Checkpointing Web Services

A thesis submitted in 2015 to the University of Hyderabad in partial fulfilment of the award of a Ph.D. degree in School of Computer and Information Sciences

by

CH.VANI VATSALA

SCHOOL OF COMPUTER AND INFORMATION SCIENCES
UNIVERSITY OF HYDERABAD
(P.O.) CENTRAL UNIVERSITY
HYDERABAD - 500 046, INDIA

April 2015

CERTIFICATE

This is to certify that the thesis entitled "**Checkpointing web services**" submitted by **CH.Vani Vatsala** bearing Reg. No. 09MCPC02 in partial fulfilment of the requirements for the award of **Doctor of Philosophy in Computer Science** is a bonafide work carried out by her under my supervision and guidance.

The thesis has not been submitted previously in part or in full to this or any other University or Institution for the award of any degree or diploma . It is also free from any plagiarism.

Prof Arun K.Pujari
Dean
School of Computer and
Information Sciences
University of Hyderabad

Prof Hrushikesha Mohanty
Supervisor
School of Computer and
Information Sciences
University of Hyderabad

DECLARATION

I, **CH.Vani Vatsala**, hereby declare that this thesis entitled "**Check-pointing web services**" submitted by me under the guidance and supervision of **Prof. Hrushikesha Mohanty** is a bonafide research work and is free from any plagiarism. I also declare that it has not been submitted previously in part or in full to this University or any other University or Institution for the award of any degree or diploma. I hereby agree that my thesis can be submitted in Shodganga/INFLIBNET.

A report on plagiarism statistics from the University Librarian is enclosed.

Date:

Name: **CH.Vani Vatsala**

Regd. No. **09MCPC02**

//Countersigned//

Signature of the supervisor:

ACKNOWLEDGEMENTS

I would like to start this section by expressing my deepest gratitude to my supervisor **Prof. Hrushikesha Mohanty** whose valuable guidance and supervision has enabled me to carry out the research and complete this dissertation. I am immensely thankful to his continuous encouragement and motivation through out. This dissertation is the outcome of numerous formal and informal discussions with him. These discussions helped me gain thought provoking insights without which this research would have been impossible. More importantly, apart from technical know-how, I could learn the importance of ethics and values that we, as human beings, have to follow. I am extremely indebted to him for his wonderful guidance. I am blessed to have him as my guide.

I express my heartful of thanks to my doctoral review committee members **Prof. Chakravarthy Bhagvati** and **Prof. Rajeev Wankar** for meticulously analysing my research work and giving me helpful suggestions. Their constructive criticism and illuminating views have improved the contributions of this work.

I would like to thank dean, **Prof. Arun.K.Pujari** for providing facilities and infrastructure to carry out this research.

I would like to thank **Prof. Madhusudan Reddy**, advisor of CVR college of engineering and, **Dr. C.V Raghava**, chairman of CVR college of engineering for providing me facilities like weekly offs and study leaves without which carrying this research work would have been a herculean task. I am extremely thankful for their constant encouragement and support. I would also like to thank **Prof. L.C.Siva Reddy**, HOD, CSE dept, CVR college of engineering for extending his help and support through out.

I express my warm thanks to my husband **A.Narasimha Rao** and daughters **Ramya** & **Sreya** for understanding my research commitments and for wishing me good luck through out. I am extremely thankful to my **father, mother, mother-in-law, sisters and sisters-in-law** for extending their constant support, love and encouragement.

It is my best friend **H.N.Lakshmi**, who has actually inspired me to pursue my Ph.D at University of Hyderabad. I take this opportunity to thank her for giving such an invaluable suggestion and for being such a wonderful friend. She has extended her help and rendered moral support on innumerable occasions. Our hour long discussions in car, about everything right from technical to political issues, family to college matters, remain as beautiful memories to cherish.

B.Rajitha, my closest friend, has extended moral support all the time and has been a constant source of motivation. Her wise suggestions have helped me take right decisions many a times. I express my heartful thanks to her. I would also like to thank my friend **Bhagya Sree** for encouraging me through out.

I extend my thanks to my friends and lab mates **V.Supriya, A. B. Sagar** and **Abhaya Pradhan** for their help and support through out. Their timely help several times has made my job easier. Also, our discussions with our supervisor over tea shop are really unforgettable.

April 2015 **CH.Vani Vatsala**

ABSTRACT

A web service is a piece of software that provides a service and is made available on Internet. If a web service implements a business process without invoking other web services then it is called an atomic web service. Otherwise it is a composite web service. Web services which are part of a composite web service are called as constituent web services. They exchange messages among each other to accomplish a business task and this sequence is specified in a design document called as choreography document. A composite web service whose message exchange sequence is given in a document is called as choreographed web service. After the initial stage of designing a composite web service, the actual constituent services would be developed and then deployed on their respective web servers.

Since constituent services of a composite web service operate on Internet where reliability cannot be guaranteed, providing fault handling to composite web services is of primary importance. Web service composition languages provide means for exception handling, there is still need for handling transient faults to provide resilient execution, as transient faults are asynchronous in nature. Checkpointing and recovery is a time tested technique which has been applied successfully to several applications in distributed computing and databases to recover from transient faults. In this research work we attempt to provide fault handling to choreographed web services using checkpointing and recovery. We capture essential elements of a choreography in our interaction pattern model. We identify different sequences in which web services interact with each other and term them as patterns of interaction. We then use these patterns to identify checkpoint locations.

We propose checkpointing in three stages. In the first stage we propose check-

pointing locations using choreography document. In this stage checkpoints are placed in such a way that web services performing non repeatable actions are not invoked again in the event of failure of the invoking web service.

Web services provide services to their consumers in accordance with terms and conditions laid down in a document called as Service Level Agreement (SLA). In second stage of checkpointing called as deployment time checkpointing, we propose checkpointing locations in a composite web service such that time and cost deadlines as specified in SLA are met even in case of transient failures and subsequent recovery. Web services advertise their Quality of Service attribute values in order to aid consumers in selecting web services that suit their requirements. We use these advertised values in deciding checkpoint locations. These checkpoints and checkpoints generated at design time are inserted into web services in their code before they are deployed on their servers.

For certain web services, actual QoS values tend to deviate from advertised QoS values (particularly response time values). For certain other composite web services, some of the constituent web services are selected dynamically. In such cases, there is a need to revise checkpointing locations at run time for efficient execution of web services. Hence we propose run time checkpointing which revises checkpointing locations in a composite web service at run time. It uses predicted response time values to decide on checkpoint locations. Hence we propose two methods for predicting response time values of web services: 1)Black box approach that uses HMM (Hidden Markov Model) and 2)White box approach that uses HMM and Bayesian networks. We establish the accuracy of our prediction empirically too.

Thus this thesis provides a scheme of three stages for finding checkpoint locations. These three stages comprise design, deployment and execution stages of web services. The implication of the proposed scheme is analysed experimentally. In a nutshell, this thesis provides a comprehensive approach for checkpointing web services.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ACGT	Automatic Checkpoint Generation Tool
API	Application Programming Interface
BPEL	Business Process Execution Language
DOM	Document Object Model
DTC	Deployment Time Checkpointing
EM	Expectation Maximization
FMM	Failure Management Module
HMM	Hidden Markov Model
IDE	Integrated Development Environment
IET	Instruction Execution Time
MAE	Mean Absolute Error
PM	Prediction Middleware
PRT	Predicted Response Time
QoS	Quality of Service
RMSE	Root Mean squared Error
RT	Response Time
RTC	Run Time Checkpointing
SAX	Simple API for XML
SD	Statically Discovered set of services
SOA	Service Oriented Architecture
SPM	Server side Prediction Middleware
UDDI	Universal Description, Discovery and Integration
UML	Unified Modeling Language
WS	Web Service
WSDL	Web Service Description Language

WTCPT Waiting Time Conditional Probability Table

XMI XML Metadata Interchange

XML Extensible Markup Language

NOTATIONS

ξ	Web service participating in a choreography, participant in short
ξ_s	Participant that sends a message
ξ_r	Participant that receives a message
σ	Operation performed by a participant
τ	Local action
γ	Interaction
Ψ	Sequence of operations
p	A pattern
ps	A pattern string
ip	Invocation Point
sp	Service Point
vp	Must Save Point
s	Sequential Component
r	Recovery Component
T_C	Checkpointing time
T_L	Message logging time
T_R	Message replay time
T_{CR}	Time to restore a checkpointed state
$s.t_{rt}$	Response time of a component s
$s.vl$	Vulnerability of a component s
$s.ct$	Cost of service provided in s
$T_{pure}(\xi)$	Pure execution time of a component ξ
$\alpha_k(\xi)$	Failure free execution time of ξ with k deployment time checkpoints
$T_{W_k}(\xi)$	Total execution time with failure recovery with k deployment time checkpoints
$T_{rec_k}(\xi)$	Recovery time overhead of ξ with k deployment time checkpoints
$C_{rec_k}(\xi)$	Recovery cost overhead of ξ with k deployment time checkpoints
C_{total}	Failure free cost of ξ
$r.cs$	Checkpointing score of recovery component r
pt	Prediction tuple
λ	Failure rate of ξ
Θ	Parameters of HMM
Π	Initial Probability
A	State Transition Matrix
B	Emission Probability Matrix
O	Observation sequence
u	Input vector

CHAPTER 1

Introduction

1.1 Introduction to Web services

A web service is a piece of software that provides a service and is accessible over Internet. The role of web services in today's world is continuously growing. This is not only due to the use of web services by organizations having distributed operations but also for its operational ease. This ease of operation is achieved because web services implement what is called as Service Oriented Architecture (SOA).

Service-oriented architecture is a form of distributed systems architecture (Berson and Alex) which is basically characterised by separation of service specification from service implementation. The main stakeholders in this architecture are: service provider, service consumer and a registry of services. Service providers who want to make their services public, publish about their service in a registry. Service consumers search for required services in the registry and invoke a suitable service. Figure 1.1 depicts SOA framework.

SOA and hence web services have the following properties: 1) Only a logical view of actual programs (called operations) which implement business processes and databases is made public. The logical view defines what these operations do but does not specify how they do. 2) A published service operation can be invoked by a service consumer using platform independent messages (called **interactions**). These messages and operations are described in a document called as WSDL which is published in the registry. These properties enable web services to provide reusable business functionality irrespective of the platform on which they are working.

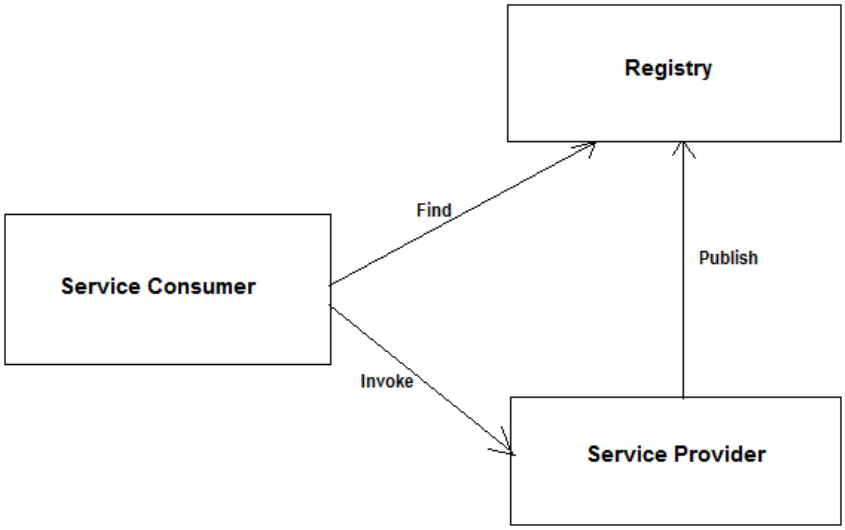


Figure 1.1: Service Oriented Architecture

A web service that implements a business process, without invoking another web service, is called as **atomic web service**. A simple business process may be accomplished by invoking only one web service. But complex business processes are built by discovering, invoking and/or composing other services available on web. These composite applications may expose themselves as services on web, resulting in what are known as service compositions or **composite services**. The web services which participate in such a composition are called as **constituent services** or simply, **participants**.

Two main terms used frequently in the area of web service composition are **Choreography** and **Orchestration**. In order to make our discussion complete, we next continue explaining and differentiating these two concepts in the following subsection.

1.1.1 Choreography and Orchestration of Web services

Choreography and Orchestration are the two words which are used most oftenly in the context of web service composition. The word **Choreography** is used to

describe a collaboration among multiple services which interact with each other in order to achieve a common business goal. The description includes a list of roles of participating services, sequence in which messages are to be exchanged for interacting with each other and data elements to be exchanged. But choreography does not describe the implementation details of any individual service.

The described choreography model may be implemented in two ways (Refer to figure 1.2): 1) There is a central coordinator which controls the sequence in which participating services are invoked. Such a composition with a central coordinator is called as an **Orchestration**. This type of centralised orchestration is particularly advantageous when amount of data that is transferred between services through central coordinator is not voluminous. But if the amount of data transmitted is huge, or if such a centralised implementation is not feasible (due to business level constraints) choreography model has to be implemented in a decentralised fashion. 2) In decentralised implementation all the participating services are treated equally, with no central coordinator controlling the sequence of interactions. Each service is aware of its role in its interactions with the partner services. No single service has a global view of the overall integration. Such a decentralised implementation is less commonly called as Choreography.

In this work, we use the term "Choreography" to refer to design document that describes a composition. In the following section, we describe three main stages in the development of a composite web service.

1.2 Engineering a Composite Web Service

A business application to be developed as a composite web service is engineered as follows:

1. **Design:** As a first step, application design is specified in a document called as choreography document Barros *et al.* (2005). Choreography specification

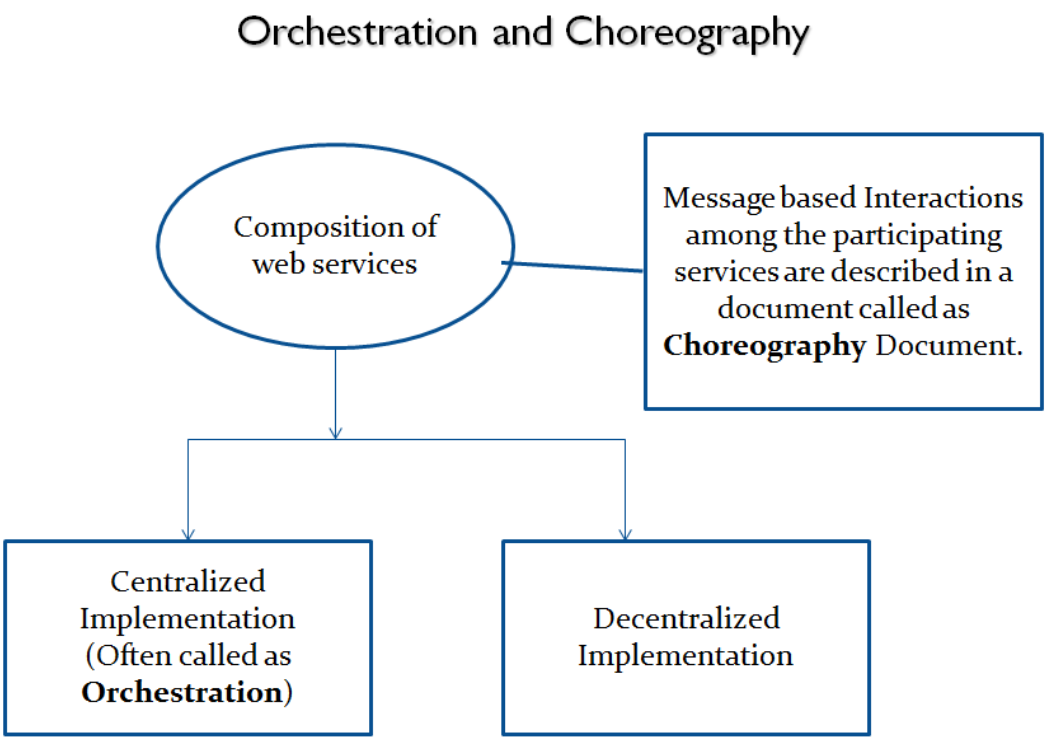


Figure 1.2: Orchestration and Choreography of Web Services

is generally given in languages like: WSCDL (Nickolas Kavantzas (2005)), BPMN (OMG (2011)), UML Activity Diagram. Choreography specified in any one of the above can be converted into another. For the purpose of discussion, we use design documents specified using UML Activity diagrams. In this design document specification of the following is given: 1)List of roles participating in the composition. For each participating role, the actions to be performed are also specified. Some of these actions are repeatable (ex: a simple computation), while others are non repeatable (ex: database write). 2) Sequence of interactions(message exchanges) 3) Details of data items exchanged in each of the interactions.

Refer to Figure 1.3 for an example choreography (*Online Booking*). In this choreography, participants are: *Online Booking service, Payment Service, Theatre Service* and *client*. The choreography describes message exchange sequence to book tickets online for a theatre. Actions like *Fetch details,*

Query Seat availability are repeatable and, *Book seats and Receive payment* are non repeatable actions. Information on non repeatable actions in a composite service is used later in this thesis to decide checkpointing locations in the service.

2. **Selection, Composition and Deployment:** Next step is to select the exact web services which implement various roles in the choreography. These web services would be either those services that are newly developed or already available services over the Internet. After composing the selected services into a composite web service, operations of the composite service are published in a WSDL (Roberto Chinnici (2007)).

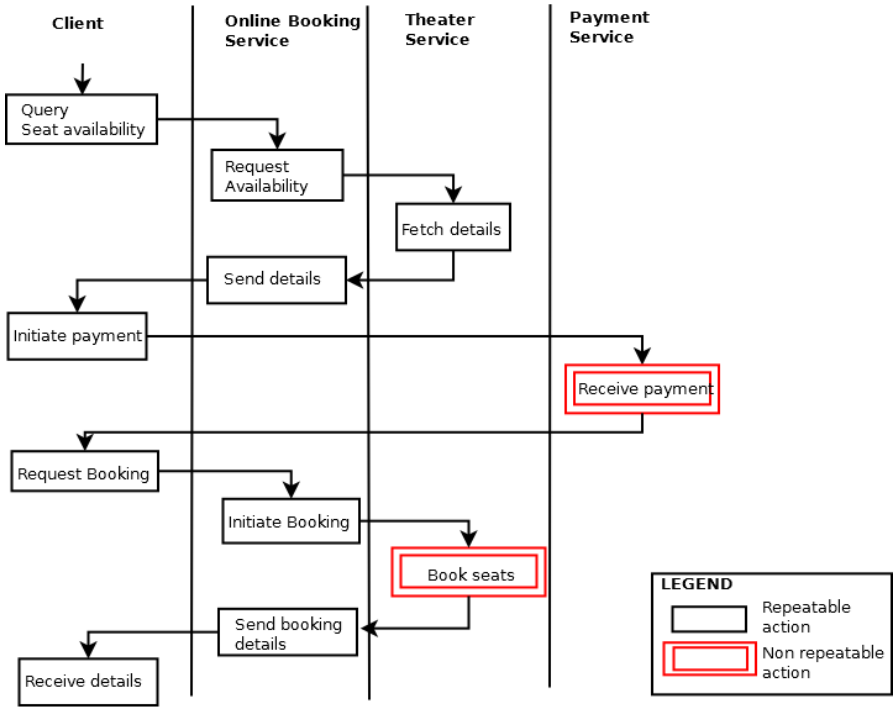


Figure 1.3: Example choreography: Online Booking

When a web service is installed on its web server and is ready to accept requests, we say that the web service is **deployed**. We say that a composite service is deployed when its constituent services are deployed on their respective servers. In case of orchestration, central coordinator also has to be deployed. If all the constituent web services are selected before deployment

then the composition is called as **static composition**. Web services framework allows selection of some of the constituent services at runtime also. In such a case, the composition is called as **dynamic composition**.

In any case, the service provided should be reliable, accurate, quick and robust. With the availability of abundant functionally similar services over the web, it becomes difficult for service consumers to select a particular web service. Web services advertise their Quality of Service(QoS) attributes like response time, cost of service, availability and reliability, trust etc to aid service consumers in selection of suitable services.

3. **Execution:** When a service consumer discovers, selects (using its QoS or otherwise) and invokes a composite web service, all the constituent services are executed in the sequence given in the design document. Upon successful execution of the services, we say that the service request is successfully completed or processed.

Web services operate over the Internet where reliability and speed cannot be guaranteed, hence providing fault handling mechanism to web services is of primary importance. In the following section, we introduce and discuss this issue.

1.3 Fault Management in Web services

Web Services use service-oriented architecture using which complex business processes are built by discovering and integrating services already available over the Internet. Web services need to be equipped with a fault handling mechanism to provide reliable services over the unreliable and dynamic Internet.

A web service is said to fail when it does not deliver the requested service or when the service delivered is not what is expected. A fault is an abnormal condition or a defect in a web service or in its environment which may cause a

service failure Avizienis *et al.* (2004). There are several faults that may appear in web services Chan *et al.* (2009), Liu *et al.* (2010):

- **Development or Content faults**: Bugs in coding and design come under this category. The service being invoked returns incorrect values. Ex: A book-query service returns incorrect book price given a book title.
- **Logical faults**: These faults are listed in WSDL document of a service. They are predefined faults, specified by application designers. They are thrown deliberately by the service on unsuccessful completion. Ex: A Loan cannot be granted due to ineligibility of a customer.
- **Temporary hardware or transient faults**: Power failure or link failure, temporary unavailability of servers. These faults are asynchronous, and self healing techniques like rollback and recovery enable the system to carry forward work. This thesis considers these types of faults and prescribes solutions for.
- **Permanent hardware faults**: Server hosting the service crashes down.
- **Interaction faults**: Parameter or interface incompatibilities come under interaction faults.
- **SLA faults**: SLA (Service Level Agreement) specifies deadlines for expected response time, cost of service, reliability etc upon which both the service consumer and service provider agree. These faults occur when the invoked service does not satisfy SLA. Ex: A service gives back its response within 20 seconds whereas according to SLA it must give back its response within 10 secs.

Content faults and interaction faults need modification in code without which they are bound to resurface. Permanent hardware faults can be handled using mechanisms like redundant servers. Logical faults are various situations that arise

in business applications. Composite web service development languages like BPEL provide fault(exception) and event handlers which execute specified actions when a predefined fault/event occurs at run time. The faults and corresponding actions to be taken to handle those faults have to be specified explicitly by application designers.

The proposed research aims to consider situations that arise due to **transient faults** which are inadvertent and unavoidable in nature. The proposed solution to handle transient faults is generic in nature (can be applied to all types of applications irrespective of their domain or implementation platform) and does not require the intervention of application designers.

When transient faults terminate the execution of a web service, the service is to be quickly restored back for continuing its execution. But this recovery has to be done in a manner that is transparent to the user. Transient faults result in SLA faults when recovery actions take considerable additional time and cost. Hence web services have to take necessary steps to avoid SLA faults (violations), and this should be the case even when transient faults occur. It becomes very important for the service providers to avoid or reduce the possible delays generated due to transient faults. Hence, in this thesis, we present an approach that enables service providers to satisfy constraints on the two most important QoS attributes: **time and cost**. The approach can be extended to avoid violations of constraints on other QoS attributes like reliability, trust etc.

In order to recover from transient faults the techniques available in the literature can be categorised into two types Avizienis *et al.* (2004): Backward recovery and Forward recovery. In backward recovery technique a web service is restored back to a previously saved state(**checkpoint**) when a fault occurs during its execution. The failed web service continues its execution from the checkpointed state. In forward recovery techniques, instead of rolling back to a saved state, other alternatives are tried to make the composite service continue its execution. Invoking functionally equivalent services H.E.Mansour and T.Dillon (2011); Ezenwoye and

Sadjadi (2007); Liu *et al.* (2010); Ben Halima *et al.* (2008) (**substitution**), designing fault compensation are the commonly used forward recovery techniques in web services. In the following subsections we discuss each of them and their applicability to composite web services.

1.3.1 Checkpointing

Checkpointing is a good old strategy to achieve fault tolerant computing. Checkpointing is a time tested technique that has been used in several areas like databases, distributed computing etc for handling transient faults. **Checkpointing** is a proactive technique which prescribes to save the state of an application so as to enable its recovery in case of any failure of the application at a later time. A failed application rollsback to a previously checkpointed state and continues its execution from there on. Checkpointing and recovery can be applied to web services also, due to the following reasons:

- Checkpointing and recovery is a general scheme and is **application or platform independent.**
- Checkpointing and recovery is efficient in case of recovery from *transient faults* because when a service is rolled back and is continued from a checkpointed state, transient faults do not resurface.
- In case of *permanent server crash*, the approach of checkpointing all executing service instances and availability of redundant servers will greatly reduce the amount of rework. All the running instances should be restored from their latest checkpoints and their execution should be resumed on the available redundant server.

Several distributed checkpointing algorithms Chandy and Lamport (1985); Silva and Silva (1992); Netzer and Xu (1995); Elnozahy *et al.* (2002); Koo and Toueg (1987); Bronevetsky *et al.* (2003); Robert *et al.* (2012); Lin and Ahamad

(1990); Netzer *et al.* (1997) have been proposed in literature, but they are not readily applicable to web services. A survey of these algorithms and detailed reasoning for their non applicability to web services is presented in section 2.2.1.

In the following subsection we discuss forward recovery techniques used for recovery of web services and their limitations.

1.3.2 Forward recovery and its limitations

Substitution and fault compensation are the two forward recovery techniques mostly used to handle transient faults (referred to as faults from here on) in web services. When a service is invoked by another web service (say caller), there is always a possibility that either the caller or the invoked service fail. In the event of *failure of the invoked service*, forward recovery using substitution may be employed by the caller. The invoked faulty service is substituted with an equivalent web service by the caller. But this *approach fails when the calling web service itself fails*. The reason is articulated in the following paragraph.

A **service instance** is a web service in execution. When two or more requests for a web service are received simultaneously from users, multiple instances of this service would be initiated. Suppose a web service A invokes another web service B to fulfill the requests that the service A receives from its users. At a given point of time, the web server hosting the service A would be executing several instances of A, some of which would have already placed a call to service B (Here service A is the caller and service B is the callee). At this point of time if the web server hosting service A fails, all running instances of the service A would also fail. Failed instances of the caller would have to be reexecuted from the beginning. In case of failure of the caller, substitution mechanism is not applicable. Reexecution from the beginning results in recalling the service B again in some cases, resulting in increased execution times. In such cases checkpointing technique, which avoids reinvocation of other web services, greatly improves the performance of failed

executions.

Fault compensation specifies actions that have to be executed to negate the effect of an already executed activity which is followed by an unsuccessful activity. In cases where reexecution helps completing an activity, compensation is not the required solution.

Hence checkpointing can be seen as a promising solution that can handle transient faults in web services. In the following subsection we present the issues to be considered while checkpointing web services.

1.4 Checkpointing web services: Issues to be considered

As discussed in the previous section, checkpointing a web service makes it capable of withstanding transient faults and enables it to successfully service requests with in stipulated cost and time. Any web service checkpointing scheme has to consider the following important characteristics of web services:

1. **Composite nature of a web service and non repeatability of actions**: When a business application is implemented by a set of web services working in tandem (by invoking each other), they execute and interact with each other in a specified sequence. Any checkpointing strategy should first capture this sequence so as to identify checkpoint locations in the composition. These checkpoint locations should be such that: a rollback to its checkpoint by a failed web service should not result in reinvocation of other web services that perform non repeatable actions, for business obligations.

   For ex, in figure 1.4, Online Booking service **WS1** sends payment details and invokes another web service, Theater booking web service **WS2** which performs a non repeatable action of booking a seat in a specified theatre. A

failure of **WS1** at time **T1** would require a rollback of **WS1** to one of its checkpoint locations. Rollback to checkpoint at location **L1** results in reinvocation of **WS2**(not desirable) where as rollback to checkpoint at location **L2** does not result in reinvocation of **WS2**(desirable). Hence checkpoint locations should be based on interactions between web services.

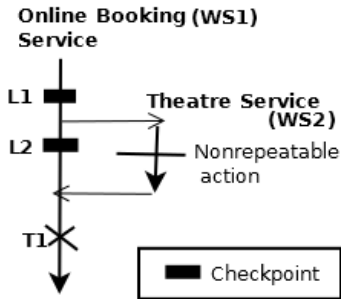


Figure 1.4: Checkpoint locations and their impact

2. **Compliance to SLA**: Web services must work in compliance with SLA (Service level agreements). In web services paradigm each service makes available the details of its QoS attributes like response time, cost of service, availability and reliability, trust etc. These attribute values give necessary insights into the environment in which the constituent web services would be executing. In order to see that a composite web service does not violate its SLA, it is very important to plan its checkpointing and recovery based upon the promised QoS values of its constituent services.

3. **Dynamic selection of web services**: Web services framework provides the facility to select some of the constituent services even dynamically, i,e at the time of execution. Hence a checkpointing strategy that enables a composite web service to meet its time and cost constraints should as well consider the following fact: the checkpointing strategy requires QoS values of all the participants to decide on checkpointing locations. But in a dynamically composed service, some of the participants would be decided only at run time and hence their QoS values also would be available only at run time. In such cases, checkpointing locations also have to be determined dynamically so as

to consider the QoS values of the dynamically selected services.

4. **Dynamic nature of the Internet and web server environments**: The traffic that flows through Internet is highly dynamic in nature which might result in large variations in the response times of web services. Also, number of pending requests at a web server will have an impact on how quickly a request is processed by the web server. This dynamic nature of the environment in which web services operate also has to be considered while taking checkpointing decisions.

A composite web service checkpointing strategy should thus consider 1) Specification of the composite web service that includes sequence of interactions and location of non repeatable actions 2)QoS values of participants to ensure conformation to SLA in case of failure and recovery 3)Refinement of checkpointing locations at run time to consider dynamic environments under which web services operate.

A checkpointing strategy that automatically generates checkpoints based on these requirements is desirable. Also a strategy that offloads most of the checkpointing tasks from run time to earlier stages of development of a composite web services is required. Taking checkpointing decisions at run time proves to be a costly affair for efficient execution of web services.

There has been some recent work on checkpointing of web services. Few papers Marzouk *et al.* (2010); SusanD. *et al.* (2010) have been published discussing the need and techniques for checkpointing web services. *These papers promote checkpointing at user-defined locations leaving the issue to service provider*. Checkpointing schemes presented in Rukoz *et al.* (2012); H.E.Mansour and T.Dillon (2011); Sen *et al.* (2005) identify checkpointing locations either at interaction points or using reliability of services involved. But they do not propose a holistic approach which considers all the above essential characteristics of web services in taking checkpointing decisions.

We have not come across any work on policy-based checkpointing of composite web services to ensure efficient handling of transient faults and subsequently avoid SLA faults. Hence in this thesis we take upon this issue and proceed to work in this direction.

Checkpointing technique is widely used for managing fault tolerance in database management systems so that data inconsistencies in execution of concurrent transactions do not persist. Checkpointing helps to identify transactions that are aborted while in execution and then recovery process decides to undo the changes done by such a transaction or resume the transaction execution leading to its completion. In practice, during execution of a transaction, taking decision on locations for checkpointing entirely lies with users. Usually the heuristic on importance of data for a usage is taken into consideration.

Checkpointing and recovery of web services are conceptually similar to that of database checkpointing and recovery. Here we will highlight the difference that web service execution has in this context. Database checkpointing process takes into account individual transactions that are different. Checkpointing process strategises to ensure consistency for each transaction execution. Though constituent web services of a composite service are executed at different locations, they are considered as one monolithic application interwoven with message interaction. The message interaction sequence has to be considered for checkpointing to ensure resilience in entire application.

Secondly checkpoints in a database transaction are issued by the designer, i.e. a transaction is programmed with checkpoints. But in case of web services, the proposed strategy here, automates the selection of checkpoints considering two factors i.e. non repeatable actions and service level agreement.

Distributed computing has striking similarity with web service execution for both of them share same event action abstraction model. Receipt of messages can be modelled as events and subsequent execution of processes as actions. Though

web service can be thought of as a variant of distributed computing, but its distinctness is due to its usages. Web services are designed for business transactions, so needs to follow business rules and practices. In this, thesis two such issues are considered. These are 1) allowing non repeatable actions i.e once done, it cannot be undone or redone. 2) adherence to service level agreements. Whereas in case of resilient distributed computing, consistency in partial order in message sending and receiving is the research focus.

Considering the issues described at the beginning of this section for web service checkpointing, we present in the following subsection, a comprehensive view of the work carried out in this thesis.

1.4.1 Problem Statement

The central objective of our research work is to : **"Automatically checkpoint a given web service composition, considering both static and dynamic aspects of web service compositions, to handle transient faults so that they do not lead to execution time and cost violations."**

We have worked out to achieve our objective by framing the following questions:

Q1: How do we propose checkpointing locations given a design document of a composite web service, so as to avoid reinvocation of web services performing non repeatable actions? (**static aspect)**

- *Q1.1:* How and what do we capture about interaction centric design of a composite web service?
- *Q1.2:* What would be the probable checkpoint locations given the design document of a composite web service?
- *Q1.3:* What would be the checkpointing rules and recovery rules that would

enable recovery of a failed web service?

Q2: How do we propose checkpointing locations given the constituent web services that participate in the choreography, and their advertised QoS values, so as to ensure that transient faults do not lead to time and cost violations? (**static aspect)**

- *Q2.1:* What are the QoS attributes that impact checkpointing decisions?
- *Q2.2:*What would be the checkpointing rules that use QoS values of constituent web services?
- *Q2.3:* What experiments need to be conducted to prove the utility of these checkpoints?

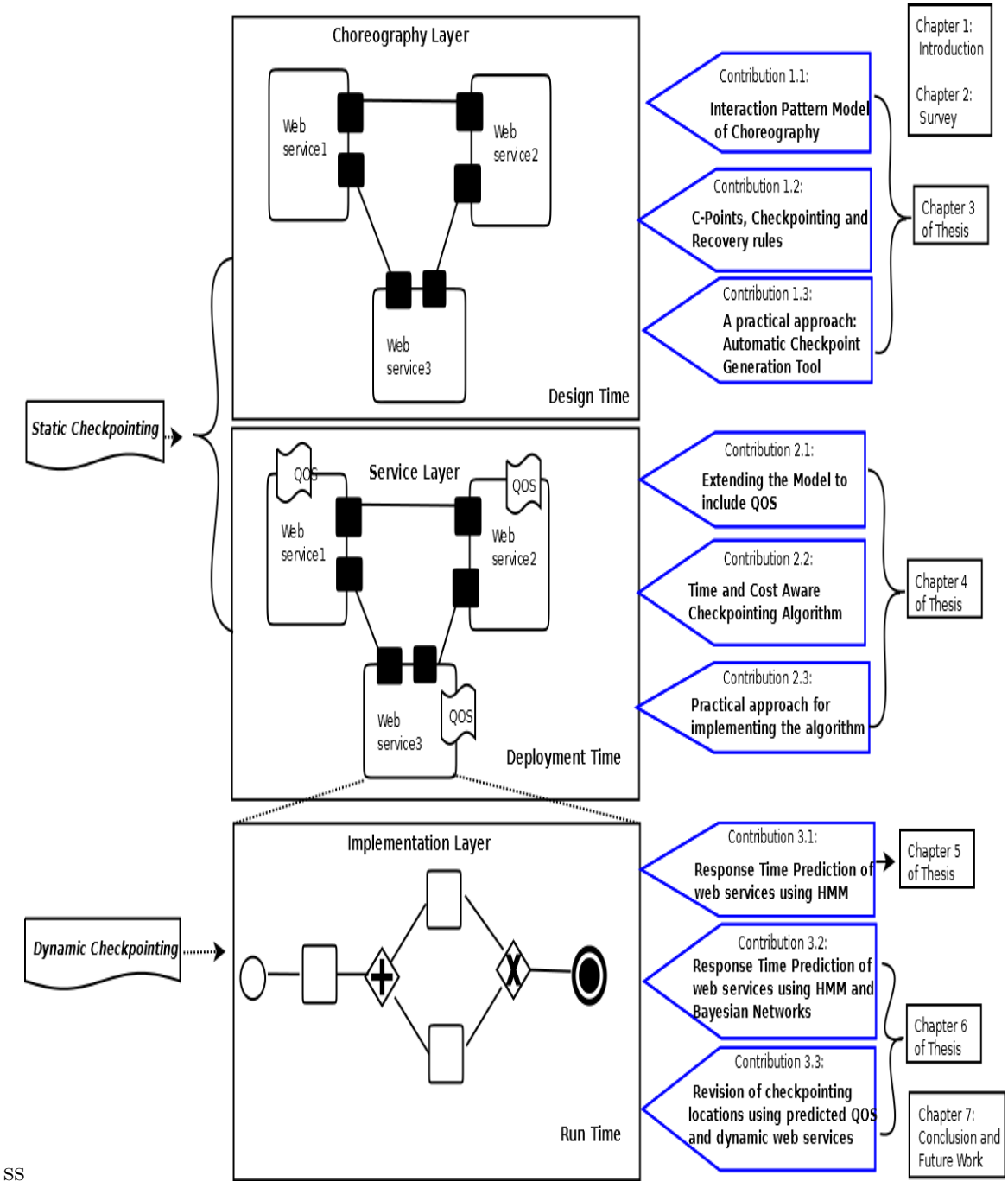
Q3: How do we revise checkpointing locations at run time considering dynamic characteristics of participating web services? (**Dynamic aspects)**

- *Q3.1:* What are the QoS values that change during run time and need to be predicted?
- *Q3.2:* How do we predict QoS values of web services at run time?
- *Q3.3:* How do we revise checkpointing locations using predicted QoS values and dynamic composition of web services, if present?

Answers to these questions make the contributions of this thesis. The thesis contributions are depicted pictorially in the Figure 1.5). Next, we will detail on these contributions and subsequently this describes organisation of the thesis too.

1.4.2 Proposed Solution and Thesis Organisation

We have worked out to achieve our objective by answering the questions posed above. We propose the following three stage checkpointing strategy:



SS

Figure 1.5: Thesis Contributions

- **Design time checkpointing:** Checkpoint locations are based on interactions among web services and location of non repeatable actions.

- **Deployment time checkpointing:** Checkpoints are added using QoS values of constituent services to avoid SLA faults that might raise due to transient faults..

- **Run time (Dynamic) checkpointing:** Checkpoints are revised considering dynamic operating environments of web services to avoid SLA faults.

Our proposed solutions are summarised as under:

- **Solution1: (Chapter 3 Vani Vathsala and HrushikeshaMohanty (2014*a*))** We have captured the interaction centric design of a choreographed web service by proposing an interaction pattern model. Five atomic interaction patterns and five composition operators are proposed that are used to model essential elements of a choreography. We identify probable checkpointing locations using the proposed patterns and term them as C-points. Proposed design time checkpointing rules convert these C-Points into checkpoints to avoid reinvocation of web services that perform non repeatable actions.(*Static Checkpointing)*. To demonstrate the applicability of our approach, we present a tool called as Automatic Checkpoint Generation Tool. It takes as input a choreography of web services in the form of UML activity diagram, applies the proposed checkpointing rules to the composition and, gives as output the activity diagram appended with design time checkpoints. We advocate recovery rules that ensure recovery without initiating reinvocation of other services that perform non repeatable actions.

- **Solution2: (Chapter 4 Vani Vathsala and HrushikeshaMohanty (2015))** We identify response time, vulnerability and cost of service as the

QoS attributes which have to be considered while taking checkpointing decisions. We then amend the proposed interaction pattern model to include these QoS attributes. Using these attributes and other quantities measurable at deployment time like: time taken to checkpoint, time taken to log a message and, restoration time etc, we present Time and Cost aware checkpointing algorithm that proposes checkpoint locations in a composite service. (*Static Checkpointing).* This algorithm computes recovery overhead in terms of time and cost and then recommends to take a new checkpoint only when time and cost constraints are not met when the composite web service fails and recovers. We demonstrate the working of our proposed algorithm experimentally using Oracle SOA Suite and present an analysis of results.

- **Solution3: (Chapter 5 and 6 Vani Vathsala and HrushikeshaMohanty (2012), Vani Vathsala and HrushikeshaMohanty (2014*c*))** For certain web services actual QoS values vary largely from advertised values. We identify that among all the selected QoS attributes, it is the response time of web services that varies from one request to another request. Hence we propose two approaches for response time prediction : 1) A black box approach and 2) A white box approach. The first approach is a block box approach, in which service implementation details(internal structure of the web service, web server environment details like average waiting time in a given time interval) of an invoked service are unknown to the caller. In this approach, we make use of the popular Hidden Markov Model(HMM) for prediction. Prediction accuracy can be improved if implementation and environment details are also made available to the caller. In case the details are known, the proposed white box approach can be used to for response time prediction. This approach makes use of HMM and Bayesian network for prediction. We establish the accuracy of our predictions experimentally. We present revision of checkpointing locations based on the prediction results and dynamic composition of web services. (*Dynamic Checkpointing).*

The strategy used for dynamic checkpointing is as follows: Select a time interval t (say 10 mins) that characterises web service traffic. At the beginning of each time interval t, predict response time of all invoked web services. Compute recovery overhead using these predicted values and revise checkpoint locations so that time and cost deadlines are met. These revised locations are used to checkpoint all instances of the web service that are initiated in the interval t.

Since dynamic checkpointing algorithm is run at the beginning of every time interval t, and not during execution, performance overhead reduces by and large. We have used the following approach to convert probable checkpointing locations into checkpoints at run time. At every C-Point, checkpointing statement is embedded within an *if* statement, and the variables used in the *if* statement are set by checkpointing algorithms. If the condition in *if* statement is true, only then web service state is saved. This is depicted in the following Figure 1.6

```
<if name="s1">
    <documentation> invocation point check</documentation>
    <condition>$nip=1</condition>
            <extensionActivity>
                    <bpelx:dehydrate name="DehydrateMV"/>
            </extensionActivity>
</if>
```

Figure 1.6: Code snippet for converting a C-Point into checkpoint

Last chapter summarises the work done, discussing its relative merits and demerits and scope for future work.

These contributions have led to the following publications:

1.5 List of publications

1. **Vani Vathsala, A. and Hrushikesha Mohanty (2014b)**. **A survey on checkpointing web services**. Proc of the 6th International Workshop on Principles of Engineering Service Oriented and Cloud Systems, pp 11 to 17. Hyderabad, 2014. Proceedings in ACM digital library. **(Indexed in DBLP).** http://dl.acm.org/citation.cfm?doid=2593793.2593795.

2. **Vani Vathsala, A. and Hrushikesha Mohanty (2014a). Interaction patterns based checkpointing of choreographed web services.** Proc of the 6th International Workshop on Principles of Engineering Service Oriented and Cloud Systems, pp 28 to 37. Hyderabad, 2014. Proceedings in ACM digital library. **(Indexed in DBLP).**
   http://doi.acm.org/10.1145/2593793.2593797.

3. **Vani Vathsala, A. and Hrushikesha Mohanty (2015)**. **Time and cost aware checkpointing of choreographed web services**. Proc of the 11th International Conference on Distributed Computing and Information Technology, pp 207 to 219. Bhubaneshwar, 2015. Proceedings in LNCS, Springer, Vol 8956. **(Indexed in DBLP).** http://dx.doi.org/10.1007/978-3-319-14977-6_17.

4. **Vani Vathsala, A. and Hrushikesha Mohanty (2012). Using hmm for predicting response time of web services.** Proc of CUBE International Conference, pp 520 to 525. Pune, 2012. Proceedings in ACM digital library. **(Indexed in DBLP).** http://doi.acm.org/10.1145/2381716.2381816. **(Has 4 citations).**

5. **Vani Vathsala, A. and Hrushikesha Mohanty (2014c). Web service response time prediction using hmm and bayesian network.** Intelligent Computing, Communication and Devices, 327 to 335. Bhubaneshwar, 2014. Proceedings in Springer book Advances in Intelligent Systems and Computing, Vol 1. http://www.springer.com/us/book/9788132220114.

6. **Vani Vathsala, A. (2012). Global checkpointing of orchestrated web services.** Proc of 1st International Conference on Recent Advances in Information Technology, pp 461 to 467. Dhanbad, 2012. Proceedings in IEEE Xplore. **(Indexed in DBLP).**
http://dx.doi.org/10.1109/RAIT.2012.6194622.

7. **Vani Vathsala, A. (2011). Optimal call based checkpointing for orchestrated web services.** International Journal of Computer Applications, 36(8), pp 44 to 50. **(Has 2 citations).**

In the following chapter we present a detailed survey on various checkpointing techniques found in literature.

CHAPTER 2

Literature Survey

Abstract This chapter[1] presents a literature survey conducted in the following areas: 1) Checkpointing algorithms in distributed computing and their applicability to web services. 2) Checkpointing in web services. 3) QoS aware checkpointing in web services and other related areas. 4) Dynamic checkpointing in web services and other related areas. 5) Web service response time prediction approaches.

2.1 Introduction

With the advent and widespread use of web services, complex business processes are being built by discovering and composing services already available on Internet. Such composite web services operate over the Internet where reliability and speed cannot be guaranteed, hence providing fault handling mechanism to composite web services is of primary importance. Several fault handling and recovery techniques have been proposed in literature in the context of distributed systems. In this chapter we present a survey on various distributed and web service checkpointing techniques discussing their applicability, strengths and weaknesses. In section 2.2 we present a survey on distributed systems checkpointing techniques.

Checkpointing strategies proposed for distributed systems are not directly applicable to web services due to differences in the two paradigms. The characteristics of web services which are to be considered while checkpointing them have been discussed in chapter 1. Few papers have been published discussing the need

[1] **Vani Vathsala, A. and Hrushikesha Mohanty (2014b)**. A survey on checkpointing web services. Proc of the 6th International Workshop on Principles of Engineering Service Oriented and Cloud Systems, pp 11 to 17. Hyderabad, 2014. Proceedings in ACM digital library.

and techniques for checkpointing web services. In section 2.3 we discuss about checkpointing techniques in web services. In that section, we list out various features that a web service checkpointing approach should possess and then present a comparision of latest works carried out in this area.

As mentioned in Introduction chapter, we have worked on time and cost aware checkpointing of web services and also on checkpointing web services dynamically. Hence in sections 2.4 and 2.5 we discuss related works carried out by other researchers in the areas of QoS aware checkpointing and dynamic checkpointing. Our proposed dynamic checkpointing requires predicted response time values for checkpointing. For this purpose we have also proposed two approaches for predicting response time of web services. Hence we present a survey of web service response time prediction approaches in section 5.5.

To begin with, we introduce in the following section, the basic terminology used in the field/area of checkpointing, various checkpointing techniques proposed for distributed systems and their applicability to web services.

2.2 Checkpointing as a Fault Handling Technique

Checkpointing is a time tested technique that has been used in several areas like databases, distributed computing etc for backward recovery from faults. Checkpoint is a saved program state and can be used to restore execution of the program in case of its temporary suspension.

Checkpointing strategies proposed in literature for individual applications try to strike a trade off between recovery overhead in case of failure and checkpointing overhead in case of failure free executions of the application. Different checkpointing strategies are: Checkpoint based on 1)*time threshold* 2) *Volume of work done* 3) *criticality/priority of work done.* In distributed applications several processes run parallely at different locations and interact with each other mostly by send-

ing and receiving messages. In addition to the checkpointing strategies specified above, event driven (receipt of a message or sending a message) checkpointing is used as another strategy in distributed checkpointing.

Most of the checkpointing works Chandy and Lamport (1985); Silva and Silva (1992); Netzer and Xu (1995); Elnozahy *et al.* (2002); Koo and Toueg (1987); Bronevetsky *et al.* (2003); Robert *et al.* (2012); Lin and Ahamad (1990); Netzer *et al.* (1997) on distributed applications are based on the notion of taking **consistent checkpoints**, i.e, saved state of the distributed application (global checkpoint) should not result in any information or message loss nor should there be any duplication of information. In other words, a message is said to be an orphan message if it is delivered but not recorded as sent. A global checkpoint is consistent if there are no orphan messages with respect to it.

2.2.1 Distributed checkpointing techniques and their applicability to web services

As discussed in Elnozahy *et al.* (2002) there are three checkpointing techniques in distributed computing: *coordinated, uncoordinated,* and *communication induced* checkpointing. These checkpointing techniques require some of the ***other processes also*** to rollback to their saved states in case of failure of a process. Checkpointing coupled with message logging is another backward recovery technique.

Coordinated checkpointing is essentially a consensus based strategy that ensures a collective action of checkpointing a distributed application. In coordinated checkpointing, there would be a central initiator that issues control messages to all processes to take checkpoints. Coordinated checkpointing is performed in either of the two modes: blocking mode and non-blocking mode. In blocking mode (Tamir and Sequin (1984),Elnozahy *et al.* (2002)) once a process receives a checkpointing request from the initiator, it can no longer send messages to other processes, has to stop its computation, take tentative checkpoint and send back the acknowl-

edgement message to the initiator. Once the initiator receives acknowledgement messages from all processes it signals all processes to commit their checkpoints and continue. Temporary blocking of all executing processes is the main drawback of this approach.

The fundamental approach used by blocking algorithms is preventing processes from receiving application messages after checkpointing phase is initiated so that the resulting global checkpoint is consistent. In non-blocking approach (Chandy and Lamport (1985),Silva and Silva (1992), Bronevetsky *et al.* (2003)), the initiator transmits a marker message after it takes a checkpoint. Upon receiving the marker message, each process takes a checkpoint and retransmits the marker message before sending any application message. This method requires additional control messages for taking a consistent global checkpoint. In both modes of coordinated checkpointing all the processes rollback to their latest checkpoints in case of failure of a process, to maintain a globally consistent state.

In uncoordinated checkpointing (Bhargava and Lian (1988), Elnozahy *et al.* (2002)), processes take their checkpoints independent of each other. But in case of failure of one of the processes a recovery line has to be computed with global coordination negating the advantage of independence of processes. All the processes which are part of the recovery line are notified and they have to rollback to their respective checkpoints that are part of the recovery line. Moreover, this approach may sometimes result in domino effect (Elnozahy *et al.* (2002)) where in rollback of one process results in the rollback of another process and this may continue till all processes are rolled back to their beginning. ***In the context of web services requesting partner services to rollback at runtime is a practical proposition. Particularly for service oriented applications it is applicable for resiliency in service delivery.***

Communication induced checkpointing (WANG (1997), Russell (1980), Netzer and Xu (1995)) tries to avoid domino effect but does not require all checkpoints to be coordinated. These algorithms force each process to piggyback its checkpointing

information in the application messages sent to other processes. The receiver then uses this information to decide if it should take a checkpoint or not. These algorithms define what are called as useless checkpoints (that will never be part of a global consistent state) and try to avoid them by detecting and breaking some patterns in communication. (like Z-Paths and Z-Cycles Netzer and Xu (1995)). ***Although these algorithms avoid domino effect, they also require some of the other processes to rollback in the event of failure of one of them, to maintain consistent state.***

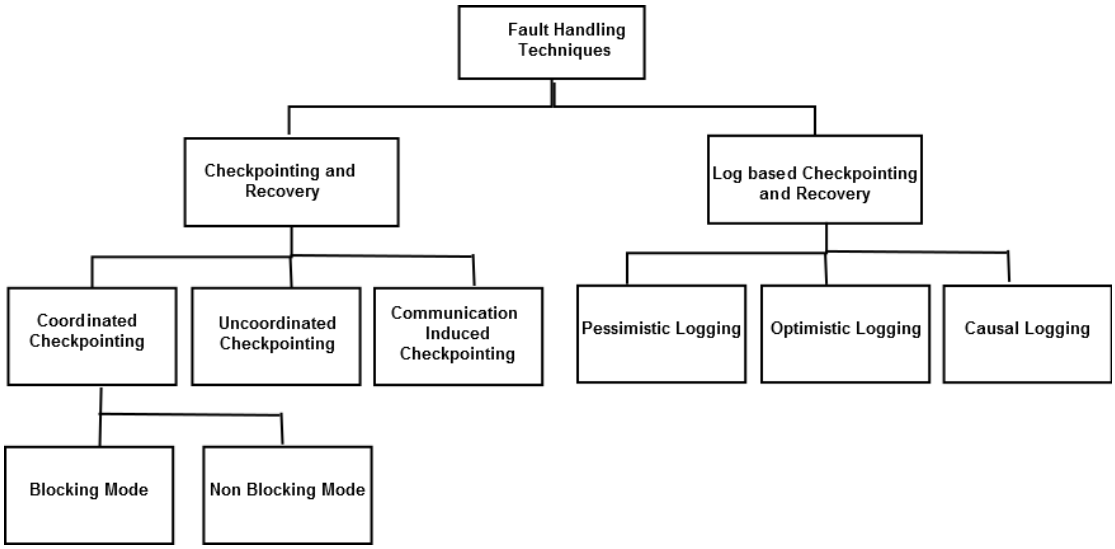


Figure 2.1: Classification of Fault Handling Techniques in Distributed Systems

Log based recovery (Strom and Yemini (1985), Zwaenepoel and Johnson (1987), Johnson and Zwaenepoel (1990), Alvisi *et al.* (2002), and Elnozahy *et al.* (2002)) in its simplest form i)requires all interacting processes to log every message received and ii) allows processes to take checkpoints independent of each other. In case of failure of a process, the failed process needs to rollback to its latest checkpoint and replay all the messages received and logged after taking its last checkpoint. Surviving processes need not rollback but can continue their execution as far as possible (till the point where they are waiting for a message from the failed process). Thus message logging technique requires only the failed process to rollback. There are three flavours of log based recovery (Elnozahy *et al.* (2002)): *Pessimistic logging,*

Optimistic logging and Causal logging. Figure 2.1 depicts various checkpointing and logging techniques in the area of distributed computing. Pessimistic logging (Strom and Yemini (1985)) dictates that when a message is received, execution of the application has to be suspended temporarily, the message has to be logged on to a permanent storage, then the message can be processed by the application. This requirement results in performance overhead. In optimistic logging (Strom and Yemini (1985), Johnson and Zwaenepoel (1990)) the received messages are logged asynchronously (application can be run parallely with logging) so as to reduce the performance overhead. As a consequence, optimistic logging results in the creation of orphan processes (Elnozahy *et al.* (2002)) (a process whose state depends on a message whose receipt is not logged) whereas pessimistic logging does not. Also the processes to which orphan process has sent messages before its failure also have to be rolled back. However optimistic logging ensures that orphans do not exist by the time recovery is complete. Hence this method also requires rolling back processes other than failed process. Surviving processes have to help the failed process in its recovery by resending the messages that were sent to but not logged at the failed process.

In causal logging (Alvisi *et al.* (2002), and Elnozahy *et al.* (2002)), the benefits of both pessimistic and optimistic logging are retained. Like optimistic logging it logs the messages asynchronously. Like pessimistic logging it avoids creation of orphan processes by dictating that messages received by a process have to be logged before sending a message to other processes. But these benefits come at the cost of a complex recovery procedure. If a process fails, surviving processes have to aid in recovery of the failed process by resending the messages that were not logged, along with their order of replay. For this purpose, this technique requires each process to maintain the global order in which messages are to be replayed. Every process builds this causal ordering of messages in the form of a dependency graph and exchanges these graphs with other processes. ***This again results in performance overhead which is not acceptable for any web service if it***

has to provide a quality service.

When a set of web services work in tandem with each other to form a composition, failure in one of the web services should not require rollback of other partner web services during recovery. Checkpointing with message logging seems promising in the context of web services where it is possible to rollback only the failed process without requiring remaining processes to rollback. But the costs (performance overhead of logging each and every message, orphan processes, and sending of dependency graphs) associated with each of the three logging approaches would reduce their applicability to web services.

Hence checkpointing and recovery of web services requires an approach that advocates 1) not to request rollback of processes other than failed process 2) use pessimistic logging to avoid creation of orphan processes but, not to log all the messages (since it results in considerable performance overhead) and 3) to consider the characteristics of web services in designing a strategy.

In the following section we discuss recent works that were proposed for checkpointing web services, and their limitations.

2.3 Checkpointing in Web Services: Literature Survey

Having discussed the non applicability of distributed checkpointing techniques to web services in previous section, we now discuss in this section, different techniques available for checkpointing web services and highlight their limitations/drawbacks. There has been some active work on checkpointing of web services. Few papers (Marzouk *et al.* (2010); SusanD. *et al.* (2010); Sen *et al.* (2005); Rukoz *et al.* (2012); Bland *et al.* (2013); H.E.Mansour and T.Dillon (2011)) have been published discussing the need and techniques for resilient service execution.

Marzouk *et al.* (2010) propose a mechanism for moving orchestrated process instances from a web server to another working web server in case of failure of the former or QoS violations. Executing instances have to be checkpointed to aid in their migration. After migration all the suspended instances would continue from their latest checkpoints. Mobility is ensured by adding mainly two management services called the Web Service Invocation Manager (WSIM) and the Web Service Checkpoint Manager (WSCM). WSIM takes up the responsibility of routing calls from clients to alternate web servers in case of migration of the executing instances of a composite web service. WSCM takes up the responsibility of checkpointing the executing instances and their recovery.

Launching mobility, checkpointing and resuming after migration are modelled as aspects. Deployment of these aspects is assigned to WSIM(first aspect) and WSCM(last two aspects). Aspects facilitate taking checkpoints at run time. Authors list out checkpointing periodically, checkpointing at migration or forced checkpointing as probable checkpointing options but the exact option to be selected is left as out of scope of the paper. Thus positioning these checkpoints is left to the user and the amount of rework done in case of failure hinges on the capability of users in positioning the checkpoints at right place. Also WSIM and WSCM have to be run on reliable servers, their failure would result in costly reinvocations.

Assurance Points (APs) along with integration rules are used by SusanD. *et al.* (2010) to provide a method of checking business level constraints and responding to execution errors in orchestrated web services. An AP is a combined logical and physical checkpoint, which during normal execution, stores execution state and invokes integration rules that check preconditions and postconditions. Depending on the satisfaction of pre and post conditions, integration rules specify either backward recovery or forward recovery. In case of backward recovery the application is rolled back to the nearest assurance point. Forward recovery is tried by attempting alternative execution paths or by reinvoking the failed web service.

Assurance points have to be defined by the user at design time and there is no policy based on which assurance points can be taken automatically.

Sen *et al.* (2005) provide a checkpointing scheme for Inter Organisational Work flows (IOWS). They define an MSP (Maximum Sequential Path) as a *collection of tasks that can be executed sequentially in a workflow.* End of each MSP is treated as a suitable checkpointing location. It is argued that end of an MSP is a suitable location for rolling back in a recovery scheme since it represents a logical end point. Hence this scheme checkpoints work flows whenever there is a control flow branch in the workflow which may result in too many checkpoints.

Rukoz *et al.* (2012) provide recovery scheme for composite web services using checkpoints. When a web service fails, all other constituent web services are allowed to execute as far as possible, and then their state is checkpointed. *But the web service that has failed has no checkpoint in it and hence has to redo all its lost computation.* Other checkpointed web services are allowed to resume when the faulty web service resumes its execution. They propose a three layer architecture which includes additional software components called execution engine and engine threads that run simultaneously along with web services. These components are used to capture and monitor the states of constituent web services and there by implement forward and backward recovery schemes. Execution of these components comes as an additional overhead at run time.

Another web service fault handling mechanism, an alternative to checkpointing, is substitution of faulty web services (H.E.Mansour and T.Dillon (2011); Ezenwoye and Sadjadi (2007); Liu *et al.* (2010); Ben Halima *et al.* (2008)).

H.E.Mansour and T.Dillon (2011) propose a service-oriented reliability model that dynamically calculates the reliability of composite web services. They propose a fault handling approach for composite web services with a central coordinator. They represent the choreography of web services in the form of a graph wherein each node represents a constituent service and an edge is placed from one node

i to another node j if the service j is to be executed after the completion of service i. For an edge (i, j) between two services they define a quantity $E(i, j) = M - R - t$ where M is the maximum expected recovery time, R is the actual recovery time from previous checkpoint and t is the expected execution time of service j. If $E(i, j)$ is negative then a checkpoint is inserted between service i and j, otherwise the next edge is tested for probable placement of a checkpoint. If a constituent service is found to be faulty the calling service is rolled back to its latest checkpoint and an alternative service is invoked. In their model there is no place to include QoS attributes of constituent services. Also dynamic nature of intermediate network and web service environment are not considered while taking checkpointing decisions.

Liu *et al.* (2010) propose a hybrid fault-tolerance mechanism to improve the reliability of composite services especially in case of partner service faults. It identifies four partner service faults: logical faults, system faults, SLA faults and content faults. To handle these faults it identifies eight possible error handling strategies like ignore, retry, notify, replicate, wait etc. They adopt Event-Condition-Action (ECA) rules to describe when and how to use the exception handling strategies. But they do not address the issue of transient faults in the web service that invokes partner services for recovery.

Ezenwoye and Sadjadi (2007) propose a framework by name RobustBPEL that can be used to automatically generate a fault tolerant version of an existing BPEL process. This work also is based on the principle of substitution to provide fault tolerance to composite web services. If one of the partner services fails, then the whole process is subject to failure. To avoid such situations, an adapt-ready process monitors the behaviour of it partners and tries to tolerate their failure by forwarding the failed request to its proxy, which in its turn finds an equivalent service to substitute the failed one. Two versions of RobustBPEL were developed. In static RobustBPEL a static proxy is used, whereas dynamic RobustBPEL uses a dynamic proxy. A static proxy replaces the faulty web service call with a call to a

functionally equivalent service. Information about equivalent services is hardwired into the code of this proxy at the time it was generated. But dynamic proxies can find equivalent services at run time.

Ben Halima *et al.* (2008) propose a framework for self-healing of web services. They have developed a non-intrusive solution for observing exchanged messages between the distributed web services that compose a given application. A four step procedure is devised to monitor, diagnose and detect, plan repair, and execute repair actions for providing self healing. The SOAP messages are extended with QoS parameters and their values observed by monitor components running at receiver and provider ends. When a performance degradation is detected using a specified policy, alternate partner services are invoked. Their reconfiguration algorithm relies on dynamic binding and substitution policies.

The approaches presented by H.E.Mansour and T.Dillon (2011); Ezenwoye and Sadjadi (2007); Liu *et al.* (2010); Ben Halima *et al.* (2008) are to be used when an invoked service fails: invoked service is substituted by an equivalent service. The main drawback of this approach can be seen when the invoking service itself fails. The failed instances would have to be reexecuted from the beginning. This implies recalling the invoked services again, resulting in increased execution times.

Summarising our findings, we present a comparision of all the contemporary works on web services fault handling in Table 2.1. We use the following criteria for comparision:

- Backward /Forward Recovery : Whether the proposed work implements backward recovery or forward recovery. Some of the works implement both of them but specify some criteria for selecting a suitable option at run time.
- Partner service faults or Coordinator faults: Does the proposed fault handling mechanism deal with Partner service faults or coordinator faults or both?

- Automatic(A) or User specified(U): Are the checkpoints automatically generated or manually specified by user at design time?
- Time dependency of WS : In case web service replies are time specific, is this issue considered?
- QoS of web services : Are Quality of Service attributes like response time, reliability, cost of service etc., of web services deliberated while checkpointing?
- Dynamic selection of checkpoints : Can the checkpoint locations be selected or modified at run time?
- Can support dynamic selection of web services : In case constituent services are selected at run time, does the checkpointing algorithm consider this issue while taking checkpointing decisions?

After presenting a survey on general strategies for handling faults in web services, we now proceed to present a survey on QoS aware checkpointing in web services and related areas.

2.4 QoS Aware Checkpointing

Considering QoS values of the constituent services while taking checkpointing decisions for a choreographed web service is pivotal in meeting promised deadlines. We propose to use Quality of Service attributes to decide checkpointing locations so that, execution time and cost of service meet deadlines in case of faults, and in case of failure free executions they are minimal. We advocate QoS aware checkpointing strategy to make choreographed web services resilient to faults, while meeting the promised deadlines. In our survey we have not come across any web service checkpointing strategy that focuses on this issue. Hence we present a survey on QoS aware checkpointing strategies proposed in other related areas.

QoS aware checkpointing has been proposed in various areas like embedded systems (Chen *et al.* (2009)), mobile computing (P.J.Darby and Tzeng (2010)) and multimedia network systems (Osada and Higaki (2001)). Chen *et al.* (2009) propose QoS aware message logging based checkpointing of embedded and distributed systems. They formulate the problem of finding an optimal checkpoint interval that maximizes overall quality as a Mixed Integer Non-linear Programming (MINLP) problem and provide an algorithm for finding the solution. They do not consider QoS values of other web services participating in the composition, which is crucial in case of web services, in checkpointing decisions. The work proposed by P.J.Darby and Tzeng (2010) is for mobile computing environment where hosts going out of access range transfer their checkpointed data to other hosts. Reliability of a mobile link is calculated using factors like signal strength, distance between the two MHS etc. It uses link reliability values to dynamically maintain superior checkpointing arrangement.

Osada and Higaki (2001) propose a checkpointing protocol for multimedia network systems, which is similar to our approach where in checkpoints are at the points of interaction. It considers time to checkpoint should be limited to a range that is predefined. This time represents a QoS attribute. Number of packets that can be lost is also limited and is taken as a QoS attribute. It proposes a checkpoint protocol which ensures that quality of service is maintained and there are no lost or orphan messages in the system. The QoS attributes that they have chosen are not the QoS attributes that are specific to web services (like response time, cost of service etc). They also allow certain parts of a large message to be lost in transition to maintain the required timeliness, which is not desirable in case of web services.

Considering QoS values of the constituent services while taking checkpointing decisions is pivotal in meeting promised deadlines . In our survey we have not come across any web service checkpointing strategy that focuses on this issue. Hence we advocate time and cost aware checkpointing strategy that makes use of

QoS values of constituent services, to decide on checkpoint locations while meeting the promised deadlines.

Large variations in response times of constituent services and the need to compose web services dynamically drive us to review and revise checkpointing locations at run time. We propose an algorithm that revises checkpointing locations dynamically, using predicted response time values of constituent services and, advertised QoS values if services are selected dynamically. In this context we have surveyed dynamic strategies for checkpointing choreographed web services, in the following section we present these details.

2.5 Dynamic Checkpointing

We have proposed to use advertised QoS values of constituent services to take checkpoint decisions in a composite web service. But at run time there would be large variations between advertised QoS values and actual QoS values. Hence we propose to revise the checkpointing locations at run time using predicted QoS values that would be very close to actual Qos Values. Dynamic composition of web services and varying failure rate of web services are other reasons that demand dynamic checkpointing. In this section we present a survey on dynamic checkpointing strategies proposed in web services and related areas.

Chtepen *et al.* (2009) propose an adaptive algorithm named MeanFailureCP+ that deals with checkpointing of grid applications with execution times that are unknown apriori. The algorithm modifies its parameters, based on dynamically collected feedback on its performance. MeanFailureCP+ monitors dynamically the number of jobs processed during a monitoring interval of predefined length and based on this feedback modifies subsequent job length estimates in such a way that the checkpointing overhead is minimized without significantly penalizing the system fault-tolerance. The approach allows for periodic modification of checkpointing intervals at run-time, when additional information becomes avail-

able. This algorithm increases checkpointing interval by a value given by the end user when remaining execution time is lesser than average failure interval. Else, if job failure interval is lesser than remaining execution time, it reduces checkpoint interval by a value given by end user. In modified version of the algorithm it does not ask the user to give exact job length value, but estimates it from current number of requests. If number of jobs in last interval $\geq$ number of jobs in current interval, it reduces the job length by 0.1%, otherwise increases it by 0.1%.

Nazir *et al.* (2009) provide an adaptive task checkpointing based job scheduling scheme for grid environments; Whenever a grid resource broker has tasks to schedule on grid resources, it makes use of the fault index (No of jobs not successfully completed gives fault index). They propose to maintain and update the fault index of all available resources of the grid. The fault index of the grid resource will suggest its vulnerability to faults (i.e., higher the fault index, higher is the failure rate). The Fault Tolerant Schedule Manager (FTScheduleManager) maintains fault index history. A centralised checkpoint manager CPManager maintains information of partially executed tasks by the grid resources. It maintains details of last successful checkpoints taken by jobs. Grid resource broker allocates jobs to resources based on fault index.

Zhang and Chakrabarty (2003) present an integrated approach that provides fault tolerance and dynamic power management for a real-time task executing in an embedded system. Adaptive checkpointing is then combined with a dynamic voltage scaling scheme to achieve power reduction. The resulting energy-aware adaptive checkpointing scheme uses a dynamic voltage scaling criterion that is based not only on the slack in task execution but also on the occurrences of faults during task execution. Checkpointing locations are updated based on the frequency of fault occurrences and the amount of time remaining before the task deadline. The proposed adaptive checkpointing scheme is tailored to handle not only a random fault-arrival process, but it is also designed to tolerate up to k fault occurrences.

Work proposed by Zhou *et al.* (2010) uses predicted server load to adjust checkpoint frequency for high throughput data services. The authors present programming and runtime support called SLACH for building multi-threaded high-throughput persistent services. In order to keep in-memory objects persistent, SIACH employs application-assisted logging and checkpointing for log-based recovery while maximizing throughput and concurrency. SIACH adaptively adjusts checkpointing frequency based on log growth and throughput demand to balance between runtime overhead and recovery speed. A programmer can select one of the following policies to control checkpointing frequency: the number of logged records, log file size, or checkpointing time interval, and can also specify the allowable threshold lower bound and upper bound [LB, UB]. SLACH determines the triggering threshold as follows. When the predicted server load drops below a low watermark (LW), the overhead of checkpointing is negligible and the lower bound is used as a threshold to perform checkpointing more frequently. On the other hand, when the predicted server load exceeds a high watermark (HW), the upper bound is used to adjust the checkpoint frequency in order to perform checkpointing as sporadically as possible. When the number of logged records on disk exceeds this threshold, the checkpointing process is triggered to reduce the number of logged records. When the predicted server load lies between LW and HW, threshold is computed through a nonlinear function of the server load.

H.E.Mansour and T.Dillon (2011) propose a service-oriented reliability model that dynamically calculates the reliability of a composite web service and places checkpoints in the composite web service using expected recovery time. ***We propose a holistic approach to decide on checkpoint locations which considers 1)dynamic QoS values of all the invoked web services 2)failure rate variations of the web service to be checkpointed and 3) provision of dynamic composition of the web service to be checkpointed. We have not come across any work that proposes dynamic checkpointing for composite web services considering all the above stated issues.***

As predicted response time values are required for implementing dynamic checkpointing, we have proposed two methods for predicting response time of web services. In this context we have surveyed literature for web service response time prediction methods and in the following section we present this survey.

2.6 Response Time Prediction Approaches

Our approach for dynamic checkpointing requires predicted response times of invoked web services. Hence we have proposed two response time prediction approaches. Using our approaches a service user can predict the response time of a web service that he invokes. First method is a **black box approach** which uses Hidden Markov Model(HMM) for prediction and, second method is a **white box approach** that uses both Bayesian network and HMM for prediction.

In general, web service response time prediction approaches can be divided into two types: black box approaches and white box approaches. In black box approach internal structure of a web service, whose response time has to be predicted, is unknown to the prediction module. In white box approach (Marzolla and Mirandola (2007),Laranjeiro *et al.* (2009)) internal structure of the web service along with environment details like average waiting time, would be available to the prediction module while making predictions.

Black box approaches can be further divided into i)static methods(use only historic data) ii)dynamic methods (consider current traffic status also). Most commonly used static methods are those that employ a)Artificial Neural Networks(Gao and Wu (2005), R.D.Albu *et al.* (2013), R.D.Albu (2014), R.D.Albu (2013)) b)User similarity metrics (Li *et al.* (2010),Chen *et al.* (2011*a*)) c) Mathematical approximations (Cheung *et al.* (2011), Balogh *et al.* (2006)), (Chen *et al.* (2011*b*) and d) petrinets (Geebelen *et al.* (2014)). Popular dynamic methods employ a) time series modeling (Yan Hai (2012)) and b)Markov Models (Yan and Liu (2013), Vani Vathsala and HrushikeshaMohanty (2012) Vani Vathsala and

HrushikeshaMohanty (2014*c*)), for prediction.

Marzolla and Mirandola (2007) propose an approach for performance assessment of web service workflows that use BPEL constructs. This is a white box approach that requires a knowledge of internal structure of the web service and can be used by developers to predict performance of the service being developed. It's a static approach and does not consider status of a dynamically varying network and web server traffic during prediction.

In order to predict timing failures, Laranjeiro *et al.* (2009) propose to use a graph based approach. He analyses a service code and builds a graph to represent its logical structure. He then gathers time-related performance metrics during runtime. This is used to predict the possibility of service execution in given time. This also is a white box approach which works when service is executing. If prediction results are required just before execution, as in our case, this method is not applicable.

Li *et al.* (2010) and Chen *et al.* (2011*a*) propose to employ web services similarity statistics and user similarity metrics to predict web service performance. When a user wants to predict QoS for a web service, the authors propose to first select a set of web services that have the highest degree of similarity with the target service. They do so by comparing the target service with the other services used by target user. Then they obtain users of these similar services and their QoS values. Integrating all these QoS values gives the required predicted value.

Balogh *et al.* (2006) presented a knowledge based approach for predicting runtime of stateful web services. To predict the execution time of a web service instance, it maintains a knowledge base of possible different past cases for different combinations of input parameters. Given a web service instance, Euclidean distances are used to find out most similar past cases. The runtime for the given web service instance is predicted to be the average output value of the most similar past cases.

Similar to our concept, Cheung *et al.* (2011) also focuses on predicting the response time of a web service from clients perspective. He proposes an approach that collects data from performance testing, and applies interpolation at low workloads and extrapolation at high workloads, to predict the response time. These are black box approaches, and they do not consider dynamic behaviour of network and web server, which is very much needed for accurate prediction.

Work presented by Chen *et al.* (2011*b*) proposes a QoS model to capture the time based QoS attributes, based on which QoS of composite services are aggregated. It considers time while modeling time dependent QoS attributes like response time and availability, as opposed to other models which do not consider time. Although it uses HMM for prediction of QoS values, it uses static data, which might not reflect current network traffic and server status.

Artificial Neural Networks have also proved themselves to be reliable predictors. Zhengdong Gao uses Back Propagation Neural Networks to predict the runtime of a given web service (Gao and Wu (2005)). He uses availability, network bandwidth, response time, reliability of a given web service as inputs to a neural network which produces predicted execution duration as output. R.D.Albu *et al.* (2013) have published a set of research papers (R.D.Albu (2014),R.D.Albu (2013)) that compare prediction accuracy of different varieties of neural networks on a data set that consists of real world web services response time and declare one of them as the most suitable design for response time prediction. Although neural networks have established themselves as powerful predictors, HMM scores an edge over NNs when it comes to prediction in dynamic environments. For our purpose we need response time prediction approaches that consider current state of web services and underlying network environments for giving accurate prediction results.

Geebelen *et al.* (2014) present a two-step approach to predict QoS values of composite services. The first step, which is based on petri nets, deals with QoS aggregation by deriving QoS values of a workflow composition from those of its

participating elementary services. In a second step, they apply time series prediction on the aggregated historical QoS values to accurately predict if a composition complies with an SLA. Again, they do not consider dynamic characteristics of constituent web services in predicting their response times. The approach has only training data and proposes to minimise the expected risk in prediction by using quantile estimation.

Yan Hai (2012) point out that predicting the QoS properties only by their current state is a flaw, a dynamic prediction based on previous results of QoS is necessary. The network load always changes and hence web service response time also changes. By observing the patterns of changes in response times in subsequent intervals of time, they propose autocorrelation based approach for predicting response time of web services. They consider only dynamic behaviour of underlying network in this prediction process i.e, they consider variations in only network delay component of response time and ignore varying waiting time component that results due to several pending requests at the web server.

Yan and Liu (2013) authors put forward a QoS dynamic prediction method based on Semi-Markov Processes(SMP) and Case-based Reasoning(CBR). This method at first uses semi-markov process to predict business state of web service in the future, and then applies the technology of CBR to predict web service QoS, for example, when the service deals with a specific operation. But this work does not consider network delay at all to predict response time. It assumes constant data transfer rates, which is far from reality. They consider only server load metrics to determine the web service state, where as we use other metrics also like database server load and input values to perform the prediction.

In our black box approach, we are using HMMs for prediction purpose because of their proven suitability for modelling dynamic systems. The research works presented by Hassan and Nath (2005) to Netzer *et al.* (2008) propose methods for using HMM as a predictor. Hassan and Nath (2005) use HMM to predict share prices in stock markets. Gonzalez *et al.* (2005) use IOHMM to analyse

and forecast electricity spot prices. Netzer *et al.* (2008) model the dynamics of customer relationships and the subsequent buying behavior of the customers using non-homogeneous HMM. To the best of our knowledge there is no work which concentrates on predicting Response Times of web services dynamically using HMM.

In our white box approach, we divide response time to be predicted into two main components: Time spent on web server(called as execution time) and network delay. Execution time is further divided into waiting time(time spent by the request waiting for common resources) and instruction execution time. Network delay is predicted using HMM, as states of intermediate network are invisible to the service requester who performs prediction. Waiting time prediction is done using data available at web server, like number of pending requests, input vector etc. Waiting time prediction is done using bayesian network as it provides a distribution of estimated value rather than a point estimate. This helps very much in achieving prediction accuracy. In cases where required data like number of pending requests etc is unavailable, black box approach has to be used for prediction.

Our work (Vani Vathsala and HrushikeshaMohanty (2012)) has been cited by Ahmed and Wu (2013), and Tan *et al.* (2014). Ahmed and Wu (2013) propose a method for QoS metrification based on Hidden Markov Models. The technique can be used to measure and predict the behaviour of web services under several criteria (like servers behaving badly, improper mapping of data) and can thus be used to rank services quantitatively rather than just qualitatively. It is explained with experiments how HMM can be used to analyse and predict QoS attributes of a web service in terms of data discrepancy, i.e what they predict is probability of getting incorrect data(o/p) in the next time interval, they do not predict any of the QoS attributes.

Tan *et al.* (2014) propose an analysis and prediction model based on time series using Particle Swarm Optimization based Back Propagation Neural Network

(PSO-BPNN) model, to predict the dynamic performance of workflow systems. It proposes a black box approach for monitoring and prediction of QoS values of workflow components. HMM scores an edge over BPNNs also, since it considers current state of web services and underlying network environments also for giving accurate prediction results.

2.7 Conclusion

In the beginning of this chapter we have presented a survey on various checkpointing techniques proposed in distributed systems and discussed their applicability to web services. List of characteristics that web service checkpointing algorithms should possess is then presented. Recent works on checkpointing web services are also discussed and they are analysed based on the listed characteristics. QoS aware checkpointing and dynamic checkpointing works proposed in related areas like mobile networks and embedded systems are also surveyed. Finally, web service response time prediction approaches proposed in literature are discussed. In all these sections, we have compared our proposed work with available literature and established the need and applicability of our work.

We present the first module of our work, i.e design time checkpointing, in the next chapter.

Table 2.1: Comparision of Web services fault handling approaches

Reference	Backward(B) Forward(F) Recovery	Partner service faults (P) or Coordinator faults (C)	Choice of Checkpoint Locations: (A) Or (U)	Time dependency of WS	QoS of WS	Dynamic selection of checkpoints	Dynamic Composition of Web Services
Checkpoints using Aspects Marzouk *et al.* (2010)	B	C	U	X	Y	Y	Y
Checkpointing workflows Sen *et al.* (2005)	B	C	A	X	X	X	X
Checkpointing using Petrinets Rukoz *et al.* (2012)	B	C	A	X	X	Y	X
Time based Checkpointing H.E.Mansour and T.Dillon (2011)	B and F	P and C	A	Y	Y(Only Reliability)	Y	Y
Substitution using web service features Liu *et al.* (2010)	F	P	U	X	Y	-	Y
Assurance Points SusanD. *et al.* (2010)	B and F	C	U	X	X	X	X
Substitution using proxy server Ezenwoye and Sadjadi (2007)	F	P	-	X	X	-	Y
Pattern based Checkpointing [Our Approach]	B	Pand C	A	X	Y	Y	Y
X= Does not support Y = Supports/handles							

CHAPTER 3

Design Time Checkpointing

Abstract In this chapter[1] we first present our interaction pattern model which captures various elements of a choreography that are essential for our checkpointing. Then we define what is a pattern, atomic pattern and composite pattern. Using these patterns we identify probable checkpointing locations in a choreography by defining what are called as C-points. Subsequently we present our design time checkpointing and recovery rules which ensure that web services performing non repeatable actions are not reinvoked during recovery of the calling service in the event of its failure. We demonstrate that the proposed idea is implementable through a framework called as Automatic Checkpoint Generation.

3.1 Introduction

As discussed in *survey* chapter, well understood techniques for checkpointing and recovery of distributed applications are not readily applicable to web services. Unlike distributed applications, aborted web services can't be reinvoked, may be for strategic business decisions or simply on cost consideration. Hence checkpointing composite web services is of interest.

There has been some recent work on checkpointing in the web services world. Few papers Marzouk *et al.* (2010); SusanD. *et al.* (2010); Sen *et al.* (2005); Rukoz *et al.* (2012); Bland *et al.* (2013); H.E.Mansour and T.Dillon (2011) have been published discussing the need and techniques for checkpointing web services. *But*

[1]**Vani Vathsala, A. and Hrushikesha Mohanty (2014a).** Interaction patterns based checkpointing of choreographed web services. Proc of the 6th International Workshop on Principles of Engineering Service Oriented and Cloud Systems, pp 28 to 37. Hyderabad, 2014. Proceedings in ACM digital library.

these works provide a framework for placing checkpoints in composite web services and user has to identify checkpointing locations, they do not propose any checkpointing policy. **On the contrary, we propose a checkpointing technique which decides on checkpoint locations automatically, without user intervention.**

A set of services working in tandem with each other to achieve a business goal are called as choreographed web services. They interact with each other according to a predefined sequence specified in a design document called as choreography document. Based on this document, we propose a **design time** checkpointing policy that introduces checkpoints in the choreography. In the event of transient failures this checkpoint arrangement avoids reinvocation of web services that perform non repeatable actions.

To identify non repeatable actions and to avoid reinvocation of web services, we need a model that captures the actions and invocations of constituent services of a choreography. (The only way to invoke a web service is to ***interact*** with it by sending a message to it). We propose such a model and name it as **pattern based interaction model.** If a web service provides its service to a service requester without interacting with another web service, then we call that interaction as **atomic interaction**, otherwise it is called as **composite interaction**. On closely observing these message based interactions, it can be seen that there are some patterns in which web services interact with each other. In this chapter, we identify *five atomic interaction patterns* which are used to define *composite interaction patterns*. These atomic and composite interaction patterns are then used to model a given choreography. This act of modelling a given choreography in terms of interaction patterns helps us in identifying probable checkpoint locations that can avoid reinvocation of web services that perform non repeatable actions.

In order to identify checkpointing locations in a given choreography, we define C-points which are probable checkpointing locations. Checkpointing rules specify whether a C-Point is to be converted into a checkpoint or not. We then present

a framework for automatic checkpoint generation. Through this framework we demonstrate that the proposed interaction pattern model and checkpointing rules can be used to automatically generate checkpointing locations in a given web service choreography.

Organisation of this chapter is as follows: We capture essential details of a choreography from it's design document (section 3.2), propose atomic interaction patterns, composite patterns and then model choreographies using the proposed interaction patterns (section 3.3). Then we proceed to develop a design time checkpointing technique (section 3.4) that places checkpoints using the proposed interaction patterns. In the section (section 3.5) that follows, we present pattern identification algorithm that performs the task of modeling a given choreography diagram in terms of proposed patterns. Next section (section 3.6) presents design time checkpointing algorithm that inserts checkpoints into the given choreography by applying the rules specified in (section 3.4). Recovery rules are presented in the section that follows (section 3.7). We present a proof of correctness of our proposed checkpointing and recovery rules in section 3.8. We demonstrate that design time checkpointing is implementable through the development of a framework called ACG (Automatic checkpoint Generation) (section 3.9). Conclusion is given in section 3.10.

3.2 Modelling Service Choreographies

As described in the earlier section, the design of a choreographed web service is specified in a document called as choreography document. In this section we present a model which captures essential details of a choreography document for the purpose of checkpointing a given choreography.

In a choreography document, specification of the following is given: 1)List of participating services 2) Actions performed by each of the participants and 3) Sequence of interactions(message exchanges).

UML Activity diagram is one of the popular languages used for modelling business processes Russell *et al.* (2006). In this thesis we capture essential elements of choreography modelled using UML activity diagram.

Figure 1.3 depicts design document of a choreographed web service **On line booking** modelled using UML activity diagram. In this choreography, there are four participants: Online Booking Service, Payment service, Theatre Service and Client. Each participant performs some local actions(represented in rectangular boxes) and performs some interactions (represented by horizontal arrows) with other participants. Non repeatable actions are enclosed in red colored rectangular boxes.

In a UML activity diagram participants are represented by swimlanes, actions are represented using rectangular boxes and arrows between swimlanes depict interactions. To mark an action as a non repeatable action, we use red colored rectangular boxes. Arrows with in same swimlane represent sequential execution of actions by a participant.

In the following subsections we capture essential details of a choreography.

3.2.1 Participant

A web service participating in a choreography is in short called as a **participant**. In our proposed model, a **participant** ξ is specified as a tuple $(i, Pname)$, where $Pname$ is a short description of the participant, and i is the unique identification number given to the participant, $1 \leq i \leq n$ where n is the total number of participants in the choreography. i^{th} participant is in short represented by ξ_i. In the example choreography, there are four ($n = 4$) participants, $\xi_1 =$ *client*, $\xi_2 =$ *Online Booking Service*, $\xi_3 =$ *Theatre Service* and $\xi_4 =$ *Payment service*. A set of participants SP $= \{(i, Pname)\}$.

Each participant of a service choreography performs some local actions and

communicates necessary information with other participants. Thus, broadly, actions of a service are classified into two categories: *Local action* and *Interaction*.

3.2.2 Local action

An action performed locally by a participant, without intervention of other participants is called as a **local action**. A local action τ performed by a participant would be either **Repeatable** (ra) that can be repeated any number of times) or **Nonrepeatable** (ra). Different types of local actions are:

- A local computation. (Repeatable)
- A database/file query/read operation. (Repeatable)
- Database/File update or write (Nonrepeatable)
- Manual/Physical action like delivering goods etc. (Nonrepeatable)
- Wait for an interaction (Repeatable)

For ex, local actions performed by the participant *Theatre Service* are: *Fetch details and Book seats*. Among these local actions, *Book seats* is specified as a non repeatable action. *Fetch details* is a repeatable local action.

3.2.3 Interaction

An **interaction** γ is defined as a message that is used for sending information from one participant (sender) to another (receiver). An interaction between two participants can be one of the following three types.

- **Request** γ_{rq}: A participant of a choreography invokes another participant by sending a *request* message and waits for a corresponding reply.

- **Invoke** γ_{in}: A participant invokes another participant by sending an *invoke* message but does not wait for a reply.

- **Reply** γ_{rp}: A participant which receives a *request* message or an *invoke* message sends back a response to its requester by using *reply* message. This type of interaction should have a matching request or invoke.

Variables ξ_s, ξ_r are used to represent sender and receiver of an interaction. In the example choreography the messages sent by *Online Booking Service* to *Theatre Service* are of type γ_{rq}, and the corresponding reply messages sent back are of type γ_{rp}.

3.2.4 Operation

Actions performed by a participant of a choreography, viz, local actions and interactions are termed as **operations** performed by the participant. Thus, behaviour of a participant can be specified by a set of operations, where each operation σ is defined as a tuple as follows:

$\sigma = (OName, tp, doer, doesfor, stp)$ where

- $OName$ is a short description of the operation

- tp is the type of operation. If the operation performed is a local action then $tp = \tau$; $tp = \gamma$ if the operation performed is an interaction.

- $doer$ is the participant who performs(in case of local actions) or invokes(in case of interactions) this operation.

- $doesfor$ is the participant with whom $doer$ interacts. It is null if $tp = \tau$.

- stp indicates subtype of the operation.

  - If $tp = \tau$, stp indicates whether it is a repeatable action ra, or a non repeatable action na, i.e $stp \in (\tau_{ra}, \tau_{na})$)

– If $tp = \gamma$, stp indicates whether it is a request rq or an invoke in or a reply rp, i.e $stp \in (\gamma_{rq}, \gamma_{in}, \gamma_{rp})$

An operation is now formally specified as $\sigma = (OName, tp, \xi_s, \xi_r, stp)$, if $tp = \gamma$ then $stp \in (\gamma_{rq}, \gamma_{in}, \gamma_{rp})$ and if $tp = \tau$ then $stp \in (\tau_{ra}, \tau_{na}))$ and $\xi_r = null$.

In the above example, the local action 'Fetch details' performed by *Theatre Service* is formally modelled as : $\sigma = ('Fetch \ \ details', \tau, \xi_3, null, \tau_{ra})$. Similarly, the request message sent by *Online Booking Service* to *Theatre Service* is modelled as: $\sigma = ('Request\ availability\ details', \gamma, \xi_2, \xi_3, \gamma_{rq})$.

A **Sequence** of operations $\sigma_1, \sigma_2, ...\sigma_n$ is represented by Ψ. Here after a sequence of operations is superscripted (wherever required) with id of the participant which performs the operation. Using this sequence of operations, we define the concept of interaction patterns in the following section.

3.3 Interaction Pattern Model

Communication among the participants of a choreography in terms of the three types of interactions listed above make different patterns. In the following subsections these patterns are introduced before delving into checkpointing and recovery.

3.3.1 Interaction pattern

An **interaction pattern**, or a **pattern** in short, is defined as a Ψ wherein the first operation is an interaction of type γ_{rq} or γ_{in}, and is optionally ended by γ_{rp} For a given γ_{rq} or γ_{in} sent by ξ_s, if there is a corresponding γ_{rp} sent back by ξ_r then the pattern is called as a *two way pattern* else the pattern is called as a *one way pattern.*

3.3.2 Atomic patterns

A pattern is called as an **atomic pattern** if the pattern contains only one request or invoke message. For simplicity, we call an atomic two way pattern as **two way pattern** and an atomic one way pattern as a **one way pattern**.

One special pattern, P_0, called as *No interaction* pattern is defined, where-in the initiator of the pattern does only local actions and does not perform any interaction. A sequence of operations in which none of the operations is an interaction is represented by Ψ_0. 2 two way patterns and 2 one way patterns are identified.

Two way patterns : A two way pattern starts when its initiator sends a request and ends when it gets back a reply to the request sent. Two way patterns are further classified into two types viz Request Reply and Invoke Reply which are labelled as P_1 and P_2 respectively. Figure 3.1(a) depicts two way patterns.

1. Pattern P_1 *(Request Response):* A caller/ initiator ξ_s requests for a service from a service provider ξ_r. ξ_r executes a sequence Ψ_0^r of operations and sends back the reply message to ξ_s. The sequence of operations that get executed in the pattern P_1 are: $\gamma_{rq}^s, \Psi_0^r, \gamma_{rp}^r$, as shown in Fig. 3.1.1

2. Pattern P_2 *(Invoke with Reply):* An initiator ξ_s places invoke message and continues to execute a sequence of operations Ψ_0^s, before getting a reply from ξ_r. Upon receiving the invoke message, ξ_r executes a sequence of operations Ψ_0^r and sends a reply message back to ξ_s. The sequence of operations that get executed in the atomic pattern P_2 are: $\gamma_{in}^s, (\Psi_0^s || \Psi_0^r), \gamma_{rp}^r$ as shown in Fig. 3.1.2. The symbol || represents parallel execution of operations.

One way patterns: A one way pattern is said to start when its initiator ξ_s places the invoke message and ends when both ξ_s and ξ_r end their execution. Figure 3.1(b) depicts one way patterns.

- Pattern P_3 *(Invoke and continue):* A participant ξ_s invokes another partic-

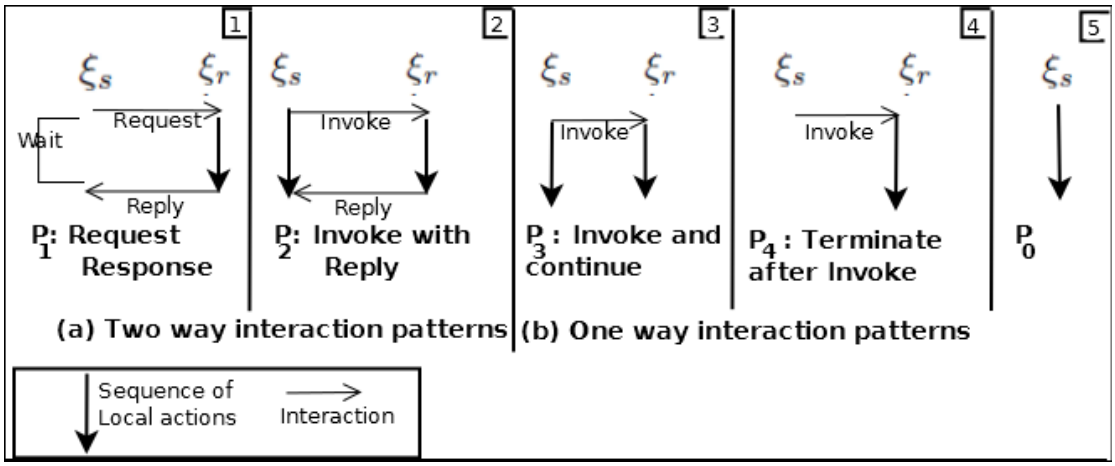


Figure 3.1: Atomic Patterns of Interaction

ipant ξ_r and continues to execute a sequence of operations Ψ_0^s. ξ_r receives the request and executes a sequence of operations Ψ_0^r and does not send a reply message back to ξ_s. The sequence of operations that get executed in the atomic pattern P_3 are: $\gamma_{in}^s, (\Psi_0^s || \Psi_0^r)$ as shown in Fig. 3.1.3

- Pattern P_4 *(Terminate after Invoke):* A participant ξ_s invokes another participant ξ_r and terminates its execution. ξ_r executes a sequence of operations Ψ_0^r and does not send a reply message back to ξ_s. The sequence of operations that get executed in the atomic pattern P_4 are: γ_{in}^s, Ψ_0^r as in Fig. 3.1.4.

It may be noted that the proposed atomic patterns are represented using capital letters $P_1, P_2...$, and small letters $p, p_1, p_2...$ are variables of type pattern. The patterns are superscripted (wherever required) with three numbers as p^{xyz}, where x indicates the pattern type, y indicates the id of its initiator and z indicates the id of its receiver respectively. For example, p^{112} indicates that type of pattern p is P_1, participant ξ_1 has initiated the pattern and participant ξ_2 is the receiver which provides the service requested. In case superscripts yz are not necessary for illustration, we omit them.

3.3.3 Composite patterns

A pattern is called as a **composite pattern** if the pattern contains more than one request or invoke messages. Atomic patterns may be combined in different ways using composition operators to give composite patterns. Five kinds of composite patterns are proposed: Sequential pattern, Nested pattern, Iterative pattern, Concurrent pattern and Choice pattern. A **Pattern String** represents a choreographed web service which is expressed in terms of patterns and composition operators.

Sequential pattern '.' : If a pattern p_2 starts after the completion of another pattern p_1, then the resultant pattern is called as a *sequential pattern*. Sequential composition is represented using the operator '.' Pattern string representing a sequential pattern is: $p_1.p_2$. Refer to Figure 3.2(a) for an example of a sequential pattern, where the patterns p_1^1,and p_2^0 are sequentially executed.

Nested pattern '[ ]' : If the receiver ξ_r in a pattern p_1 initiates another pattern p_2 before completion of the pattern p_1, then the pattern p_2 is said to be nested inside the pattern p_1. In this case Ψ^r contains an interaction. The pattern p_2 is called as inner pattern and the pattern p_1 is called as outer pattern. The resultant pattern is called as a *nested pattern*. Nested composition is represented using the operator [ ], written as $p_1[p_2]$ and read as p_1 ***contains*** p_2. Refer to Figure 3.2(b) for an example of a nested pattern, where p_2^1 is nested in p_1^1.

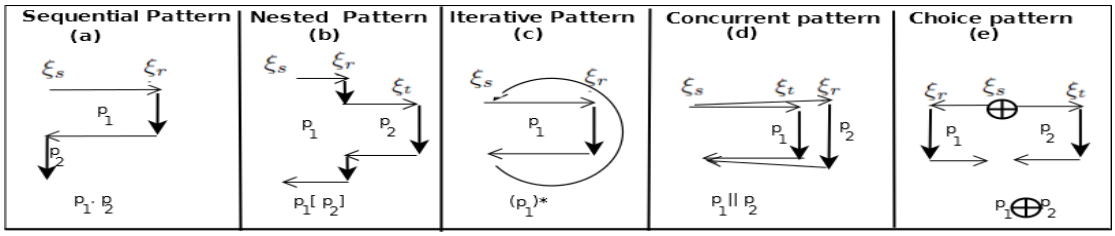


Figure 3.2: Examples of Composite Patterns

Nesting notations: When a pattern, say p_1^{1sr} is nested in another pattern, say p_3^{1rt}, the actual sequence of operations that get executed are :$\gamma_{rq}^s, \Psi_0^r, \gamma_{rq}^r, \Psi_0^t, \gamma_{rp}^t$,

Ψ_0^r, γ_{rp}^r. When represented in terms of patterns, it becomes $p_1^1[p_2^0.p_3^1.p_4^0]$. This expression in short is represented as $p_1[p_3]$, after omitting p^0 patterns which by default prefix and suffix any inner pattern of a nested composition. Figure 3.3 depicts all possible nesting notations. Each component in this diagram is labelled with the corresponding Expanded Pattern string (EPS) and its equivalent pattern string (PS). For the pattern type P^4 it can be seen that $p_i^4[p_j]$ is equivalent to $p_i^4.p_j$.

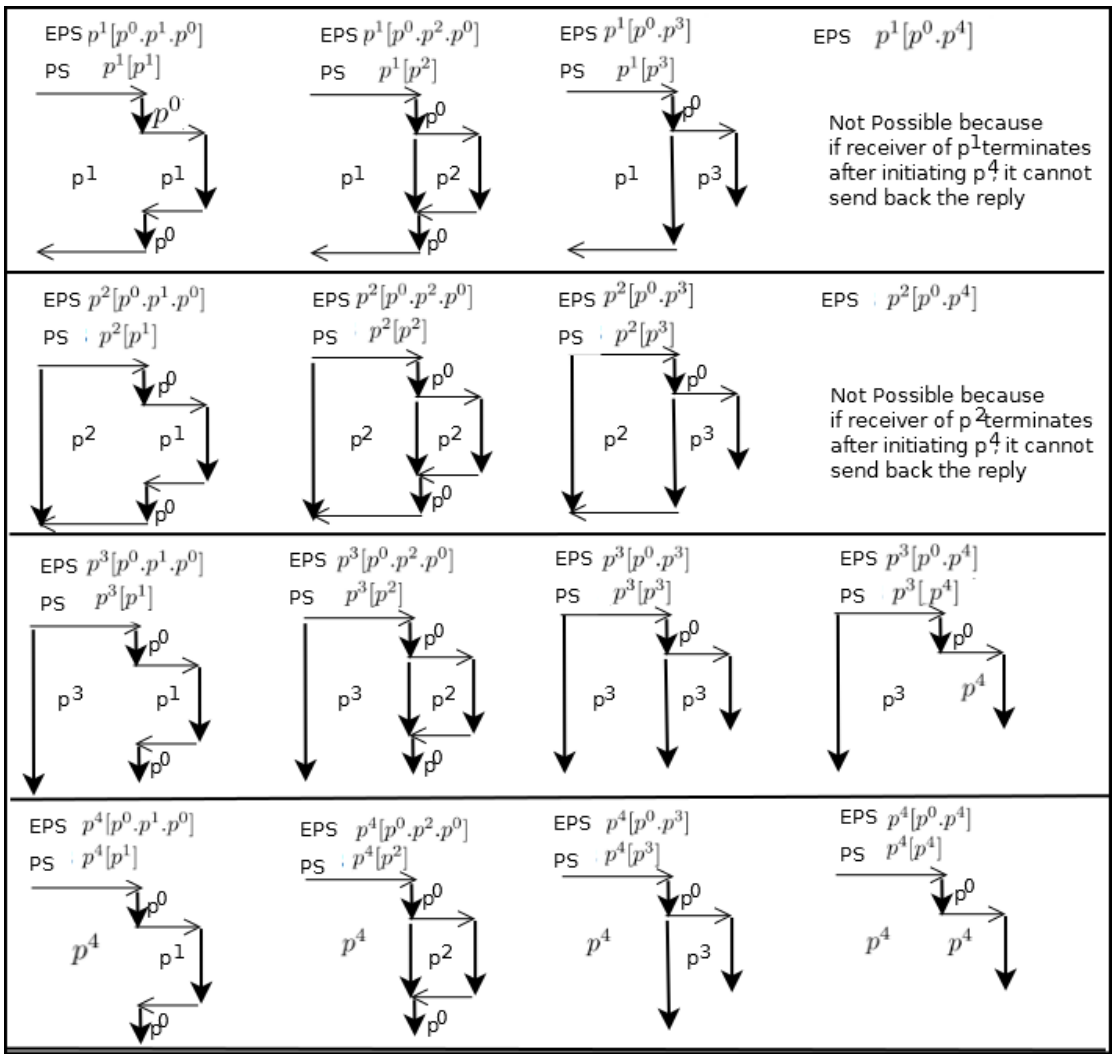


Figure 3.3: Nesting notations

Iterative pattern '∗' : If a pattern p is sequentially executed m number of times then the resultant pattern is called as an *iterative pattern*. Iteration is represented using the operator '∗'. Pattern string representing an iterative pattern is: $(p_1)^*$.

Figure 3.2(c) depicts an iterative pattern.

Conurrent Operator '||': If a participant ξ_s initiates more than one pattern simultaneously, then the resultant pattern is called as a *concurrent pattern*. Concurrent composition is represented using the operator: '||'. Pattern string representing a concurrent pattern is: $p_1||p_2$. Figure 3.2(d) depicts a concurrent pattern.

Choice pattern '⊕' : If either pattern p_1 or pattern p_2 is executed, but not both, then the resultant pattern is called as a *choice pattern*. Choice composition is represented using the operator '⊕'. Pattern string representing a choice pattern is: $p_1 \oplus p_2$. Refer to Figure 3.2(e) for an example of a choice pattern.

In the following subsection a formal model for atomic and composite patterns is presented which is used later for automatic checkpoint generation.

3.3.4 Formal model for patterns

An **atomic pattern** p is described as a tuple $(i, \Psi, \xi_s, \xi_r, tp)$ and is in short represented as p_i^{tpsr} where

i is a unique id given to the pattern.

Ψ is the sequence of operations that are performed in the pattern.

ξ_s is the initiator of the pattern.

ξ_r is the service provider invoked by the initiator of the pattern.

tp is the type of the pattern where $tp \in \{0, 1, 2, 3, 4\}$.

Ex: Pattern p_1^{2sr} is described as the tuple $(1, \Psi, \xi_s, \xi_r, 2)$ where $\Psi = \gamma_{rq}^s, \Psi_0^r, \gamma_{rp}^r$.

A **composite pattern** cp is described as a tuple (i, op, SCP, ξ_p) where

i is a unique id given to the pattern.

op is a composition operator and $op \in \{., ||, *, [\,], \oplus\}$.

SCP is the set of constituent patterns. $SCP = \{p_1, p_2\}$. For an iterative pattern $p_2 = null$.

ξ_p is the initiator of the pattern, $\xi_p = p_1.\xi_s$

3.3.5 Modelling a service choreography as a composition of patterns

When a choreography of web services is designed as a composite web service, a requested service is collectively provided by the constituent services. Such a composite service can be modelled by composing the service patterns using the proposed composition operators. Thus a *composite web service* can be modelled as a *composite pattern*.

Figure 3.4 a) depicts an example choreography and Figure 3.4 (b) depicts modelling of the choreography as a composite pattern. $p_1^{01}.p_2^{112}.p_3^{123}[p_4^{01}]$ represents pattern string for the choreography.

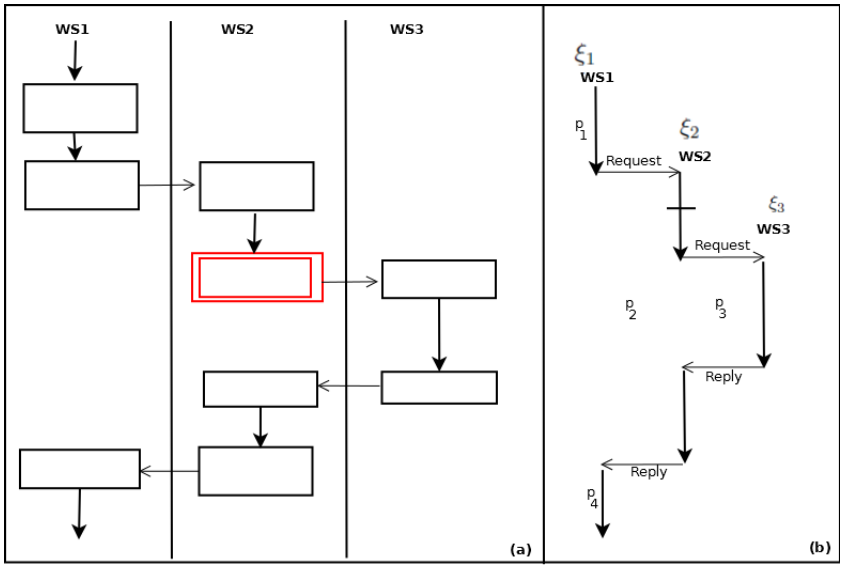


Figure 3.4: A choreography modeled as a composite pattern

In the following subsection we discuss the adequacy level of proposed interac-

tion patterns.

3.3.5.1 Adequacy of interaction patterns

The proposed interaction pattern model is aimed at capturing sequence in which constituent web services interact with each other. As specified in section 3.2, we capture this information from a given UML activity diagram. UML activity diagrams are well suited for capturing control flow aspects of a choreography (Russell *et al.* (2006)). In a UML activity diagram control flow is modelled using the following five basic control patterns (Russell *et al.* (2006)): Sequence, parallel split, synchronisation, exclusive choice and simple merge.

- Sequence: depicts sequence of activities.
- Parallel split: depicts splitting of single thread of control into multiple threads of control.
- Synchronisation: depicts merging of multiple threads of control into a single thread of control.
- Exclusive choice: depicts selection of one of several choices.
- Simple merge: depicts merging of alternative branches.

A pattern of interaction, as defined in section 3.3, starts with a request or invoke activity and optionally ends with a reply. Sequence pattern of UML can be mapped directly to our sequence pattern. Parallel split and synchronisation patterns of UML when combined together, map to our concurrent pattern. Exclusive choice and simple merge patterns when combined together map to our choice pattern. Thus it can be clearly seen that all the common patterns of web service composition can be modelled using our interaction pattern model.

UML activity diagram contains advanced patterns like multiple choice, multiple merge and discriminator (Russell *et al.* (2006)) which when combined result

in other possible patterns of interaction. As mentioned in our future work chapter, we propose to present modelling elements to capture these advanced UML patterns, as part of our future work.

3.4 Design Time Checkpointing

Our main purpose of identifying patterns and modelling choreographed web services using these patterns, as detailed in the previous section, is to define *probable checkpointing locations* based on these patterns. Such points are called **C-Points** and the decision for converting such points into checkpoints is governed by certain rules called checkpointing rules. C-points are described in the next subsection followed by checkpointing rules.

3.4.1 C-Points

In a step towards identifying possible checkpointing locations in a given choreography, each pattern is associated with what are called as C-Points. A C-Point is a probable checkpointing location. Three types of C-Points are defined: Service Point, Must Save Point (not same as checkpoint) and, Invocation Point.

- ***Service Point (SP)****:* If a pattern is a two way pattern then it is associated with a service point. It is marked in the initiator of the pattern after the reply message is received. A service point marks the end of a two way pattern; also it marks the end of a service requested by the initiator. If the initiator of a pattern fails after this service point, it is rolled back to this service point and thus reexecution of the pattern is avoided.

- ***Invocation Point (IP)****:* Both one way and two way patterns are associated with an invocation point. It is marked in the initiator of a pattern after it sends an invoke/request message. An invocation point is utilised differently

for two way and one way patterns which is explained later.

- ***Must Save Point (VP):*** Must Save point is marked in a participant of a pattern after a non repeatable action, if any, is performed.

If a pattern is a two way pattern then it contains both service point and invocation point. A one way pattern is associated with only one point, i.e. invocation point. Both the type of patterns may be associated with must save points. Figure 3.5 depicts C-Points inserted into the example composite web service. (Hollow box represents a C-Point)

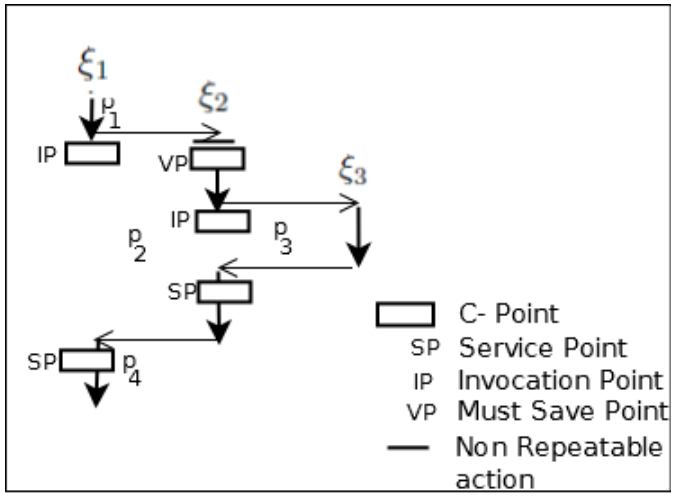


Figure 3.5: Example: C-Points in the example choreography

3.4.2 Checkpointing policy

Design time checkpointing policy is proposed in this subsection that identifies checkpointing locations in the patterns described in previous section. Checkpointing policy is based on the following two principles:

1. A participant executing a non repeatable action, must be checkpointed immediately after performing the non repeatable action.

2. If the receiver in a pattern executes a non repeatable action, the initiator of the pattern should not initiate the interaction again in case of its failure.

According to the proposed design time checkpointing policy :

1. A participant ξ of a pattern is checkpointed in the following scenarios: (refer to Figure 3.6 (Filled box represents a checkpoint))

   - (Rule 1 (CR1)) Checkpoint a participant ξ at its must save point. This rule is applicable to all the five pattern types. Remaining rules are not applicable to the pattern type P_0.
   - (Rule 2 (CR2)) Checkpoint isnitiator ξ_s of a pattern p if ξ_r of p is checkpointed. ξ_s is checkpointed at its invocation point so that in case of failure of ξ_s, ξ_s is rolled back to the checkpoint and does not reinvoke ξ_r which would have been already checkpointed. This rule is applicable to all the pattern types except P_4. This rule is demonstrated in the Figure 3.6 using the two way pattern P_2.

2. Messages are logged according to the following rules:

   - (Rule 3 (LR1)) Log *request/ invoke* message at its receiver(ξ_r), sent just before being checkpointed by the initiator ξ_s of a pattern. This is required for recovery of ξ_r) in case of it's failure. Refer to Figure 3.6.
   - (Rule 4 (LR2)) Log any *reply* message at receiver (ξ_s), sent after taking a checkpoint by (ξ_r). This is required for recovery of (ξ_s) in case of it's failure.
   - (Rule 5 (LR3)) Log *invoke* message at ξ_r, sent by ξ_s of a pattern p if ξ_s is checkpointed due to a must save point between sending the request message and receiving the reply message from ξ_r. This is required for recovery of the ξ_r in case of it's failure.
   - (Rule 6 (LR4)) Log any *reply* message at receiver (ξ_s), sent by ξ_r of a pattern p if ξ_s is checkpointed due to must save point between sending the request message and receiving the reply message from ξ_r. This is required for recovery of the ξ_s in case of it's failure.

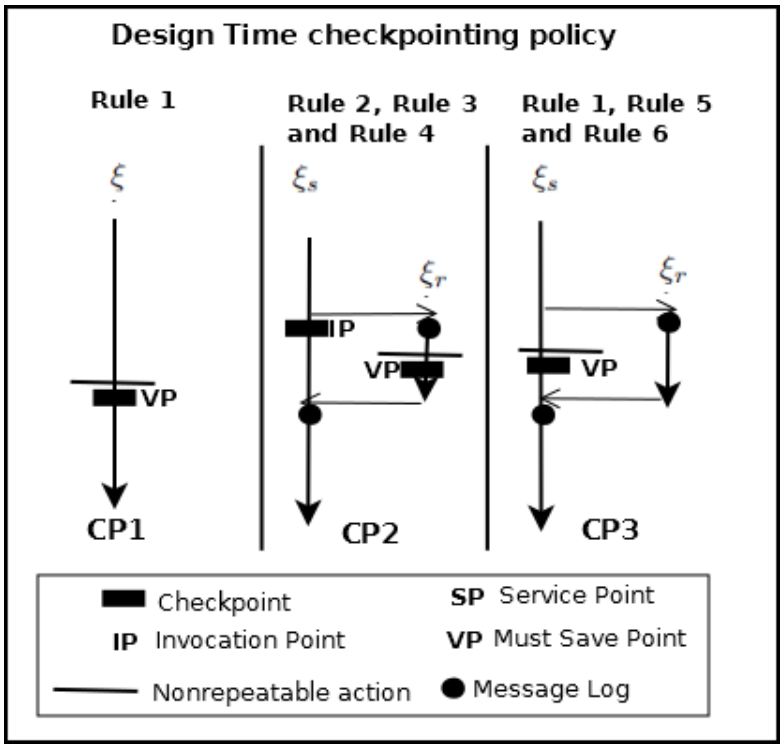


Figure 3.6: Checkpointing Patterns

On applying rules of checkpointing and logging to the atomic patterns of choreography, the patterns that evolve are categorized into three types CP1, CP2, and CP3 which are named as checkpointing patterns. These patterns are used to specify recovery patterns that describe recovery rules in case of failure of participants.

3.4.3 Extending formal model with C-Points

In this subsection we extend the formal definition of atomic patterns presented in subsection 3.3.4 to amend specification for patterns with C-Points.

Specification for atomic patterns needs to be augmented to accommodate markings for C-Points. For the sake of simplicity we recommend to model one must save point for each participant in an atomic pattern. However this need not be a limitation for our purpose. We identify two generic locations in a pattern p for placing must save points, as detailed below.

- *Location ML1:* in the initiator of p after sending the request message and before receiving the reply message.
- *Location ML2:* in the receiver of p after receiving the request message.

Location ML1 is valid for the pattern types P_2, P_3, P_0 and location ML2 is valid for all the five patterns. In this extended definition certain variables are used to indicate whether C-points of a pattern p are checkpointed or not.

An **atomic pattern** p is described as a tuple $(i, \Psi, \xi_s, \xi_r, tp, ip, sp, vp1, vp2, lrq, lrp)$ where

i is the id given to the pattern.

Ψ is the sequence of operations that are performed in the pattern.

ξ_s is the initiator of the pattern.

ξ_r is the service provider invoked by the initiator of the pattern.

tp is the type of the pattern

ip is a boolean variable which is initially set to false indicating that invocation point of the pattern p is still not converted into a checkpoint. When the invocation point is converted to a checkpoint, ip is set to true.

sp is a boolean variable which is initially set to false indicating that service point of the pattern p is still not converted into a checkpoint. When the service point is converted to a checkpoint then sp is set to true. For one way patterns, sp is always false.

$vp1$ is a variable that can take three values. It is set to 0 initially if there is no non repeatable action at $ML1$ of the pattern p, else it is set to 1. When the must save point is converted to a checkpoint then $vp1$ is set to 2. For pattern types P_1, P_4 $vp1$ is always set to 0.

$vp2$ is set similar to $vp1$ for $ML2$.

lrq is a boolean variable which is initially set to false indicating that request/invoke message of the pattern is not logged at ξ_r after receiving it. It is set to true when the message is logged at ξ_r.

lrp is a boolean variable which is initially set to false indicating that reply message of the pattern is not logged at ξ_s after receiving it. It is set to true when the message is logged at ξ_s.

In case checkpoints are inserted into a web service either at design time or later, the pattern string that reflects the choreography must reflect the checkpoint locations also. Hence the following notation is used: For a pattern p if its invocation point is converted to a checkpoint then symbol ! is added to the left of it, if its service point is converted to a checkpoint then symbol ! is added to the right of it. If must save point at location *ML1* of pattern p is converted into checkpoint then the symbol (!) is added to the left of it. If must save point at location *ML2* of pattern p is converted into checkpoint then the symbol (!) is added to the right of it. The pattern string that contains inserted checkpoint marks is called as **"Annotated pattern string"**. For example if invocation point of pattern p_2 in the pattern string $p_1.p_2$ is checkpointed then the annotated pattern string that reflects this checkpoint is $p_1.!p_2$.

In order to automate the process of inserting checkpoints into a given choreography, the first step that must be performed is to convert the given choreography diagram into its equivalent pattern string. We propose an algorithm called **Pattern identification algorithm** that performs this task, and the following section presents this algorithm.

3.5 Pattern Identification Algorithm

Pattern identification algorithm scans through a choreography given in the form of a diagram. We have considered UML activity diagram for this purpose, other equivalent diagrams may also be used. The algorithm constructs a pattern string PS equivalent to the given choreography. Also, it generates an array of atomic patterns identified in the choreography. The algorithm makes use of five main fucntions: $getNextOperation(d)$, $makeString(i, j, \sigma.\xi_s, \sigma.\xi_r)$, $Append(PS, c)$, $setnonRepeatable(SP,$ and $addPattern\ (SP, \sigma, i, j)$. Algorithm 5 depicts this algorithm. The algorithm also makes use of a stack to remember the sequence of operations in case of 1) nesting and 2)identification of pattern types P_3, P_2, P_4. The algorithm works for sequence and nesting composition operations and it can be extended on similar lines to identify remaining compositions.

The function $getNextOperation(d)$ scans through the composition given in the form of diagram d and returns the next operation in sequence. In case of invoke operation, the next operation to be fetched is that executed by ξ_s.

The function $makeString(i, j, \sigma.\xi_s, \sigma.\xi_r)$ returns the string p_i^{jsr}.

The function $Append(PS, c)$ appends the string c to the pattern string PS.

The function $setnonRepeatable(SP, i, j)$ sets vp1 of i^{th} pattern in the set of patterns SP to 1 if $j = 1$ and sets vp2 to 1 if $j = 2$.

The function $addPattern(SP, \sigma, i, j)$ constructs a new pattern whose initiator and receiver are set to those of σ. The function sets all C-Points to their default values. Then the new pattern is inserted into the array SP.

Algorithm 1 $addPattern(SP, \sigma, i, j)$ Atomic Pattern Listing Algorithm

Input The array SP to which a new pattern p_i has to be added. Participants are extracted from σ.

Output The array SP to which p_i is added

1: $p.i \leftarrow i$
2: $p.\xi_s \leftarrow \sigma.\xi_s$
3: $p.\xi_r \leftarrow \sigma.\xi_r$
4: $p.tp \leftarrow j$
5: $p.ip \leftarrow false$
6: $p.sp \leftarrow false$
7: $p.vp1 \leftarrow 0$
8: $p.vp2 \leftarrow 0$
9: $p.lrq \leftarrow false$
10: $p.lrp \leftarrow false$
11: $Insert(SP, p)$

Algorithm 2 Pattern Identification Algorithm

Input The diagram d which depicts the choreography modelled as a composite pattern.

Output Pattern string PS, Set of patterns SP

1: i=1; $PS = null$;
2: **while** $((\sigma = getNextOperation(d))! = null)$ **do**
3: **if** $(top()! = "@" || top() = "[")$ **then**
4: $Append(PS, ".")$
5: **end if**
6: **if** $(\sigma.tp = \tau)$ **then** // operation is local action
7: **if** $isStackEmpty()$ **then**
8: $push("@")$
9: **else**
10: **if** $(nonRepeatable(\sigma)$ **then**
11: $setnonRepeatable(SP, i, 2)$ //sets $vp2$ of pattern i to 1.
12: **end if**
13: $pop()$
14: **end if**
15: **if** $top()! = "@"$ **then** // Construct pattern of type P_0
16: $p \leftarrow makeString(i, 0, \sigma.\xi_s, null); Append(PS, p);$
17: $addPattern(SP, \sigma, i, 0)$; i++;
18: **if** $(nonRepeatable(\sigma)$ **then**
19: $setnonRepeatable(SP, i, 1)$ //sets $vp1$ of pattern i to 1.
20: **end if**
21: **end if**
22: **else if** $(\sigma.stp = \gamma_{rq})$ **then** // operation is request
23: **if** $(top() = "@")$ **then** // if no other interaction in stack
24: push("A")
25: **else if** $(top() = "rp")$ **then** // if top of stack is reply,i.e reply is not yet read
26: $Append(PS, "["); push(\tau); push("[");$
27: **end if**
28: $p \leftarrow makeString(i, 1, \sigma.\xi_s, \sigma.\xi_r); Append(PS, p);$
29: $addPattern(SP, \sigma, i, 1)$;i++; //Pattern type P_1
30: $push("rp"); push(\tau)$ // expecting local action, reply
31: **else if** $(\sigma.stp = \gamma_{in})$ **then** // operation is invoke
32: **if** $(top() = "@")$ **then**
33: push("A")
34: **else if** $(top() = "rp")$ **then**
35: $Append(PS, "["); push("[");$
36: **end if**
37: $\sigma \leftarrow getNextOperation(d)$
38: **if** $(\sigma = null)$ **then**
39: $p \leftarrow makeString(i, 4, \sigma.\xi_s, \sigma.\xi_r);$

Pattern Identification Algorithm(continued)

```
40:       else if (σ = τ) then
41:         if (checkNextReply(d)) then //checks if there is a corresponding reply
42:           p ← makeString(i, 2, σ.ξ_s, σ.ξ_r); push("rp"); push(τ)
43:           addPattern(SP, σ, i, 2) //Pattern type P_2
44:         else
45:           p ← makeString(i, 3, σ.ξ_s, σ.ξ_r); push(τ)
46:           addPattern(SP, σ, i, 3) //Pattern type P_3
47:         end if
48:         Append(PS, p); i++;
49:       end if
50:     else if (σ.Stp = γ_rp) then
51:       pop()
52:       if (top() = "[" then
53:         Append(PS, "]"); pop()
54:       end if
55:       if (top()="A" then
56:         pop()
57:       end if
58:     end if
59: end while
```

For the choreography depicted in Figure 3.4, the set SP generated by the algorithm is $SP = \{p_1, p_2, p_3, p_4\}$ where:

$$p_1 = \{1, \Psi_1, 0, 1, null, false, false, 0, 0, false, false\}$$

$$p_2 = \{2, \Psi_2, 1, 1, 2, false, false, 0, 1, false, false\}$$

$$p_3 = \{3, \Psi_3, 1, 2, 3, false, false, 0, 0, false, false\}$$

$$p_4 = \{4, \Psi_4, 0, 1, null, false, false, 0, 0, false, false\}$$

The pattern string PS generated by the algorithm is $p_1^{01}.p_2^{112}[p_3^{123}].p_4^{01}$.

After generating an equivalent pattern string for a choreography given in the form of a diagram, the next step is to apply the proposed chechpointing rules and insert checkpoints into the pattern string. This task is performed by **Design time checkpointing algorithm** which is presented in the following section.

3.6 Design Time Checkpointing Algorithm

Algorithm 6 applies design time checkpointing and logging rules presented in subsection 3.4.2 to the given pattern string and set of atomic patterns, and inserts checkpoint locations into the pattern string. It inserts checkpoints by setting pattern's appropriate checkpointing variables. For ex: if a pattern p has a must save point at location ML1 then, it is converted into a checkpoint by setting $p.vp1 = 2$. The algorithm makes use of three main functions $getNextPattern(PS)$, $getEnclosingPatterns(p, PS)$, $insertCkp(PS, p, n)$.

The function $getNextPattern(PS)$ parses the pattern string PS from left to right and returns the next pattern in PS each time it is called. When called initially, it returns first pattern, returns second pattern in next call and so on. For ex: $getNextPattern(p_1^0.p_2^1[p_3^1].p_4^0)$ returns p_1^0 when called initially, in the next call it returns p_2^1, and in the next call it returns p_3^1 and finally it returns p_4^0.

The function $getEnclosingPatterns(p, PS)$ returns an array of patterns which enclose pattern p in the pattern string PS, from innermost enclosing to outermost enclosing patterns. For ex: $getEnclosingPatterns(p_3^1, p_1^0.p_2^1[p_3^1].p_4^0)$ returns the array $[p_2^1]$.

The function $insertCkp(PS, p, n)$ inserts the character ! in the pattern string PS depending on the value of n. $n = 1$ indicates checkpoint is inserted at invocation point of p, $n = 2$ indicates checkpoint is inserted at service point of p. Similarly $n = 3$ and $n = 4$ indicate must save points at ML1 and ML2 are converted into checkpoints respectively. Hence if $n = 1$ the symbol ! is inserted before p, if $n = 2$ the symbol ! is inserted after p, if $n = 3$ the symbol (!) is inserted before p and if $n = 4$ the symbol (!) is inserted after p. The function call $insertCkp(p_1^0.p_2^1[p_3^1].p_4^0, p_3^1, 4)$ returns the pattern string $p_1^0.p_2^1[p_3^1(!)].p_4^0$. Algorithm 6 presents the proposed design time checkpointing algorithm.

Algorithm 3 Design time checkpointing algorithm

Input Pattern string PS that represents the given choreography. Set of patterns generated by Pattern identification algorithm.

Output Pattern string PS annotated with checkpoints. Set of patterns with inserted checkpoints.

```
1: while (p = getNextPattern(PS)!=null) do
2:   if p.vp1 = 1 Possible only for pattern types P2, P3, P0 then
       // Must Save point at location 1
3:     p.vp1 ← 2 // Applying CR1
4:     insertCkp(PS, p, 3)
5:     if p.tp ≠ 0 then
6:       p.lrq ← true // Applying LR3
7:       if p.tp ≠ 3 then
8:         p.lrp ← true // Applying LR4
9:       end if
10:    end if
11:  end if
12:  if p.vp2 = 1 Possible for all pattern types except P0 then
       // Must Save point at location 2
13:    p.vp2 ← 2 // Applying CR1
14:    insertCkp(PS, p, 4)
15:    p.lrq ← true // Applying LR1
16:    if p.tp ≠ 4 then
17:      p.ip ← true // Applying CR2.
18:      insertCkp(PS, p, 1)
19:      if p.tp ≠ 3 then
20:        p.lrp ← true // Applying LR2
21:      end if
22:    end if
23:  end if
24:  if p.vp2 = 2||p.vp1 = 2 then
25:    enclosingpatterns[ ] ← getEnclosingPatterns(p)
26:    for each pattern q in enclosingpatterns[ ] do
27:      q.lrq ← true // Applying LR1
28:      if q.tp ≠ 3 and 4 then
29:        q.lrp ← true // Applying LR2
30:      end if
31:      if q.tp ≠ 4 then
32:        q.ip ← true
33:        insertCkp(PS, q, 1)
34:      end if
35:    end for
36:  end if
37: end while
```

For the choreography depicted in Figure 3.4, the set SP as modified by the algorithm is $SP = \{p_1, p_2, p_3, p_4\}$ where:

$p_1 = \{1, \Psi_1, 0, 1, null, false, false, 0, 0, false, false\}$

$p_2 = \{2, \Psi_2, 1, 1, 2, true, false, 0, 2, true, true\}$

$p_3 = \{3, \Psi_3, 1, 2, 3, false, false, 0, 0, false, false\}$

$p_4 = \{4, \Psi_4, 0, 1, null, false, false, 0, 0, false, false\}$

The pattern string PS generated by pattern modification algorithm is modified by this algorithm as $p_1^{01}.!\ p_2^{112}(!)[p_3^{123}].p_4^{01}$. (Checkpoints are marked using the character ! in the pattern string.)

After proposing checkpointing rules, next step in handling faults is proposing recovery rules, and the next sections delves into this issue.

3.7 Fault Asynchrony and Recovery

After viewing a choreographed service as a composition of patterns and proposing checkpointing rules, it is desirable to achieve seamless execution of the composition. For this purpose identifying possible points of failure and planning recovery are essential. This section takes up these issues and recommends a set of recovery rules.

In case of any local transient faults, the checkpointed services would rollback to previously saved state and continue from there on upon recovery. Faults may appear anywhere in the choreography, i.e, there would be different points of failure in the execution of a choreographed web service. Either the initiator of a pattern might fail or the invoked participant might fail. Figure 3.7 depicts four scenarios where the first, second scenarios depict points of failure(t1,t2) in the initiator

and the third and fourth scenarios depict points of failure(t3, t4) in the invoked participant.

At points where failures occur after receiving a request/ reply message, the logged message is replayed at recovery stage. Similarly for a failure after non repeatable action which is checkpointed, the saved state is restored at recovery stage. Thus reinvocation of a participant is avoided along with redoing of work to carry on services seamlessly after failures.

3.7.1 Recovery patterns

Depending on the location of points of failure and type of checkpointing pattern (CP1, CP2 and CP3), recovery rules are presented which can be used to recover the participants from their failures.

A *recovery pattern* specifies the rules to be applied for recovery of participants of a checkpointing pattern in case of their failure. Each recovery rule specifies the sequence of operations to be executed for recovery. These recovery patterns are derived using the proposed checkpointing patterns and points of failure. Checkpointing patterns CP1, CP2 and CP3 lead to recovery patterns RP1, RP2 and RP3.

3.7.1.1 Recovery pattern RP1

Recovery pattern RP1 presents the rules to be applied to the participant of checkpointing pattern CP1, for its recovery. In pattern CP1, the participant is checkpointed at its must save point and there are no intermediate interactions. Recovery in this case is trivial: Rollback to the latest checkpoint if any, or to the beginning in case of no prior checkpoints, and continue from there on.

3.7.1.2 Recovery pattern RP2

Recovery pattern RP2 presents the rules to be applied to participants of checkpointing pattern CP2 for their recovery. In pattern CP2, receiver is checkpointed at its must save point due to which initiator of the pattern is checkpointed at its invocation point. The following rules specify recovery actions for CP2 depending on points of failure.

- Rule 7 (RR1): If the initiator of CP2 fails after taking the checkpoint and before receiving reply message, if any, rollback to the latest checkpoint and continue from there on.
- Rule 8 (RR2): If the initiator of CP2 fails after taking the checkpoint and reply message is also logged, rollback to the latest checkpoint, redo unsaved activities, replay the logged message and continue from there on.
- Rule 9 (RR3): If the receiver of CP2 fails before taking the checkpoint, rollback to the beginning, replay the logged request message and start execution.
- Rule 10 (RR4): If the receiver of CP2 fails after taking the checkpoint, rollback to the latest checkpoint, and continue execution from there on.

Figure 3.7 depicts checkpointing pattern CP2 and its recovery rules for the points of failure t1, t2, t3 and t4. For failure at t2(which is a failure in initiator after taking a checkpoint C1 and logging received reply message L2), Rule RR2 is used to perform recovery. Thus the recovery steps are: **1.** Rollback to its previous checkpoint C1, **2.** replay the logged message L2 and continue execution from there on. In the absence of this checkpointing and recovery scheme, recovery after failure at t2 would require the initiator to restart again, place costly call to ξ_1, receive reply message from it and then proceed its execution. The proposed recovery pattern aims to avoid this repetition of work in case of failure, yet it

guarantees to provide the same execution results. Similarly, recovery steps can be obtained for other points of failure as depicted in Figure 3.7.

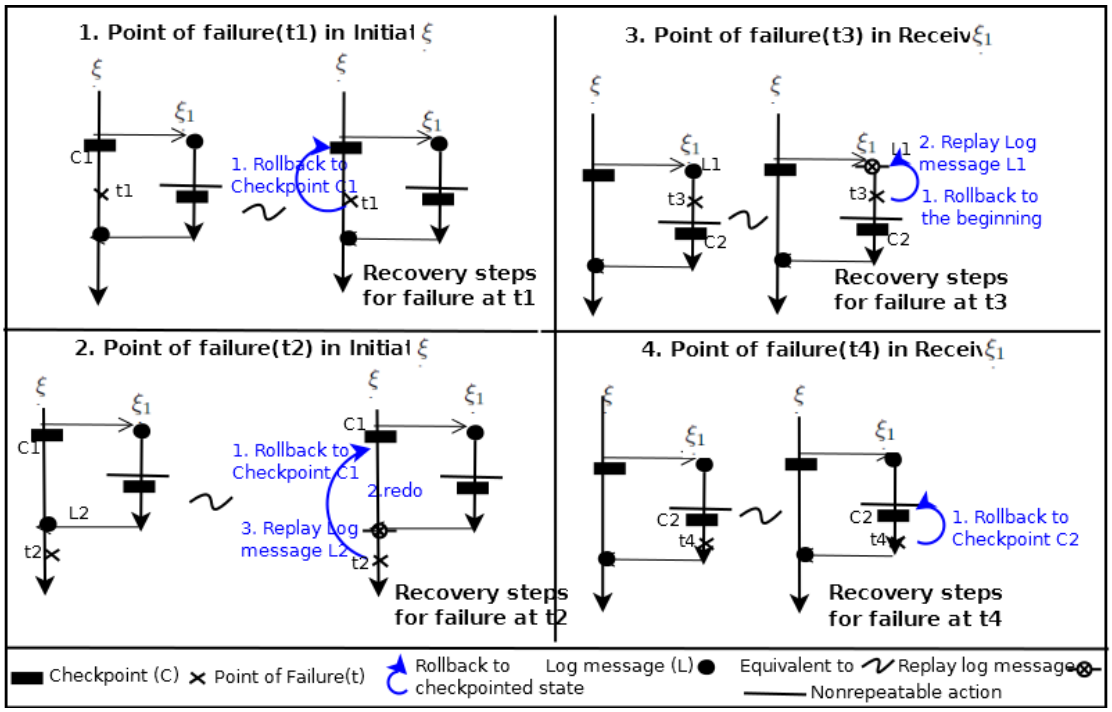


Figure 3.7: Checkpointing pattern CP2 and its Recovery Patterns

3.7.1.3 Recovery pattern RP3

Recovery pattern RP3 presents the rules to be applied to participants of checkpointing pattern CP3 for their recovery. In pattern CP3, initiator is checkpointed at its must save point due to which request message is logged at the receiver of the pattern. The following rules specify recovery actions for CP3 depending on points of failure.

- Rule 11 (RR5): If the initiator of CP3 fails before taking the checkpoint, rollback to the latest checkpoint if any, or to the beginning in case of no prior checkpoints, and continue from there on. Do not place the request message again of CP3 since it is logged at the receiver.

- Rule 12 (RR6): If the initiator of CP3 fails after taking the checkpoint and before receiving reply message, rollback to the checkpoint, and continue from there on.
- Rule 13 (RR7): If the initiator of CP3 fails after taking the checkpoint and reply message is also logged, rollback to the latest checkpoint, redo unsaved activities, replay the logged message and continue from there on.
- Rule 14 (RR8): If the receiver of CP3 fails, rollback to the beginning, replay the logged request message and continue execution from there on.

In the following section we prove the correctness of our proposed checkpointing and recovery rules.

3.8 Proof of correctness of recovery

In this section we first introduce notations required for proving correctness and then use these notations for the proof.

Let p represent a pattern and p' represent the pattern p with checkpoints and logs introduced as per the proposed checkpointing rules.

For ex, consider a pattern $p = \gamma^s_{rq}.(\Psi^s_0||\Psi^r_0).\gamma^r_{rp}.\tau^s$ (pattern type P2)

Let p' represent p with checkpoints and logs, $p' = \gamma^s_{rq}.ckp^s.(\Psi^s_0||\Psi^r_0).\gamma^r_{rp}.lg^s.\tau^s$ where p' represents checkpointing pattern CP2. ckp^s represents checkpointing activity done by ξ_s and lg^s represents logging of reply message activity by ξ_s

Let $execute(p, S)$ represent sequence of state changes of initiator ξ_s of p due to the execution of operations in the pattern p when ξ_s starts in state S. A **state** of a web service, is defined as a set of its local variables and their values which affect the computations done by the service.

Execution of a local activity by a participant results in its state change. Execution of checkpointing activity and logging activity do not result in change of local variables of a participant. Hence execution of these activities by ξ_s do not change its state.

Lemma 1: If p is a pattern and p' represents p with checkpoints and logs, then $execute(p, S) \equiv execute(p', S)$

Proof of this lemma follows from the fact that when a web services takes a checkpoint or logs an input message, values of its local variables do not change, i.e, its state does not change

Let the notation $S_a \xrightarrow{\Psi^s} S_b$ indicate that execution of a sequence of operations by ξ_s results in its state change from S_a to S_b. Sometimes this sequence of operations Ψ^s can be a single local activity or an interaction.

For example, $execute(p, S) = \mathrm{S} \xrightarrow{\gamma^s_{rq}} S_1 \xrightarrow{\Psi^s_0||\Psi^r_0} S_2 \xrightarrow{\gamma^r_{rp}} S_3 \xrightarrow{\tau^s} S_4$

$$\therefore execute(p, S) = S_4. \tag{3.1}$$

Similarly, $execute(p', S) = \mathrm{S} \xrightarrow{\gamma^s_{rq}} S_1 \xrightarrow{ckp^s} S_1 \xrightarrow{\Psi^s_0||\Psi^r_0} S_2 \xrightarrow{\gamma^r_{rp}} S_3 \xrightarrow{lg^s} S_3 \xrightarrow{\tau^s} S_4$

where S_1, S_2, S_3, S_4 represent subsequent states of ξ_s when it executes pattern p' from state S.

$$\therefore execute(p', S) = S_4. \tag{3.2}$$

Thus $execute(p, S) \equiv execute(p', S)$ $\square$

Let p'' represent p' with failure τ_f. For example $p'' = \gamma^s_{rq}.ckp^s.(\Psi^s_0||\Psi^r_0).\gamma^r_{rp}.lg^s.\tau_f$ represents p' with failure in initiator ξ_s after logging the reply message.

$$execute(p'', S) = S \xrightarrow{\gamma^s_{rq}} S_1 \xrightarrow{ckp^s} S_1 \xrightarrow{\Psi^s_0||\Psi^r_0} S_2 \xrightarrow{\gamma^r_{rp}} S_3 \xrightarrow{lg^s} S_3 \xrightarrow{\tau_f} S_f \tag{3.3}$$

The above statement implies that upon failure, ξ_s enters failure state S_f.

Let $restore(p'', S_f)$ represent sequence of state changes by applying recovery rules to the pattern p'' when ξ_s is in state S_f.

Thoerem: For a pattern p' with checkpoints and logs, its execution with failure and recovery is equivalent to its execution without failures. Stated otherwise: $execute(p'', S) + restore(p'', S_f) \equiv execute(p', S)$ where + operator denotes sequential execution of its operands.

Proof:

From equation 3.3 we have $execute(p'', S) = S_f$.

$\therefore execute(p'', S) + restore(p'', S_f) = S_f + restore(p'', S_f)$

In the example considered, p'' represents CP2 with a failure in ξ_s . Hence apply recovery pattern $RP2$ for recovery of ξ_s.

According to RP2 (Rule 8 (RR2)): If the initiator of CP2 fails after taking the checkpoint and reply message is also logged, rollback to the latest checkpoint, redo unsaved activities, replay the logged message and continue from there on.

According to RP2, $restore(p'', S_f) = S_f \xrightarrow{rollckp^s} S_1 \xrightarrow{\Psi_0^s} S2 \xrightarrow{replaylog^s} S3 \xrightarrow{\tau^s} S_4$

$\therefore execute(p'', S) + restore(p'', S_f) = S_4$.

From equation 3.2 we have $execute(p', S) = S4$.

Thus $execute(p', S) \equiv execute(p'', S) + restore(p'', S_f)$.

Thus proved. □

This theorem proves the correctness of our recovery rules. Similarly, correctness of other checkpointing and recovery rules can also be proved.

We present a framework called automatic checkpoint generation, that inserts checkpointing locations in a choreography diagram, using the formal model of patterns and algorithms presented in previous sections. It also performs the task of

recovery in case of transient failures. The following section illustrates the proposed framework.

3.9 Framework for Automatic Checkpoint Generation

In this section a framework is presented that supports automatic generation of checkpoints at design time in a given choreography. It mainly consists of two modules: 1. Design time Checkpointing Module (DSM) 2. Recovery module. DSM in turn consists of two submodules: Pattern identification module and Checkpointing module. Figure 3.8 presents the proposed framework.

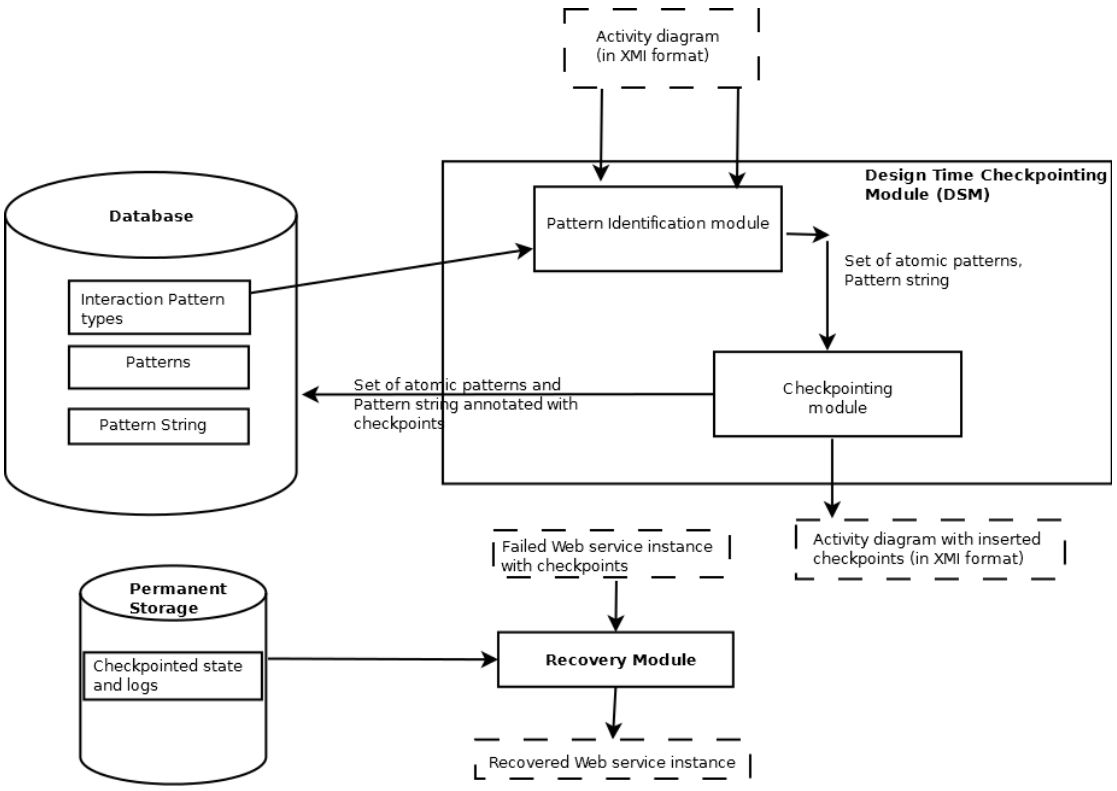


Figure 3.8: Framework for Automatic checkpoint generation

Pattern identification module takes as input, a choreography depicted in the form of a UML activity diagram and identifies atomic patterns present in the

given choreography, using the proposed patterns stored in the database. It then models the choreography as a composition of the identified patterns and expresses the given choreography as a pattern string. It outputs this pattern string PS and the set of atomic patterns SP which constitute the pattern string. It implements Pattern identification algorithm (presented in section 3.5).

Checkpointing module takes the pattern string and the set of patterns generated by pattern identification module, as input and then applies the checkpointing and logging rules to insert checkpointing locations in the choreography. It implements design time checkpointing algorithm (presented in section 3.6). Output of the checkpointing Module is the UML activity diagram which is given as input but has checkpointing locations appended to it. This output diagram helps the participating web services to insert checkpoints at appropriate locations in their code. The module also outputs the pattern string appended with design time checkpoints. This pattern string is used as input to checkpointing algorithms executed at later stages, i.e at deployment stage and execution stage. Figure 3.9 depicts the working of DSM.

Recovery module takes as input failed web service instance id. It restores the failed web service instance to the latest checkpointed state and replays logged messages as indicated by recovery rules. It may be noted here that the proposed recovery rules do not require recovery of other web services, other than the failed instance.

In the following subsection we present a possible implementation of our framework.

3.9.1 Experimentation

In this subsection an implementation of the proposed framework is presented. We have developed a tool called ACGT (Automatic Checkpoint Generation Tool) that inserts checkpoints into a given choreography.

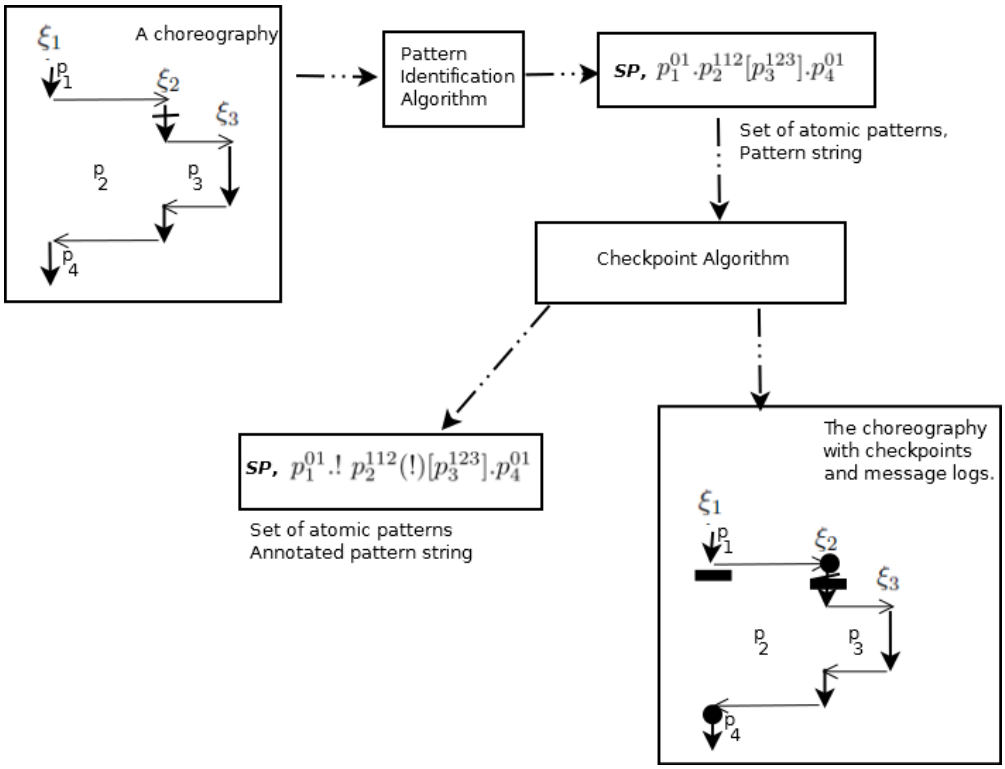


Figure 3.9: Working of DSM

ACGT is implemented in Java under Eclipse IDE. The following four step procedure is followed:

As a first step of implementation the choreography of web services is generated using UML Activity diagrams. Input to ACGT is a UML diagram that is generated using Visual Paradigm for UML 10.2 Enterprise Edition. Any other UML tool such as Rational Rose, Umbrella etc may also be used.

In the second step, the diagram is exported in XMI(XML Metadata Interchange) format. XMI is an OMG standard for exchanging metadata information which uses XML tags.

In the third step the XMI file generated in step 2 is given as input to ACGT. In order to extract patterns from XMI file, DOM (Document Object Model) and SAX(Simple API for XML) APIs are used. ACGT generates graphical elements representing checkpointing locations in the given UML activity diagram. The output again is in XMI format.

As a fourth step the XMI file generated in step three is imported in Visual Paradigm to view the diagram along with appended checkpoints.

Figure 3.10 (a) depicts an example choreography which is given as input to ACGT and Figureure3.10 (b) depicts the same choreography with checkpointing locations inserted by ACGT. In this example a choreography that consists of non repeatable actions (red coloured actions) is depicted. Here, ACGT generates checkpoints based on checkpointing rules $CR1$ and $CR2$ and message logging rules $LR1$ and $LR2$ resulting in checkpointing pattern $CP2$.

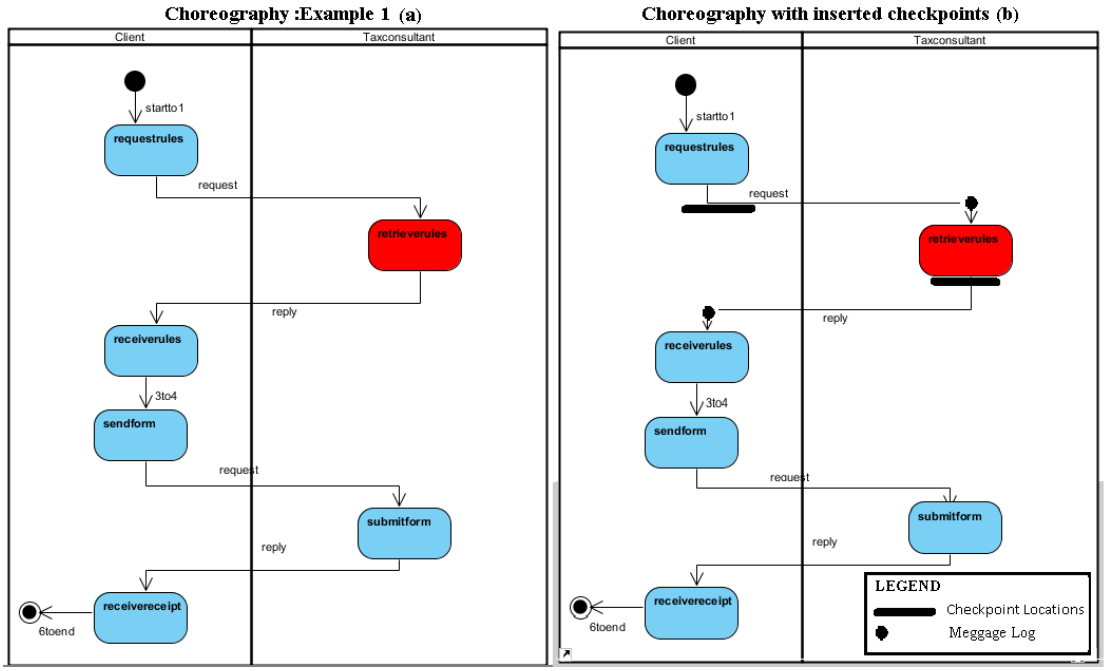


Figure 3.10: Checkpoints inserted into a choreography

3.10 Conclusion

In this chapter, we have modelled service choreographies using patterns and proposed design time checkpointing for choreographed web services. As a first step we have presented atomic patterns for modelling web service choreographies. We then presented composite patterns. We have proposed pattern based checkpointing rules that avoid reinvocation of services performing non repeatable actions.

We have also proposed recovery rules which specify sequence of operations to be executed to recover participants from their failure. We have then proposed two algorithms for generation of pattern string from a given choreography diagram and for inserting checkpoints into the given pattern string. We have implemented the idea by developing a tool called ACGT that reads a choreography document and inserts checkpoint marks into the document. ACGT models the choreography as a composition of proposed patterns. It then applies the proposed checkpointing policy to insert checkpoints into the choreography. The checkpoints generated by ACGT have to be inserted in code during the development of constituent web services. ACGT also includes a recovery module that restores back a failed web service instance to its latest checkpointed state using the proposed recovery rules.

The proposed interaction pattern model can model those choreographies which can be represented using UML activity diagrams. The complexity of design time checkpointing algorithm is $\bigcirc(n^2)$ where n represents number of atomic patterns in the pattern string. Since insertion of design time checkpoints is an offline activity, the algorithm complexity does not affect the performance of web services developed.

This chapter does not talk about satisfying execution time and cost constraints even in case of failed and recovered service instances. In the next chapter, we take up this issue and present a deployment time checkpointing algorithm that ensures constraint satisfaction in all execution instances.

CHAPTER 4

Deployment Time Checkpointing

Abstract We propose "Time and cost aware checkpointing algorithm[1]," that inserts checkpoints in a composite web service ξ so that time and cost deadlines are met even in the case of failure and recovery of ξ. The quantities used by this algorithm like checkpointing time, logging time etc can be measured only after development of the services and when they are ready to be deployed. Hence we call this stage as deployment checkpointing. For the purpose of deployment checkpointing, we define components as sequential patterns initiated by ξ. Using them we define recovery components and present the computation of checkpointing score, which is used while making checkpointing decisions. We present experimental results analysing number of checkpoints to be inserted at deployment time.

4.1 Introduction

In the previous chapter we have presented design time checkpointing that introduces checkpoint locations in a choreography, at places where non repeatable actions are performed. In the event of transient failures(temporary failures) this checkpoint arrangement avoids reinvocation of any web service that performs a non repeatable action. But, in a service oriented environment every web service has to provide the requested service with in a stipulated cost and time. Our design time checkpointing does not consider the issue of meeting execution time and cost constraints in case of transient failures. In order to address this issue we make

[1]**Vani Vathsala, A. and Hrushikesha Mohanty (2015)**. Time and cost aware checkpointing of choreographed web services. Proc of the 11th International Conference on Distributed Computing and Information Technology, pp 207 to 219. Bhubaneshwar, 2015. Proceedings in LNCS, Springer, Vol 8956.

use of *"Quality of Service"* (QoS) attributes of web services like response time, reliability, cost of service while taking checkpointing decisions.

QoS aware checkpointing has been proposed in various areas like embedded systems Chen *et al.* (2009), mobile computing P.J.Darby and Tzeng (2010) and multimedia network systems Osada and Higaki (2001). In Chen *et al.* (2009) authors propose QoS aware message logging based checkpointing of embedded and distributed systems. They do not consider QoS values of other processes participating in the composition, in checkpointing decisions. The work proposed in P.J.Darby and Tzeng (2010) is for mobile computing environment where in hosts going out of range transfer their checkpointed data to other hosts. Reliability of a mobile link is calculated using factors like signal strength, distance between the two MHS etc. It uses link reliability values to dynamically maintain superior checkpointing arrangement. Authors of Osada and Higaki (2001) propose a checkpointing protocol for multimedia network systems, which is similar to our approach where in checkpoints are at the points of interaction. It considers time to checkpoint should be limited to a range that is predefined. This time represents a QoS attribute. Number of packets that can be lost is also limited and is taken as a QoS attribute. It proposes a checkpoint protocol which ensures that quality of service is maintained and there are no lost or orphan messages in the system. The QoS attributes that they have chosen are not the QoS attributes that are specific to web services (like response time, cost of service etc). It also allows certain parts of a large message to be lost in transition to maintain the required timeliness, which is not desirable in case of web services.

Considering QoS values of the constituent services while taking checkpointing decisions for a choreographed web service is pivotal in meeting promised deadlines. In our survey we have not come across any web service checkpointing strategy that focuses on this issue. ***Hence in this chapter we advocate Time and Cost aware checkpointing that makes use of QoS values of constituent services, to decide on checkpoint locations to meet the promised deadlines.***

The proposed "Time and cost aware checkpointing algorithm" introduces minimum number of checkpoints, in addition to design time checkpoints, into a composite web service ξ so that i) execution time and cost constraints are met even in the event of transient failures and ii) failure free service instances take less execution time. To take checkpointing decisions, we divide the web service ξ into smaller execution units called **recovery components** that are bound by checkpoints. Then we compute recovery time overhead and recovery cost overhead for each of them. Recovery overhead is calculated using i)Deployment time measurable quantities like execution time, checkpointing time, message logging time and message replay time. ii) QoS values of ξ and the web services it invokes. For each of the recovery components a checkpointing score is calculated using the computed recovery overhead. We use this checkpointing score to select one of the recovery components for checkpointing. The trade-off between number of checkpoints and recovery time is experimentally analysed.

The proposed "Time and cost aware checkpointing algorithm" has to be applied to all participants of a choreography so that each participant can satisfy its time and cost constraints. But the particular sequence in which it has to be applied to participants is governed by interactions among them. This sequence ensures that failure and recovery of a participant does not result in reinvocation of a checkpointed participant. We detail upon this policy in section 4.2.2.

At the end of this chapter we describe two scenarios for which additional steps/extra care has to be taken to apply the proposed algorithm. 1) Checkpointing a web service which is atomic, i.e does not invoke any other web service. 2) Checkpointing a web service with multiple paths and invokes different set of web services in each path.

It may be noted that the terms "web service" and "participant (of a choreography)" are used interchangeably in this chapter. This chapter is organised as follows: in section 4.2 we give a motivating example and present sequence determination policy, introduce time and cost aware checkpointing strategy and extend

the interaction pattern model to include recovery components. In section 4.3 we detail on the procedure to be followed for deployment time checkpointing, describing necessary computations to be done. Formal model introduced in previous chapter is extended in section 4.4 to include concepts introduced in this chapter. Checkpointing algorithm is given in section 4.5 along with experimental results. Special scenarios are dealt in section 4.6 and Conclusion of this chapter work is discussed in section 4.7.

In the next section we present concepts required here for putting the proposed strategy at right perspective.

4.2 Checkpointing At Deployment Time

In this section we detail on concepts upon which "Time and cost aware checkpointing algorithm" is built. We assume that occurrence of transient failures in a web service follow poisson distribution with the mean failure rate given by λ. It is assumed that failures do not occur during recovery time. Each failure is immediately followed by recovery of the failed service for successful completion of its execution.

Prior to introducing our proposed strategy, we present, in the subsection which follows, an example choreography that is used for demonstrating the concepts proposed in this chapter.

4.2.1 Motivating example

The motivating example taken up in previous chapter was deliberately chosen to be simple to aid in easy comprehension of concepts. In this chapter, we take up a larger choreography as the motivating example so that it suffices to describe all the concepts introduced in this chapter. Without going into business details, Figure

4.1 presents a choreography that we use in this chapter for demonstration of the concepts. The composite web service ξ invokes six more web services ξ_1 to ξ_6. All these seven web services are called as participants of the choreography. It may be noted that participant ξ_2 performs a non repeatable action in this choreography. The pattern string for the example choreography is $p_a.p_b[p_c].p_d[p_e].p_f$. Figure 4.2 (a) depicts the choreography with checkpoints inserted at design stage. The participants of this choreography are then checkpointed at their deployment time using the proposed "*Time and cost aware checkpointing algorithm*".

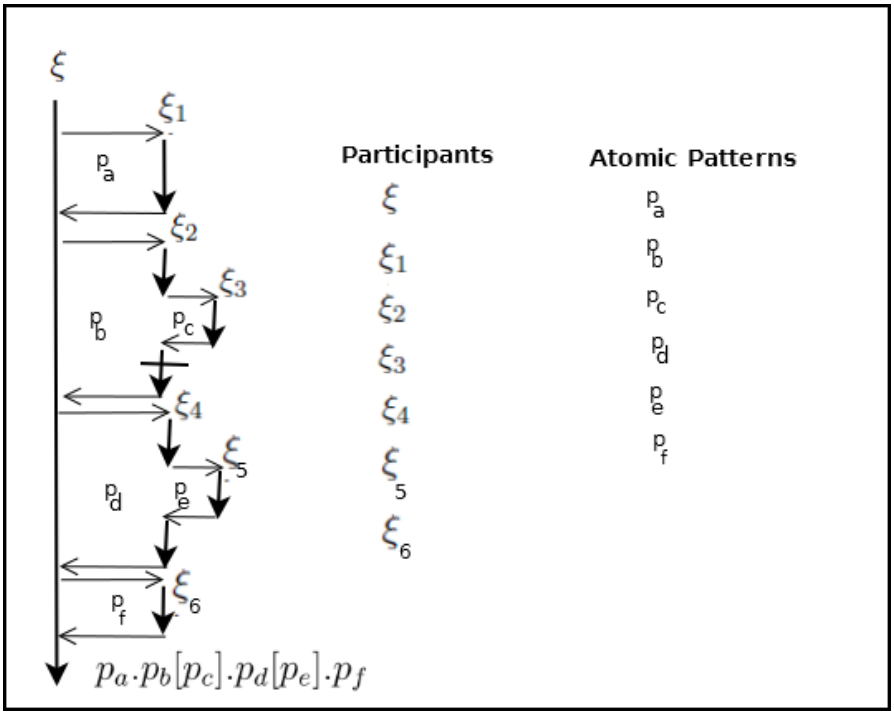


Figure 4.1: Example Choreography

Before delving into details of "*Time and cost aware checkpointing algorithm*", we first present sequence determination policy which describes the order in which the algorithm has to be applied to participants of a choreography.

4.2.2 Sequence determination policy

The proposed "*Time and cost aware checkpointing algorithm*" has to be applied at the time of deployment to all the participants of a given choreography so that

every participant meets time and cost constraints. But the sequence in which it has to be applied among the participants is given by "*Sequence determination policy*". This policy determines the sequence based upon interaction dependencies among the participants. If a web service ξ_r, invoked by a web service ξ, introduces a checkpoint in its code due to *time and cost aware checkpointing algorithm*, then its caller ξ has to insert checkpoint in its code after calling ξ_r, at the invocation point. This arrangement is required to avoid reinvocation of checkpointed web service ξ_r in the event of failure and recovery of ξ.

According to this policy:

1. Each participant has to prepare a list of its callers and also a list of the participants which it calls, i.e callee list.

2. A participant ξ can proceed with execution of "*Time and cost aware checkpointing algorithm*" after satisfying the conditions specified below.

3. Initially, each participant ξ which does not have any callee can proceed with execution of "*Time and cost aware checkpointing algorithm*".

4. *Completion step:* After executing the algorithm and fixing deployment time checkpoints, ξ has to send a *completion message* to all the participants in its callers list stating whether it has introduced any new checkpoints at deployment or not. If ξ has introduced atleast one checkpoint at deployment, then all its callers must introduce checkpoints at their respective invocation points. Also, reply message from ξ must be logged at each of its callers. ξ has to log the request messages received from each of its callers.

5. A participant which has callees can proceed with execution of "*Time and cost aware checkpointing algorithm*" only after it receives completion messages from all its callees. After completion, it has to send completion message to all its callers as detailed in the *Completion step* described above.

According to the sequence determination policy ξ_1, ξ_3, ξ_5 and ξ_6, which are atomic web services, should execute the "*Time and cost aware checkpointing algorithm*" initially and send completion messages to their callers. Then ξ_2 and ξ_4 should execute the algorithm and send messages to their caller, ξ. After getting completion messages from all its callees, i.e ξ_1, ξ_2, ξ_4 and ξ_6, ξ should execute the algorithm. **We illustrate application of the algorithm to the composite web service ξ, the algorithm is applicable to other constituent web services also.** The approach to be followed for applying the algorithm to atomic web services ξ_1, ξ_3, ξ_5 and ξ_6 is presented in subsection 4.6.2

In the following subsection we detail upon our strategy used for checkpointing web services.

4.2.3 Checkpointing strategy

As discussed in the introduction section, every web service has to deliver a requested operation within the advertised time and agreed upon cost, i.e every execution of a web service has to meet time and cost constraints. Since the execution of a composite web service involves execution of other invoked web services also total time and cost computations must include the response time and cost of invoked web services also. Any checkpointing policy must consider the following two issues:

1. **Performance during failure free executions:** One major concern while determining number of checkpoints and their locations is total execution time of the participant during failure free executions (successful executions which do not encounter any transient failures). During execution, additional activities performed to deal with failures are: 1)Save or checkpoint the current state at all identified checkpoint locations 2)Log messages which are marked for logging. These two activities induce additional time. Increase in execution time due to checkpoints should be as low as possible. Hence

placing too many checkpoints and message logs is not a good decision as it slows down execution time of failure free instances.

2. **Meeting time and cost constraints for failed and recovered executions:** Every participant has to deliver the service within the advertised time and agreed upon cost, even in case of failed and recovered instances. But, recovery of a web service has the following two overheads: Recovery time overhead and Recovery cost overhead. **Recovery time overhead** is the additional time required during recovery that includes: i)rollback to the checkpointed state ii) replay logged messaged iii) Reexecute unsaved activities. **Recovery cost overhead** is the additional overhead to be paid to reinvoke a constituent web service in case its reply is not logged. Together, recovery time overhead and recovery cost overhead are called as **recovery overhead**. Due to reexecution of unsaved activities, the total execution time and cost increase and, many a times overshoot the promised time and cost constraints.

Keeping these issues in mind we propose a checkpointing policy which **aims at inserting minimum number of checkpoints to reduce recovery overhead** in case of failures so that constraints are met. Minimum number of checkpoints ensure minimal additional execution time in case of failure free executions. Minimal recovery overhead requires more number of checkpoints to be inserted which results in undesirable increase in time for failure free executions. Hence we do not aim at minimal recovery overhead. Instead, we aim at inserting minimum number of checkpoints that are required to satisfy time and cost constraints even in case of failures, but which result in minimum overhead during failure free executions.

As described in introduction, our checkpointing strategy utilises QoS values of the web services involved. In the following subsection we define each of the QoS values considered while taking checkpointing decisions.

4.2.3.1 QoS attributes in checkpointing

Most commonly used QoS attributes of web services are response time, throughput, scalability, availability, reliability, maintainability, integrity, safety, cost etc. Certain QoS attributes have an important role in deciding checkpointing locations. For example: if the cost of an invoked operation is high, a checkpoint placed after the operation completion avoids reinvocation of the costly operation in case of failure. If the reliability of an invoked service is less and its average response time is high, appropriately placed checkpoints reduce the recovery time in case of failures. We have identified and modelled those QoS attributes which play a crucial role in checkpointing decisions. Following is the list of considered QoS attributes:

- **Response time** rt**:** Response time of a service is the difference between the service delivery time and its corresponding service invocation time. Web services advertise response time values to aid users in service selection. If not advertised, there are several third party services which provide QoS values of requested web services. Alternatively, since the web service ξ_r would have been already deployed, the service requester ξ may even experimentally invoke ξ_r and compute average response time of ξ_r for using in checkpointing decisions.

  In case of one way pattern types P_3, P_4 there would be no reply message and hence their response time is taken as 0.

- **Vulnerability** vl**:** Vulnerability of a service to failures is defined using reliability of the service. **Reliability** rl**:** is defined as the success rate of the service i.e. the ratio of number of times the service is successfully delivered to total number of service invocations. Web services advertise their reliability values using which vulnerability can be computed. $vl = 1 - (rl)$.

- **Cost of service** ct**:** Cost of a service is defined as the price to be paid for providing the requested operation. This cost of service could be obtained from service level agreement documents.

Having specified the QoS attributes required for checkpointing, we proceed to introduce the concepts required for computing recovery overhead. We define smaller execution units called as *sequential components* and *recovery components* for computing recovery overhead. These components are defined, in the following subsection, using *interaction patterns* defined in section 3.3 of the previous chapter.

4.2.4 Extension to Interaction pattern model

Choreography document describes series of interactions that are to be performed by constituent web services to accomplish common business goals. A choreography of web services is modelled as a composition of interaction patterns as described in section 3.3 of the previous chapter. In this subsection we extend this model and introduce what are called as *Sequential components* and *Recovery components* to aid in checkpointing decisions.

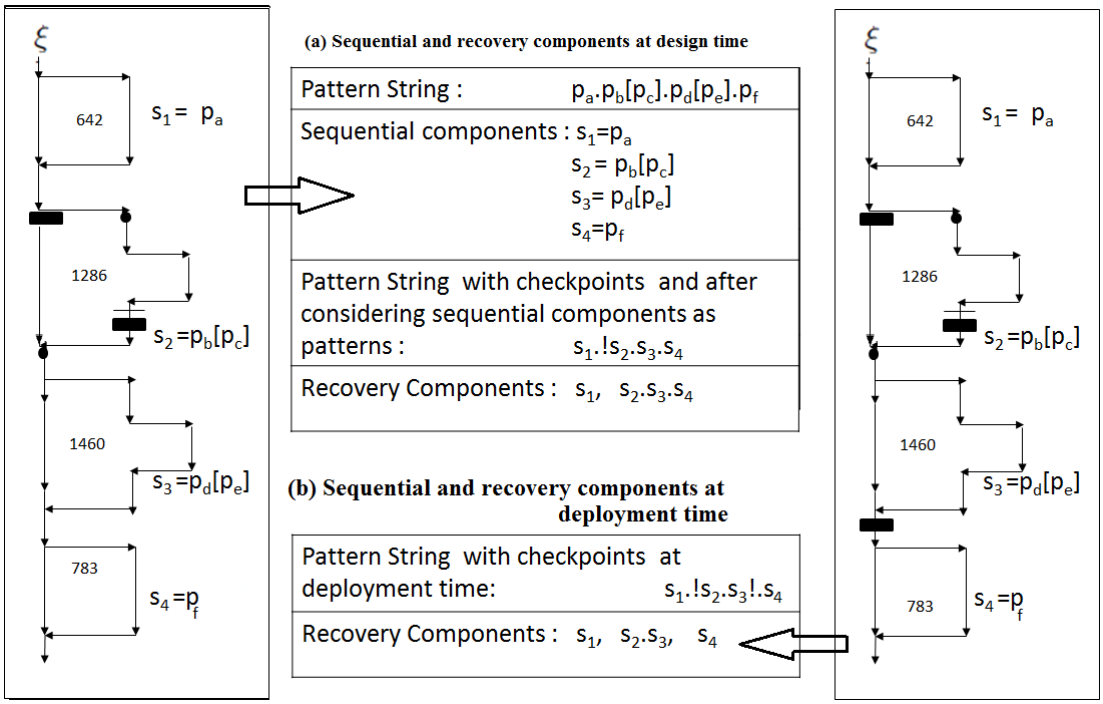


Figure 4.2: A choreography and its recovery components

4.2.4.1 Sequential components

The pattern string which depicts a constituent web service is extracted from the complete pattern string which depicts the entire choreography. For example: 1) the pattern string which depicts the composite web service ξ_2 is $p^0[p_c]p^0$. 2)the pattern string which depicts the composite web service ξ is $p_a.p_b[p_c].p_d[p_e].p_f$. To checkpoint the participant ξ, we consider the pattern string depicting the composite service ξ and identify sequential components from it. The part of pattern string which is delimited by "." operator is called as a **sequential component**, denoted by the letter "s". The sequential components for ξ, in the example, are p_a, $p_b[p_c]$, $p_d[p_e]$, p_f which are named as s_1, s_2, s_3, s_4 respectively (Fig 4.2(a)). There is only one sequential component $p^0[p_c]p^0$ for ξ_2 in the example. The sequential component that represents ξ_3 is p^0. The case of checkpointing participants with only one sequential component is trivial and is explained later. Now, we proceed with checkpointing a composite web service which contains more than one sequential component.

The participant to be checkpointed, ξ is expressed in terms of sequential components, i.e $s_1.s_2.s_3.s_4$. This string is annotated with checkpoints inserted into ξ at design time. The annotated string for ξ is $s_1.!s_2.s_3.s_4$.

Sequential components are referred to as **components** for ease of writing from here on. Each component is a pattern(can be either atomic or composite) and hence has C-points associated with it. If the web service ξ initiates n_s number of components we have a maximum of $2n_s$ C-points to be converted into checkpoints. (Every component has two C-Points that can be checkpointed in its initiator at deployment time: Invocation point and service point. All must save points are checkpointed at design time itself). It may not be possible to convert all these C-points since conversion of all the C-points increases total execution time and might result in violation of deadlines. To decide on checkpoint locations, we use recovery components which are introduced and explained in the following subsection.

4.2.4.2 Recovery components

A recovery component is defined as an execution unit that is delimited by checkpoints. A failure at anywhere in a recovery component results in rollback to the checkpoint placed at the beginning of the recovery component.

For the participant ξ, we have initially two recovery components $s_1, s_2.s_3.s_4$ for the example choreography. Figure 4.2(b) depicts the choreography and its recovery components after a service point is converted into checkpoint at deployment time. By this time we have three recovery components $s_1, s_2.s_3, s_4$.

In this section we have introduced the concepts and discussed on the issues considered while checkpointing at deployment. In the following section we demonstrate the process to be adopted for deployment time checkpointing.

4.3 Checkpointing Process

This section delves into details of checkpointing process, by resting on the concepts presented in the previous section. Deployment time checkpointing of a web service ξ is performed in three stages. 1) Measurement of execution time and other quantities of ξ at deployment time. 2)Collection of QoS values of constituent services. 3)Computation of total execution time and cost with failure recovery, and placement of checkpoints. Each of them is elucidated in the following subsections.

4.3.1 Measurement of quantities

Let ξ be the participant which is currently being checkpointed. Let T_C, T_L, T_R and T_{CR} represent checkpointing time, message logging time, message replay time and time to restore ξ to a saved state, respectively. These quantities are determined experimentally at deployment time. Let C_D represent the cost of service charged by ξ when service is provided with in the promised maximum execution time T_D.

4.3.2 Collection and computation of QoS values

The considered QoS attributes are response time, vulnerability and cost of service provision. Web services advertise these values to aid users in service selection which are collected at deployment stage, for all web services invoked by ξ.

In this subsection, we delve into details of computing QoS values for a component s. These QoS values are utilised in recovery overhead computations for each of the components and also for ξ as a whole. Component s may be an atomic pattern or a composite pattern. We illustrate here it's QoS computation for each possible case.

Case 1: If component s is an atomic pattern as shown in figure 4.3(a) and ξ_r is the service provider in the component s, then QoS values of s are derived from those of ξ_r as follows:

- **Response time** $s.t_{rt}$**:** Response time of a component s is defined as the response time of its service provider ξ_r.
- **Vulnerability** $s.vl$**:** Vulnerability of s to failures is defined as vulnerability of ξ_r which in turn is defined using reliability of ξ_r. **Reliability** rl**:** $\xi_r.vl = 1 - (\xi_r.rl)$ and $s.vl = \xi_r.vl$.
- **Cost of service** $s.ct$**:** Cost of service of s is defined as cost of service provided by ξ_r of the component. $s.ct = \xi_r.ct$.

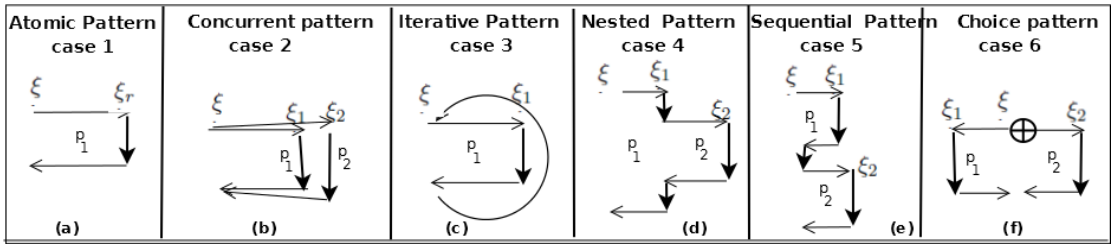


Figure 4.3: Component s and its composition cases

Case 2: If component s is a parallel composition of patterns p_1 and p_2 as shown in figure 4.3(b) and ξ_1 and ξ_2 are the service providers in these patterns

respectively, then QoS values of s are derived as follows:

- **Response time** $s.t_{rt} = max(\xi_1.t_{rt}, \xi_2.t_{rt})$
- **Vulnerability** $s.vl = 1 - min(\xi_1.rl, \xi_2.rl)$
- **Cost of service** $s.ct = \xi_1.ct + \xi_2.ct$

Case 3: If component s is an iterative composition of pattern p_1 as shown in figure 4.3(c) and ξ_1 is the service provider then QoS values of s are derived as follows: (n is the iteration count)

- **Response time** $s.t_{rt} = (\xi_1.t_{rt} * n)$
- **Vulnerability** $s.vl = 1 - (\xi_1.rl * n)$
- **Cost of service** $s.ct = \xi_1.ct * n$

Case 4: If component s is a nested composition of patterns p_1 and p_2 as shown in figure 4.3(d), then advertised QoS values of ξ_1 would have included those of ξ_2. Thus QoS values of component s would be calculated from those of ξ_1 as in case 1.

Case 5: If component s is a sequential composition of patterns p_1 and p_2 as shown in figure 4.3(e), then p_1 and p_2 would themselves be sequential components of ξ and their QoS values would be calculated as in case 1.

Case 6: If component s is a composite pattern and composition operator is $\oplus$ as shown in figure 4.3(f), then multiple paths of execution result in ξ and is presented as a special case in the subsection 4.6.2.

In certain cases where dynamic composition is allowed, computation of QoS values of service provider ξ_r in component s attracts special attention. This is discussed in detail in the following subsection.

4.3.3 Provisioning of dynamic composition

A web service composition is said to be dynamic if one or more of the constituent services are selected dynamically at runtime. There are several reasons for allowing dynamism in a composition. Important among them are:

1. **QoS Constraints:** In a quest to serve the requesters with best values for QoS attributes, a composite service might decide to select some of its constituent services dynamically. A service that has optimal values for QoS attributes will be selected dynamically.
2. **Openness:** Service Oriented Architecture (SOA) facilitates openness to all available services. This enables a composite service to support openness and select its constituent services dynamically at run time from all available services instead of being biased to preselected services.
3. **Error handling and substitution:** In case of permanent failure of one of the constituent services, a composite service has to dynamically select an equivalent service and substitute it in the place of failed service so as to ensure seamless execution of the composite service.

Dynamic composition of a composite service is done in two stages: 1) Discovery of services that match the required functionality 2) Selection of a particular service from the set of matching services. A service that provides optimal QoS values is generally selected from the pool of matching services. Both discovery and selection of matching services can be dynamic or just selection of a particular service is done dynamically from a list of statically discovered services. Selection of a suitable option depends on requirements and architecture of the application being designed and developed. The exact selection strategy and approach to be used are out of scope of our work and hence we do not detail further in this aspect. In this context what is important to us is whether there is any dynamic composition provision or not in the current participant being checkpointed. In case there is no such

provision, no extra care needs to be taken. Otherwise, special attention needs to be paid while calculating recovery overhead, which is discussed in the following subsection.

4.3.3.1 Impact of dynamic composition on deployment time checkpointing

If a composite web service provides a provision for dynamic composition, then we do not know in advance, the exact service provider ξ_r that would be invoked by the composite service ξ at run time. In such a case, the QoS values (response time, reliability and cost) of a dynamically selected service provider, cannot be known in advance. These values are required for computing recovery overhead at deployment time.

As discussed at the beginning of this subsection, dynamic selection of a service provider ξ_r can be done either from statically discovered list of services or from dynamically discovered list of services. In earlier case, we would have the list of all services, and one among them would be selected at run time. Hence advertised QoS values of all these services would be available at deployment time, and the various QoS values to be used for recovery overhead calculations are computed as follows: Let SD be the set of statically discovered list of services and $|SD| = d$.

1. **Response time:** $\xi_r.t_{rt} = max_{i=1}^{d}(\xi_i.t_{rt})$ where $\xi_i \in SD$.
2. **Reliability:** $\xi_r.rl = min_{i=1}^{d}(\xi_i.rl)$ where $\xi_i \in SD$.
3. **Cost of service:** $\xi_r.ct = max_{i=1}^{d}(\xi_i.ct)$ where $\xi_i \in SD$.

In case of dynamic discovery of services, we would not have, at deployment time, any list of services to be invoked at run time. Hence the web service ξ must set minimum permissible value for reliability and maximum permissible values for response time and cost using its QoS requirements and QoS values of other partic-

ipants. The values set should be used in computations performed at deployment time.

After detailing on QoS values derivation, we now proceed to demonstrate computation of total execution time and cost with failure recovery in the next subsection.

4.3.4 Computation of total execution time and cost with failure recovery

In this section we define **total execution time and cost with failure recovery** for ξ and detail upon their computation. We begin with computation of total execution time with failure recovery. In each of the subsections that follow we first introduce relevant terms and then proceed to define total execution time /cost with failure recovery.

4.3.4.1 Total execution time with failure recovery

For each of the components s, initiated by the participant ξ, let $s.t_{at}$ and $s.t_{rt}$ represent average local activity time and response time of the callee. These values are determined experimentally at deployment time. For every component s, there can be only one invocation point, one service point, and one or more must save points in ξ. According to our design time checkpointing policy, only invocation point and must save points of a component may be converted into checkpoints, service points cannot be. Hence, initially $s.n_{mp} \geq 0$ and $s.n_{ip} \in (0, 1)$ represent number of must save points and invocation points which are converted into checkpoints at design time. Similarly, initial value of $s.n_{sp} = 0$ represents number of service points which are converted into checkpoints. According to the design time policy, $s.n_{sp} = 0$ indicating that zero service points are checkpointed at design time.

Let $T_{pure}(s)$ represent pure execution time of a component s without includ-

ing any of checkpointing and message logging times. $T_{pure}(s) = max(s.t_{at}, s.t_{rt})$. Let $T_{fixed}(s)$ represent the fixed execution time of component s after including checkpointing and message logging times for checkpoints and logs inserted at design time. Checkpointing time is given by $(s.n_{mp} + s.n_{ip}) * T_C$. These checkpoints are never revised at deployment time and run time. Hence we call it as $T_{fixed}(s)$. If an invocation point or a must save point is converted into checkpoint, the reply message received by ξ has to be logged. Let $s.n_{ml}$ represent number of message logs in component s. At design stage, $s.n_{ml} \in (0, 1)$. $T_{fixed}(s) = T_{pure}(s) + (s.n_{mp} + s.n_{ip}) * T_C + s.n_{ml} * T_L$.

We use the notation $\alpha_0(s)$ to represent $T_{fixed}(s)$, 0 indicates that 0 C-points of s are converted into checkpoint in deployment stage. Let $T_{fixed}(\xi)$ represent fixed execution time of the participant ξ before converting any of the C-points of ξ into checkpoints in deployment stage. It is in short represented as $\alpha_0(\xi)$ where 0 indicates that 0 C-points are already converted into checkpoints in deployment stage. $T_{fixed}(\xi) = \alpha_0(\xi) = \sum_{i=1}^{n_s} \alpha_0(s_i)$ where n_s is number of components in ξ.

$\alpha_k(\xi)$ represents **failure free execution time** of ξ after k C-points are converted into checkpoints and lg messages are logged in deployment stage. $\alpha_k(\xi) = \sum_{i=1}^{n_s} \alpha_0(s_i) + k * T_C + lg * T_L$

Let $T_{W_k}(\xi)$ represent **total execution time with failure recovery** of the participant ξ with k deployment time checkpoints, which is defined as follows: $T_{W_k}(\xi) = \alpha_k(\xi) + T_{rec_k}(\xi)$, which includes $\alpha_k(\xi)$, and time $T_{rec_k}(\xi)$ to recover the web service in case of its failure. To satisfy the promised maximum execution time, $T_{W_k}(\xi)$ must be $\leq T_D$. In the next subsection we describe recovery time overhead $T_{rec_k}(\xi)$ computation.

Recovery time overhead computation:
Let $T_r(s)$ represent recovery time of a component s. In a component if its IP or VP is checkpointed, then the service $s.\xi_r$ is not invoked again. In such a case $T_r(s)$ includes i) the time taken, T_{CR}, to restore the web service from its latest

checkpoint, ii) message replay time T_R and iii) time to execute unsaved local activities. Maximum unsaved work in a component results when a failure occurs almost at the end of the component, i.e just before receiving reply.

If s's IP or a VP is already converted into checkpoint then

$$T_r(s) = T_{CR} + T_R + s.t_{at} \quad \text{else} \quad T_r(s) = max(s.t_{at}, s.t_{rt}) \tag{4.1}$$

Table 4.2 (fifth column) gives recovery time values of components for the example choreography. Let $T_{rec}(s)$ represent recovery time overhead of s when ξ fails at rate of λ.

$$T_{rec}(s) = f(s) * T_r(s) \tag{4.2}$$

where $f(s) = (\lambda * T_r(s))$ gives average number of failures in the component s.

$$T_{rec_k}(r) = \sum_{s \in r} T_{rec}(s) \tag{4.3}$$

where $T_{rec_k}(r)$ represents recovery time overhead of r when there are k number of deployment checkpoints.

$T_{rec_k}(\xi) = \sum_{i=1}^{n_k} T_{rec_k}(r_i)$ where n_k is the number of recovery components of ξ with k deployment time checkpoints.

Relative recovery time of component s is defined as: $T_{rrec_k}(s) = \frac{T_{rec}(s)}{T_{rec_k}(r)}$.
Relative recovery time of recovery component r is: $T_{rrec_k}(r) = \frac{T_{rec_k}(r)}{T_{rec_k}(\xi)}$.

4.3.4.2 Total execution cost with failure recovery

Let C_{total} indicate failure free cost of service for all the components of ξ, $C_{total} = \sum_{i=1}^{n_s} s_i.ct$.

Let $C_r(s)$ indicate recovery cost incurred in component s during recovery in case of a failure. In a component if its IP or VP is checkpointed, then the service

ξ_r is not invoked again and hence there is no additional cost of invocation. Thus,

$$C_r(s) = 0 \;\; if \;\; (s.n_{ip}||s.n_{vp1})! = 0, \;\; else \;\; C_r(s) = s.ct \tag{4.4}$$

Recovery cost overhead computation:
Let $C_{rec}(s)$ indicate recovery cost overhead incurred in component s during recovery if ξ fails at failure rate λ.

$$C_{rec}(s) = f(s) * C_r(s) \tag{4.5}$$

$$C_{rec_k}(r) = \sum_{s \in r} C_{rec}(s) \tag{4.6}$$

$$C_{rec_k}(\xi) = \textstyle\sum_{i=1}^{n_k} C_{rec_k}(r_i).$$

Relative recovery cost for s is defined as: $C_{rrec_k}(s) = \frac{C_{rec}(s)}{C_{rec_k}(r)}$.
Relative recovery cost for r is defined as: $C_{rrec_k}(r) = \frac{C_{rec_k}(r)}{C_{rec_k}(\xi)}$.
Let $C_{W_k}(\xi)$ indicate total execution cost with failure recovery of service, which is defined as $C_{W_k}(\xi) = C_{total} + C_{rec_k}(\xi)$. Cost constraint is satisfied if $C_{W_k}(\xi) \leq C_D$.

4.3.4.3 Checkpointing score

Checkpointing score cs of a component s quantifies suitability of the component for checkpointing. It is defined as the following weighted sum. Higher values of cs indicate greater need for checkpointing the component.

$$s.cs = W_1 * T_{rrec_k}(s) + W_2 * s.vl + W_3 * C_{rrec_k}(s)$$

where $\sum_{i=1}^{3} W_i = 1$ and they represent weights to be used to alter the effects of individual components on the checkpointing score. Similarly, checkpointing score cs of a recovery component r is defined as:

$$r.cs = W_1 * T_{rrec_k}(r) + W_2 * r.vl + W_3 * C_{rrec_k}(r).$$

where $r.vl$ is defined as maximum value from among the vulnerability values of its components.

4.3.4.4 Deciding on checkpoint locations

Let recovery component r_j have the highest checkpointing score and is selected for checkpointing. From among all components of r_j, the component s_o that has got maximum checkpointing score is selected for placing the checkpoint. Further, if local activity time of s_o is greater than or equal to s_o's response time the next checkpoint is placed at service point of s_o. Else the next checkpoint is placed at invocation point of s_o, i.e if s_o's response time is larger, the service provider of s_o is not reinvoked in the event of failure of ξ by placing a checkpoint at the invocation point (refer to Figure 4.4 (b,a)).

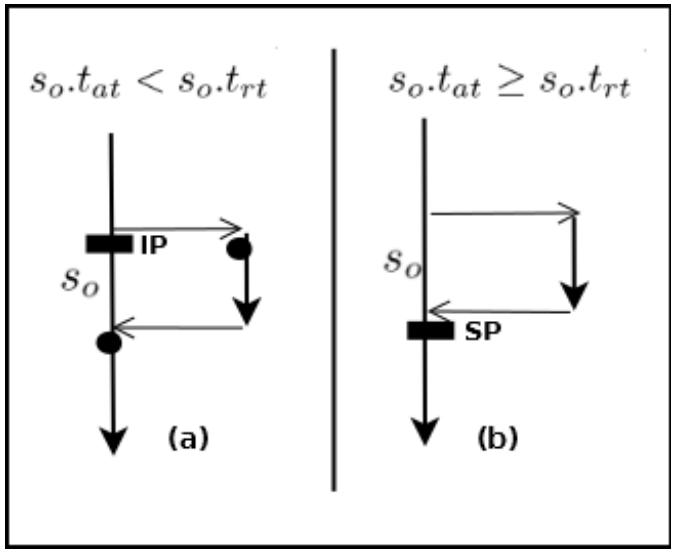


Figure 4.4: Deployment Time Checkpointing

The computations presented in this section are the crux of the proposed *"Time and Cost Aware Checkpointing Algorithm"*. Before presenting the algorithm we include the sequential and recovery components in the interaction pattern model, so as to capture all the details required for the algorithm.

4.4 Extending Interaction Pattern Model With Components

In this section, we extend the formal model proposed in section 3.4.3 of previous chapter to include the concepts introduced in this chapter. This model is used in representing computations performed by *Time and cost aware checkpointing algorithm* presented in the following section. Initially, we remodel the *participant* to accommodate its QoS values. Then we amend the model with sequential component and recovery component.

A **participant** ξ is described as a tuple $(i, Pname, t_{rt}, rl, vl, ct)$, where

$Pname$ is a short description of the participant.

i is the unique identification number given to the participant.

t_{rt} is the promised response time of ξ.

rl is defined as the reliability advertised by ξ.

vl is defined as vulnerability of ξ; $\xi.vl = 1 - (\xi.rl)$.

ct is defined as cost of service charged by ξ.

A **sequential component** s is described as a tuple $(i, \xi, \xi_r, n_{mp}, n_{ip}, f_{ip}, n_{sp}, n_{ml}, t_{at}, t_{rt}, vl, ct, cs, ps, ds)$ where

i is the id given to the component.

ξ is the initiator of the component.

ξ_r is the service provider of the component.

n_{mp} is the number of must save points converted into checkpoints. Initially $s.n_{mp} \geq 0$.

n_{ip} is the number of invocation points converted into checkpoints. Initially $s.n_{ip} \in (0, 1)$.

f_{ip} is a flag which is set to true if the invocation point of this component is converted into checkpoint at design time. If not, it is set to false. This flag is used during checkpoint revision by run time checkpointing to avoid deletion of design time checkpoint at the invocation point.

n_{sp} is the number of service points converted into checkpoints. Initially $s.n_{sp} = 0$.

n_{ml} is the number of message logs. Initially $s.n_{ml} \in (0, 1)$.

t_{at} is the average local activity time of ξ in the component s.

t_{rt} is the response time of the service provider ξ_r.

vl of s to failures is defined as vulnerability of ξ_r; $\xi_r.vl = 1 - (\xi_r.rl)$.

ct of s is defined as cost of service provided by ξ_r.

cs is the checkpointing score of s.

ps is the annotated pattern string that represents s as composition of patterns.

ds is set to 1 if ξ_r is dynamically selected from a static list, is set to 2 if ξ_r is dynamically selected from a dynamic list. It is set to 0 otherwise.

A **recovery component** r is described as a tuple (i, ξ, cs, ps) where

i is the id given to the recovery component.

ξ is the initiator of the recovery component.

cs is the checkpointing score of r.

ps is the annotated pattern string that represents r as composition of components.

Having represented the quantities essential for checkpointing in interaction pattern model, we present the *"Time and Cost Aware Checkpointing Algorithm"* in the following section.

4.5 Time and Cost Aware Checkpointing Algorithm

Time and cost aware checkpointing algorithm [Algorithm 6] incrementally converts C-points into checkpoints, one at a time. It takes as input all the quantities measured and collected at deployment stage, QoS values of services invoked by ξ, and pattern string representing the composition with checkpoints inserted at design stage.

Time and cost aware checkpointing algorithm initially invokes an algorithm called $SequentialComponents(ps)$ (Algorithm 4) which constructs all sequential components of ξ from its pattern string ps that is generated by design time checkpointing algorithm. It takes as input the pattern string ps and returns a string of components, instead of a string of patterns that represent the given composition. This algorithm makes use of the function $getNextToken(ps)$ where tokens in the input pattern string are: patterns, ".", and "!". The algorithm searches for the operator "." in pattern string ps and identifies components in the strings. It copies checkpoint locations indicated by "!" into the new pattern string of components that it constructs.

We would like to a make specific note here. If the participant contains a must save point in one of the patterns p initiated by it, it may be the case that this must save point can be anywhere from the invocation point of p to its service point. We treat a checkpoint at must save point in a pattern p as checkpoint at its invocation point, since this results in maximum recovery time in p if a failure occurs in p. Accordingly flag f_{ip} is set for the component constructed from p.

Time and cost aware checkpointing algorithm makes use of another algorithm $RecoveryComponents(ps)$ (Algorithm 5) that extracts recovery components from a given pattern string ps. (ps contains pattern string expressed in terms of sequential components). This algorithm makes use of the function $getNextToken(ps)$ where tokens in the input pattern string are: sequential components, ".", and "!".

Algorithm 4 *SequentialComponents*

Input Pattern string ps that contains a string of patterns

Output An array S of sequential components; Pattern string ps that contains a string of components

1: $i \leftarrow 1$; new pattern string $nps \leftarrow null$
2: $c \leftarrow getNextToken(ps)$
3: **if** $c =$ "(" **then** $c \leftarrow getNextToken(ps)$
4: **end if**
5: **if** $c =$ "!" **then**
6: $append(nps,$ "!")
7: $c \leftarrow getNextToken(ps)$
8: **end if**
9: **if** $c =$ ")" **then** $c \leftarrow getNextToken(ps)$
10: **end if**
11: $p \leftarrow c$
12: $s_1 \leftarrow Constructsfromp(p)$ //Constructs a component s from pattern p.
13: $insert(S[1], s_1)$ //inserts the component s_1 into the array S
14: $append(nps, S[1])$
15: **while** $(c \leftarrow getNextToken(ps)! = null)$ **do**
16: **if** $c =$ "." **then**
17: $append(nps,$ ".")
18: $c \leftarrow getNextToken(ps)$
19: **if** $c =$ "(" **then** $c \leftarrow getNextToken(ps)$
20: **end if**
21: **if** $c =$ "!" **then**
22: $append(nps,$ "!")
23: $c \leftarrow getNextToken(ps)$
24: **end if**
25: **if** $c =$ ")" **then** $c \leftarrow getNextToken(ps)$
26: **end if**
27: $i \leftarrow i + 1$
28: $p \leftarrow c$
29: $s_i \leftarrow Constructsfromp(p)$ //Constructs a component s from pattern p.
30: $insert(S[i], s_i)$ //inserts the component s_i into the array S
31: $append(nps, S[i])$
32: **end if**
33: **end while**
34: return nps

Algorithm 5 RecoveryComponents

Input Pattern string *ps* that contains a string of components
Output An array R of recovery components

```
1: i ← 0
2: comp ← null
3: while (c ← getNextToken(ps)! = null) do
4:    if c = "!" AND comp! = null then
5:       R[i] ← comp
6:       i ← i + 1
7:       comp ← null // next r starts after checkpoint.
8:    end if
9:    Append(comp, c)
10: end while
```

Algorithm starts by computing recovery overhead of all components (Line numbers 2-4). In each iteration it finds out recovery components and computes their recovery cost and time overheads (Line numbers 11,13). It then computes total execution cost with failure recovery $C_{W_k}(\xi)$ and time $T_{W_k}(\xi)$ (Line numbers 12,15). Then it checks to see if $T_{W_k}(\xi)$ is within T_D and $C_{W_k}(\xi)$ is within C_D. If so, it terminates. Generally $T_{W_k}(\xi)$ is large and is greater than T_D. After conversion of a C-point into checkpoint, $T_{W_k}(\xi)$ decreases. Also, total checkpointing time increases. It continues converting C-points into checkpoints until time and cost constraints are met. Exact checkpointing location is decided using the deployment checkpointing strategy presented in subsection 4.3.4.4. It inserts the character "!" into pattern string given as input at locations where checkpoints are inserted.

4.5.1 Experimental results

To illustrate the effectiveness of our approach, we have developed, tested and monitored the performance of sample web services using Oracle SOA Suite 11g soa (2014). We have used weblogic server which is configured to work as SOA server. Oracle JDeveloper is used for development and deployment of web services. Oracle Enterprise Manager which is also a part of Oracle SOA Suite, is used to collect performance related metrics. The weblogic server is hosted on a machine

Algorithm 6 Time and Cost Aware Checkpointing Algorithm

Inputs ξ, $T_C, T_L, T_R, T_{CR}, T_D, C_D, \lambda$ and QoS values of components of ξ, its pattern string ps given by design time checkpointing.

1: $k \leftarrow 0$; $lg \leftarrow 0$; $ps \leftarrow SequentialComponents(ps)$;
2: **for** each component $s \in ps$ **do**
3: Compute $C_{rec}(s)$, $T_{rec}(s)$ as per equations 4.2 and 4.5
4: **end for**
5: $C_{total} \leftarrow \sum_{i=1}^{n_s} s_i.ct$
6: $R[\ ] = RecoveryComponents(ps)$
7: **while** R is not $null$ **do**
8: $n_k \leftarrow |R|$ // Number of recovery components
9: **for** each recovery component $r \in R$ **do**
Compute $C_{rec_k}(r)$, $T_{rec_k}(r)$ as per equations 4.3 and 4.6
10: **end for**
11: $C_{rec_k}(\xi) \leftarrow \sum_{i=1}^{n_k}(C_{rec_k}(r_i))$
12: $C_{W_k}(\xi) \leftarrow C_{total} + C_{rec_k}(\xi)$
13: $T_{rec_k}(\xi) \leftarrow \sum_{i=1}^{n_k}(T_{rec_k}(r_i))$
14: $\alpha_k(\xi) \leftarrow \alpha_0(\xi) + k * T_C + lg * T_L$
15: $T_{W_k}(\xi) \leftarrow \alpha_k(\xi) + T_{rec_k}(\xi)$
16: **if** $(T_{W_k}(\xi) \leq T_D)$ and $(C_{W_k}(\xi) \leq C_D)$ **then**
break. // deadlines can be met
17: **end if**
//else select a component from recovery component r for checkpointing
18: **for** each recovery component $r \in R$ **do**
19: $C_{rrec_k}(r) \leftarrow \frac{C_{rec_k}(r)}{C_{rec_k}(\xi)}$
20: $T_{rrec_k}(r) \leftarrow \frac{T_{rec_k}(r)}{T_{rec_k}(\xi)}$
21: $r.cs \leftarrow CHS(r)$ //CHS computes checkpointing score of r
22: **end for**
23: $j \leftarrow$ index of recovery component whose cs is returned by $\max_{i=1}^{n}(r_i.cs)$
24: Let the recovery component r_j consist of sequential components from index l to l+h where $1 \leq h \leq n_s - l$
25: **for** each component $s \in r$ **do**
26: $s.cs \leftarrow CHSP(s)$ //CHSP computes checkpointing score of s
27: **end for**
28: $o \leftarrow$ index of component whose cs is returned by $\max_{i=l}^{l+h}(s_i.cs)$
29: **if** $s_o.t_{at} \geq s_o.t_{rt}$ **then**
30: $s_o.n_{sp} \leftarrow 1$ //Place the next checkpoint at service point of s_o
31: $insertCkp(ps, s_o, 2)$.
32: **else**
33: $s_o.n_{ip} \leftarrow 1$ //Place the next checkpoint at invocation point of s_o
34: $s_o.n_{ml} \leftarrow 1$ //log the reply message received
35: $lg \leftarrow lg + 1$
36: $insertCkp(ps, s_o, 1)$.
37: **end if**
38: $k \leftarrow k + 1$
39: $R \leftarrow RecoveryComponents(ps)$
40: **end while**
41: Print no of checkpoints inserted = k

using 4GB RAM, 2.13GHz CPU and J2SDK5. Oracle 11g Database is installed on a machine having 4GB RAM and 3.00GHz CPU. All the PCs run on Windows 7 OS and are connected via high speed LAN through 100Mbps Ethernet cards.

To aid in our experimental results, we have developed the following web services:

1. WS1: web service which invokes *currency converter* web service and *calculator* web service.
2. WS2: web service which invokes *currency converter* web service.
3. WS3: web service which invokes *calculator* web service.
4. WS4: web service takes an Indian state and returns its capital.
5. WS5: web service which returns the Indian state to which a given capital city belongs.
6. WS6: web service which invokes four web services in the order: WS4, WS3,WS2,WS5.

For testing the efficiency of the proposed algorithm, we have used web service WS6, the choreography document of which is depicted in Figure 4.2(a). Numbers in the figure depict fixed execution time, $\alpha_0(s)$, of each of the components. Table 4.1 depicts the values of various required quantities that are recorded and read into the algorithm. Table 4.2 shows checkpointing score calculation for components. Table 4.3 gives a trace of the algorithm for $k = 1$. According to computations component s_3 is selected for converting its invocation point into checkpoint.

Graphs in Figure 4.5 depict the variation of important quantities computed by the algorithm with increase in k value. It can be seen that T_W and C_W cross deadline for $k = 0$ (Fig 4.5(c),(d)). T_W and C_W decrease considerably after converting a C-point into checkpoint. Also, failure free execution time increases due to this checkpoint. But conversion of another C-point into checkpoint is not taken up because deadline is already met with $k = 1$ and also failure free execution time

Table 4.1: Execution Time Parameters for WS6

Quantity	**Value** (Time in millisecs, Cost in units)
T_C	78
T_L	24
T_R	15
T_{CR}	36
$T_{pure}, \alpha_0(\xi)$	4171, 4171+78+24=4273
T_D, C_D	6000,150
W_1, W_2, W_3, λ	0.333,0.333,0.333,.001

Table 4.2: Checkpointing score calculation for constituent web services

s	**WS**	$s.t_{at}$	$\alpha_0(s)$	$T_r(s)$	$s.vl$	$s.ct$	$C_r(s)$	cs	$f(s)$
s_1	WS4	125	642	642	0.43	25	25	(1+0.43+1)* 0.333=0.81	0.642
s_2	WS3	240	1286	240 + 36 + 15 = 291	0.56	30	0	(0.035+0.56+0)* 0.333=0.198	0.291
s_3	WS2	451	1460	1460	0.38	25	25	(0.753+0.38+0.703)* 0.333=0.611	1.460
s_4	WS5	245	783	783	0.24	20	20	(0.217+0.24+0.301)* 0.333=0.252	0.783

increases further with $k = 2$ (Fig 4.5(b)). Hence the algorithm terminates when k=1. In the following subsection we describe a practical approach for placing

Table 4.3: Algorithm 6 Trace for k=1

Quantity	**Value**
Q, Q_0	$\{s_1, s_2, s_3, s_4\}$
$R = \{r_1, r_2\}$	$r_1 = s_1$,$r_2 = s_2.s_3.s_4$
$C_{rec_k}(\xi)$	$\sum(C_{rec_k}(r_1), C_{rec_k}(r_2)) = \sum(25 * 0.642, 0 + 25 * 1.46 + 20 * 0.783) = 68$
$T_{rec_k}(\xi)$	$\sum(T_{rec_k}(r_1), T_{rec_k}(r_2)) = \sum(642*.642, 291*.291+1460*1.46+ 783 * .783) = 3241$
$\alpha_k(\xi), T_{W_k}(\xi)$	4273 +0 = 4273, 4273+3241=7514
$C_{total}(\xi), C_{W_k}(\xi)$	100,168
r_1	$T_{rrec_k} = \frac{412}{3241} = 0.145, vl = 0.43, C_{rrec_k} = \frac{16.1}{68} = .237,\ cs = .270$
r_2	$T_{rrec_k} = \frac{2829}{3241} = .8727, vl = 0.81, C_{rrec_k} = \frac{51.9}{68} = .763,\ cs = .814$
k, j, l, h	$0, 2, 2, 2$
$\max(s_2.cs, s_3.cs, s_4.cs)$	$\max(0.198, 0.611, 0.252) = 0.611$
o	3, Place checkpoint at ip of s_3 and log reply message.

checkpoints in a web service.

4.5.2 Inserting checkpoints in a web service: A practical approach

As described in previous chapter, a web service is modelled as a sequential composition of its components. In the previous chapter, every sequential component s is described as a tuple $(i, \xi, \xi_r, n_{mp}, n_{ip}, n_{sp}, n_{ml},\ t_{at}, t_{rt}, vl, ct, cs, ps, ds)$. All these details for each of the components of the web service ξ are maintained in the database. For each component s of ξ, C-points are set by checkpointing algorithms run at three different stages. According to the set values of C-points, they would be converted into checkpoints at run time. In this section we describe a practical approach using commonly used composition language, BPEL, to insert checkpointing code in a web service implementation.

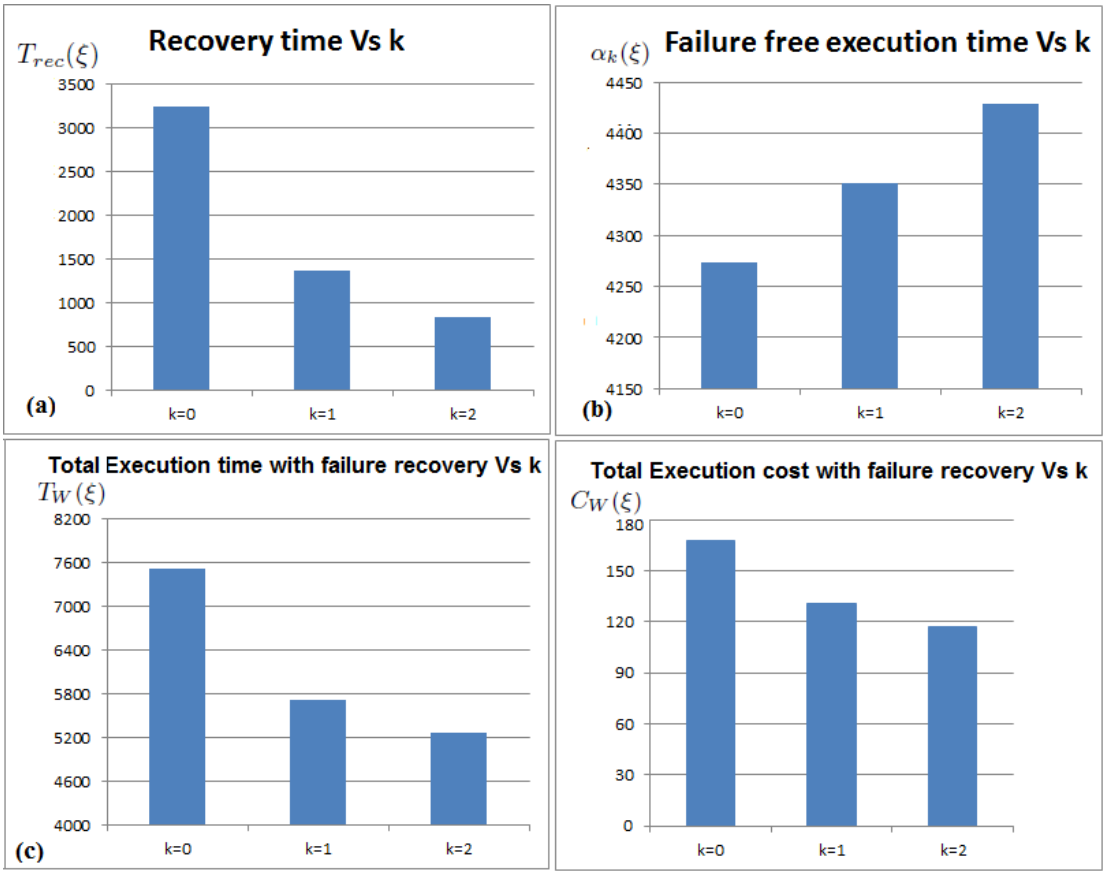


Figure 4.5: Plots of important quantities in three iterations of algorithm

In the BPEL process that represents composite web service ξ, local variables are used to reflect n_{ip}, n_{sp}, n_{vp} values for each of the components. These local variables are initialised to the corresponding C-Point values stored in the database. An invocation point of a component s is checkpointed if its corresponding $s.n_{ip}$ is set to 1. Similarly a service point would be checkpointed if $s.n_{sp}$ is set to 1. Code to test a local variable representing the value of a component's invocation point is placed just after the related invoke activity. If the value of this local variable is 1, the web service state would be saved at the invocation point. Figure 4.6 depicts sample BPEL code placed after an invoke activity. In this code snippet, the local variable that represents the invocation point status for component $s1$ is *nip*. Similarly, code to test the value of a local variable representing the status of a component's service point is placed just after the related receive activity.

```
<if name="s1">
    <documentation>invocation point check</documentation>
    <condition>$nip=1</condition>
        <extensionActivity>
            <bpelx:dehydrate name="DehydrateMV"/>
        </extensionActivity>
</if>
```

Figure 4.6: BPEL Code snippet for converting an invocation point into checkpoint

In the following section we present different scenarios where the proposed checkpointing algorithm is not directly applicable.

4.6 Checkpointing a Participant: Special Scenarios

"Time and cost aware checkpointing algorithm" described in previous section is not directly applicable to two special scenarios: checkpointing a participant which

is atomic and checkpointing a participant which has multiple execution paths. This section advocates the procedure to be adopted in each of these two scenarios.

4.6.1 Checkpointing an atomic web service

The proposed checkpointing strategy makes use of sequential components and recovery components of a composite web service to make checkpointing decisions. Hence an atomic web service too should be modelled as a composition of sequential components to apply the proposed checkpointing algorithm. An atomic web service ξ does not invoke any other web service and hence QoS values of invoked services do not have any role in checkpointing ξ. All quantities not applicable to an atomic web service are considered to be zero. Only one factor that influences checkpointing decision is the failure rate ξ.

Each local activity performed ξ is modelled as a pattern whose type is P_0. If there are nla number of local activities, then ξ is modelled as a sequential composition of nla number of patterns of type P_0. Any non repeatable local activity performed by ξ is also modelled as a local activity of type P_0. Such a pattern would be checkpointed by design time checkpointing algorithm. For example, an atomic web service with four local activities, among which second one is non repeatable is modelled using the pattern string $p_1.p_2!.p_3.p_4$ where type of each pattern is P_0. Checkpoint at the end of pattern p_2 is inserted by design time checkpointing algorithm. Thus there are four sequential components and two recovery components in this example web service. This modelling will enable us to apply the checkpointing algorithm to atomic web services too!

4.6.2 Checkpointing a participant with multiple paths

Multiple paths exist in a composite web service if the composition consists of a choice operator. Checkpointing a participant ξ with multiple paths has to be

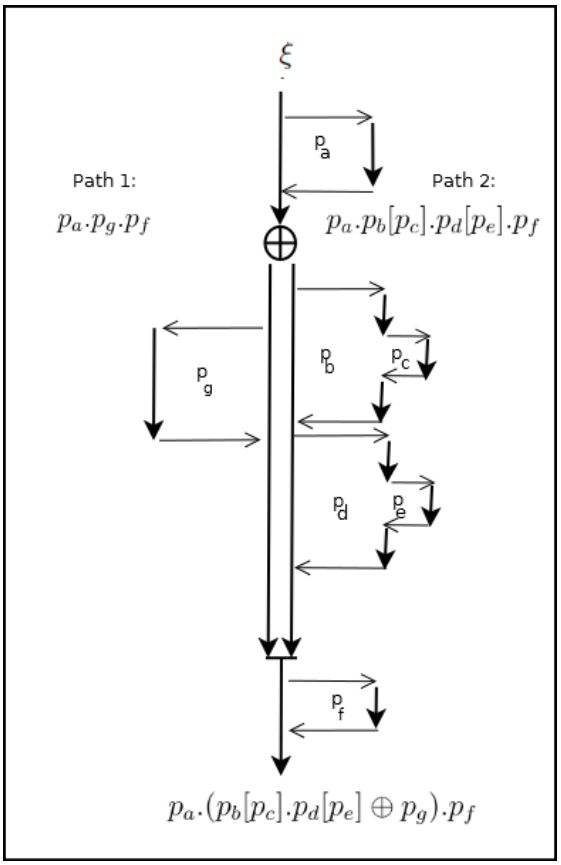


Figure 4.7: A composite web service with multiple paths

dealt with care since the algorithm presented above assumes only one execution path in ξ. In this subsection we present the *"Time and cost aware checkpointing algorithm"* which is adapted to work for a participant with multiple paths. Figure 4.7 portrays a composite web service with two execution paths represented by Path1 = $p_a.p_g.p_f$ and Path2 = $p_a.p_b[p_c].p_d[p_e].p_f$. Many a times these execution paths have some patterns in common, for ex p_a and p_f, which are termed as *common paths*. Remaining patterns are exclusive to the respective paths, for ex p_g and $p_b[p_c].p_d[p_e]$, are termed as *exclusive paths*.

Applying the algorithm to each execution path separately results in different checkpointing locations. For example, let application of the algorithm to Path1 result in $p_a!.p_g.p_f$ where as applying the algorithm to Path2 result in $p_a.p_b[p_c].p_d[p_e]!.p_f$. It can be seen that Path1 gets a checkpoint at the service point of p_a, i.e at the end of common path. But, Path2 does not get a checkpoint

at the service point of p_a. This results in an ambiguity: Should the checkpoint be placed at the service point of p_a or not? To resolve this ambiguity we propose a two pass algorithm that results in unambiguous checkpointing locations in ξ. The following subsection presents this algorithm.

4.6.2.1 Two Pass Checkpointing algorithm for multi-path web services

In this subsection we present a two pass algorithm to be applied for checkpointing multi-path web services. Pass 1 applies *"Time and cost aware checkpointing algorithm"* to each of the paths separately. If a checkpoint is inserted in any of the common paths while checkpointing an execution path, the checkpoint is inserted in all execution paths by Pass 1. This is done because if a checkpoint is inserted in a common path, it has to exist on all execution paths. Pass 2 tries to minimise number of checkpoints, inserted by Pass 1, for each of the execution paths. It accomplishes this task by calling the algorithm $CheckpointMinimisationAlgorithm$. This algorithm tries to check, for every execution path, whether one or more of the checkpoints inserted on its exclusive paths can be avoided. A checkpoint can be avoided, if after its removal, total execution time and cost with failure recovery can satisfy execution time and cost constraints. Semantics of all variables used in this algorithm as same as that described in subsection 4.3.4. Two pass checkpointing algorithm is as follows:

- Pass 1: This pass comprises of the following three main steps:
    1. Enumerate all possible execution paths through the composite service ξ listing out common and exclusive paths. Obtain pattern strings for each of the paths.
    2. Apply *"Time and cost aware checkpointing algorithm"* to each of the execution paths and obtain annotated pattern strings for each of the paths.
    3. Step 2 results in different cases:

Case 1: Step 2 does not result in checkpoints in any common path. The annotated pattern strings obtained in step 2 represent the final checkpointing locations at deployment stage, do not execute Pass 2.

Case 2: Step 2 results in 1 or more checkpoints in any of the common paths. Insert these checkpoints in common paths in the annotated pattern strings of all execution paths. For ex, step 2 for the example depicted in Figure 4.7 results in annotated pattern strings $p_a!.p_g.p_f$ and $p_a.p_b[p_c].p_d[p_e]!.p_f$, i.e, a checkpoint is placed at the service point of common path p_a by the *"Time and cost aware checkpointing algorithm"*. Inserting the checkpoint at the service point of common path p_a in all execution paths results in annotated pattern strings $p_a!.p_g.p_f$ and $p_a!.p_b[p_c].p_d[p_e]!.p_f$. These pattern strings are given as input to Pass 2.

- Pass 2: For each of the annotated pattern string ps obtained in step 3 of Pass 1, apply Algorithm 7 i.e, call $CheckpointMinimisationAlgorithm(ps)$

4.7 Conclusion

In this chapter we have detailed on *Time and cost aware checkpointing algorithm* that has to be adopted at deployment stage for resilient execution of choreographed web services. We define units of execution called as recovery components and compute their checkpointing score by using execution time and other quantities measurable at deployment time and QoS values of constituent services. It aims at introducing minimum number of checkpoints in a web service so that time and cost constraints are met for both failure free executions as well as failed and recovered executions. Our proposed algorithm is novel in its approach as it makes use of QoS attributes of the invoked services also, to decide on checkpoint locations.

Web services operate in highly dynamic behavioural conditions of Internet, hence response times are bound to vary from the advertised values. We intend to

Algorithm 7 CheckpointMinimisationAlgorithm

Inputs Read $T_C, T_L, T_R, T_{CR}, T_D, C_D, \lambda$ and QoS values of components, pattern string ps.

1: SC = set of checkpoints in exclusive paths of ps
2: **for** each checkpoint $c \in SC$ **do**
3: remove checkpoint c from ps
4: **for** each component $s \in ps$ **do**
5: Compute $C_{rec}(s)$, $T_{rec}(s)$ as per equations 4.1 and 4.3
6: **end for**
7: $C_{total} \leftarrow \sum_{i=1}^{n_s} s_i.ct$
8: $R[\,] = RecoveryComponents(ps)$
9: $n_k \leftarrow |R|$ // Number of recovery components
10: **for** each recovery component $r \in R$ **do**
Compute $C_{rec_k}(r)$, $T_{rec_k}(r)$ as per equations 4.2 and 4.4
11: **end for**
12: $C_{rec_k}(\xi) \leftarrow \sum_{i=1}^{n_k}(C_{rec}(r_i))$
13: $C_{W_k}(\xi) \leftarrow C_{total} + C_{rec_k}(\xi)$
14: $T_{rec_k}(\xi) \leftarrow \sum_{i=1}^{n_k}(T_{rec}(r_i))$
15: $\alpha_k(\xi) \leftarrow \alpha_0(\xi) + k * T_C + lg * T_L$
16: $T_{W_k}(\xi) \leftarrow \alpha_k(\xi) + T_{rec_k}(\xi)$
17: **if** $!(T_{W_k}(\xi) \leq T_D$ and $C_{W_k}(\xi) \leq C_D)$ **then**
18: insert checkpoint c in ps in its original location// deadlines cannot be met
19: **end if**
20: **end for**

take up run time revision of checkpoint locations in the next chapter. It is mainly intended to consider actual values of response times and dynamic composition of web services to revise checkpoint locations.

CHAPTER 5

Dynamic Checkpointing

Abstract In this chapter we discuss the need for dynamic checkpointing and, various factors to be considered while revising checkpoint locations dynamically. Subsequently we present a "Run time checkpointing algorithm" for revising checkpoint locations in a composite web service ξ using predicted response time values of the web services invoked by ξ. We propose a framework that implements run time checkpointing algorithm. We also discuss in this chapter a practical approach for inserting checkpoints into a web service at run time. We compare failure free execution time and recovery time of web services in the presence and absence of run time checkpointing, and argue upon the need for dynamic checkpointing. We also discuss in this chapter, about the approach to be followed to revise checkpoint locations in a dynamically composed web service.

5.1 Introduction

Design time checkpointing of choreographed web services, proposed in Chapter 3, introduces checkpoints in a given choreography using design document. The scheme primarily intends to avoid reexecution of non-repeatable actions a service does and checkpoints them at places of non-repeatable actions. However, the checkpointing scheme does not consider the issue of meeting execution time and cost constraints in case of failed and recovered executions. Hence in chapter 4 we have proposed deployment time checkpointing that ensures satisfaction of execution time and cost constraints. The strategy makes use of advertised QoS values of the invoked services to decide on checkpointing locations at deployment time.

But, at run time due to variations in underlying network traffic and web server traffic, there would be differences in advertised and actual QoS values. If this

difference is considerable, checkpoint locations have to be revised at run time. Another scenario which demands dynamic revision of checkpoints is when a web service is composed dynamically. Also, when failure rate of the web service to be checkpointed varies largely from its failure rate computed at deployment, checkpoint locations need to be readjusted. In this chapter we urge on the need for checkpointing web services dynamically and describe our strategy for dynamic revision of checkpoints. We demonstrate empirically the effect of dynamic revision of checkpointing locations.

There are certain dynamic checkpointing strategies proposed in literature. Among them, Chtepen *et al.* (2009) provides dynamic checkpointing for dynamic grids by considering number of faults that can still be tolerated. Nazir *et al.* (2009) provides adaptive checkpointing for grid environments; their work talks more of grid resource scheduling and architecture to be used. Adaptive checkpointing is based on fault index(number of jobs not successfully completed gives fault index) of resources and amount of job left. Even in Zhang and Chakrabarty (2003), checkpointing locations are updated based on the frequency of fault occurrences and the amount of time remaining before the task deadline. Work proposed in Zhou *et al.* (2010) uses predicted server load to adjust checkpoint frequency for high throughput data services. In H.E.Mansour and T.Dillon (2011) authors propose a service-oriented reliability model that dynamically calculates the reliability of a composite web service and places checkpoints in the composite web service using expected recovery time. ***We propose a holistic approach to decide on checkpoint locations which considers 1)dynamic QoS values of all the invoked web services 2)failure rate variations of the web service to be checkpointed and 3) provision of dynamic composition of the web service to be checkpointed. We have not come across any work that proposes dynamic checkpointing for composite web services considering all the above stated issues.***

We propose a run time checkpointing algorithm in this chapter that discusses

how to revise checkpoint locations at run time, which are inserted at deployment stage by time and cost aware checkpointing algorithm, in a composite web service ξ. Run time checkpointing algorithm takes the holistic approach presented in the previous paragraph, to take these decisions. We demonstrate experimentally the usefulness of run time checkpointing algorithm in cases where there are considerable variations in advertised and actual QoS values of web services invoked by ξ.

Among all the QoS values considered for deployment time checkpointing (which are listed in section 4.2.3.1 of previous chapter), it is only response time that might change from one request to another request. Reliability and cost of operations usually do not change during service execution. Hence our dynamic checkpointing strategy is devised to consider only changes in response times to revise locations of checkpoints. This strategy requires actual response time values of web services to be invoked by ξ. As actual values would be available only after invocation, we propose to use predicted response times which are very close to actual response times, for taking checkpointing decisions. We propose two historic data based prediction schemes which are detailed in next chapter.

Response times of web services differ from time to time, because they operate on the Internet which has highly dynamic network traffic conditions. We propose a traffic aware response time prediction strategy that considers equi-length time intervals (of length T time units). Each interval t characterises a network condition. We propose to predict response times once at the beginning of every time interval so that the predicted response time of a web service in the time interval is almost equal to its actual response times in that time interval.

We propose a framework that implements the proposed run time checkpointing algorithm. It mainly consists of prediction module and dynamic checkpointing module. Prediction module uses the prediction strategy which is detailed in next chapter. It predicts at the beginning of each time interval t, response times of all web services that would be invoked by ξ. Dynamic checkpointing module is

also invoked once for each time interval, just after prediction module submits the predicted response times. Using these predicted values it proposes revision in checkpoint locations of ξ which are inserted in all executions of ξ in that time interval t.

The proposed run time checkpointing algorithm is not applicable to ξ if it invokes one or more dynamically discovered and selected services. This is because, for dynamically discovered services QoS values cannot be predicted at run time due to non availability of historic data. Hence we devise a strategy to revise checkpointing locations at run time for such dynamically composed services.

This chapter is organised as follows: Section 5.2 elucidates the need for dynamic checkpointing, and section 5.3 explains the propounded run time checkpointing strategy, section 5.4 presents the proposed algorithm. Section 5.5 presents the framework designed to perform dynamic checkpointing. Section 5.6 discusses the experiments conducted and analyses experimental results. The approach to be followed if discovery of any of the constituent services is done dynamically is given in section 5.7. Having presented frameworks for implementing checkpointing algorithms at three different stages, we propose to merge all the three frameworks and present a unified framework in section 5.8. Section 5.9 presents conclusion of the chapter.

5.2 Need for Dynamic Checkpointing

Deployment time checkpoint described in previous chapter introduces checkpoint locations in a composite web service ξ with the aim of meeting constraints on execution time and cost even in case of a failure and subsequent recovery of the service. It decides on checkpoint locations in ξ by making use of QoS values of ξ and also those of invoked web services. But revision of these checkpoint locations at run time is required for those web services for which the following conditions hold good: 1. Actual response time varies largely from the advertised response

time. 2. Dynamic composition of constituent web services of ξ is facilitated. 3. Failure rate of ξ deviates significantly from the projected failure rate at deployment. In the following subsections we detail upon these three scenarios.

5.2.1 Actual Vs advertised response time values

Deployment checkpointing proposed in the previous chapter, makes use of response time of the invoked services in deciding checkpointing locations for a composite web service ξ. The considered response times are either advertised by providers or measured by the service requesters/ consumers. In any case, response time value considered at deployment time represents an average of a set of values collected under varied conditions. However, at run time, response times vary from the considered average values. At times, this difference may be considerable due to variations in the the underlying network traffic and computing environment.

In case there is significant difference between the considered average response time values and the actual values at run time of a constituent web service of ξ, revising checkpoint locations in ξ would improve its performance. There are two important scenarios to be considered here: 1. Actual Response time of a web service is significantly **greater than** its advertised average response time. This scenario demands addition of checkpoint locations to satisfy constraints on execution time even when an instance of ξ fails and recovers. 2. Actual Response time of a web service is significantly **lesser than** its advertised average response time. In this scenario, removal of some checkpoint locations results in improved execution time of failure free instances of ξ.

Hence to improve performance of a checkpointed composite web service we propose to revise checkpointing locations at run time. In the following subsection we state another reason which urges for dynamic revision of checkpoint locations.

5.2.2 Dynamic composition

Certain composite web services allow for dynamic selection of some of its constituent web services. As discussed in subsection 4.3.3 of the previous chapter, dynamic composition can be done 1)either from a list SD of statically discovered (at deployment time) web services or 2) from a dynamically discovered and selected web services. In both the cases, its not known at deployment time which web service would be selected at run time. Hence in first case, we have proposed to use the worst case advertised QoS values (maximum value in case of response time and cost, and minimum value in case of reliability) of the web services which are in the static list SD. In case of discovery of a service dynamically, we would not even have a list of shortlisted web services at deployment time and hence we propose to use maximum permissible QoS values.

In both the cases, there would be large differences in QoS values considered at deployment and those considered at run time. Hence checkpoint locations have to be revised dynamically to allow for better performance of the composite web service ξ.

5.2.3 Change in failure rate

After deployment, there may be changes in the failure rate of the composite web service ξ. If difference between the new failure rate and the failure rate projected at deployment is significant, there would be considerable differences in recovery overhead of each of the recovery components. Hence to tune the performance of ξ accordingly, checkpoint locations have to be revised.

In this chapter we advocate revision of checkpoint locations dynamically, due to the above specified reasons. In the following section we describe the proposed strategy for revision of checkpoint locations.

5.3 Dynamic Checkpointing Strategy

Dynamic checkpointing of a composite web service ξ is carried out at run time due to the following reasons: 1)difference between actual and advertised response time values of web services invoked by ξ, 2) provisioning of dynamic composition by ξ and, 3) possible changes in failure rate of ξ. In the following subsection the strategy adopted for dynamic checkpointing is presented.

Actual response time of a web service can be obtained only after its invocation. But for the purpose of dynamic checkpointing, actual response time of a web service is needed before invoking it. Hence predicted response time which would be almost equivalent to actual value is used to decide whether revision of checkpoints is to be carried out or not.

Since predicted response times are needed for dynamic checkpointing, we propose response time prediction approaches in the next chapter. For the sake of clarity of discussion of concepts here, we briefly explain our process of prediction in the following paragraph. Detailed approach is presented in chapter 6.

Response times of web services differ from time to time, mostly due to varying network traffic conditions. At peek business hours response time may increase due to heavy traffic, leading to slow message flow. We propose a traffic aware response time prediction strategy that considers equi-length time intervals (of length T time units). Each interval characterises a network condition. Here, for discussion, we have considered 10mins time interval. In practice, the interval can be decided considering a day long traffic patterns. If $T = 10mins$, then each hour would be divided into 6 intervals making up to a total of 144 time intervals per day. At the beginning of each time interval, response time of the web services invoked by ξ are predicted.

Since predicted response time values remain same in an interval, the proposed run time checkpointing algorithm is run at the beginning of every time interval

to adjust checkpoints in ξ. If the web service ξ is dynamically composed from a statically discovered set SD of services, then we predict response time of each of the web services in the set SD. If service provider ξ_r in a component s of ξ is dynamically selected from a static list SD_s where $SD_s \in SD$, then instead of using maximum value of the ***advertised*** response times of the probable web services SD_s, we propose to use maximum value of the ***predicted*** response times for computation of recovery time overhead for the component s.

If ξ is dynamically composed from a set of services discovered at run time, there would be no list of probable web services at our disposal and hence we cannot predict any response time values. Hence, in this case, we propose to use maximum permissible values initially at the beginning of each time interval. When actual execution of ξ takes place and actual response time of a dynamically selected constituent service of ξ becomes available, we propose to adjust checkpoint locations in the remaining components to be executed. This strategy is not included in run time checkpointing algorithm which is run at the beginning of each time interval, but is instrumented into the code of web service ξ. Current failure rate of ξ is also considered for revision of checkpoints.

In the following section, we introduce run time checkpointing algorithm which decides whether checkpoints introduced at deployment in web service ξ have to be revised or not.

5.4 Run time checkpointing algorithm

The proposed algorithm uses the concept of recovery overhead introduced in subsection 4.3.4 of the previous chapter. Using the predicted response time values and failure rate of ξ for the current interval t, it recomputes fixed execution time of the participant ξ, $\alpha_0^{new}(\xi)$ and, recovery time overhead $T_{rec_k}^{new}(r)$ for each recovery component r. Then it computes recovery time overhead $T_{rec_k}^{new}(\xi)$ for ξ and **new total execution time with failure recovery** $T_{W_k}^{new}(\xi)$. Since there would not

be any difference in cost of service attribute value at deployment time and run time, we do not recompute recovery cost overhead at run time again.

If $T^{new}_{W_k}(\xi)$ is greater than T_D (execution time deadline), then it invokes time and cost aware checkpointing algorithm with predicted response time values for adding new checkpoints. This is needed since new checkpoint results in reduced total execution times with failure recovery. If $T^{new}_{W_k}(\xi) < T_{W_k}(\xi)$, there is a possibility of removing one or more checkpoints to reduce failure free execution times. Hence the algorithm tries to select and delete checkpoints iteratively until there is violation of time and cost constraints. In each iteration it selects a checkpoint whose deletion results in least possible total execution time with failure recovery for ξ, $T_{W_{k-1}}(\xi)$. Algorithm 8 depicts the proposed run time checkpointing algorithm. It does not delete checkpoints which are inserted during design stage since they are required to avoid reinvocation of web services that perform non repeatable actions.

Run time checkpointing algorithm takes as input $T_C, T_L, T_R, T_{CR}, T_D, C_D$ since they have to be given as input to time and cost aware checkpointing algorithm when it is invoked. It also takes predicted response time values of all web services invoked by ξ, to revise checkpoint locations. Additionally the algorithm takes as input the annotated pattern string ps for ξ represented in terms of components, and the components, generated by time and cost aware checkpointing algorithm at deployment time. It makes use of the field ds introduced in formal representation of components that is presented in section 4.4 of previous chapter. For a component s, $s.ds$ is set to 1 if the service provider ξ_r of the component s is dynamically selected from a static list SD_s, $s.ds$ is set to 2 if ξ_r is dynamically selected from a dynamic list. It is set to 0 otherwise. SD_s is a subset of web services in SD and contains those web services that are functionally equivalent and one of them would be selected at run time as the service provider ξ_r of the component s.

Algorithm 8 depicts the proposed run time checkpointing algorithm. Line numbers 1,2,3 in the algorithm set response time of each component s to the maximum of predicted response times of web services in set SD_s. Line numbers 4,5,6 recom-

pute recovery time overhead and total execution time with failure recovery for ξ using the new predicted response times of invoked web services. Line numbers 7,8 check to see if deadlines are not met for newly recomputed total execution time with failure recovery. If so, time and cost aware checkpointing algorithm is invoked. If deadlines are met, algorithm performs iterative deletion of checkpoints in line numbers 12 to 25 until deadlines are not met. In each iteration i, it selects a checkpoint c_i and constructs a new patter string ps^i by deleting the checkpoint c_i from pattern string ps (line no 15). Then, the algorithm finds new set of recovery components (line no 15). As one checkpoint is deleted, number of checkpoints now is $k-1$. For each recovery component r, it computes total execution time and cost with failure recovery $T^i_{W_{k-1}}(\xi), C^i_{W_{k-1}}(\xi)$, and checks to see if deadlines are met. If so, adds the checkpoint c_i to set C which is the set of probable checkpoints to be deleted (line no 20). Finally it selects a checkpoint from the set C and deletes it from pattern string ps. The selected checkpoint for deletion would be the one which results in least total execution time with failure recovery.

The following section presents the framework proposed for dynamic checkpointing.

5.5 Framework for revision of checkpoints

In this section, we describe the proposed framework required for introducing checkpoints at run time in a composite web service ξ. Three main components of this framework are Prediction Middleware(PM), Failure rate Measurement Module (FMM) and Dynamic Checkpointing Module(DCM). Figure 5.1 depicts the framework required for revision of checkpoints at runtime.

Prediction Middleware(PM) predicts response time values of web services invoked by ξ. At the beginning of each time interval t, PM would predict response time values for each of the web services possibly invoked by ξ, according to the given choreography. We propose two approaches for prediction that are detailed

Algorithm 8 Run time checkpointing algorithm

Input t, $T_C, T_L, T_R, T_{CR}, T_D, C_D, \lambda_t$, Predicted QoS values of components of ξ, Components and annotated pattern string ps generated by deployment time checkpoint.

Output Annotated pattern string ps with revised checkpoint locations

1: **for** every component s whose $s.ds = 1$ **do**
2: $s.t_{rt} \leftarrow max_{i=1}^{d}(\xi_i.t_{prt})$ where $\xi_i \in SD_s$ and $d = |SD_s|$.
3: **end for**
4: Compute $\alpha_0^{new}(\xi)$ and, $T_{rec_k}^{new}(r)$ for each of the recovery components, using predicted response time values of each of its components.
5: Add them to get $T_{rec_k}^{new}(\xi)$ and compute $T_{W_k}^{new}(\xi)$
6: $input \leftarrow \xi, T_C, T_L, T_R, T_{CR}, T_D, C_D, \lambda_t$, predicted QoS of components of ξ, ps.
7: **if** $T_{W_k}^{new}(\xi) > T_D$ **then** // A new checkpoint to be inserted
Invoke Time and Cost Aware Checkpointing Algorithm($input$)
8: **else if** $T_{W_k}^{new}(\xi) < T_{W_k}(\xi)$ **then** // One or more checkpoints may be deleted.
9: SC = set of checkpoints in ps
10: **while** true **do**
11: $C \leftarrow \emptyset$
12: **for** each deployment checkpoint $c_i \in SC, 1 \leq i \leq k$ **do**
13: // Computes $T_W^{k-1}(\xi)$ assuming c_i is deleted.
14: Let the component that contains this C-Point be s_i
15: $ps^i \leftarrow ps - c_i$ and $R^i[\,] \leftarrow RecoveryComponents(ps^i)$
16: Compute $T_{rec_{k-1}}^i(r), C_{rec_{k-1}}^i(r)$ for each $r \in R^i$, using predicted values.
17: Add them to get $T_{rec_{k-1}}^i(\xi), C_{rec_{k-1}}^i(\xi)$ respectively.
18: compute , $T_{W_{k-1}}^i(\xi), C_{W_{k-1}}^i(\xi)$
19: **if** $T_{W_{k-1}}^i(\xi) < T_D$ and $C_{W_{k-1}}^i(\xi) < C_D$ **then**
20: $C \leftarrow C \cup c_i$
21: **end if**
22: **end for**
23: **if** $C = \emptyset$ **then** break
24: **else**
25: delete checkpoint $c_j \in C$ in ps where $j \leftarrow \delta_{i=1}^{|C|}(T_{W_{k-1}}^i(\xi))$ and δ is an operator that returns the index which minimises the bracketed expression.
26: $ps \leftarrow ps^j$ and $k \leftarrow k - 1$
27: Update the component s_j to reflect the deletion of checkpoint
28: **end if**
29: **end while**
30: **end if**

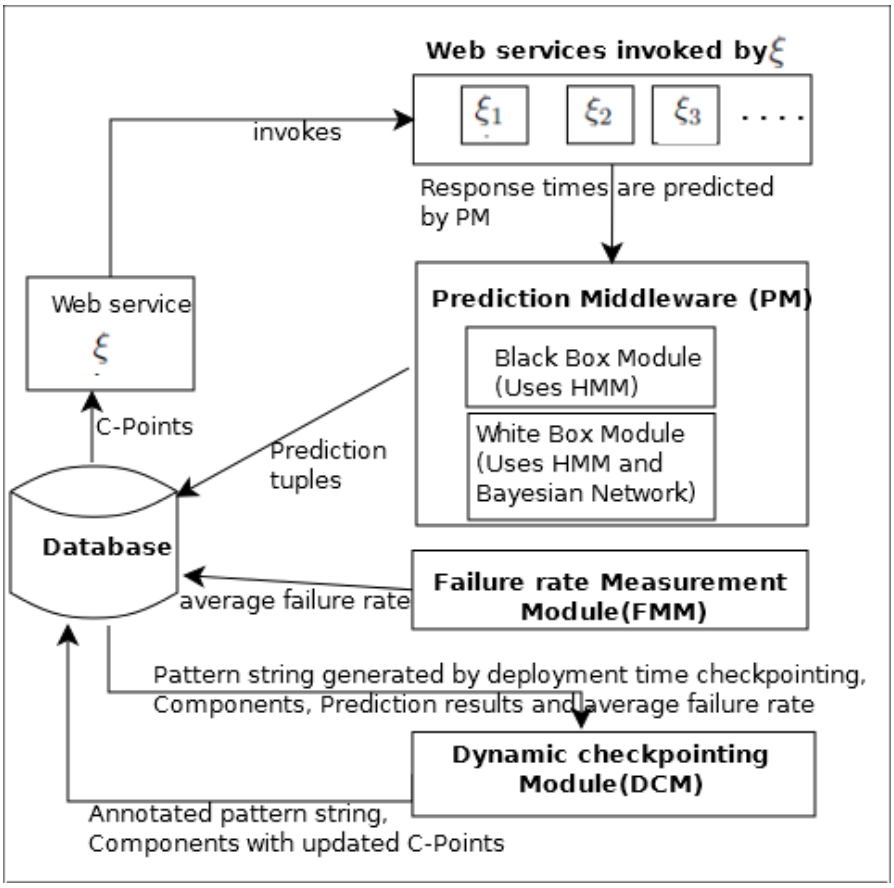


Figure 5.1: Framework for Dynamic checkpointing

in next chapter. Accordingly, PM mainly consists of two modules: Black Box Module and White Box Module.

If a web service ξ_r, invoked by ξ does not reveal its implementation details (internal structure of the web service, web server environment details like average waiting time in a given time interval), then PM uses black box approach to predict response time of ξ_r. If a web service ξ_r, invoked by ξ does reveal its implementation details, then PM uses white box approach to predict response time of ξ_r. The prediction results are made available in the form of tuples called as prediction tuples and are stored in the database at ξ. These predicted values are used for checkpointing decisions made during the time interval t.

Each prediction tuple pt is represented as (t, ξ_r, ta, t_{prt}) where

t is the time interval.

ξ_r is the invoked web service.

ta is the type of approach used. $ta = 1$ for black box approach and $ta = 2$ for white box approach.

t_{prt} is the predicted response time in msecs.

If white box approach is used for predicting response time of a web service ξ_r, for each time interval t, as many predictions would be done as there are equivalence classes for input vector of ξ_r (All input vector values that result in same execution path are grouped under one equivalence class. Each equivalence class results in different response time.) But maximum predicted response time value among them for t would be used to represent the predicted response time of ξ_r for the time interval t.

The latest expected failure rate λ_t of ξ would also be made available by a module called as Failure Management Module(FMM) at the beginning of each time interval t. We do not discuss here, about the approaches to be used for computing λ_t; any of the classic approaches may be used to determine λ_t.

Dynamic Checkpointing Module(DCM) takes up the task of revising checkpoint locations at run time by invoking Run time checkpointing algorithm. DCM invokes the algorithm at the beginning of every time interval t, after prediction tuples are made available by PM. DCM finally updates the components with modified C-points, and stores them in the database. When an instance of ξ starts executing, it saves its state at all places where C-points are set.

In the following subsection we describe the process of modifying checkpoint locations in a web service at run time.

5.5.1 Updating C-Points in a web service at run time: A practical approach

In many web service implementation environments like Oracle SOA Suite, preference variables are available which can be used to set values of variables used by a BPEL process at runtime. These preference variables help us to make changes in variables of running BPEL process without having to change its code and redeploy it. Values of these preference variables can be set from Enterprise Manager.

As described in section 4.5.2 of previous chapter, in the BPEL process that represents composite web service ξ, local variables are used to represent C-Point values for each of the components. Preference variables are set to C-Point values at the beginning of each time interval. Web service ξ is appended with code to read preference variables values into its local variables. Code to test a local variable would be placed just after the related activity to convert the C-Point into a checkpoint.

Thus without having to modify code of a web service at run time and redeploy it, it would be possible to modify the values assigned to its local variables.

5.6 Comparision of execution times with and without dynamic checkpointing

We have used the same experimental set up that is used for deployment time checkpointing, i.e using Oracle SOA Suite 11g (soa (2014)) 4.5.1.

For testing the efficiency of algorithms proposed in this chapter, we have used web service WS6 which invokes four web services in the order: WS4, WS3,WS2,WS5. Table 5.1 depicts the response time values of web services WS4, WS3, WS2 and WS5 recorded for various time intervals. Response time values of WS4, WS3 are closer to average response time values considered for deployment checkpointing.

Response time values for WS2 and WS5 tend to vary in certain time intervals. Column **DTC** in the table represents average response time values considered for deployment time checkpointing. Time intervals t_1, t_2 and t_3 record lesser response times for WS2 than the average response time recorded at deployment time. Hence we see that, for these time intervals run time checkpointing algorithm removes the additional checkpoint inserted at deployment time. Time interval t_4 records almost same response times for WS2 as the average response times recorded at deployment time. Hence run time checkpointing algorithm does not alter the deployment time checkpoints. For time intervals t_5, t_6 and t_7, response times for WS5 increase higher than its average response times in these time intervals which results in *total execution time with failure recovery* crossing time constraint. Run time checkpointing algorithm inserts one more checkpoint in these time intervals. For all these time intervals, failure rate λ of ξ is 0.001, same value as the one taken at deployment time. All the components of ξ are statically composed.

Table 5.1: Time Interval Wise Predicted Response Times (PRT)

Quantity	**t=1**	**t=2**	**t=3**	**t=4**	**DTC**	**t=5**	**t=6**	**t=7**
PRT of WS4	651	637	654	647	642	630	667	656
PRT of WS3	1167	1145	1032	1246	1286	1310	1290	1329
PRT of WS2	623	711	613	1370	1460	1520	1460	1460
PRT of WS5	754	701	767	811	783	900	1226	1302

Figure 5.2 depicts number of checkpoints generated by Deployment time checkpointing (DTC) algorithm and Run time checkpointing algorithm (RTC) in each of the time intervals. It can be seen that while DTC generates only one checkpoint in ξ for these time intervals, RTC generates different number of checkpoints depending on predicted response time values of web services invoked by ξ, in these intervals. In intervals t=1,t=2 and t=3, $T^{new}_{W_0}(\xi) < T_D$ hence RTC deletes one checkpoint inserted by DTC. In interval t=4, $T^{new}_{W_0}(\xi) > T_D$ but $T^{new}_{W_1}(\xi) < T_D$ hence it does not alter any checkpoints inserted by DTC. Similarly for intervals t=5,t=6 and t=7 $T^{new}_{W_1}(\xi) > T_D$ but $T^{new}_{W_2}(\xi) < T_D$, hence RTC inserts one additional checkpoint into ξ in these three time intervals.

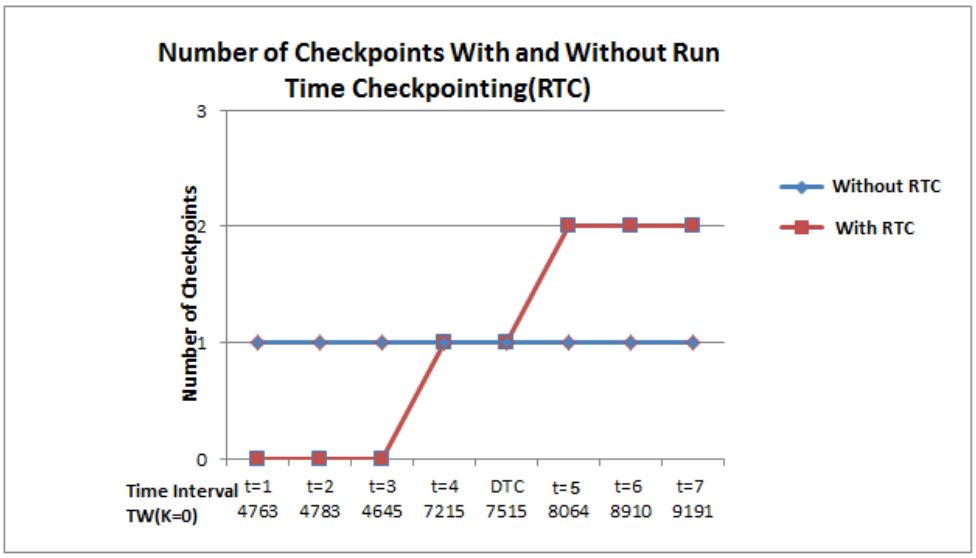


Figure 5.2: Number of checkpoints generated with and without RTC

Figure 5.3 depicts failure free execution time of ξ with and without run time checkpointing algorithm. It can be seen that RTC reduces failure free execution time for time intervals t=1 to t=3 as it deletes one checkpoint inserted by DTC. For t=6 to t=8 failure free execution time increases marginally due to the time required for inserting an additional checkpoint and a message log.

Figure 5.4 depicts *total execution time with failure recovery* of ξ with and without run time checkpointing algorithm. It can be seen that for time intervals t=6 to t=8, RTC reduces the *total execution time with failure recovery* below T_D by introducing an additional checkpoint.

Hence we conclude this section saying that run time checkpointing algorithm revises checkpoint locations in a composite web service ξ based on predicted response times of web services invoked by ξ and its current failure rate. The following section discusses how to place checkpoints in case ξ is dynamically composed.

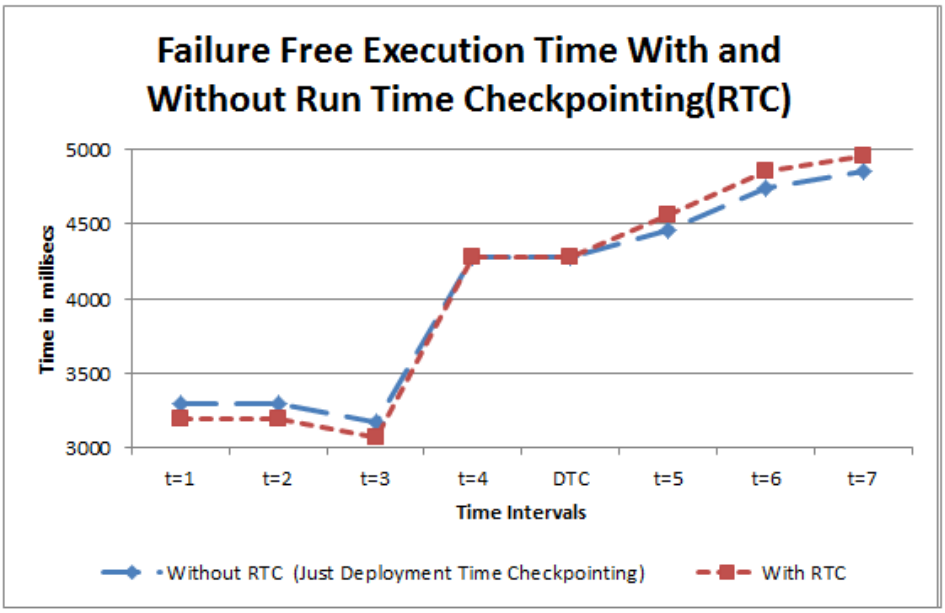


Figure 5.3: Failure free execution time of ξ with and without RTC

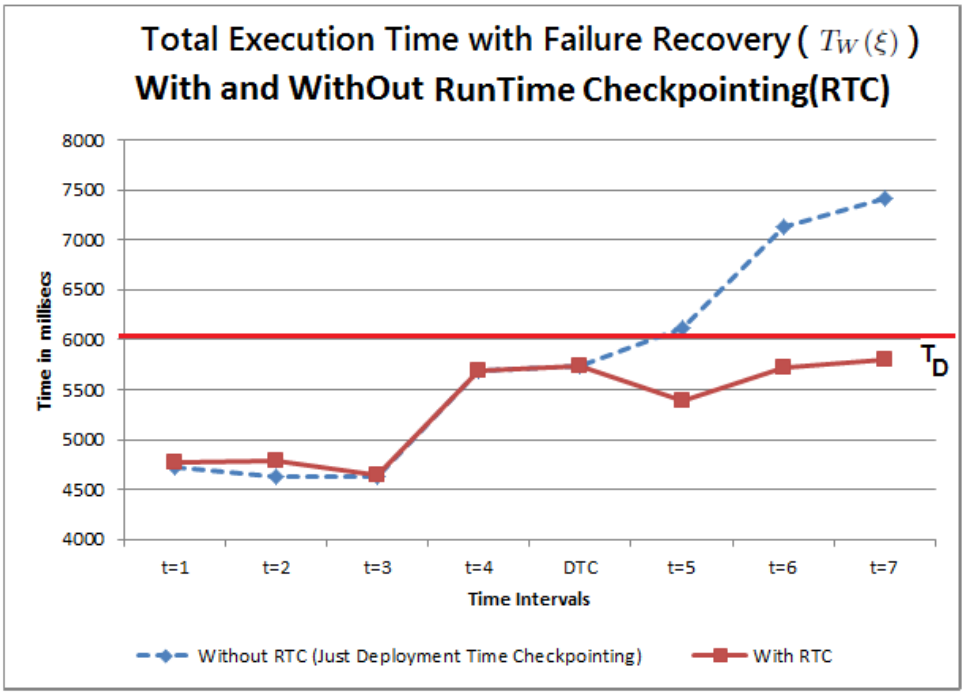


Figure 5.4: Total execution time with failure recovery of ξ with and without RTC

5.7 Handling dynamic composition at run time

If ξ_r in a component s of ξ is dynamically selected from a static list, **maximum predicted response time** from set SD_s is considered as $s.t_{rt}$ for recovery time overhead computation. Even if s is dynamically selected from services discovered at run time (dynamic list), the **maximum permissible response time** is considered by run time checkpointing algorithm. Hence if a web service ξ_i with lesser response time is actually selected at run time, i.e $\xi_i.t_{rt} < s.t_{rt}$, recovery time overhead would not increase at run time. Hence there would be no need to insert a new checkpoint while an instance of ξ is executing. But the other case is possible; if $\xi_i.t_{rt} < s.t_{rt}$ recovery time overhead will decrease and it might be possible to delete one or more checkpoints from the remaining code to be executed.

To determine whether or not to delete a checkpoint in the remaining code, we *assume* that for each checkpoint c in the remaining code, c is removed and the two recovery components preceding and succeeding c are merged. Then we recompute $\alpha_0^{new}(\xi)$ and, $T_{rec_{k-1}}^{new}(r)$ for each of the resulting recovery components, using actual response time of dynamically selected component. We then compute $T_{rec_{k-1}}^{new}(\xi)$ and $T_{W_{k-1}}^{new}(\xi)$. If $T_{W_{k-1}}^{new}(\xi) < T_D$ then we delete the checkpoint c. This is achieved by not invoking any algorithm, but this code is injected into ξ at deployment time if a component of it is dynamically composed. This code is placed after receive activity of the dynamically selected component. The code needs access to response time values of all the components at run time. It is practically possible to obtain values for local variables used by a BPEL process at run time, like using preference variables. This process thus helps us to avoid additional checkpoints at run time.

Having proposed frameworks for implementing checkpointing algorithms at three different stages, we present a unified framework that would be required to implement all three stages of checkpointing for a composite web service ξ.

5.8 Unified Framework for Checkpointing Web services

We have proposed three different stages of checkpointing to achieve our objective: "Automatically checkpoint a given web service composition, considering both static and dynamic aspects of web service compositions, to handle transient faults so that they do not lead to execution time and cost violations". The proposed three stage checkpointing is incorporated into composite web service development cycle, and the complete process is depicted in the Figure 5.5.

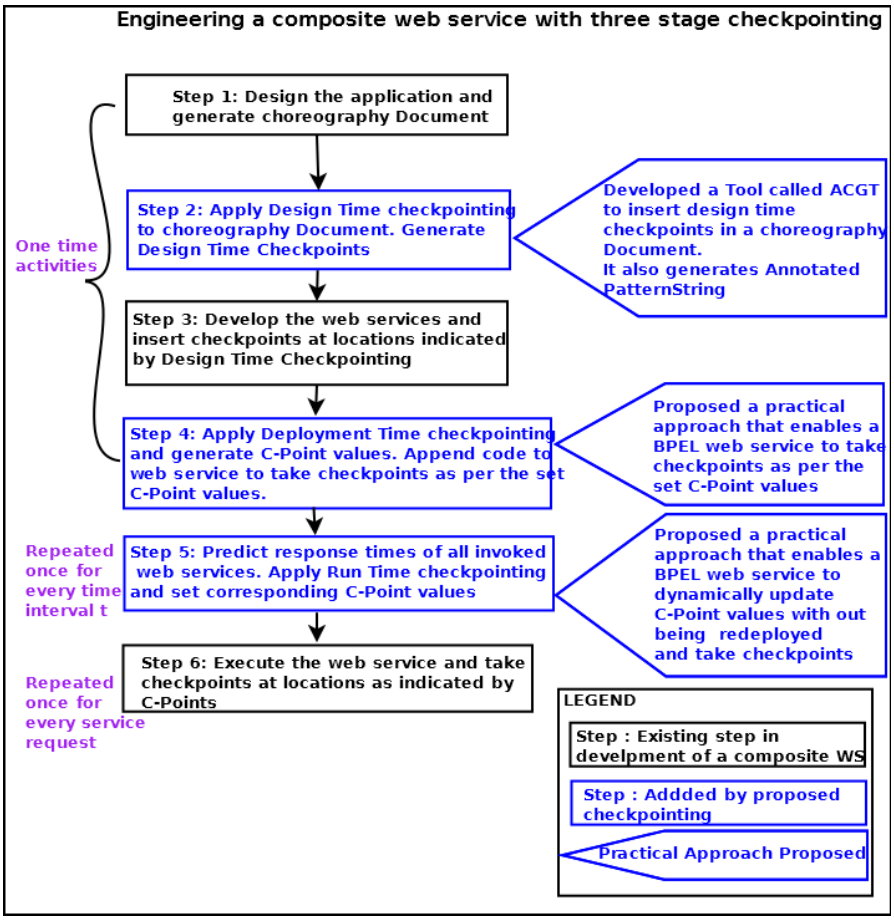


Figure 5.5: Service Engineering approach for checkpointing

Accordingly we have presented frameworks in the respective chapters to implement checkpointing algorithms proposed in those chapters. In this section we would like to unify all these frameworks and present a wholistic framework that is

required to implement all these three stages of checkpointing. Figure 5.6 depicts this unified framework.

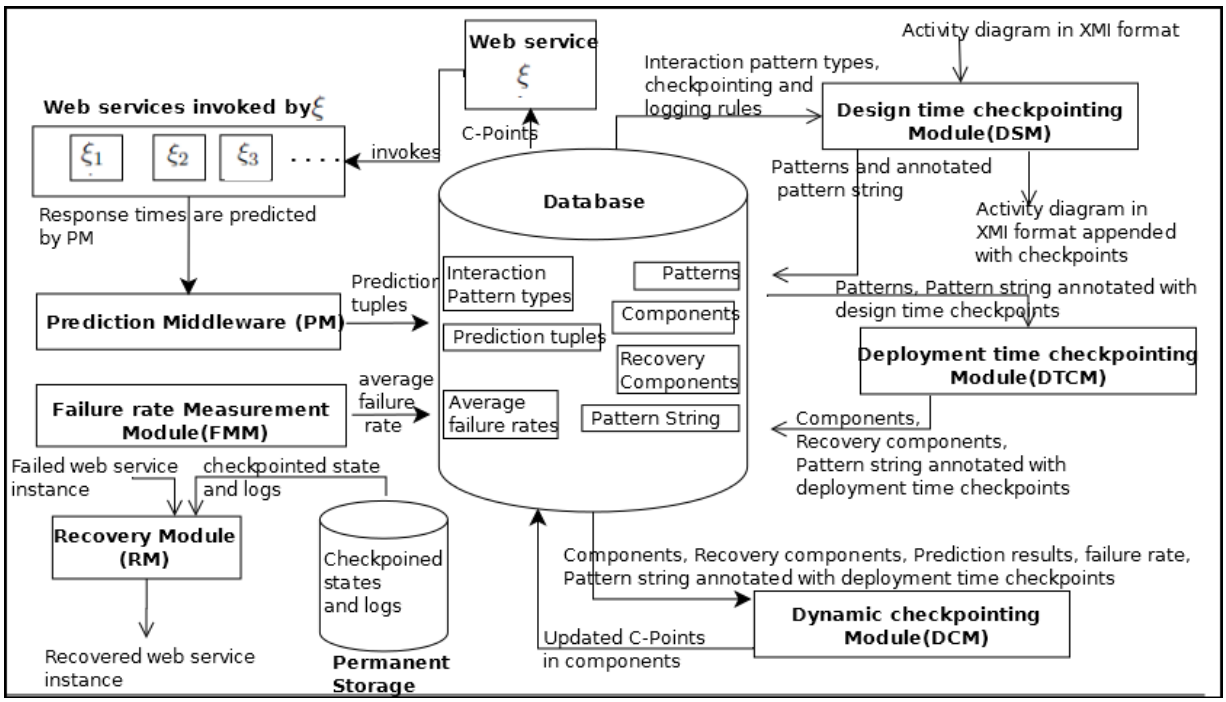


Figure 5.6: Unified framework for checkpointing ξ

The framework consists of three main modules for checkpointing: DeSign time checkpointing Module (DSM), Deployment Time Checkpointing Module (DTCM) and Dynamic Checkpointing Module (DCM). It also consists of modules for prediction (PM), failure rate management (FMM) which are required for dynamic checkpointing. Recovery module (RM) takes care of resuming execution of failed instances of ξ. The framework also contains a database that is used to store patterns, components, recovery components and pattern strings which are generated by checkpointing algorithms. It also used to store advertised and predicted QoS values all web services that are invoked by ξ. Intermediate checkpointed states and logs are stored in permanent storage.

DSM takes as input a choreography given in the form of activity diagram, generates patterns and pattern string that represents the given choreography. It then applies design time checkpointing and logging rules to generated pattern

string and generates annotated pattern string (pattern string with design time checkpoints). It also modifies the patterns to reflect the checkpoint locations. These patterns and annotated pattern string are stored in the database for further use by other checkpointing algorithms. It even inserts diagram elements into the given activity diagram which reflect positions of generated checkpoints. This diagram can be referred by all participants of the participants of choreography to insert checkpoints at the indicated locations in their code.

DTCM and DCM are to be used by each participant to fix deployment time and run time checkpoint locations respectively. Hence participant ξ also has to be equipped with these two modules. DTCM contains a submodule called as *coordinator* module which is responsible for implementing sequence determination policy. Coordinator receives completion messages from all callees of ξ after which it signals the start of deployment time checkpointing at ξ.

DTCM takes as input annotated pattern string that reflects ξ and set of patterns generated by DSM and generates components from them. Using these components and advertised QoS values it runs time and cost aware checkpointing algorithm. As a result, components modified with checkpoint locations, recovery components and pattern string annotated with deployment time checkpoints are generated. These are stored in database for further use by DCM. The checkpoints inserted in the components have to be inserted in to code of ξ.

After deployment of ξ, PM at ξ predicts response time values of all web services invoked by ξ for each time interval t. After prediction tuples are available for t, DCM executes run time checkpointing algorithm. Run time checkpointing algorithm takes the following as input: recovery components, components and pattern string annotated with deployment time checkpoints, current failure rate of ξ and prediction tuples. It applies run time checkpoint algorithm to these inputs and revises checkpoint locations. Accordingly, it modifies components and annotated pattern string to reflect revised checkpoint locations. Checkpoints can be inserted into ξ at run time according to the approach given in subsection 5.5.1

We assume that FMM is already available and is directly used in the framework. Prediction Module is discussed in detail in the next chapter

5.9 Conclusion

In this chapter we have argued upon the need to checkpoint a composite web service ξ dynamically, the reasons being: difference in actual and advertised response times of web services invoked by ξ, failure rate variation of ξ at run time and provision of dynamic composition of ξ. We have proposed to predict response times of constituent web services of ξ, which closely represent actual response times, at the beginning of each time interval t. Using these predicted values we propose a run time checkpointing algorithm which either inserts additional checkpoints, or does not alter checkpoints or removes one or more checkpoints depending on the new *total execution time with failure recovery* of ξ. It is shown experimentally that run time checkpointing helps in meeting time and cost constraints in case *total execution time with failure recovery* of ξ exceeds its deadline at run time. On the contrary, if *total execution time with failure recovery* of ξ reduces significantly, run time checkpointing algorithm removes one or more checkpoints resulting in reduced failure free execution times.

We have also presented a framework that mainly consists of three modules: prediction module, dynamic checkpointing module and failure rate management module. Finally we have presented a unified framework that shows the modules used in all three stages of checkpointing, their inputs and corresponding outputs.

In this chapter we have utilised predicted response time for checkpointing. Thus accurate prediction results are required for efficient implementation of dynamic checkpointing. For this purpose we propose two approaches for web service response time prediction, in our next chapter.

CHAPTER 6

Web Service Response Time Prediction Approaches

Abstract In this chapter we present two approaches for predicting response time of web services: Black box approach[1] and White box approach[2]. Black box approach is presented first which details on what is HMM, how HMM has been used for predictions, and how it can be used for predicting response time of web services. We then detail upon steps to be taken to finally predict response time using HMM. We also validate our prediction approach through experimental results. We also present in this chapter, white box approach for predicting response time. We describe how bayesian network is used for predicting waiting time at web server and HMM is used for predicting network delay component of response time. We compare the prediction accuracy of each of the approaches.

6.1 Introduction

Dynamic checkpointing of web services proposed in the previous chapter revises checkpointing locations of a composite web service ξ based on predicted response times of the web services that ξ invokes. In this chapter, we propose two approaches that use Hidden Markov Model(HMM) and Bayesian network for predicting response time of web services.

[1]**Vani Vathsala, A. and Hrushikesha Mohanty (2012).** Using hmm for predicting response time of web services. Proc of CUBE International Conference, pp 520 to 525. Pune, 2012. Proceedings in ACM digital library.

[2]**Vani Vathsala, A. and Hrushikesha Mohanty (2014c).** Web service response time prediction using hmm and bayesian network. Intelligent Computing, Communication and Devices, 327 to 335. Bhubaneshwar, 2014. Proceedings in Springer book Advances in Intelligent Systems and Computing, Vol 1

Response time (RT) of a web service is the difference between the service delivery time and service initiation time. In addition to the context of checkpointing, prediction of response time of a web service has several other applications. As the number of web services that offer functionally similar services is growing enormously, service consumers have the luxury of selecting services that offer high quality in less time. We present two approaches for RT prediction which would help service users in selecting web services that respond quickly.

We propose to implement a Prediction Middleware(PM) that predicts response time of web services. The middleware mainly contains two modules: The first module uses black box approach and the second module uses white box approach for response time prediction. In block box approach service implementation (web service internal structure) and web server environment (no of pending requests at the web server, average waiting time etc) details are unknown to the prediction middleware. For this approach, we make use of the popular Hidden Markov Model Resch *et al.* (2004) for prediction. Prediction accuracy can be improved if implementation and environment details are also made available to the prediction middleware. In case the details are known, our proposed white box approach makes use of HMM and Bayesian networks Heckerman (1995) for prediction. We establish the accuracy of our predictions experimentally.

We are using HMMs for prediction purpose because of their proven suitability for modelling dynamic systems. The research works presented in Hassan and Nath (2005) to Netzer *et al.* (2008) propose methods for using HMM as a predictor. Hassan and Nath (2005) uses HMM to predict share prices in stock markets. Gonzalez *et al.* (2005) uses IOHMM to analyse and forecast electricity spot prices. Netzer *et al.* (2008) models the dynamics of customer relationships and the subsequent buying behavior of the customers using non-homogeneous HMM.

Artificial Neural Networks have also proved themselves to be reliable predictors R.D.Albu *et al.* (2013), R.D.Albu (2014), R.D.Albu *et al.* (2013) and Gao and Wu (2005). Zhengdong Gao uses Back Propagation Neural Networks to predict

the runtime of a given web service (Gao and Wu (2005)). He uses Availability, Network Bandwidth, Response Time, Reliability of the given web service as inputs to the Neural Network which produces predicted execution duration as output. But HMMs score an edge over ANNs when it comes to prediction in dynamic environments.

This chapter is organised as follows: Section 6.2 details on our black box approach using HMM for predicting response time of a web service and section 6.3 presents our second approach using Bayesian network and HMM for response time prediction, along with experimental results. Section 6.4 presents the conclusion of the chapter.

In the following section we present our black box approach for response time prediction of web services.

6.2 Using HMM for response time prediction viewing the web service as block box

The approach presented in this section predicts response time of a web service. The prediction middleware views that service as a black box, i.e details of the web service implementation and the environment hosting the service are not known. In order to predict response time, well known Hidden Markov Model is used.

HMMs have been extensively used for speech recognition, pattern recognition, DNA sequencing, finger printing matching, stock market prediction, electrical signal prediction and image processing, etc. HMMs have also established themselves to be reliable predictors of the behaviour of dynamic systems. Inspired by their successful application to several areas as predictors, we use HMM for predicting the response time of a web service and found that the prediction compliments atleast by 70 % to the observed data.

In the following subsection we present a brief introduction to the process of prediction using HMM

6.2.1 Prediction using Hidden Markov Model

Hidden Markov Model was first introduced in late 1960s by L.E Baum and his team. When a system is characterised by a set of mutually exclusive finite number of states $S = \{S_1, S_2, ..., S_N\}$ and the state in which the system can be in a time slot T is determined only by the state of the system in time slot $T-1$ and not by states of the system in time intervals $1, ...T-2$, then the system is said to be a Markov Process.

In a regular markov model the state transitions are directly visible to an observer. In a Hidden Markov Model (Rabiner (1989)) the system being modelled is assumed to be a Stochastic Markov Process with unobservable or hidden states, but the output of the system in each of the time intervals, which is dependent on corresponding states of the system, is visible to the observer. We call this observable output of the system as Emissions or Observations O. State Transition Matrix A, depicts the transition probabilities among states of the system. Each state has a probability distribution over the observation symbols. Emission Probability Matrix B, governs probability distribution of the states over these emissions. Given below is a simple example (Resch *et al.* (2004)) demonstrating modelling of a system using HMM.

Two friends X and Y live in two different cities but talk to each other over phone daily. X does only one of the three things in a day : going for a walk, shopping and cleaning his house. Activity that he does depends on weather of the day. Y has no definite information of the weather of X's city, but knows general trend that it can be either Rainy or Sunny. Y tries to guess the weather of the city on a particular day from the activities reported by X over phone. Activities of X become observables for Y, whereas weather of X's city is hidden from Y. This

Table 6.1: Parameters of HMM for the example given

HMM	**Weather Prediction Example**
States $S = \{S_1, S_2, .., S_N\}$	$\{S_1, S_2\} = \{Rainy, Sunny\}$
Observations $O = \{O_1, O_2, .., O_M\}$	$\{O_1, O_2, O_3\} = \{walk, shop, clean\}$
Initial Probability Π	$\{\pi_{Rainy}, \pi_{Sunny}\} = \{0.6, 0.4\}$
State Transition Matrix A	$A = \begin{array}{c} \\ Rainy \\ Sunny \end{array} \begin{array}{c} \begin{array}{cc} Rainy & Sunny \end{array} \\ \begin{pmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{pmatrix} \end{array}$
Emission Probability Matrix B	$B = \begin{array}{c} \\ Rainy \\ Sunny \end{array} \begin{array}{c} \begin{array}{ccc} walk & shop & clean \end{array} \\ \begin{pmatrix} 0.1 & 0.4 & 0.5 \\ 0.6 & 0.3 & 0.1 \end{pmatrix} \end{array}$

entire system can be modelled using HMM as shown in Table 6.1.

Initial probability Π represents Y's belief about what would be weather of X's city on the day when he first calls X. Y believes that it may be Sunny with probability 0.4 or it may be Rainy with probability 0.6. State transition matrix represents how the weather might change in subsequent days. Emission probability matrix represents the likeliness of X's activities on each of the days.

The problem of weather prediction is trying to guess what would be the weather of X's city tomorrow based on the observations of the activities of X in the past few days. In the example given, an observation sequence $O^{'}$= $\{clean, clean, shop, walk\}$ provides info about the sequence of activities done by X on previous 4 consecutive days. Using this observation sequence and parameters $\Theta = \{\Pi, A, B\}$ of HMM, Y has to predict weather of X's city on 5^{th} day.

The HMM process of prediction is as follows. Given an observation sequence $O^{'} = \{o_1, o_2, ..., o_{T-1}\}$, the problem of predicting the observation in time slot T is concerned with finding the state sequence $S^{'} = \{s_1, s_2, ..., s_T\}$ which has the maximum likelihood of emitting the given observation sequence $O^{'}$. Here, s_n is a random variable that represents the state the system could have taken in time

slot n, $1 \leq n \leq T$. The most likely emission given by the state s_T is the predicted observation for the time slot T. Consequently Viterbi algorithm (Forney (1973)), the most frequently used algorithm for estimations in the literature, can be used for prediction. Due to space restrictions, details of the algorithms are not presented here. The interested reader is referred to Rabiner (1989).

Coming back to weather prediction example, Viterbi algorithm predicts state sequence to be $S^{'} = \{s_1, s_2, ..., s_5\} = \{Rainy, Rainy, Rainy, Sunny, Sunny\}$ for the given observation sequence $O^{'} = \{clean, clean, shop, walk\}$, i.e. it predicts $s_5 = Sunny$ to be the weather of X's city on 5^{th} day. The most likely emission $walk$, given by the state $s_5 = Sunny$ is the predicted activity of X (observation)for the next day i.e day 5 (time slot $T = 5$). When a HMM process generates a sequence of observations of length T, the result of the HMM process will be both the observations sequence $\{o_1, o_2, ..., o_T\}$ and the states sequence $\{s_1, s_2, ..., s_T\}$. However, whereas the observations sequence is visible, the states sequence remains hidden, and no information about it is available to an observer.

Mathematically, an HMM is defined by the following elements:

1. the number of states in the model : N.

2. the number of observation symbols in the model : M.

3. an initial state probability distribution Π where each element $\Pi_i \in [0, 1]$ is defined as:
   $\Pi_i = (P(s_1 = S_i))$ where $1 \leq i \leq N$

4. the state transition probability matrix A of order N*N where each element $A_{ji} \in [0, 1]$ is defined as:
   $A_{ji} = P(s_{t+1} = S_i | s_t = S_j)$ where $1 \leq i \leq N$ and $1 \leq j \leq N$

5. emission probability matrix B of order N* M where each element $B_{jm} \in [0, 1]$ is defined as:
   $B_j(O_m) = P(o_t = O_m | s_t = S_j)$ where $1 \leq j \leq N$ and $1 \leq m \leq M$

In order to use HMM for prediction, above parameters $\Theta = \{\Pi, A, B\}$ of HMM have to be fixed first. In weather prediction example we are already provided with these values(Table 6.1). On the contrary, while modelling a dynamic system using HMM we will be provided only with a set of observation sequences, using which we are required to determine the parameters of HMM. This process of fixing the parameters of HMM is called training the HMM.

Given a set of observation sequences, called the training data, where each observation sequence is of the form $O' = \{o_1, o_2, ..., o_{T-1}\}$, the problem of adjusting an HMM is concerned with finding the set of parameters $\Theta = \{\Pi, A, B\}$ that best fits the training data. Consequently, the Expectation Maximization (EM) algorithm (Dempster *et al.* (1977), McLachlan and Krishnan (2008)) and its simplified version, due to Baum and Welch (Rabiner (1989)), are widely used in the literature to estimate the parameters of HMM.

The following subsection details on the process of using HMM for web service response time prediction.

6.2.2 Response time prediction using HMM

The two factors that contribute to response time of a web service are 1) service processing time by the corresponding web server and 2) the delay introduced by the underlying network.

Service processing time may vary from one service request to another as processing at a given time depends on current load and sharing of common resources. The network through which a service request is routed also contributes to delay in response time of the service. Delay introduced by the underlying network may be highly attributed towards the traffic load the network has. For the current purpose, both web server and underlying network are considered together as a single system (Web service system).

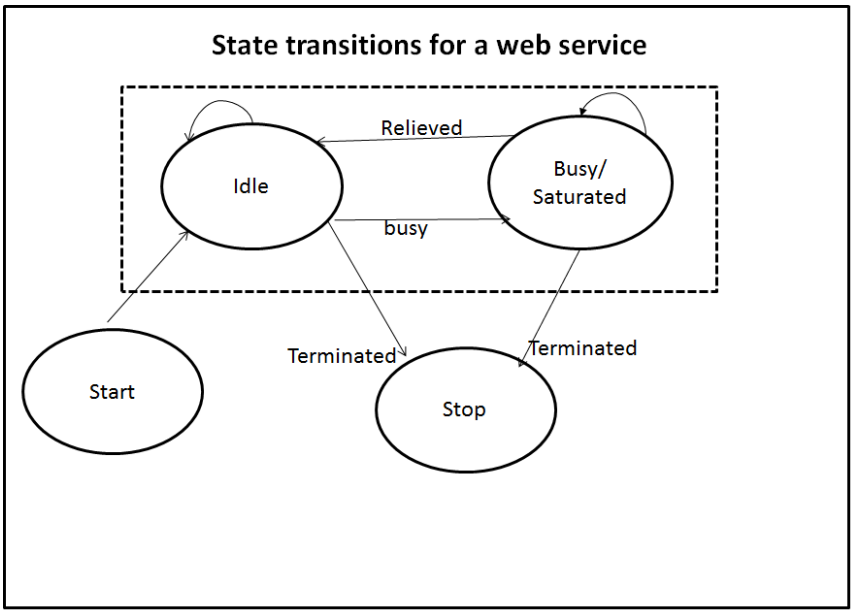


Figure 6.1: Web service state transitions

Behaviour of a web service is captured in the state transition diagram depicted in Figure 6.1. From the time of its publication at a UDDI, a web service should be available for providing service to the customers. Life cycle of a Web service begins in **start** state. A web service moves to **stop** state when it seizes to provide the service. A web service accepts service requests when it is in either idle state or busy state. The two states (start and stop) are only nominal and are included to complete the life cycle of a web service. A web service will be in busy state if any of its components (web server or network) is busy, and the web service will be in idle state if both of its components are in idle state. Thus any web service can be deemed to have atleast two states: **busy** and **idle**. When a web service is in busy state, its subsequent callers witness higher response times. In an overloaded system, the turn of execution of a service is delayed resulting in longer response time. More the system is busy, longer is the response time. When the web service gets back to its idle state response times reduce. This shows the relation between observation symbols i.e response time and hidden states viz. busy and idle. Relation between the two is reflected in Matrix B as shown in Table 6.2.

In a HMM there are unobservable hidden states that produce observable emissions. Our decision to use HMMs for web service response time prediction has been motivated by this fact. The states of a web service that are not visible i.e which are hidden from external world, are considered as hidden states S of HMM. Since the response times of a web service can be observed and measured by a service user, they are considered as the emissions O of HMM. Since response times can take arbitrary non-negative real values, we quantize them into a finite set of observation symbols or emissions. For the purpose of prediction, response times of a web service in continuous intervals of time, say from 0 to $T-1$, are collected. These values are then used to predict response time of the web service in time interval T.

HMM model for a web service is depicted in Table 6.2. Our reasoning for selecting two states for a web service is essayed in next paragraph. As detailed in previous section, values for the parameters $\Theta = \{\Pi, A, B\}$ are determined using EM algorithm. Π_i indicates the probability of moving to a web service state S_i at time $T = 1$. A_{ji} is the probability of moving to the web service state S_i at time T given the web service state S_j at time $T-1$. $B_j(O_m)$ is the probability of emitting response time O_m from the state S_j. From the dataset we have used for validating our approach, we depict in Table 6.2 the values computed by EM algorithm for the parameters $\Theta = \{\Pi, A, B\}$ when web service 5 is called by user with id=1.

Response times observed by a user of a web service differ significantly from that observed by another user of the web service due to unpredictable communication links and underlying heterogeneous network environments. When a web service ξ would like to predict the response time of its service provider ξ_r, it has to construct a HMM that captures the states of ξ_r and the intervening network. In the following subsection we present the framework necessary for predicting response time when a web service ξ_r is called by its service user ξ.

Table 6.2: HMM Parameters for a Web service

HMM	**Web Service States**
States $S = \{S_1, S_2, ..., S_N\}$	States of the web service $S = \{S_1, S_2\} = \{idle, busy\}$
Observations O	Observations are response Times of web service which are positive real values. Hence they are quantized into M observation symbols
Initial Probability Π	$\{\pi_{Idle}, \pi_{Busy}\} = \{0.75, 0.25\}$
State Transition Matrix A	$A = \begin{array}{c} \\ Idle \\ Busy \end{array} \begin{array}{c} \begin{array}{cc} Idle & Busy \end{array} \\ \begin{pmatrix} 0.7333 & 0.2667 \\ 0.7214 & 0.2786 \end{pmatrix} \end{array}$
Emission Probability Matrix B	$B = \begin{array}{c} \\ Idle \\ Busy \end{array} \begin{array}{c} \begin{array}{cc} O_1 & O_2 \end{array} \\ \begin{pmatrix} 0.9063 & 0.0938 \\ 0.8999 & 0.1111 \end{pmatrix} \end{array}$

6.2.3 Framework for prediction

Prediction middleware at ξ should be configured to have HMM Module that performs the task of response time prediction. It mainly consists of three submodules: RT prediction module R, EM and Viterbi algorithms (Refer to Figure 6.2).

Predicting response time of web service ξ_r by ξ is a three step procedure as depicted in Figure 6.3:

1. Training the HMM: This step includes estimating the parameters $\Theta = \{\Pi, A, B\}$ of the HMM. To estimate these parameters, training data should be provided to RT prediction module R. Training data should consist of response times of the web service, ξ_r, when called by the user ξ at regular intervals of time. EM and Viterbi algorithms used in our experiments require sequence of discrete and finite observation symbols as their input. Response times can take arbitrary non-negative real values, hence R determines the set of finite and discrete observation symbols from training data containing real response times. We term

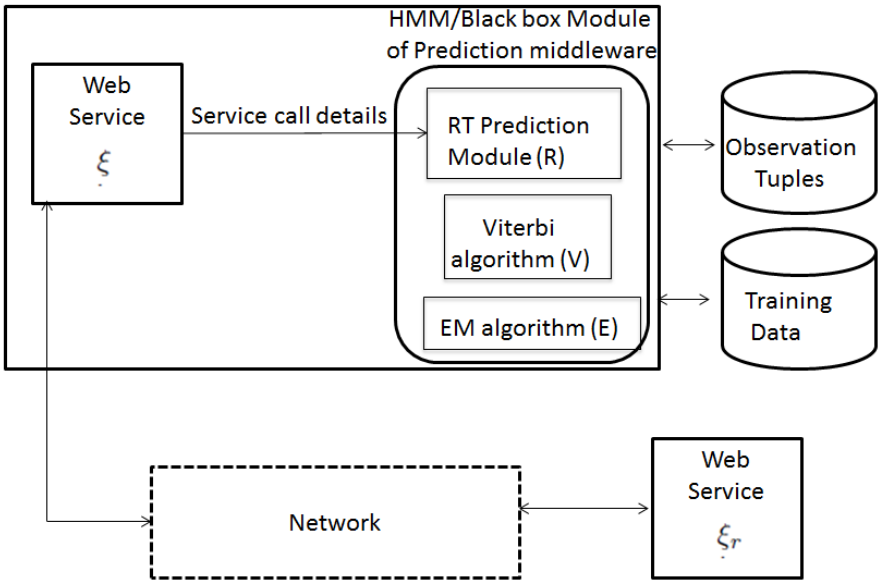


Figure 6.2: Framework for response time prediction using black box approach

this process of determining finite number of observation symbols from real valued response times as quantization. Then it converts sequence of observations containing real values into corresponding sequence of discrete observation symbols. These sequences are then fed as input by R to EM algorithm (E in Figure 6.2) that fixes the parameters $\Theta = \{\Pi, A, B\}$ of the HMM. This is a one time processing step and once the model parameters are estimated, they can be used for subsequent predictions. EM algorithm determines the parameters of HMM, but the values it fixes are directly dependent on training data. Hence training data should truly capture the dynamism of the web service system for appropriate predictions.

2. Collection of response times: In order to aid in response time prediction, service call details are maintained by R in its database, these details are collected each time ξ calls the web service ξ_r. The service call details contain the following fields :

1. i, the unique identification number of the web service ξ_r.
2. Local timestamp of the service requester ξ at the time call is made. (Local timestamp should be according to the ticks of a global clock. The issue of

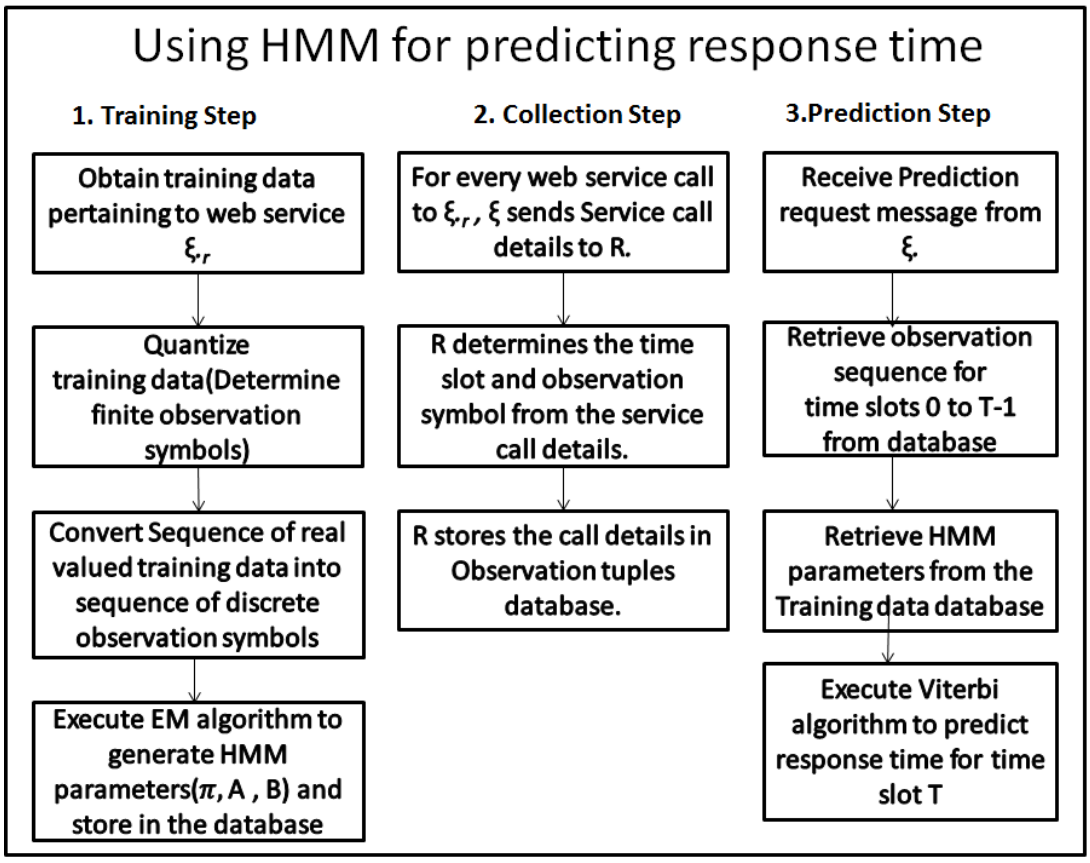


Figure 6.3: Using HMM for response time prediction

clock management is out of scope of this work).

3. Response time i.e, the time-lapse between the time at which the service request is placed and the time at which the reply is received.

R would maintain time intervals of length t time units each. For example, if $t = 10mins$, then each hour would be divided into 6 intervals making up to a total of 144 time intervals per day. For each call made to ξ_r , it determines the time slot to which the service call belongs to from the corresponding service call details. For example, if Local timestamp value $= < 00 : 07 : 56 >$, it is put in the time slot T1. (because $< 00 : 07 : 56 > \in$ first time slot in the first hour of a day)

R uses Response time field of service call details to determine the corresponding observation symbol and stores the message details in its database. Thus the

quantised response times of the web service ξ_r when invoked by user ξ, time slot wise, would be maintained by the RT prediction module R in its database. These details are termed as *observation tuples*.

3. Prediction: In order to predict response time in time slot T, the user ξ sends a request message to R. The request message should contain i, service Provider Id. Response times of ξ_r when called by ξ in preceding time intervals i.e from 0 to T-1 are are available in its database. Depending on the time slot T to which the request message belongs, R retrieves the observation sequence $O = \{o_1, o_2, ..., o_{T-1}\}$ of quantized response times for the web service ξ_r from the database. Corresponding HMM parameters are also retrieved by R from the database. Response time o_T of ξ_r at time T is predicted by R using Viterbi Algorithm (V in Figure 6.2). The predicted response time would then be sent back to the web service ξ.

6.2.4 Experimentation and results

In order to conduct experiments for training HMM and subsequent prediction of web service response time, we have used WSDream dataset (Zhang *et al.* (2011)). The dataset used contains response times of around 4532 web services recorded in 64 time intervals when invoked by 142 users. We have conducted experiments by preparing subsets of data, where each subset contains data pertaining to a web service when it is called by the same user in 64 time intervals. Length of each time interval is 15mins. We have compared the accuracy of our prediction against the recorded response time for 64^{th} time interval available in the subset.

While using HMM, number of hidden states that model the selected system have to be decided first. The problem of selecting optimal number of hidden states is not yet given a proper solution (Cappé and Moulines (2005)). The prediction accuracy along with computational complexity is expected to increase with the order of HMM, however HMM with more number of states does not guarantee

Table 6.3: Predictions for User Id 1

WS Id	**Matrix A**	**Matrix B**	**Predicted RT(secs)**	**Observed RT(secs)**
2	0.8667, 0.1333; 0.8600, 0.1400	0.2500, 0.2813, 0.1563, 0.1250, 0.0625, 0.1250; 0.2500, 0.2812, 0.1429, 0.1384, 0.0625, 0.1250	0.250000	0.2580000
5	0.7333, 0.2667; 0.7214, 0.2786	0.9063, 0.0938; 0.8999, 0.1111	0.430000	0.438000
11	0.8000, 0.2000; 0.7999, 0.2111	0.8594, 0.1094, 0.0313; 0.8300, 0.1288, 0.0313	0.41000	0.695000

optimal results (Cappé and Moulines (2005)). Hence we have used two state HMM for prediction. The proposed solution can be tailored to work with three state HMM for web services.

Table 6.3 shows the values of State Transition Matrix(A), Emission Probability Matrix(B), predicted response time and observed response time for 64^{th} time interval for User Id=1. Each row corresponds to a different web service that the user has called and thus we can see different values for HMM parameters in each of the rows. Matrix B contains different number of columns in each of the 3 rows; number of columns are determined by the number of symbols generated as a result of quantization of the recorded response time values.

We have predicted response times for 100 web services, user wise using MATLAB. We depict in Charts shown in Figure 6.4 the comparisions done for web services with ids 2, 5 and 11. These charts present the comparisions between our predicted value and observed value for response times in 64^{th} time interval when these web services are invoked by different service users. Out of 142 service users we have considered only those service users whose calling records are available for all 64 time intervals.

From the charts it is evident that our predictions match with observations in most of the cases. In certain cases where the difference is more than 50%, the

reason is as follows: if the training data captures all possible patterns in changes of the response time, then our prediction matches well with the observation. If the response time observed in 64^{th} time interval does not fall with in the range of training data, then our prediction does not match with observed value.

In the following section we present white box approach for response time prediction of a web service using Bayesian networks and HMM.

6.3 Using HMM and Bayesian networks for response time prediction viewing the web service as white box

The approach presented earlier has to be used for response time prediction in case implementation and web service environment details are inaccessible, but if these details can be provided, the approach provided in this section has to be used. This approach performs response time prediction with greater accuracy since finer details are at its disposal.

The two factors that contribute to response time of a web service are 1) service processing time/execution time taken by the corresponding web server and 2) the delay introduced by the underlying network.

Service processing time may vary from one service request to another as processing at a given time depends on current load, input parameter values and sharing of common resources. The network through which a service request is routed to the service provider also contributes to delay in response time of the service. Delay introduced by the underlying network may be highly attributed towards the traffic flowing through the network. Thus service processing time depends upon web server environment and processing logic where as network delay depends upon underlying network environment. Response time for a service

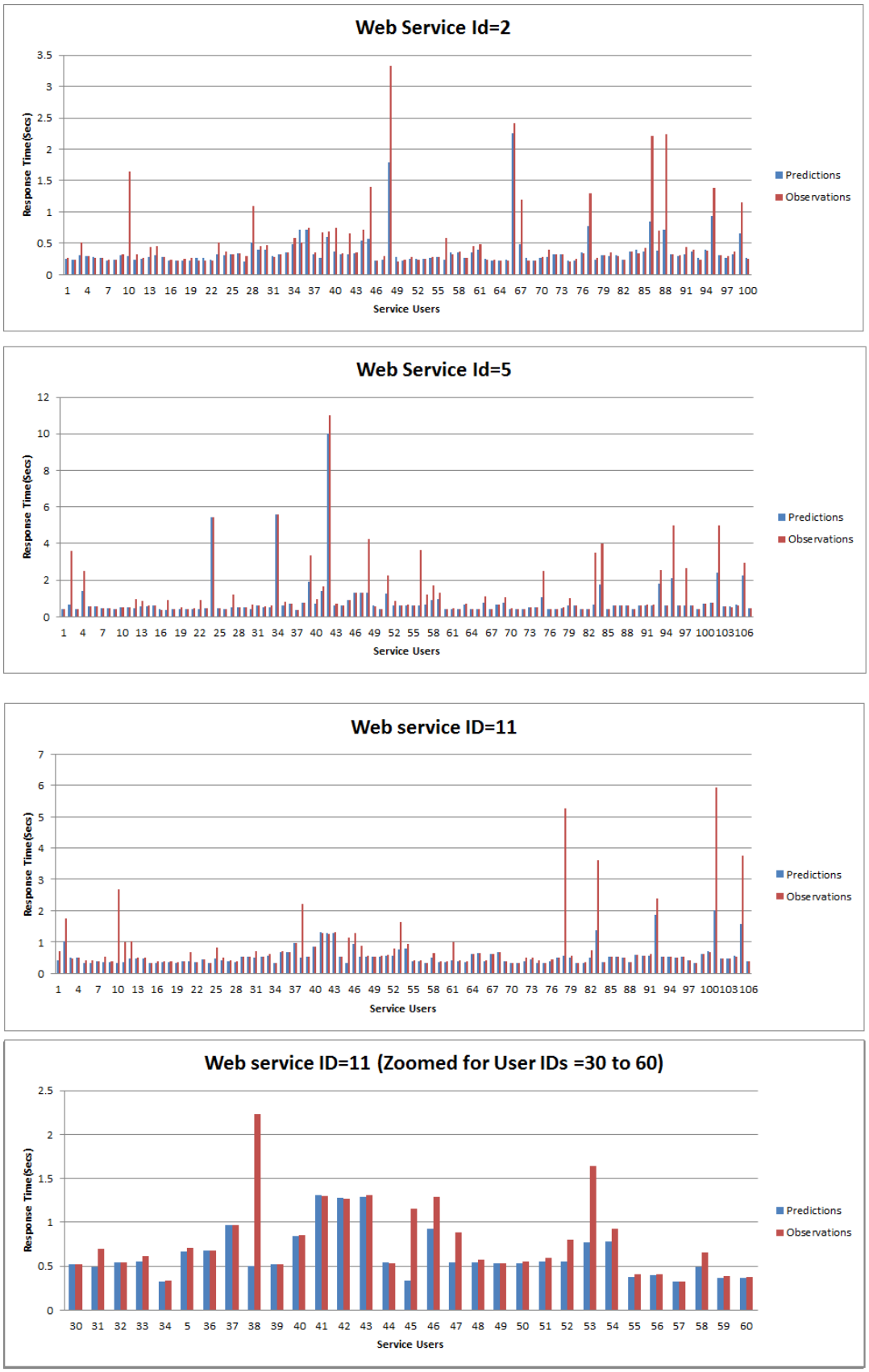


Figure 6.4: Predictions Vs Observations

request i can be segmented as follows: $rt_i = e_i + d_i$, where :

rt_i is the response time for the service request i.

e_i is the time taken by web server to process the service request.

d_i is the delay introduced by network, through which the service request is routed.

Hence we propose the following approach:

1. Predict service execution time for a given service request by modelling web server environment.
2. Predict delay introduced by underlying network by modelling the network through which service request is routed

If a web service ξ would like to predict the response time of another web service ξ_r using white box approach, then ξ should have the proposed white box module of prediction middleware PM installed (Figure 6.5). Web service ξ_r should have the Server side Prediction Middleware component called as SPM installed at its side. Prediction is performed as follows: Execution time prediction is done at ξ_r by SPM using Execution Time Prediction Module and network delay prediction is done at ξ by PM using Network Delay Prediction Module. To perform prediction, PM should send its prediction request to SPM at ξ_r which sends back the predicted execution time and PM predicts network delay and finally computes predicted response time.

In the following subsection we present Execution Time Prediction Module and in subsequent section we detail on Network Delay Prediction Module. In all these subsections, the sample data presented in tabular format is of web service WS1, details of which are given in section 4.5.1.

6.3.1 Execution time prediction

We first list various factors that significantly affect the execution time of a web service. Then we devise bayesian network based approach for predicting execution time.

6.3.1.1 Factors influencing execution time

Execution time of a service request is the sum of **instruction execution time** and **waiting time**. Instruction execution time is the time taken to execute the code within the web service. Waiting time is the time spent by the service request in queues waiting for the processor, other shared resources and for receiving reply from other invoked web services(if any).

The most obvious factor that determines instruction execution time is the execution path taken by the current request. Distinct values for input parameters lead to different execution paths through a web service which have totally different execution times. Hence input parameter values determine the instruction execution time of a service request.

A web server may have several requests at a given time, made by different service requesters. No of service requests pending at the web server when a current request is made affect waiting time of the current request. A web server will be configured to process more than one service request simultaneously. We define *MaxLimit* as the maximum number of service requests that a web server can process concurrently. As long as current number of service requests are less than *MaxLimit*, web server can accept new service requests with out having them to wait. Once current number of requests surpass *MaxLimit*, subsequent requests are made to wait. *Current service load* metric of the web server is a crucial factor in determining the waiting time of a service request. Time spent waiting for other shared resources like database server also add on to the waiting time of service request. In this work we consider only database server as a shared resource. The

approach may be extended to other shared resources like file servers without loss of generality. Current load at the database server is another factor in determining the waiting time of a service request. Time spent waiting for access to shared resources is termed as *resource waiting time*.

For a composite web service, major portion of its waiting comprises the time that it spends waiting for replies from other web services it invokes. This time is termed as *receive time* in this chapter. Thus waiting time of ξ_r is the sum of resource waiting time and receive time.

Procedures for predicting two components of execution time (instruction execution time and waiting time) are presented independently in the following subsections.

6.3.1.2 Predicting instruction execution time

This subsection illustrates the procedure to be adopted to predict Instruction execution time, given an input vector. Let u represent the input vector of the web service ξ_r. $u = \{i_1, i_2, ..i_n\}$ where n indicates number of input parameters in the input vector.

Server side prediction middleware has to be provided with all possible combinations of input vector values that result in different instruction execution times, which it stores in the form of a table, called IET Table (Instruction Execution Times) in its database. Thus, table IET contains (input vector, observed instruction execution time) pairs. These values may be obtained by running the web service on all possible combinations of input vector values under no server load, without invoking constituent services, if any, of ξ.

6.3.1.3 Predicting waiting time

Unlike instruction execution time, waiting time cannot be predetermined for a given set of input parameters as it depends on current server load, database server load and receive time also. The most important factor that determines waiting time is the execution path taken by the current request. Distinct equivalence classes of input vector lead to different execution paths through a web service which have totally different execution times. Especially if the web service ξ_r is a composite web service, different equivalence classes invoke different constituent web services and result in different receive times. Hence input vector classes also determine the waiting of a service request.

The data structures: load details table and Waiting time conditional probability table (WTCPT) and the logs: access logs are used for the prediction of waiting time. Contents and usage of each of them is given below.

The data structure WTCPT is prepared by *SPM at the time of initial set up* and is used later during prediction. The data structure Load details table which is also prepared by *SPM at the time of initial set up* is used for WTCPT construction.

Web servers write all the details of execution instances in **access logs** and so does ξ_r. Each access log is represented by a tuple (s, u, st, et, ξ) where s indicates the serial number of an entry, st and et are the starting time and ending time of execution, ξ is the web service making the request, u is the input vector. Using st, et of access logs, average web server load y and average database server load r in a time interval t can be obtained. This task of computing average server loads at regular time intervals of length t is done by SPM. At the end of each time interval, SPM computes these average values and inserts a tuple in the **Load details** (t, y, r) **table** that SPM maintains in its database.

SPM prepares WTCPT by making use of access logs(training data) and IET table stored in its database. WTCPT contains the following columns: Serial No s,

Input Vector class u, Server load y, Database server load r, and Waiting time n. Access logs and WTCPT collected for a sample web service are shown in Table 6.4 and Table 6.5 respectively. From WTCPT it may be noted that for a given input vector class, server load and database server load, there might be several possible waiting times due to nonmeasurable factors like delay introduced by scheduling algorithm, network delay etc. WTCPT construction Algorithm used by SPM is given in Algorithm 9.

Table 6.4: Sample Access Logs

S.No	**Input Vector**	**Start Time**	**End Time**
1	U_1	12:30:540	12:31:106
2	U_1	12:31:236	12:31:889
3	U_2	12:32:701	12:33:103
4	U_2	12:34:345	12:34:775
5	U_3	12:36:876	12:39:409
6	U_3	12:37:590	12:39:276
7	U_4	12:38:266	12:38:405

Algorithm 9 WTCPT Construction Algorithm

Inputs n access logs (s_i, u_i, st_i, et_i), $1 \leq i \leq n$, Load details table (t, y, r) and IET (u_i, iet_i).

1: **for** each access log i=1 to n **do**
2: Enter i under column S. Retrieve input vector u_i and enter it under U.
3: Determine the time interval t_i from st_i and et_i.
4: Determine execution time $e_i = et_i - st_i$.
5: Then retrieve server load y and database server load r from Load details table for the time interval $t = t_i$. Enter them under columns y, r respectively.
6: For the input vector u_i retrieve instruction execution time iet_i from IET.
7: Populate waiting time under column N using $n_i = e_i - iet_i$.
8: **end for**

Load details table and WTCPT are prepared by SPM at the beginning of each time interval using current access logs. Load details table contains as many entries as there are number of time intervals per day. Entries in WTCPT are not

Table 6.5: Sample WTCPT

S.Nos	**Input Vector u**	**Server load y**	**Database server load r**	**waiting time n**(msecs)
1	U_1	14	8	486
2	U_1	14	8	523
3	U_2	14	8	351
4	U_2	14	8	459
5	U_3	14	8	1412
6	U_3	14	8	1555
7	U_4	14	8	168

limited since there might be many possible combinations of u, y, r. During run time, at the end of each time interval $t-1$, SPM retrieves access logs, computes y_{t-1}^{new} and r_{t-1}^{new} and, updates the entry for time interval $t-1$ in Load details table. Then it takes average of (y_{t-1}^{new} and y_t) and (r_{t-1}^{new} and r_t) to update values for y_t and r_t respectively in Load details table. This means that server loads for time interval t are updated with average of server loads for time interval t of previous day and time interval $t-1$ of current day. Then, entries corresponding to all calls made by ξ in $t-1$ are added to WTCPT by SPM. The framework for prediction is depicted in Figure 6.5. Observation tuples seen in this figure are used for network delay prediction and is described in the subsection 6.3.2.

Having prepared the data structures required for prediction, SPM can perform waiting time prediction.

When a new request (u, t), where u indicates input vector class and t indicates time interval, is submitted to SPM by PM for response time prediction, execution time prediction is done by summing up instruction execution time and waiting time. Instruction execution time for the given input vector class u is retrieved from IET, where as waiting time has to be estimated using WTCPT and load details table. WTCPT prepared initially with training data provides priors.

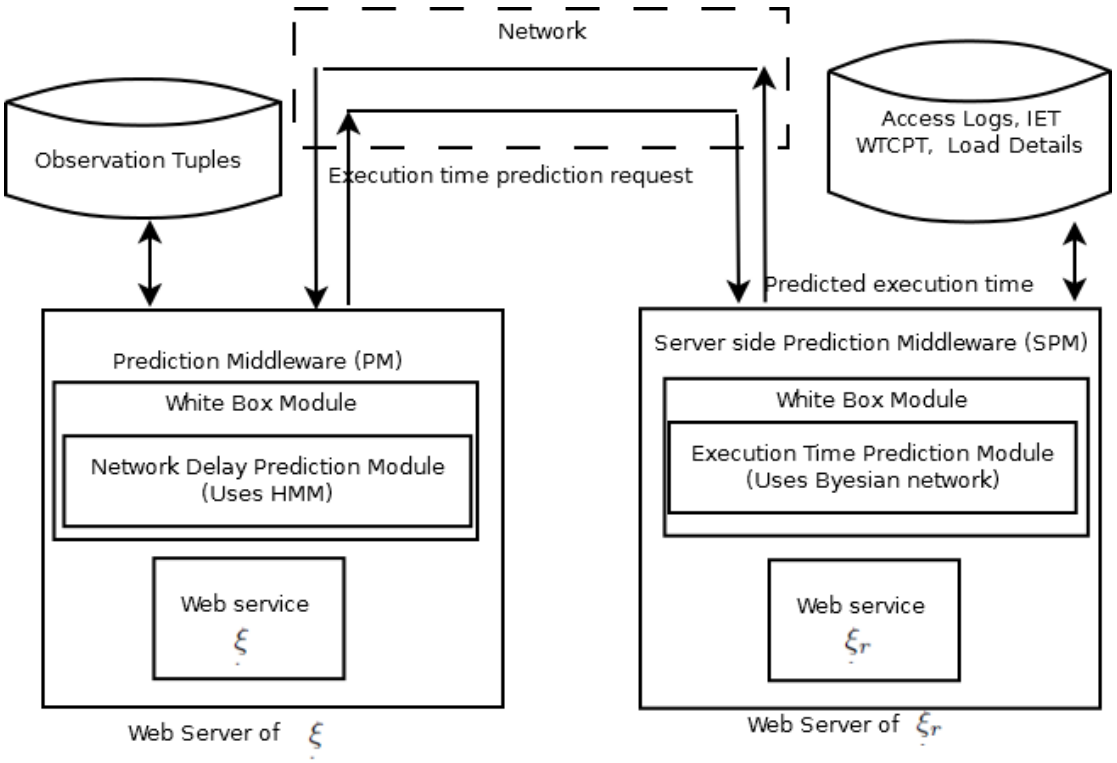


Figure 6.5: Framework for response time prediction using white box approach

In literature, Maximum likelihood estimation and Bayesian estimation have been used successfully for estimating quantities. Maximum likelihood estimation gives a point estimate for a parameter θ (whose value is to be estimated). In the bayesian approach, parameter θ to be estimated is treated as a random variable and its probability distribution is computed.

Bayesian estimation has the advantage that it allows us to specify prior distribution, which reduces the amount of time required for the model to converge and result in more accurate estimates. Hence in cases where prior probability given by $Pr(\theta)$, for is available for parameter θ it is advantageous to use bayesian estimation. Hence we propose to use bayesian approach for estimating waiting time.

Waiting time spent by a service request in time interval t depends upon current server load and database server load. It also depends upon input vector class u of the service request since input vector class determines the execution path to be

taken which in turn determines the receive time. Waiting time has to be estimated using these quantities. The bayesian network that represents this relationship is depicted in Figure 6.6.

Predicting waiting time using Bayesian network

Let u_t represent a random variable that represents the input vector class for service request submitted to the web server in time interval t. y_t is a random variable that represents the server load in time interval t. Random variable r_t represents database server load in time interval t. n_t is another random variable that represents waiting time spent by the service request at web server. The Bayesian network O computes waiting time probabilities conditioned on the input u_t, server load y_t, and database server load r_t.

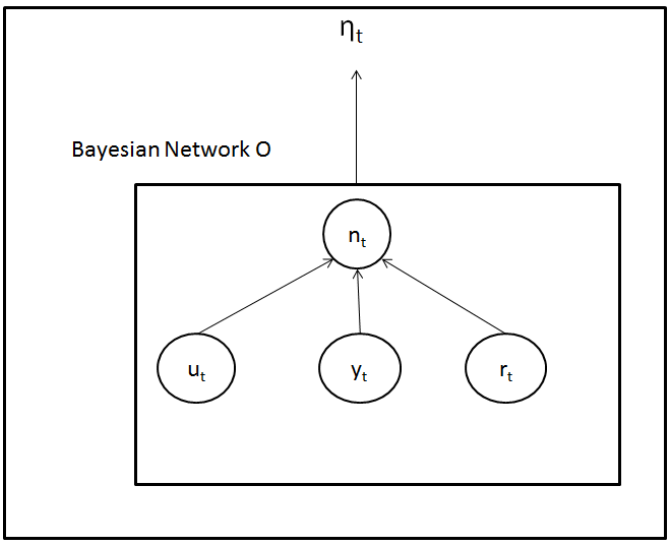


Figure 6.6: Bayesian Network for learning waiting time

Let η be the expected value for probability distribution of waiting time n_t. Thus η = expected value of probability distribution $Pr(n_t|u_t, r_t, y_t)$ as depicted in Figure 6.6.

Let $h(u_t, n_t, y_t, r_t) = Pr(n_t|u_t, r_t, y_t)$

Let D represent data in WTCPT.

θ_j = represents probabilities of all k_j possible discrete values that n_t would take where $1 \leq j \leq m$ and m is number of different possible combinations of values of u_t,y_t and r_t . These different possible discrete values for n_t are obtained using WTCPT. Thus θ_j is a vector of values = $\{\theta_{j1}, \theta_{j2}...\theta_{jk_j}\}$.

Posterior probabilities for each of θ_j could be calculated independently.

Assuming Dirichlets distributions $Dir(\theta_j|\alpha_{j1}, ...\alpha_{jk_j})$, for priors of the parameters θ_j (Dirichlet distributions are often used as prior distributions of multivariate normal distributions represented by Bayesian networks and α_{ji}, $1 \leq i \leq k_j$ are hyper-parameters of Dirichlet distribution that represent expert knowledge), we have posterior probabilities as:

$Pr(\theta_j|D) = Dir(\theta_j|\alpha_{j1} + N_{j1}, ..., \alpha_{jk_j} + N_{jk_j})$

where N_{ji} is the number of cases in D that have i^{th} possible discrete value represented by probability distribution θ_j .

Thus the expected value of the probability distributions η is computed as follows:

$$\begin{aligned} \eta_j &= \text{Expected value of } h(n_t, (u_t, y_t, r_t)^j) \\ &= \sum_i h(n_t^i, (u_t, y_t, r_t)^j).n_t^i \end{aligned} \tag{6.1}$$

where

$(u_t, y_t, r_t)^j$ is j^{th} possible combination of (u_t, y_t, r_t) in D where $(u_t = u, y_t = y, r_t = r)$.

$h(n_t^i, (u_t, y_t, r_t)^j) = \frac{\alpha_{ji}+N_{ji}}{\alpha_j+N_j}$.

$\alpha_j = \sum_{i=1}^{k_j} \alpha_{ji}$ and

$N_j = \sum_{i=1}^{k_j} N_{ji}$ □

After estimating waiting time, execution time can be easily predicted as detailed in Algorithm 10.

Algorithm 10 Execution Time Prediction Algorithm

Input new request (u, t), Load details table (t, y, r), WTCPT, LWS.

1: For the requested time interval t retrieve server load y and database server load r from Load details table.
2: Estimate waiting time η using y, r, u as given in equation 6.1.
3: Predicted execution time $e = iet + \eta$.

In the following subsection, we slightly modify the HMM based prediction approach presented in previous section, to predict network delay.

6.3.2 Network delay prediction using HMM

Previous subsection details on prediction of execution time by SPM. In this subsection we present an HMM based approach for inferring delay induced by network based on observed metrics. A network can be in any one of two possible states: **congested** and **non congested**. Several factors have their role in determining current state of a network: number of packets that are currently being routed through it, network bandwidth, routing and congestion control algorithms used etc. But Prediction middleware has no direct access to all such details and thus network state remains hidden to Prediction middleware. What could be measured is the delay that is introduced by the network in transmission of request and response messages between sender and receiver.

We propose historical data based prediction approach. In order to perform network delay prediction PM at ξ prepares **observation tuples** table, whenever ξ_r is invoked. i^{th} **observation tuple** for web service ξ_r is represented by (i, t_i, u_i, rt_i) and $1 \leq i \leq n$ where: t_i = time interval, u_i is the input vector submitted to ξ_r by ξ, rt_i = response time and n = total number of observations. Prediction middleware PM has to be given n such observations in its database as training data. When a new request (u, t) is submitted, Network Delay Prediction module has to

predict average network delay d_t in the time interval t.

In order to predict network delay d_t in a time interval t, PM makes use of well known Hidden Markov Model (HMM).

6.3.2.1 HMM for underlying network

States of the underlying network are not visible to PM and hence can be mapped to hidden states of a HMM. Since delay introduced by the network can be measured and hence, are visible, they become emissions of the HMM. Table 6.6 presents an analogy between the HMM and Network states.

Network Delay Prediction module predicts average network delay in the time

Table 6.6: Analogy between HMM and Network States

HMM	Network
States $S = \{S_1, S_2, ..., S_N\}$	States of the network $S = \{noncongested, congested\}$
Observation o_t	network delay d_t in time interval t
π_i	Probability of moving to a network state S_i at time $t = 1$
a_{ji}	Probability of moving to the network state S_i at time t given the network state S_j at time $t - 1$
$b_j(O_m)$	Probability of emitting a network delay $O_m = d_t$ from the state S_j

interval t by performing following tasks:

1. Extract network delay for each of the observation tuples given in training data. For each observation tuple i, extract network delay by using the formula $d_i = rt_i - e_i$.

2. Compute average network delay for each time interval by taking average of

network delays available for the time interval t and prepare training data containing average network delay time interval wise.

3. Run the $EMAlgorithm$ on training data, i.e n observation tuples provided, and train the HMM(i.e, determine the parameters of the HMM). This is a one time task which is done initially when the framework is set up.

4. Consolidate average network delay, similar to steps 1 and 2. This is a continuous procedure which has to be performed at the end of each time interval.

5. Execute the $ViterbiAlgorithm$ for network delay prediction upon receiving a request.

6.3.3 Experimentation and Results

The proposed white box approach for predicting response time of web services has been tested using the same experimental set up specified in subsection 5.6 of previous chapter. Web services WS1, WS2, and WS5 were used for predictions. Web services WS2 and WS5 have only one equivalence class for input vector. Web service WS1 has four equivalence classes U_1 to U_4. For WS1 if U=1, currency convertor web service is called, U=2, calculator web service is called U=3, both the web services are called, U=4, none of them are called. For each of the web services, IET contains one entry per equivalence class. Training data set has been so generated that for every (u, y, r) triplet, there are around 5 entries in WTCPT having different values for waiting time. While calculating waiting time η using equation 6.1 for a given (u, y, r) tuple, if exact matching entry is not available in WTCPT, all i rows with same u value are first selected. Then rows with minimum Euclidean distance from the required y, r are selected and used for calculating η, i.e., rows with minimum value of $\sqrt{(y - y_j)^2 + (r - r_j)^2}$ where $1 \leq j \leq i$, are selected.

Figure 6.7 depicts the predicted response time values using both black box and

white box approaches against actual recorded values. Prediction was carried over for web services WS1, WS2, and WS5 in the time intervals t=1 to t=7. The results for WS2, WS5 and WS1 are depicted in Figure 6.7 (a),(b) and (c) respectively. It may be noted that these intervals are not continuous, they do not belong to same hour of the day. To analyse the correctness of predictions, we have computed Mean Absolute Error (MAE) and Root Mean Square Error(RMSE) for predictions of both approaches. Figure 6.7 (d) depicts these results. Lesser values for these errors for white box approach than black box approach clearly indicate that white box approach is more accurate than black box approach.

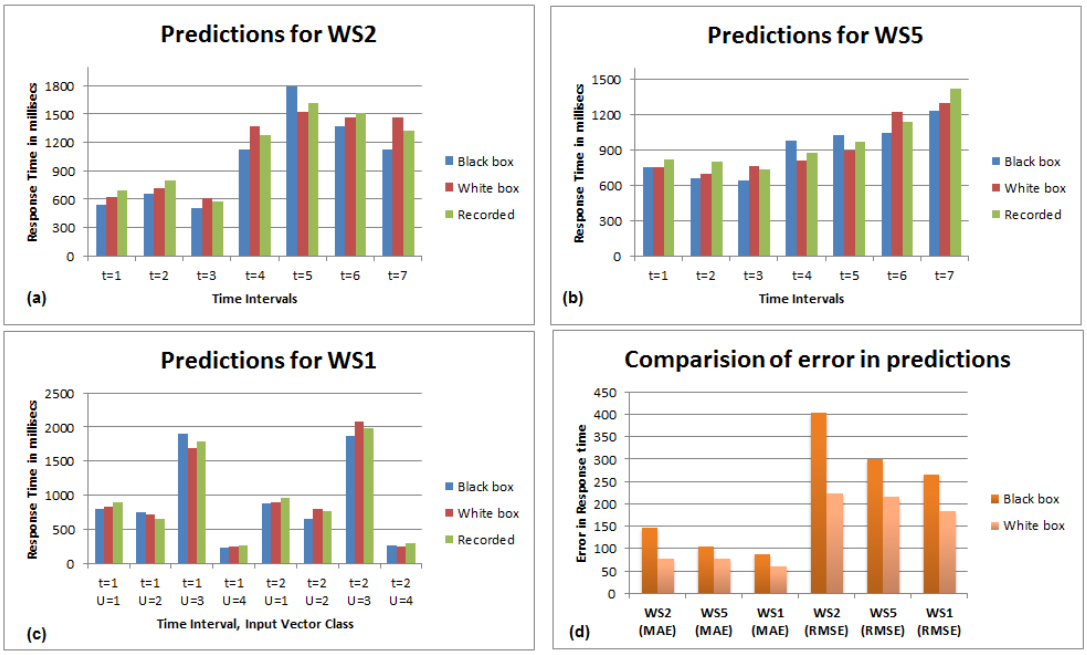


Figure 6.7: Comparision of prediction results

6.4 Conclusion

In our previous chapter we have argued upon the need to checkpoint a composite web service ξ dynamically, the reasons being: difference in actual and advertised response times of web services invoked by ξ, failure rate variation of ξ at run time and provision of dynamic composition of ξ. We have proposed to predict response

times of constituent web services of ξ, which closely represent actual response times, at the beginning of each time interval t. Using these predicted values we do run time checkpointing.

We have proposed two prediction approaches: i)a black box approach using HMM and ii)a white box approach using HMM and Bayesian network for response time prediction. We have demonstrated empirically that white box approach is superior to black box approach in prediction accuracy as it has finer implementation details at its disposal. But the proposed black box approach comes to our rescue when such implementation and server environment details are not available to us.

CHAPTER 7

Conclusion and Future Work

Checkpointing and recovery is a general fault handling mechanism that has been successfully applied to distributed applications to deal with transient faults. Although web services operationally behave like distributed systems, they differ from distributed systems because of their specific characteristics like maintaining QoS attributes, provisioning of dynamic composition etc. Hence, the checkpointing strategies designed for generic distributed systems are not readily applicable for the purpose. Below, we present such characteristics justifying this study on web service checkpointing. Subsequently we summarise specific problems and the solutions proposed in this research:

- Distributed checkpointing and recovery algorithms require processes other than failed process also to rollback to maintain consistent state of the entire application. In service domain, requesting partner services to rollback is not desirable, or even not permissible for business obligations. Hence we propose a checkpointing and recovery scheme where only the failed web service has to roll back. We do so by capturing the essential elements of design document of composite web services, called choreography document. We model different ways in which web services interact with each other in our "Interaction Pattern Model". We use the proposed patterns to propose checkpointing locations. Design time checkpointing rules identify checkpointing locations in such a way that failure in one service does not request rollback of other services. We have demonstrated that the rules are practically implementable through the development of a tool called ACGM.

- Web services have to deliver their services according to terms and conditions laid down in SLAs. SLAs define deadlines for maximum response time, cost

of service, minimum reliability to be provided etc. In our second module, which we call as deployment time checkpointing, we take up the idea of applying checkpointing and recovery to composite web services so that web services can meet time and cost deadlines even in the presence of transient faults. In order to take checkpointing decisions we consider the Quality of Service attribute values of all services participating in the choreography. The checkpoint locations are determined at deployment time since the QoS values of all participants would be available by this time. We propose to introduce minimum number of checkpoints so that failure free executions are not penalised. At the same time, the checkpoints introduced ensure that cost and time constraints are satisfied event in the case of failed and subsequently recovered instances.

- In our deployment time checkpointing module we consider advertised QoS values of the constituent services to take checkpointing decisions. But for certain web services actual QoS values would deviate much from advertised QoS values. This is more so with response time. This might be due to varying network traffic or due to varying number of pending requests at the corresponding web server. In case of higher response times, more number of checkpoints have to be introduced to keep up with time constraints in "failed and recovered" scenario. In case of reduced response times, some of the checkpoints can be removed so that failure free executions achieve improved performance in terms of speed.

  For certain other web services, some of the constituent web services would be dynamically selected. We would not have with us advertised QoS values for the service which is to be selected dynamically. Hence, at deployment time, we use maximum permissible values for QoS attributes to take checkpointing decisions. But at run time, when these values are available to us, checkpoint locations can be revised accordingly. Hence we propose dynamic checkpointing in our third module, which takes up this revision of check-

pointing locations at run time.

To perform this run time checkpointing, we need predicted response time values which would be as much close as possible to actual response time values. Hence we have proposed two approaches to predict response times of constituent web services: 1)Black box approach using HMM 2)White box approach using HMM and Bayesian networks. We use history data based prediction in black box approach. We model the web service whose response time has to be predicted and its underlying network using HMM. We initially apply EM algorithm to learn the parameters of HMM from historical data. Then we apply Viterbi algorithm to observations of previous $n-1$ intervals to predict the response time in n^{th} time interval. Predictions can be done more accurately if details like number of outstanding requests at the web server, input vector values etc are known to the prediction module. Our white box approach which has access to all these details, performs accurate response time prediction in two steps: i) predict waiting time at web server using Bayesian networks and use this predicted value in computing execution time, and ii)predict network delay using HMM. Depending on the details available, we apply one of the proposed approaches for predicting response time values.

Summarising, we have proposed a three stage checkpointing strategy to handle transient faults and prevent SLA faults. Design and deployment time checkpointing are one time activities and do not incur much overhead. Dynamic checkpointing has to be used when there is significant difference in advertised and actual response times or when dynamic composition is facilitated. We have also proposed two approaches for response time prediction: Black box approach and White box approach. White box approach gives accurate predictions, but when web service environment details are not accessible, black box approach comes handy.

7.1 Future work

Our proposed Interaction pattern model can capture choreographies that can be specified by UML activity diagrams. But choreographies specified using other choreography specification languages like WSCDL and BPMN contain certain other patterns which cannot be captured using our model. For example, consider the following choreography: An auctioning service invites bids by sending messages to 10 bidders. It does not wait for replies from all ten bidders, but proceeds when it receives bids from atleast 3 bidders. Our pattern type P1 can be used to model sending a request message to a bidder and waiting for his reply. Concurrent pattern can be used to model sending messages to 10 bidders simultaneously. But our model does not have enough elements to specify that the auctioning service proceeds after receiving reply from *atleast 3 bidders*.

Another scenario which cannot be captured by our model can be explained by modifying the above example as follows: An auctioning service invites bids by sending messages to 10 bidders. It does not wait for replies from all ten bidders, but proceeds with whatever replies it gets *within 3 days*. Our model does not have enough elements to specify such time specific replies.

Hence as part of our future work we propose to extend our Interaction pattern model so as to model m out of n replies, time specific replies etc.

Our proposed Time and cost aware checkpointing algorithm decides checkpointing locations based on deadlines for execution time and cost of service. It can be revised to consider deadlines on other Quality of service attributes like reliability and trust also.

REFERENCES

1. *Oracle SOA Suite 12c A Detailed Look*. Oracle, 2014. URL `http://www.oracle.com/us/products/middleware/soa/suite`.

2. **Ahmed, W.** and **Y. Wu** (2013). Probabilistic qos analysis of web services. *Network and Parallel Computing*, **8147**, 393–404.

3. **Alvisi, L.**, **K. Bhatia**, and **K. Marzullo** (2002). Causality tracking in causal message-logging protocols. *Journal of Distributed Computing*, **15**(1), 1–15.

4. **Avizienis, A.**, **J.-C. Laprie**, **B. Randell**, and **C. Landwehr** (2004). Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, **1**(1), 11–33.

5. **Balogh, Z.**, **E. Gatial**, **M. Laclavik**, **M. Maliska**, and **L. Hluchy** (2006). Knowledge-based runtime prediction of stateful web services for optimal workflow construction. *Parallel Processing and Applied Mathematics*, **3911**, 599–607.

6. **Barros, A.**, **M. Dumas**, and **P. Oaks** (2005). A critical overview of the web services choreography description language. *BPTrends Newsletter*, **3**, 1–24.

7. **Ben Halima, R.**, **K. Drira**, and **M. Jmaiel** (2008). A qos-oriented reconfigurable middleware for self-healing web services. *IEEE International Conference on Web Service*, 104–111.

8. **Berson** and **Alex**, *Master Data Management & Data Governance*. McGraw-Hill Education (India) Pvt Limited, . URL `https://books.google.co.in/books?id=P-Upga3zz_4C`.

9. **Bhargava, B.** and **S.-R. Lian** (1988). Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach. *Seventh Symposium on Reliable Distributed Systems, 1988. Proc.*, 3–12.

10. **Bland, W.**, **P. Du**, **A. Bouteiller**, **T. Herault**, and **Bosilca** (2013). Extending the scope of the checkpoint-on-failure protocol for forward recovery in standard mpi. *Concurrency and Computation: Practice and Experience*, **25**(17), 2381–2393.

11. **Bronevetsky, G.**, **D. Marques**, **K. Pingali**, and **P. Stodghill** (2003). Automated application-level checkpointing of mpi programs. *Proc of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 84–94.

12. **Cappé, O.** and **E. Moulines** (2005). Inference in hidden markov models.

13. **Chan, K.**, **J. Bishop**, **J. Steyn**, **L. Baresi**, and **S. Guinea** (2009). A fault taxonomy for web service composition. *Service-Oriented Computing Workshops*, **4907**, 363–375.

14. **Chandy, K. M.** and **L. Lamport** (1985). Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, **3**(1), 63–75.

15. **Chen, L.**, **Y. Feng**, **J. Wu**, and **Z. Zheng** (2011*a*). An enhanced qos prediction approach for service selection. *IEEE International Conference on Services Computing*, 727–728.

16. **Chen, L.**, **J. Yang**, and **L. Zhang** (2011*b*). Time based qos modeling and prediction for web services. *Service-Oriented Computing*, **7084**, 532–540.

17. **Chen, N.**, **Y. Yu**, and **S. Ren** (2009). Checkpoint interval and system's overall quality for message logging-based rollback and recovery in distributed and embedded computing. *ICESS*, 315–322.

18. **Cheung, L.**, **L. Golubchik**, and **F. Sha** (2011). A study of web services performance prediction: A client's perspective. *IEEE 19th International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems*, 75–84.

19. **Chtepen, M.**, **B. Dhoedt**, **F. De Turck**, **P. Demeester**, **F. Claeys**, and **P. Vanrolleghem** (2009). Adaptive checkpointing in dynamic grids for uncertain job durations. *Proc of the ITI 2009 31st International Conference on Information Technology Interfaces*, 585–590.

20. **CurrencyConvertor**, *webservicex.net*. URL `http://www.webservicex.net/CurrencyConvertor.asmx`.

21. **Dempster, A. P.**, **N. M. Laird**, and **D. B. Rubin** (1977). Maximum likelihood from incomplete data via the em algorithm. *Journal of The Royal Statistical Society*, **39**(1), 1–38.

22. **Elnozahy, E. N.**, **L. Alvisi**, and **Y.-M. Wang** (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, **34**(3), 375–408.

23. **Ezenwoye, O.** and **S. M. Sadjadi** (2007). Trap/bpel: A framework for dynamic adaptation of composite services. *Proc of WEBIST*, 216–221.

24. **Forney, J., G.D.** (1973). The viterbi algorithm. *Proc of the IEEE*, **61**(3), 268–278.

25. **Gao, Z.** and **G. Wu** (2005). Combining qos-based service selection with performance prediction. *IEEE International Conference on e-Business Engineering*, 611–614.

26. **Geebelen, D.**, **K. Geebelen**, **E. Truyen**, **S. Michiels**, **J. A. Suykens**, **J. Vandewalle**, and **W. Joosen** (2014). Qos prediction for web service compositions using kernel-based quantile estimation with online adaptation of the constant offset. *Information Sciences*, **268**(0), 397 – 424.

27. **Gonzalez, A.**, **A. Roque**, and **J. Garcia-Gonzalez** (2005). Modeling and forecasting electricity prices with input/output hidden markov models. *IEEE Transactions on Power Systems*, **20**(1), 13–24.

28. **Hassan, M. R.** and **B. Nath** (2005). Stockmarket forecasting using hidden markov model: A new approach. *Proc of the 5th International Conference on Intelligent Systems Design and Applications*, 192–196.

29. **Heckerman, D.** (1995). A tutorial on learning with bayesian networks. *Microsoft Research*.

30. **H.E.Mansour** and **T.Dillon** (2011). Dependability and rollback recovery for composite web services. *IEEE Transactions on Services Computing*, **4**(4), 328–339.

31. **Johnson, D. B.** and **W. Zwaenepoel** (1990). Recovery in distributed systems using optimistic message logging and check-pointing. *J. Algorithms*, **11**(3), 462–491.

32. **Koo, R.** and **S. Toueg** (1987). Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 23–31.

33. **Laranjeiro, N.**, **M. Vieira**, and **H. Madeira** (2009). Predicting timing failures in web services. *Database Systems for Advanced Applications*, **5667**, 182–196.

34. **Li, Z.**, **Z. Bin**, **N. Jun**, **H. Liping**, and **Z. Mingwei** (2010). An approach for web service qos prediction based on service using information. *International Conference on Service Sciences*, 324–328.

35. **Lin, L.** and **M. Ahamad** (1990). Checkpointing and rollback-recovery in distributed object based systems. *20th International Symposium Fault-Tolerant Computing*, 97–104.

36. **Linthicum, D.** (). Service oriented architecture (soa). *Microsoft developer network*. URL `http://msdn.microsoft.com`.

37. **Liu, A.**, **L. Qing**, **L. Huang**, and **M. Xiao** (2010). Facts: A framework for fault-tolerant composition of transactional web services. *IEEE Transactions on Services Computing*, **3**(1), 46–59.

38. **Marzolla, M.** and **R. Mirandola** (2007). Performance prediction of web service workflows. *Software Architectures, Components, and Applications*, **4880**, 127–144.

39. **Marzouk, S.**, **A. J. Maâlej**, and **M. Jmaiel** (2010). Aspect-oriented checkpointing approach of composed web services. *Proc of the 10th International Conference on Current Trends in Web Engineering*, 301–312.

40. **McLachlan, G. J.** and **T. Krishnan**, *The EM Algorithm and Extensions*. Wiley, 2nd Edition, 2008.

41. **Nazir, B.**, **K. Qureshi**, and **P. Manuel** (2009). Adaptive checkpointing strategy to tolerate faults in economy based grid. *The Journal of Supercomputing*, **50**(1), 1–18.

42. **Netzer, M. R.**, **D. Manivannan**, **R. H. B. Netzer**, and **M. Singhal** (1997). Finding consistent global checkpoints in a distributed computation. *IEEE Transactions on Parallel and Distributed Systems*, **8**, 623–627.

43. **Netzer, O.**, **J. M. Lattin**, and **V. Srinivasan** (2008). A hidden markov model of customer relationship dynamics. *Marketing Science*, **27**(2), 185–204.

44. **Netzer, R. H.** and **J. Xu** (1995). Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and distributed Systems*, **6**(2), 165–169.

45. **Nghiem, A.**, *IT Web Services: A Roadmap for the Enterprise*. Prentice Hall PTR, 2003.

46. **Nickolas Kavantzas, D. B.** (ed.), *A Critical Overview of the Web Services Choreography Description Language Version 1.0*. W3C Recommendation, 2005. URL `http://www.w3.org/TR/ws-cdl-10/`.

47. **OMG** (ed.), *Business Process Model and Notation*. Object Management Group, 2011. URL `http://www.omg.org/spec/BPMN/2.0`.

48. **Osada, S.** and **H. Higaki** (2001). Qos-based checkpoint protocol for multimedia network systems. *Proc of the Second IEEE Pacific Rim Conference on Multimedia: Advances in Multimedia Information Processing*, 574–581.

49. **P.J.Darby** and **N. F. Tzeng** (2010). Decentralized qos-aware checkpointing arrangement in mobile grid computing. *IEEE Transactions on Mobile Computing*, 1173–1186.

50. **Rabiner, L.** (1989). A tutorial on hidden markov models and selected applications in speech recognition. *Proc of the IEEE*, **77**(2), 257–286.

51. **R.D.Albu** (2013). A comparative study for web services response time prediction. *The 9th International Scientific Conference*, **1**, 656–665.

52. **R.D.Albu** (2014). Input projection algorithms influence in prediction and optimization of qos accuracy. *International Journal of Computers Communications and Control*, **9**(2), 131–138.

53. **R.D.Albu**, **I. Felea**, and **F. Popentiu Vladicescu** (2013). On the best adaptive model for web services response time prediction. *20th International Conference on Systems, Signals and Image Processing*, 39–42.

54. **Resch, B.** *et al.* (2004). Hidden markov models a tutorial for the course computational intelligence. *J. Algorithms*.

55. **Robert, Y.**, **F. Vivien**, and **D. Zaidouni** (2012). On the complexity of scheduling checkpoints for computational workflows. *IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops*, 1–6.

56. **Roberto Chinnici, J. J. M.** (ed.), *Web Services Description Language (WSDL) Version 2.0*. W3C Recommendation, 2007. URL `http://www.w3.org/TR/wsdl20/wsdl20.pdf`.

57. **Rukoz, M.**, **Y. Cardinale**, and **R. Angarita** (2012). Faceta*: Checkpointing for transactional composite web service execution based on petri nets. *Procedia Computer Science*, **10**, 874–879.

58. **Russell, D.** (1980). State restoration in systems of communicating processes. *Proc of IEEE Transactions on Software Engineering*, **6**(2), 183–194.

59. **Russell, N.**, **W. M. van der Aalst**, **A. H. Ter Hofstede**, and **P. Wohed** (2006). On the suitability of uml 2.0 activity diagrams for business process modelling. *Proceedings of the 3rd Asia-Pacific conference on Conceptual modelling*, **53**, 95–104.

60. **Sen, S.**, **H. Demirkan**, and **M. Goul** (2005). Towards a verifiable checkpointing scheme for agent-based interorganizational workflow system" docking station" standards. *Proc of the 38th Annual Hawaii International Conference*, 165–173.

61. **Silva, L.** and **J. G. Silva** (1992). Global checkpointing for distributed programs. *Proc of 11th Symposium on Reliable Distributed Systems*, 155–162.

62. **Strom, R.** and **S. Yemini** (1985). Optimistic recovery in distributed systems. *Proc of IEEE Transactions on computers*, **3**(3), 20–226.

63. **SusanD., U.**, **G. Le**, **S. Rajiv**, and **C. Andrew** (2010). Achieving recovery in service composition with assurance points and integration rules. *On the Move to Meaningful Internet Systems: OTM*, **6426**, 428–437.

64. **Tamir, Y.** and **C. H. Sequin** (1984). Error recovery in multicomputers using global checkpoints. *Proc of the International Conference on Parallel Processing*, 32–41.

65. **Tan, W.**, **L. Li**, **Y. Sun**, and **W. Tan** (2014). Performance prediction and analysis of quality of services for cross-organizational workflows. *Proc of IEEE 11th International Conference on e-Business Engineering*, 145–150.

66. **Vani Vathsala, A.** (2011). Optimal call based checkpointing for orchestrated web services. *International Journal of Computer Applications*, **36**(8), 44–50.

67. **Vani Vathsala, A.** (2012). Global checkpointing of orchestrated web services. *Proc of 1st International Conference on Recent Advances in Information Technology*, 461–467.

68. **Vani Vathsala, A.** and **HrushikeshaMohanty** (2012). Using hmm for predicting response time of web services. *Proc of CUBE International Conference*, 520–525.

69. **Vani Vathsala, A.** and **HrushikeshaMohanty** (2014*a*). Interaction patterns based checkpointing of choreographed web services. *Proc of the 6th International Workshop on Principles of Engineering Service Oriented and Cloud Systems*, 28–37.

70. **Vani Vathsala, A.** and **HrushikeshaMohanty** (2014*b*). A survey on checkpointing web services. *Proc of the 6th International Workshop on Principles of Engineering Service Oriented and Cloud Systems*, 11–17.

71. **Vani Vathsala, A.** and **HrushikeshaMohanty** (2014*c*). Web service response time prediction using hmm and bayesian network. *Intelligent Computing, Communication and Devices*, 327–335.

72. **Vani Vathsala, A.** and **HrushikeshaMohanty** (2015). Time and cost aware checkpointing of choreographed web services. *Proc of the 11th International Conference on Distributed Computing and Information Technology*, 207–219.

73. **WANG, Y.** (1997). Consistent global checkpoints that contain a set of local checkpoints. *Proc of IEEE Transactions on computers*, **46**(4), 456–468.

74. **Yan, H.** and **Z.-Z. Liu** (2013). An approach for web service qos dynamic prediction. *Journal of Software*, **8**(10), 2637–2643.

75. **Yan Hai, G. L., Luoheng Yan** (2012). Dynamic prediction qos-aware web service composition model. *International Journal of Digital Content Technology and its Applications*, **6**(12).

76. **Zhang, Y.** and **K. Chakrabarty** (2003). Energy-aware adaptive checkpointing in embedded real-time systems. *Proc of the Conference on Design, Automation and Test in Europe*, **1**, 918–923.

77. **Zhang, Y.**, **Z. Zheng**, and **M. R. Lyu** (2011). Wspred: A time-aware personalized qos prediction framework for web services. *Proc of IEEE Symposium on Software Reliability Engineering*, 210–219.

78. **Zhou, J.**, **C. Zhang**, **H. Tang**, **J. Wu**, and **T. Yang** (2010). Programming support and adaptive checkpointing for high-throughput data services with log-based recovery. *IEEE/IFIP International Conference on Dependable Systems and Networks*, 91–100.

79. **Zwaenepoel, W.** and **D. Johnson** (1987). Sender-Based Message Logging. *Proc of the Seventeenth International Symposium on Fault-Tolerant Computing*, 14–19.