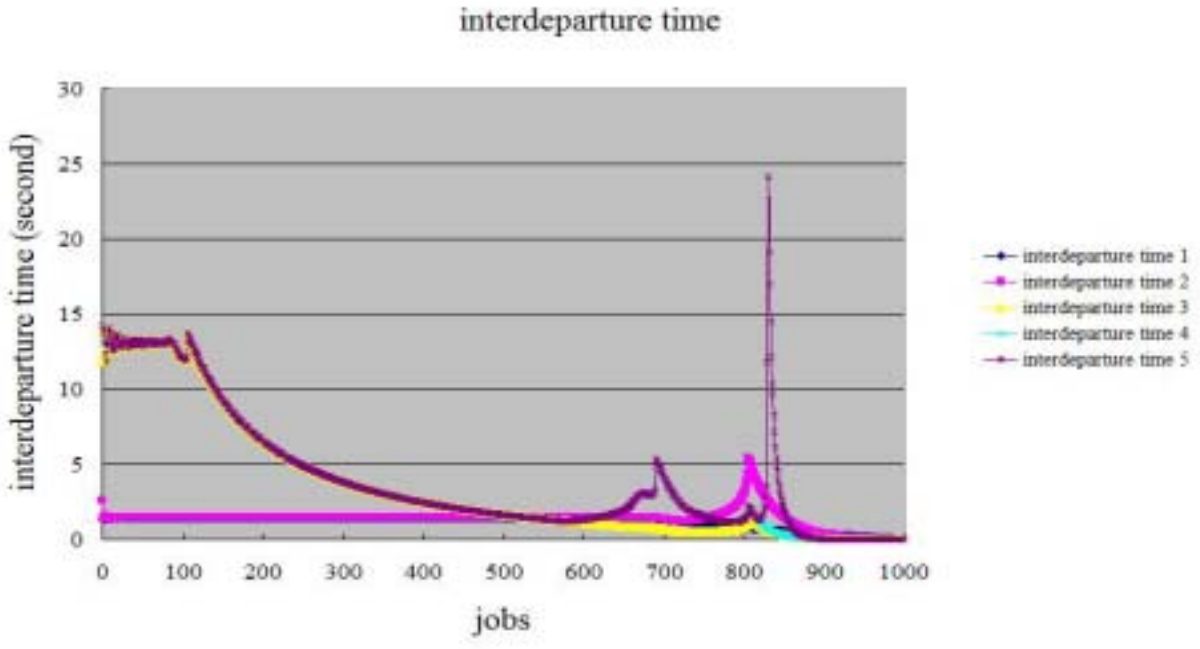

Figure 5.28: The interdeparture times of the 5 services in the experiment 3-2

Here are the interdeparture times of the 5 services in the experiment 3-3:

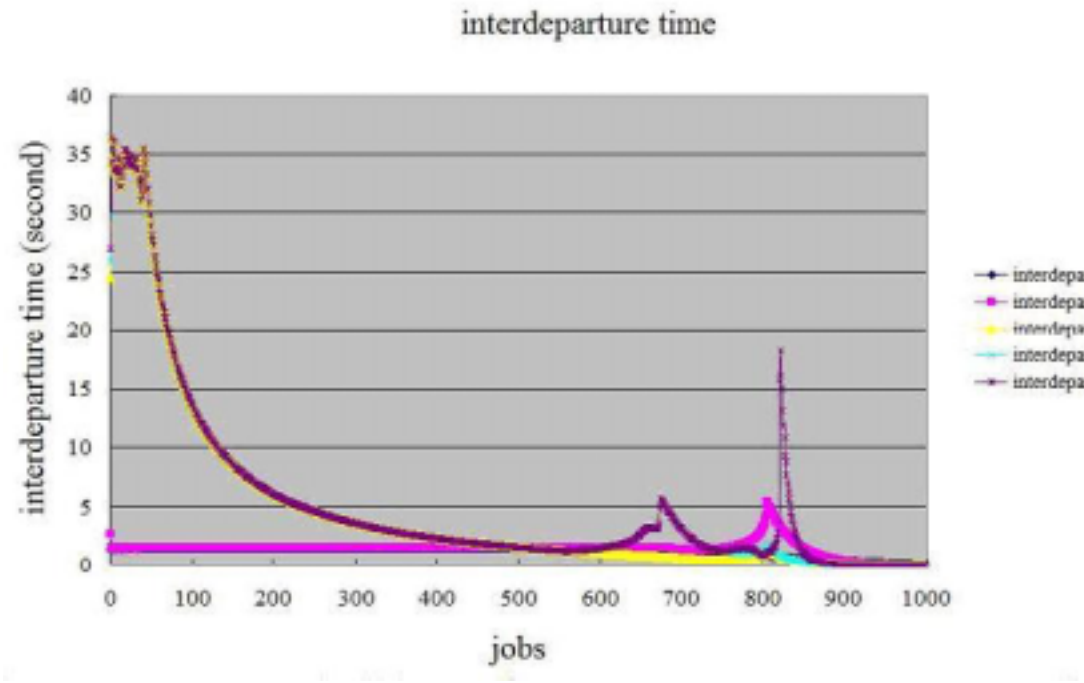

Figure 5.29: The interdeparture times of the 5 services in the experiment 3-3

Here are the interdeparture times of the 5 services in the experiment 4-1:

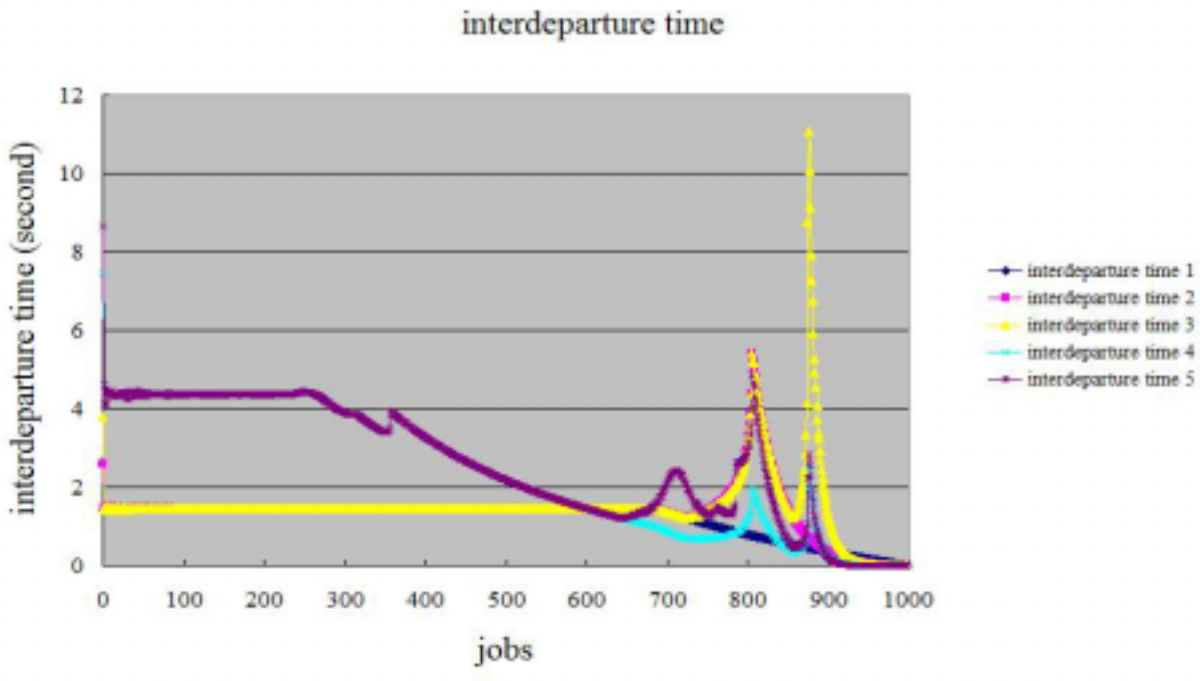

Figure 5.30: The interdeparture times of the 5 services in the experiment 4-1

Here are the interdeparture times of the 5 services in the experiment 4-2:

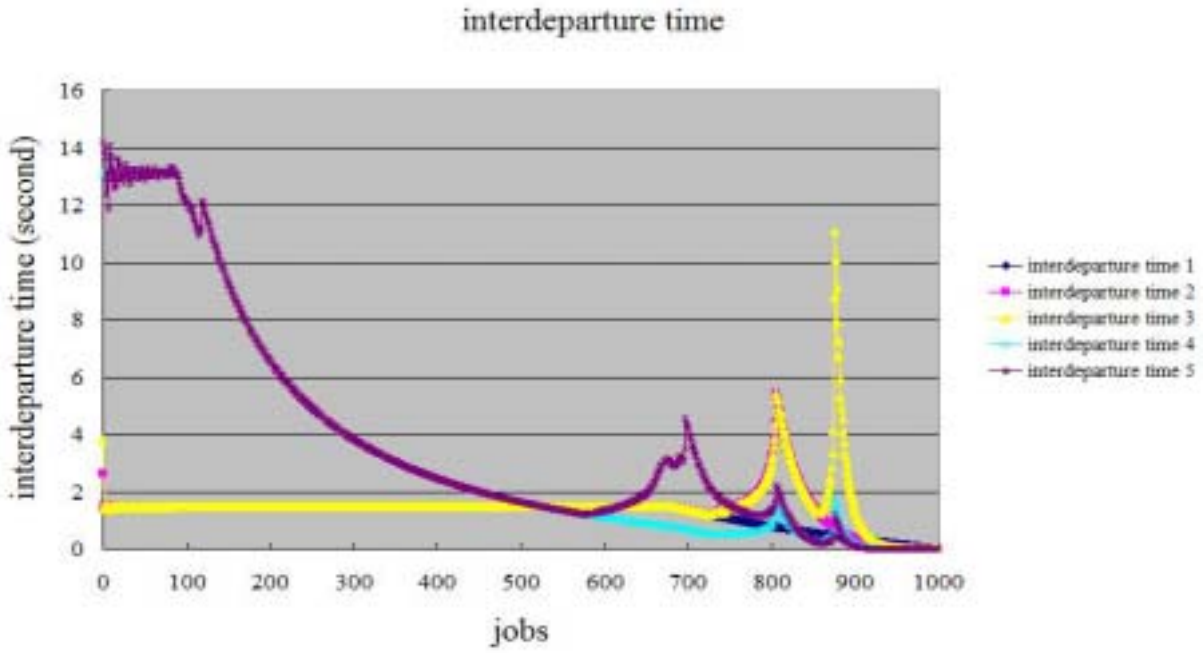

Figure 5.31: The interdeparture times of the 5 services in the experiment 4-2

Here are the interdeparture times of the 5 services in the experiment 4-3:

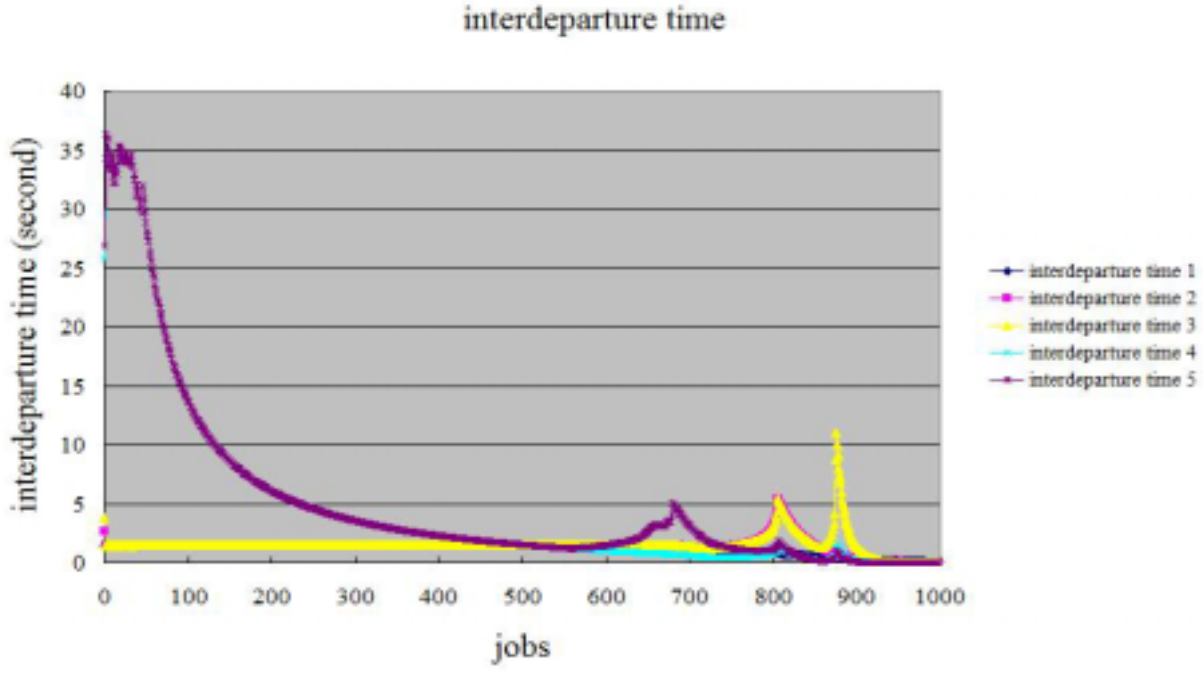

Figure 5.32: The interdeparture times of the 5 services in the experiment 4-3

Here are the interdeparture times of the 5 services in the experiment 5-1:

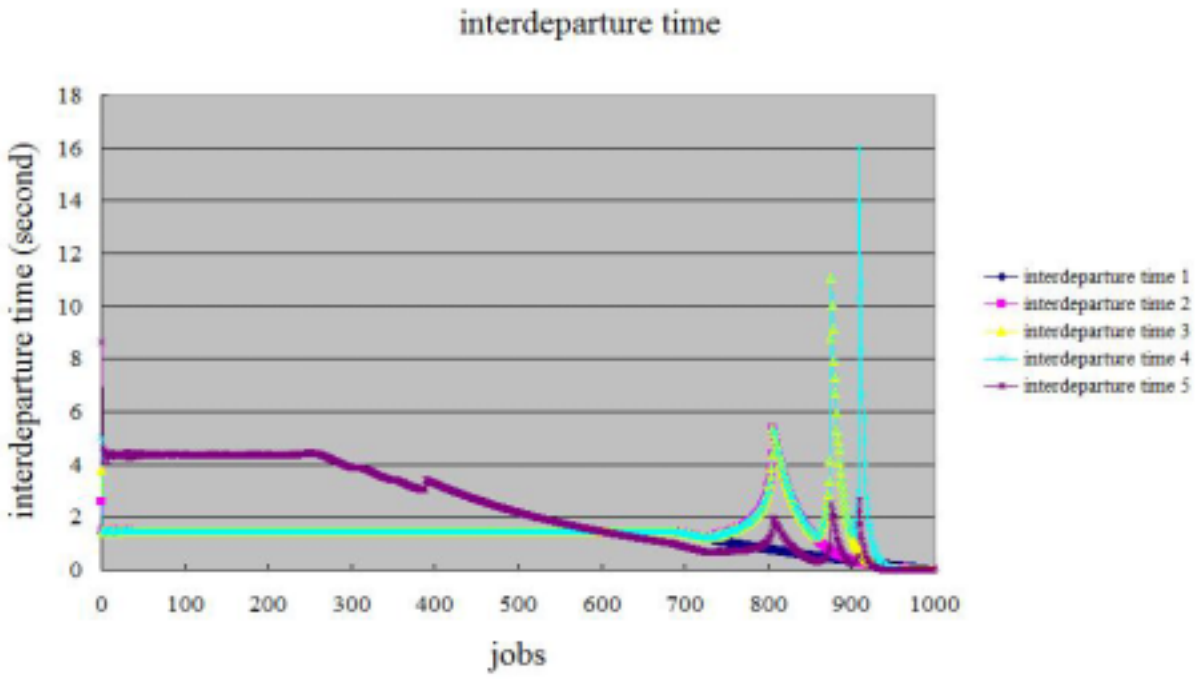

Figure 5.33: The interdeparture times of the 5 services in the experiment 5-1

Here are the interdeparture times of the 5 services in the experiment 5-2:

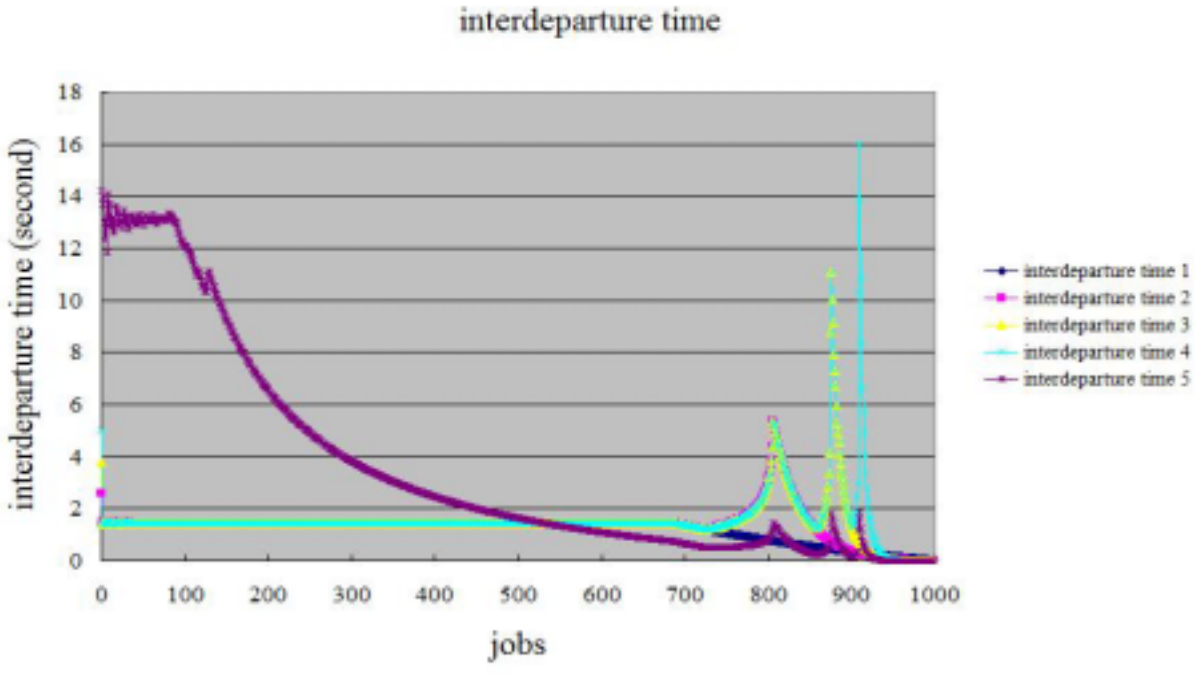

Figure 5.34: The interdeparture times of the 5 services in the experiment 5-2

Here are the interdeparture times of the 5 services in the experiment 5-3:

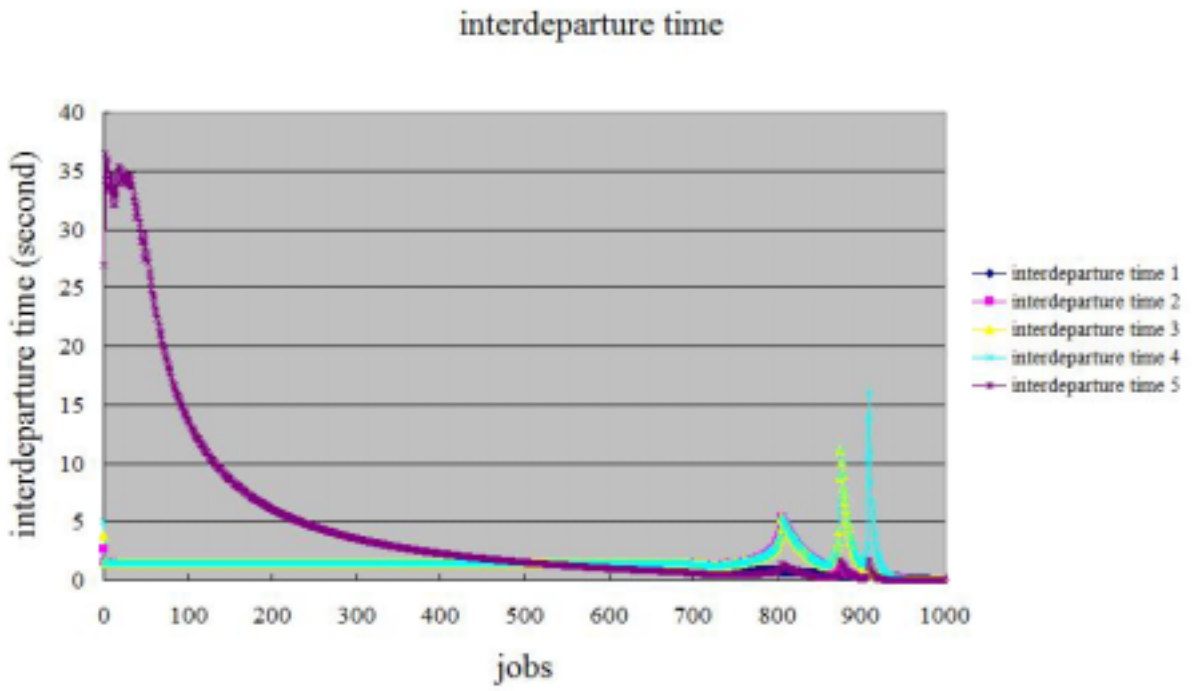

Figure 5.35: The interdeparture times of the 5 services in the experiment 5-3

In those results above, as discussed in section 5.2.1.1, the departure rate of the next service provider differs from the previous one, only after the point when the jobs depart from the previous provider faster than 1.2 seconds/job (1.2 second is the service time). After this point, the next service's interarrival time is smaller than its service time, so it is overloaded, which leads to slowdown in the departure rate (> 1.2) [14]. This situation is shown as a spike in the results. In Summary, the service 1 is overloaded at first, while the service 2, 3, 4, 5 are not fully utilized ("underload") for some period until the middle or the end of the experiment.

When a service is included in loops, its load is heavier. When a service loops for once, twice, and three times, the load increases when the number of loops increases. For example, in the results of experiments1-1, 1-2, 1-3, when the service 1 loops for once, twice, and three times, the

completion time of the workflow is larger than when no loop exists, the curves of interdeparture times of the 5 services are different, too.

As will be discussed in Section 5.2.1.4, the service 1 is the most important service in sequential workflow, and the service 2, 3, 4, 5 are less and less important on the performance of the whole workflow. This is also because that the service 1 is overloaded at first, while the service 2, 3, 4, and 5 are not fully utilized (under load) at the beginning of the experiments for some period, if there is no loop. This feature still holds when the sequential workflow has loops: when the most overloaded service (service 1) is included in loops, the performance of the whole workflow is affected most seriously; while when not fully utilized services (service 2, 3, 4, and 5) are included in loops, the load on them is relatively lighter.

The completion times of the workflow with different parameters are as following:

Table 5.2: The completion time of different workflows with loops

Simulation experiments of loops	The completion time of the workflow (second)
0	1699.17
1-1	3451.91
1-2	4718.09
1-3	5933.39
2-1	3103.02
2-2	4400.22
2-3	5629.22
3-1	2803.76
3-2	4085.57
3-3	5310.22
4-1	2604.88
4-2	3873.75
4-3	5091.91
5-1	2405.00
5-2	3605.00
5-3	4805.00

The results reconfirmed the feature in conclusion above, e.g., the result of experiment 4-2 is slightly larger than in experiment 5-2, and the result of experiment 4-3 is slightly larger than in experiment 5-3, and so on. In summary, the completion time of the workflow is longer when there are more loops in a certain service; and when the service 1 is included in loops, it has the largest completion time and shows the most important impact on the performance of the whole workflow.

5.2.1.4 CWS's basic model for performance optimizing

The sequential workflow is modeled not only for simulation experiments, but also for performance optimizing as described in this section.

CWS are built in AnyLogic, and are run in simulation experiments with different parameters. The performance of CWS are compared in different experiments with different parameters, to find the most suitable parameter which lead to best performance. Therefore, CWS could be configured optimally in a real system according to simulated best configuration. Developers can either get the best performance with limited resource, or use less resource to get the same performance.

In this research, the "CPU processing capacity", or "processor power", is used to describe the model parameter, and to represent the resource used for WS. For example, when the CPU processing capacity is 100% at first, the WS's service time is 1 second; when the CPU processing capacity is reduced to 70%, the WS's service time is increased to 1/70%= 1.429 seconds. In real system, such reduces means using a slower machine or a busy machine that has to do other jobs for deploying WS.

In this research, the "completion time" is used to represent the performance of CWS. When the "completion time" is small, the performance is considered as good.

Such simulations experiments have already been done with CWS in sequential workflows, and simulation experiments in other workflows, such as, loops and split, are going to be done.

The simulation model with a CWS in sequential workflow is built with 5 services (this system setup is similar to experiment B and C, but with longer sequential workflow). Totally 1000 requests are sent to the CWS with a constant interarriveral time. At first, all services are the same, suppose each service is using 100% CPU processing capacity. Therefore the total CPU processing capacity is 500%, and all services have the same service time for same jobs. Running simulation experiments get the results in the form of "completion time". At first, with five 100% CPU processing capacity servers, the "completion time" is 17.5084 seconds.

The simulation experiments aim to reduce resource consumption, without reducing too much performance, i.e., without increasing "completion time" too much; or get better performance with less resource. These changes are made to model parameter in different simulation experiments, by using parameter variation experiments in AnyLogic. The results with changed parameters are usually compared to the original results with five 100% CPU processing capacity servers.

The simulation results with CWS in sequential workflow are as below:

(a) When the first service has 100% power, while the other four have 86%, the time cost is 19.1255 seconds. Total CPU processing capacity is (1+4*0.86)=4.44=444%, save total CPU processing capacity (5-4.44)/5=11.2%, cost (19.1255-17.5084)/17.5084=9.24% more completion time.

(b) When the first four services have 100% power, while only the fifth service has 86%, the time cost is 17.6842 seconds. Total CPU processing capacity is (4+0.86)=4.86=486%, save total

CPU processing capacity (5-4.86)/5=2.8%, cost (17.6842-17.5084)/17.5084=1.004% more completion time.

Those two results prove that one can reduce some resources without reducing too much performance. For example, in (a), the resource is reduced by 11.2%, with the performance reduced by only 9.24%.

Other simulation experiments were also performed. In each experiment only one service's resource is reduced (to 84% for example), and other four services' resource are kept as 100%; the weakened service is different in each experiment. From the results, a useful performance rule is concluded as following: for sequential workflow, the first service is the most important, and it should have as enough resource as possible; then the 2nd, 3rd, 4th, 5th service is less and less important. Therefore, reducing the fifth service's resource is most practical, and then is the fourth, third, second one.

In simulation experiment (a) and (b), we already followed such rule, and succeeded in reducing some recourse without decreasing the performance that much.

Such rule is also used to get better performance with the same total resource. The results are shown here as (c), (d):

(c) In the model, the first service's CPU processing capacity is 1+c, and fifth service's CPU processing capacity is 1-c.

When c is 0, every service has 100% resource; time cost (completion time) is 17.5084 second.

When c is set larger than 0, the total resource is still 500%. Different values of c were tested, and the results proved that the time cost could be reduced.

When c is 0.0972, the time cost is 14.4916 seconds, which is the smallest value. It cost no more total CPU power (still be 500%), but can save (17.5084-14.4916)/17.5084=17.23% completion time.

(d) In the model, the first service's CPU processing capacity is 1+c/2, the second, third, forth service's CPU processing capacity is 1+c/6, and the fifth service's CPU processing capacity is 1-c.

When we set c as larger than 0, the total resource is still 500%. Different values of c are tested, and results prove that the time cost can be reduced.

When c is 0.16, the time cost is smallest, about 15.

For (c) and (d), the total resource is still 500%, but with shorter completion time. The performance of the workflow is improved without adding additional resources. The best result of (c) is even better than the best result of (d), which also reconfirms the performance rule for sequential workflow. The reason is that: in (c), all the reduced resource from the fifth service has been added to the first service, which has the most important impact on a sequential workflow's performance; while in (d), half of the reduced resource from the fifth service has been added to the second, third, forth service, and the second, third, forth service have less important impact on in sequential workflow performance than the first service. In (d), some of the resources are used to improve the performance of the second, third, forth service; while in (c), all the resources are used to improve the performance of the first service, which has better effect on improving the performance of the workflow.

In summary, for sequential workflows, the first service's performance is the most important one for the workflow's performance, and the first service should have as enough resource as possible; then the 2nd, 3rd, 4th, and 5th service is less and less important. Therefore, reducing

the fifth (the last) service's resource is most practical, and then is the fourth, third, second one. This rule helps to reduce some recourse without decreasing the performance that much, or improve the performance with the same or even less resource.

5.2.2 Modified Model for Experiment B, C, D, E

In a modified model, a server with "noise" presents each WS in AnyLogic.

The atomic WS's modified model is discussed in the previous sections. It is known that this modified model can simulate the "noise" in real systems, and its results are similar to experiment results.

The CWS could be also built in AnyLogic with the modified model. The CWS can be built in a certain kind of workflow, for example, sequential workflow, loops, and so on. The simulation experiments run in AnyLogic to simulate the performance of different kinds of workflows. The workloads can be constant or non-constant.

5.3 Simulation Experiments with Timeouts

In the previous experiments, we set the "interarrival" time only a little smaller than the service time of the WS. Therefore, the WS is only slightly overloaded, and no "timeout" happens.

If the WS is overloaded, it may timeout.

The timeout limit and queue size for the queue can be set in the AnyLogic model, according to a real system. Also, the maximum number of threads in the WS is limited, which is defined as the capacity of the server in AnyLogic. With such improvement, the model will be closer to real WSs. Timeout or overload may happen with the limited resources.

Six simulation experiments are done for studying timeout. Totally 1000 requests are sent to an atomic WS in all the simulation experiments. Queue size is 100 in all the simulation experiments,

since only less than 40 requests will be waiting in the queue according to simulation experiments results. The queue size is enough since the service is only slightly overloaded.

The parameter settings of the simulation experiments are: the interarrival time, service time, server's capacity (maximum number of threads), queue sizes, timeout limit (the time limit set in the queue). Those (except for queue sizes) are different in each simulation experiment.

Timeout happens in all the simulation experiments, but the numbers of requests that are successfully processed (noted as "success_num" for short) are different, that is, the number of requests that are discarded after timeout happens (noted as "timeout_num for short) are different, too. Those are recorded as the results of the simulation experiments. Also, "outnum" is used as a notation for all the requests that departure, both successfully processed and timeout. Note that: outnum=success_num+timeout_num, and outnum=999 in all the results, since total number of requests is 1000, and only 999 are recorded in the model. Therefore, "success_num" and "timeout_num" are related. After all the requests departured, the "completion time" is also recorded in the results of the simulation experiments.

All simulation experiments' parameter settings and results are in Table 5.3 (the interarrival time, service time, timeout limit, and completion time are recorded in the unit of second(s)).

In Table 5.3, experiment 1's parameter settings and its results are referred to by every other experiment. Compared to experiment 1, only one parameter's value is changed in experiment 2 to 6, and the results changed, too.

In detail, compared to experiment 1, experiment 2's server's capacity is smaller, experiment 3's timeout limit is smaller, experiment 4's interarrival time is exponentially distributed with the same average value, experiment 5's service time is smaller, and experiment 6's interarrival time is smaller. As a result, experiment 2 to 6 has larger number of timeouts than experiment 1.

From all the results, it can be concluded that: interarrival time, service time, server's capacity, timeout limit all have affect on the “timeout_num”. The “timeout_num” shows how many the timeouts happens, and reflects the performance of the system in a perspective.

Table 5.3: Simulation experiments’ parameter settings and results for timeout

No. of experiment	Interarrival time	Service time	Server's capacity	Timeout limit	success _num	time-out_ num	Completion time
Experiment 1	1.7 (constant)	2.0 (constant)	200	60	976	23	1954
Experiment 2	1.7 (constant)	2.0 (constant)	100	60	928	71	1858
Experiment 3	1.7 (constant)	2.0 (constant)	200	50	972	28	1858
Experiment 4	1.7 (exponential)	2.0 (constant)	200	60	924	75	1866.79
Experiment 5	1.7 (constant)	2.2 (constant)	200	60	901	98	1984.4
Experiment 6	1.6 (constant)	2.0 (constant)	200	60	936	63	1874

Note that: here, the “completion time” does not means the performance of the system is very good, since when timeout happens, some of the requests are discarded directly without being processed, which shortened the overall “completion time”. In the results, experiment 2 to 4, 6 has smaller “completion time” than experiment 1; while only experiment 5 has larger “completion time” than experiment 1, since the service time increased.

The simulation of timeout is done as following:

Since a service is overloaded, the requests cannot be processed in time, some of them have to wait in the service. More and more requests wait in the service, which happens from the beginning of the experiments.

Since the server's capacity (maximum number of threads) is limited, for example, is 200, and the service is overloaded, the service will be full of waiting requests soon. Then newly arrived requests have to wait in the queue after the server is full, which usually happens in the middle of the experiments.

When the requests have waited in the queue for a long time, a timeout happens, the requests are discarded by the queue, which usually happens near the end of simulation.

In conclusion, when the load is the same, reduced server's capacity or timeout limit causes more timeouts. Large service time or smaller interarrival time causes heavier load, and causes more timeouts. Compared to constant interarrival time, interarrival time that is exponentially distributed with the same average value causes more timeouts.

5.4 Simulation Experiments with Network Latency

Network latency exits on real systems. It is the time it takes for information to be transferred between computers in a network. A minimum bound on latency is determined by the distance between communicating devices and the speed at which the signal propagates in the circuits (typically 70-95% of the speed of light). Actual latency is much higher, due to packet processing in networking equipment, and other traffic.

For atomic WS or CWS, the network latency can be included in the model in AnyLogic. A new component named as “delay” is added between different WSs and clients. Therefore, the network latency can be simulated.

The new model with “delay” used different interarrival time, different network latency in the “delay” component, and different capacity of the “delay” component. The results show the effects of network latency.

5.4.1 Simulation Model of Atomic WS with Network Latency

A simulation model of atomic WS with constant network latency is built. When network latency is non-constant, the rate at which requests arrive at the server is different from the rate at which requests are sent from the client. For example, when the interarrival time is constant, and the network latency follows exponential distribution, the rate at which requests arrive at the server is not constant anymore. A new arriving rate can be used to replace the old rate from the client and the latency in models, or non-constant network latency component can be added to the model. In this section, we focus on constant network latency.

In the model, the "delay" component is used to represent network latency. The "delay" component has two main parameters: the delay time that defines how long the network latency is, and the capacity of the "delay" component that means how many jobs can stay in the component at the same time. The capacity of the "delay" component should be large enough to simulate only network latency without considering packet processing in networking equipment, but it can also be small in models and be used to simulate some component that between the client and server that makes jobs delay in it and only has limited capacity. Such "delay" component with small capacity can be routers or some switch components in the network. In this section, both the delay time and the capacity of the "delay" component will be discussed.

When the capacity of the "delay" component is large enough, the constant network latency do not make results different from the results when there is no network latency. For example, when the capacity of the "delay" component is 1000, which is equal to the maximum number of requests being sent in an experiment, the results are the same as with no latency. (In fact, for the experiments in this section, the maximum number of jobs that are staying in the "delay" component is always no more than 5, according to the observation of simulation results.)

When the capacity of the “delay” component is not large enough, the results are different from results when there is no network latency. When the interarrival time is constant, if the time of network latency is equal or smaller than the interarrival time, the results that with network latency are the same as the results that do not include network latency; while if the time of network latency is larger than the interarrival time, the results that with network latency are different. For example, when the interarrival time is 1 second, if the time of network latency is 0.1 second, 0.5 second, 1 second, the results are the same as with no latency; if the time of network latency is 1.1 second, 1.2 second, 1.5 second, 2 second, and the capacity is only 1, the “new” interarrival time at the server end is actually changed into 1.1 second, 1.2 second, 1.5 second, 2 second, which is equal to the latency, and the interdeparture time changed into 1.3 second, 1.2 second, 1.5 second, 2 second, depending on the service time and the “new” interarrival time.

When the capacity of the “delay” component is not large enough, and the capacity is only 1 (additionally, if the capacity of the “delay” component is larger than 1, more than 1 requests can stay at the “delay” component, which makes things more complex), and then only one request can stay at the “delay” component at a time. If the time of network latency is smaller than or equal to the interarrival time, the newly arrived request can always come into the “delay” component immediately; while if the time of network latency is larger than the interarrival time, the newly arrived request have to wait for the request that already in the “delay” component to leave. If the time of network latency is noted as L, and the constant interarrival time is noted as A ($A<L$), the first request arrive at the “delay” component at time T, then the second request arrive at the “delay” component at time T+A. The second request has to wait until the time T+L ($T+L>T+A$) to enter the “delay” component, after the second request left the “delay” component.

The second request is delayed for time L before leave the “delay” component. Therefore, the first request leaves the “delay” component and arrives at the server at time T+L, and the second one arrives at (T+L)+L. The “new” interarrival time at the server end is actually changed into L (L>A), instead of A.

In summary, when the capacity of the “delay” component is only 1, and when the time of network latency is larger than the constant interarrival time, the “new” interarrival time at the server end is actually changed into the time of network latency, which makes the results different. While when the capacity of the “delay” component is large enough, or when the time of network latency is smaller than or equal to the constant interarrival time, the results are the same as with no latency.

When the interarrival time is exponentially distributed, the rule concluded above still holds, although the detail of results looks slightly different. For example, when the average value of interarrival time is 1 second, and the time of network latency is 1 second (with capacity is only 1), since the interarrival time is exponentially distributed, it can be smaller than, equal to, or larger than 1 at a certain time between two adjacent requests. When the interarrival time at a certain time is smaller than 1 second, “new” interarrival time at the server end is actually changed into the time of network latency, which is constant as 1 second; when the interarrival time at a certain time is equal to, or larger than 1 second, it remain unchanged and is still exponentially distributed. For the result of interdeparture time, when “new” interarrival time at the server end is 1 second, it is almost stable; when “new” interarrival time at the server end is still exponentially distributed, it is also exponentially distributed. Therefore, the result has some stable parts that is similar to the result when interarrival time is 1 second (then the result should be about 1.4 seconds), and also has some exponentially distributed parts that has different high

spikes that above 1.4 second. This can be shown in the result of experiment 1 below in this section (in Figure 5.36).

If the average value of interarrival time is still 1 second, and the time of network latency is only 0.5 second (with capacity is only 1), the probability that interarrival time at a certain time is below 0.5 second is smaller than it is below 1 second. Therefore, the probability that "new" interarrival time at the server end remains exponentially distributed is larger. This can be shown in the result of experiment 2 below in this section (in Figure 5.37).

Experiment 1, 2, and 3 shows the time of delay has effect on the result of interdeparture time. In experiment 1, 2, and 3, the average value of interarrival time is always 1 second, while the time of network latency is set as 1 second, 0.5 second, and 0.1 second, and the results of interdeparture time are different. When there is no network latency, the interdeparture time fits exponential distribution quite well. In experiment 1, the network latency is as large as 1 second, and the interdeparture time has many small parts that are almost stable. In experiment 2, the network latency is 0.5 second; the interdeparture time is still different from the result when it has no network latency. In experiment 3, the network latency is as small as 0.1 second; the interdeparture time looks more similar to the result when it has no network latency.

In experiment 4, I keep the time of delay as 1 second, but increase both the average value of interarrival time and the service time to twice the value as before. Therefore, the load is still 120% as in experiment 1, and the time of delay is half of the average value of interarrival time. As a result, the curve of interdeparture time of experiment 4 is very similar to experiment 1, too, except that the value of it is about twice the value in experiment 1.

In experiment 1, 2, 3, and 4, the capacity of the "delay" component is 1. That is, only one job can stay in the "delay" component. In experiment 5, the capacity of the "delay" component is

changed from 1 into 1000. Other parameters in experiment 5 are the same as in experiment 1: the interarrival time is exponential(1), service time is 1.2 seconds, and time of delay is 1 second. The result of experiment 5 is similar to the result when it has no network latency. The interdeparture time fit the exponential distribution quite well (in fact, it fits exponential(1.398741374)).

Here are the summary of parameters in experiment 1, 2, 3, 4, and 5:

Table 5.4: The summary of parameters in experiment 1, 2, 3, 4, and 5

Simulation experiment	Interarrival time (second)	Service time (second)	Time of delay (second)	Capacity of delay component
Experiment 1	Exponential(1)	1.2	1	1
Experiment 2	Exponential(1)	1.2	0.5	1
Experiment 3	Exponential(1)	1.2	0.1	1
Experiment 4	Exponential(2)	2.4	1	1
Experiment 5	Exponential(1)	1.2	1	1000

Here are the result of experiment 1, 2, 3, 4, and 5:

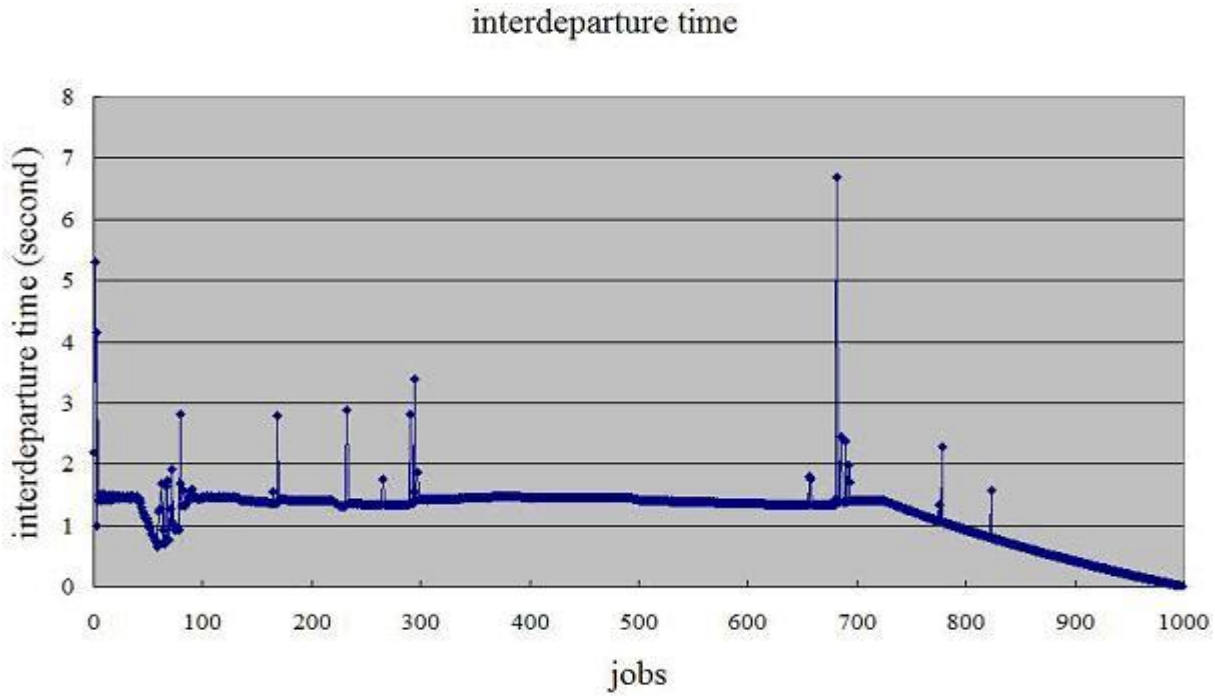

Figure 5.36: Result 1 - when the network latency is 1 second

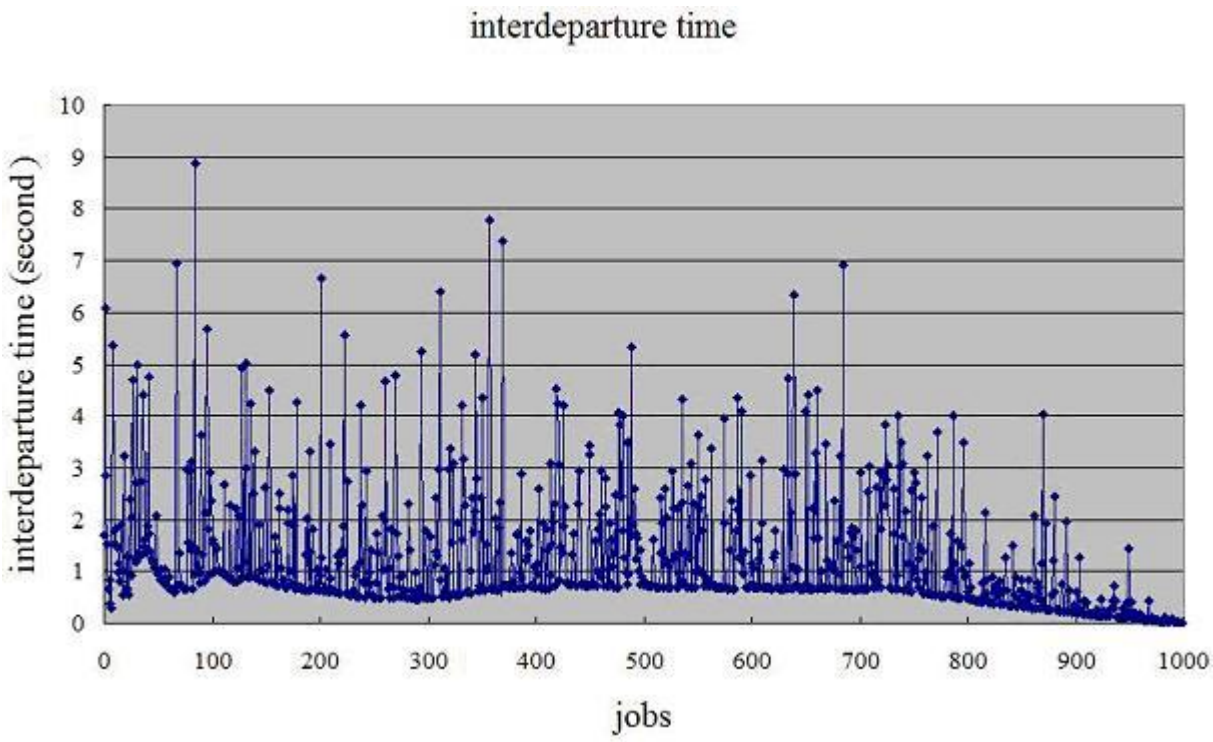

Figure 5.37: Result2 - when the network latency is 0.5 second

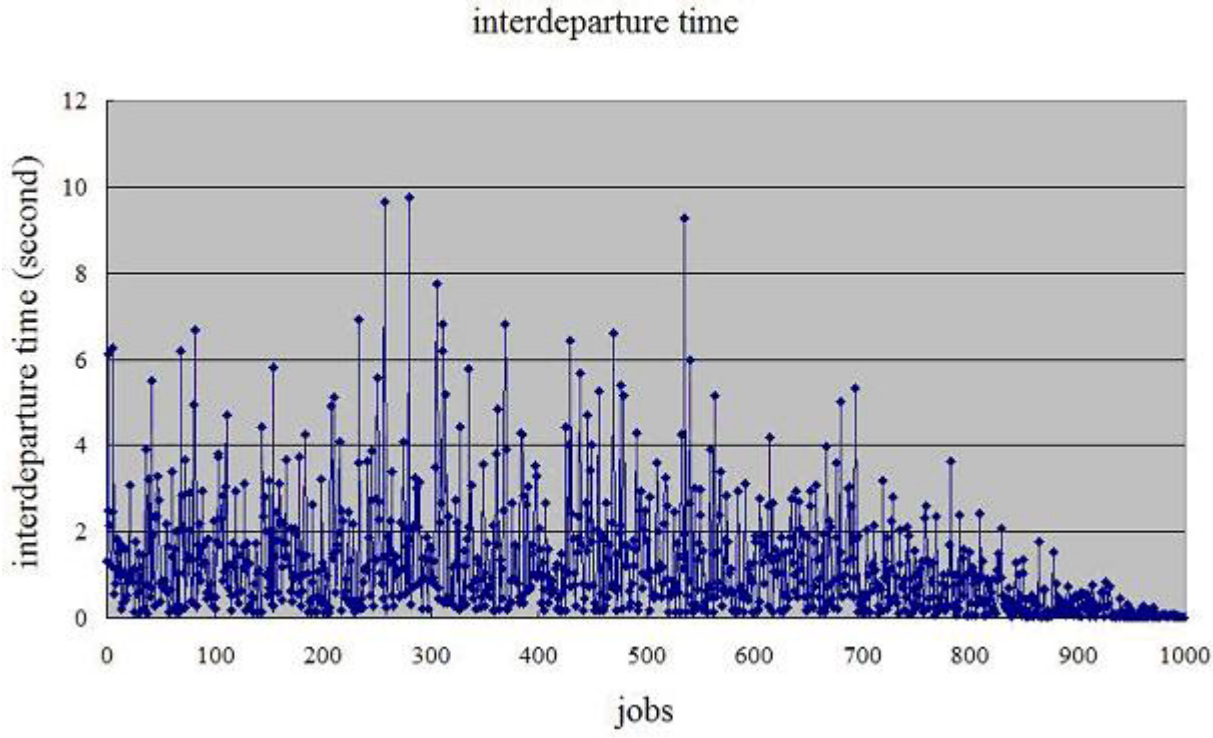

Figure 5.38: Result 3 - when the network latency is 0.1 second

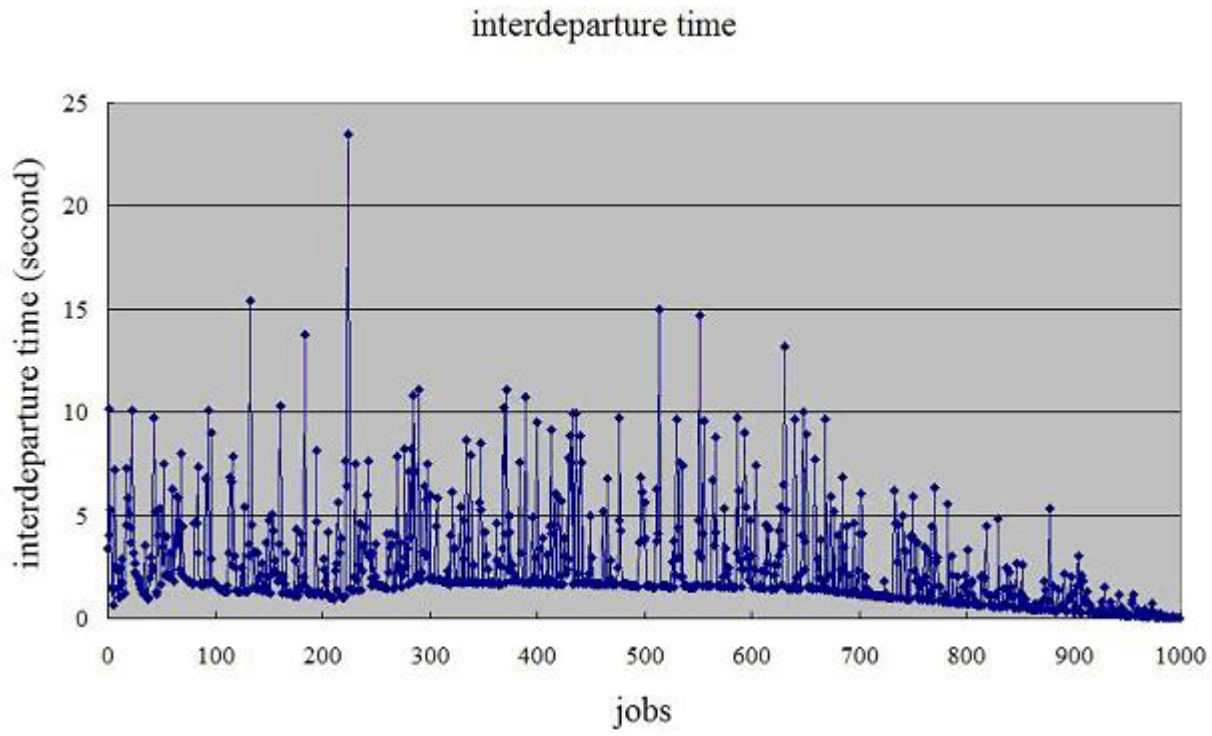

Figure 5.39: Result 4 - network latency is 1 second, but interarrival time is twice

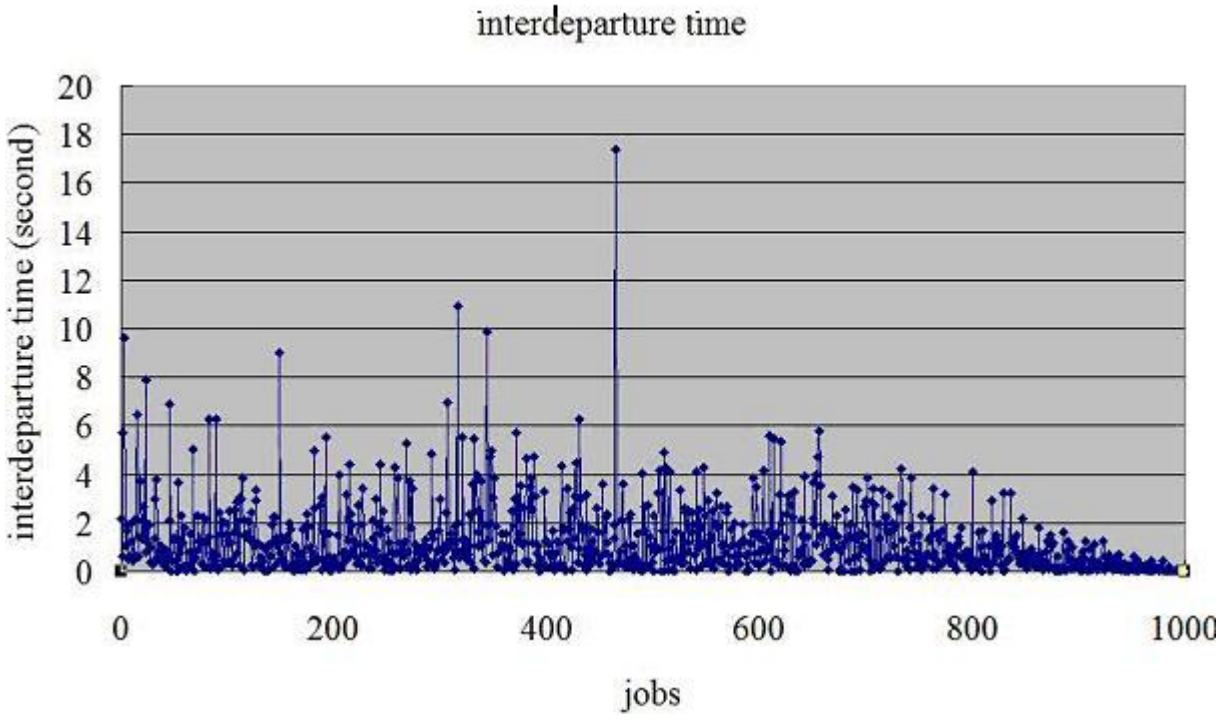

Figure 5.40: Result 5 - network latency is 1 second, but capacity is 1000

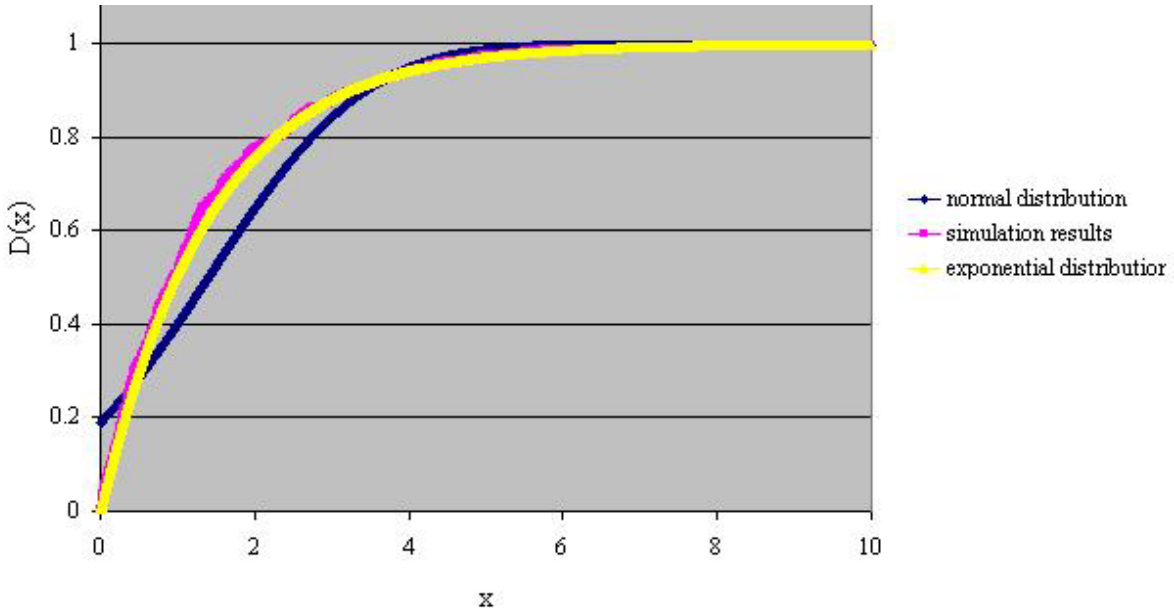

Figure 5.41: The distribution of results 5 when network latency is 1 second

5.4.2 Simulation Model of CWS with Network Latency

A simulation model of CWS with constant network latency is also built. The model of CWS has 5 WSs in a sequential workflow. Each WS's service time is 1.2 seconds. The interarrival time is constant as 1 second. Totally 1000 requests are sent to the workflow. Therefore, the model is similar to the CWS model of sequential workflow in previous sections; the only difference is: between the client and the service 1, and between service n and service n+1 (n=1, 2, 3, 4), there is a "delay" component.

The rule concluded in section 5.4.2 with atomic WS still holds. When the capacity of the "delay" component is large enough (such as 1000), or when the time of network latency is smaller than or equal to the constant interarrival time, the results are the same as with no latency; while when the capacity of the "delay" component is 1, and when the time of network latency is

larger than the constant interarrival time, the "new" interarrival time at the server end is actually changed into the time of network latency, which makes the results different.

The first experiment setting has 5 WSs in a sequential workflow, the capacity of the "delay" component is 1 only for the 1st service, and the capacity of the other "delay" components is 1000 for the 2nd, 3rd, 4th, and 5th service. The time of network latency for the 5 WSs is 1 second, and the interarrival time at the client end is constant at 0.5 second. In the result of the experiment, the "new" interarrival time at the 1st server is actually changed into the time of network latency, which is 1. This result is the same as the result that has constant interarrival time of 1 second, with a result of stable interdeparture time that is about 1.45 seconds at the beginning of the experiment.

The second experiment setting is similar to the first one, but the interarrival time at the client end is constant at only 0.1 second, the result is the same as that of the first experiment, since the "new" interarrival time at the 1st server is still actually changed into the time of network latency.

The third experiment setting is similar to the first one, but the capacity of the "delay" component is 1 only for the 2nd service, and the capacity of the other "delay" components is 1000 for the 1st, 3rd, 4th, and 5th service. The result is the same as that of the first experiment.

The fourth experiment setting is similar to the first one, but the interarrival time at the client end is exponentially distributed, with average value of 1 second. The result of the 1st service is very similar to the result shown in section 5.4.1 (in Figure 5.36), and the results of the 2nd, 3rd, 4th, and 5th service are changed with the result of the 1st service.

In summary, the results of CWS reconfirmed the rule concluded in section 5.4.2 with atomic WS.

Additionally, in this section, the model allows the requests will wait in other component, but not to be lost or discarded, even the capacity of the “delay” component is not large enough. Some complex situations such as discarding request packages are not included in this research. When the capacity of the “delay” component is larger than 1, but not enough, the “new” interarrival time can be different from the time of network latency, which need to be calculated according to detailed value of parameters. Such situations are not included in this section, too.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

The WS technology allows creating new dynamic applications. With the emerging languages (like BPEL), CWSs can be defined in different ways. However, since WSs can experience transient overload, the performance of CWS is difficult to predict.

This research aims to find the features of runtime behavior of WSs that are under transient load, build models that can simulate WSs well, and predict the performance of CWS. Also, the model parameters can be varied in order to find best setting of resources in CWS, which helps to design CWS, and to improve the performance with limited resources.

In this research, atomic WS as well as CWS are implemented on Java 6, Axis 1.4, Axis 2. The services are slightly overloaded with constant or exponentially distributed load. Sequential workflow is tested.

Models are built for atomic WS and CWS in AnyLogic to get simulation results. CWS in sequential workflow and with loops are built and tested. Constant or exponentially distributed load are used for the WSs.

From the result of experiments and simulation results, the following run time behavior of atomic WS and CWS are observed:

When the interarrival time is constant, and the total number of request is certain, the interdeparture time is not a constant. The interdeparture time has very short warm-up period, a stable period and a decreasing period. In simulation experiments results, the stable period has almost constant interdeparture time; while in real systems, the results in the stable period are

normally distributed. The results' average value is similar to the simulation results. "Noise" exists in real systems, and it can be added to the simulation models. Modified simulation models with "noise" in uniform distribution have results similar to experiments results, which is also normally distributed.

When the interdeparture time decreases, the next WS in a sequential workflow or loops will have jobs that arrive very fast, and experience heavy load. At this time, the next WS's job departures slow down, and a high strike of interdeparture time is shown in the results.

When the WSs in a sequential workflow have equal service time, the first WS experience heaviest load in most of the period of the experiments, and has most important impact on the performance of the workflow. The second, third, fourth, fifth services have less and less important impact on the performance of the workflow. Therefore, with limited resources for all the WSs in a sequential workflow, the first WS should have more resources, to improve the performance of the workflow. The best amount of resources that should be added to the first WS can be tested from simulation experiments with different parameters.

Loops in a workflow add additional load to the service that has loops. A CWS of 5 WSs in sequential workflow and with loops is studied. Since the first WS has most important impact on the performance of the workflow, and the second, third, fourth, fifth services have less and less important impact on the performance of the workflow, loops on the first WS has most important impact, too. However, when the second, third, fourth, fifth WS loops for only a couple of times, the results can be the same as the results when there is no loop. When the second, third, fourth, fifth WS loops for many times, the results are different from the results when there is no loop.

When the interarrival time of atomic WS is exponentially distributed, the result of interdeparture time is also exponentially distributed. The experiments results and simulation results are similar. No “noise” needs to be added to the simulation model.

The experiments results with “noise” show chaotic behavior of WSs. For atomic WSs, the simulation results are simple and clear, and the experiment result can be identified as normally distributed, and its average value is similar to the simulation results. For a CWS with three or more WSs in a sequential workflow, the simulation results have a few spikes, and the pattern of experiments results with “noise” are not very easy to identify. When loops are added to the CWS, the simulation results are complex, and the experiments results with “noise” are more difficult to understand. When the number of WSs in a CWS is larger, and when the load is exponentially distributed or “bursts”, the behavior of WSs can be quite complex. In summary, the WSs have chaotic behaviors when they are overloaded. When the workflow is longer with more WSs, and when loops exist in the workflow, the situation is even worse. Therefore, it is important to avoid overload in some important situations.

When the queue of the WS has timeout limit and limited queue size, and when the maximum number of threads in the server is limited, timeout can happen when the WS is overloaded. Timeout limit, queue size, maximum number of threads in the server, and load can change to number of timeouts that happen in an experiment.

When there is no workflow server in a CWS, if the WS is only slightly overloaded, no timeout happens or only a few of timeouts happen; while if the WS is in quite heavy load, the number of timeouts is very large. When adding a workflow server to the sequential workflow, the number of timeouts is relatively small, when the WS is slightly overloaded or in quite heavy load.

However, when WS is overloaded, the expected decreasing period of the interdeparture time is not shown in the results of CWS with a workflow server.

Network latency is studied with models in AnyLogic. It has no impact on the atomic WS and CWS when their load is constant or exponentially distributed load. However, when the load is exponentially distributed, if the network latency is simulated with a "delay" component with limited capacity, a few of the request may be stocked, and the results are different from the results when no network latency exists.

The experiments results are compared to simulation results for validation, and were found to be similar. Some complex situation of CWS is not implemented in real systems, but some of them are implemented and tested in simulation models, and get simulation results. More simulation models can be built and run in AnyLogic, and the results can be used to predict the behavior of CWS.

Based on the above, the most valuable contributions of the thesis are:

- Studying the features of runtime behavior of WSs that are under transient load.
- Building models that can simulate WSs well and predict their performance.
- Finding best settings for CWS to improve the performance with limited resources.
- Identifying chaotic behaviors in CWS.

In the future, more complex scenarios of CWS need to be studied. More services will be used in the workflow, and the workflow can be more complex patterns with loops and split. The services can be implemented on the same computer, or computers that are far away from each other. The requests to a service may come from N (N can be a large number, e.g., 20 or even larger) clients, and the arrival rate of each client can be different. Therefore, the load will be heavier and more complex. The time when timeout happens in real systems can be recorded, and

studied. The platform of services in a workflow can be different, and the timeout limits and queue sizes can be different, too.

REFERENCES

[1] The ActiveBPEL engine. http://www.active-endpoints.com/active-bpel-engine-overview.htm.

[2] Amazon Web Services, http://www.amazon.com/gp/aws/landing.html.

[3] Apache Axis 1.4. http://ws.apache.org/axis/.

[4] Apache Axis2/Java Version 1.2. http://ws.apache.org/axis2/.

[5] Apache Tomcat. http://tomcat.apache.org/.

[6] BPWS4J engine. http://www.alphaworks.ibm.com/tech/bpws4j.

[7] Business Process Execution Language (BPEL). http://www.service-architecture.com/web-services/articles/business_process_execution_language_for_web_services_bpel4ws.html.

[8] G. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Decentralized orchestration of Composite Web Services. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 134-143, May 17–22, 2004.

[9] G. Chafle, S. Chandra, N. Karnik, V. Mann, and M. G. Nanda. Improving performance of Composite Web Services over a wide area network. In *Proceedings of the 2007 IEEE Congress on Services*, pages 292-299, July 9-13, 2007.

[10] S. Chandrasekaran, G. Silver, J. A. Miller, J. Cardoso, and A. P. Sheth. Web Service technologies and their synergy with simulation. In *Proceedings of the Winter Simulation Conference*, Vol. 1, pages 606 – 615, Dec. 8-11, 2002.

[11] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web Services Description Language (WSDL) Version 2.0. http://www.w3.org/TR/wsdl20/, June 2007.

[12] T. Clements. Overview of Soap. http://java.sun.com/developer/technicalArticles/xml/webservices/, Jan. 2002.

[13] R. W. Conway, W.L. Maxwell, and L. W. Miller. *Theory of Scheduling*. Addison-Wesley, 1967.

[14] D. Dyachuk and R. Deters. The impact of transient loads on the performance of service ecologies. In *Proceedings of the 2007 Inaugural IEEE International Conference on Digital Ecosystems and Technologies*, pages 245 – 250, Feb. 21-23, 2007.

[15] D. Dyachuk and R. Deters. Improving performance of Composite Web Services. In *Proceedings of IEEE International Conference on Service-Oriented Computing and Applications*, pages 147 – 154, June 19-20, 2007

[16] D. Dyachuk and R. Deters. Optimizing performance of Web Service providers. In *Proceedings of 21st International Conference on Advanced Information Networking and Applications*, pages 46 – 53, May 21-23, 2007.

[17] D. Dyachuk and R. Deters. Transparent scheduling of Web Services. *In Proceedings of Web Information Systems & Technology,* pages 112-119, 2007.

[18] Enterprise Java Beans Specification (EJB) 2.1. http://java.sun.com/products/ejb/.

[19] G. Gehlen and L. Pham. Mobile Web Services for peer-to-peer applications. In *Proceedings of the Consumer Communications and Networking Conference,* pages 427-433, Jan. 3-6, 2005.

[20] J. M. Govern, S. Tyagi, M. Stevens, and S. Mathew. *Java Web Service Architectur*e. Morgan Kaufmann, 2003.

[21] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation,* Volume 4, pages 22-22, October 22 - 25, 2000.

[22] H.–U. Hess and R. Wagner. Adaptive load control in transaction processing systems. In *Proceedings of the 17th International Conference on Very Large Databases*, pages 47-54, Barcelona, Spain, September 1991.

[23] http://www.ibm.com/developerworks/java/library/j-jtp11253/.

[24] H. Huang and R. A. Mason. Model checking technologies for Web Services. In *Proceedings of the Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems and the 2006 Second International Workshop on Collaborative Computing, Integration, and Assurance.* 6 pages, April 27-28, 2006

[25] Java SE 6. http://java.sun.com/javase/6/.

[26] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, 1984.

[27] N. Mitra. Y. Lafon. Soap Version 1.2 Part 0: Primer (Second Edition). http://www.w3.org/TR/soap12-part0/, April 27, 2007.

[28] M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing execution of Composite Web Services, In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 170 – 187, October 24-28, 2004.

[29] NetBeans IDE. http://www.netbeans.org/.

[30] E. Newcomer and G. Lomow. *Understanding SOA with Web Services*. Addison-Wesley, 2004.

[31] Probability and Statistics. http://mathworld.wolfram.com/.

[32] P. Savur and M. Sum. Business Process Execution Language, Part 2: partnerLinkType and partnerLink. http://developers.sun.com/jsenterprise/nb_enterprise_pack/reference/techart/bpel2.html, October 30, 2006

[33] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Transactions on Internet Technology*, 6(1): 20–52, February 2006.

[34] W. E. Smith. Various optimizers for single-state production. *Naval Research Logistics Quarterly*, 1956.

[35] I. Stojmenovic. Recursive algorithms in computer science courses: Fibonacci numbers and binomial coefficients. *IEEE Transactions on Education*, 43(3): 273-276, August 2000.

[36] W. T. Tsai, D. Zhang, Y. Chen, H. Huang, R. Paul, and N. Liao. A software reliability model for Web Services, *SEA 2004*, pages 144-149, 2004.

[37] UDDI data structure reference v1.0. http://uddi.org/pubs/DataStructure-V1.00-Published-20020628.pdf, Jun 2002.

[38] UDDI version 3.0.2, http://uddi.org/pubs/uddi_v3.htm, Oct. 2004.

[39] W. M. van der Aalst. Workflow verification: finding control-flow errors using Petri-net-based techniques. *Business Process Management*, pages 161–183, 2000.

[40] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3): 5-51, 2003.

[41] B. Wassermann. Business Process Execution Language, http://sse.cs.ucl.ac.uk/.

[42] Web Services Business Process Execution Language Version 2.0, Committee Draft, 01. http://www.oasis-open.org/committees/download.php/14616/wsbpel-specification-draft.htm, September 2005.

[43] Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/wsdl#_wsdl

[44] XJ Techologies. AnyLogic 5.5. http://www.xjtek.com/.

[45] L. Zhu, I. Gorton, Y. Liu, and N. B. Bui. Model driven benchmark generation for Web Services. In *Proceedings of the 2006 International Workshop on Service-oriented Software Engineering*, pages 33-39, May 27–28, 2006.