Acknowledgments

In the first place I would like to record my gratitude to Dr. Brahim Medjahed for his supervision, advice, and guidance from the very early stage of this research as well as giving me extraordinary experiences throughout the work. He provided me with unflinching encouragement and support in various ways. His truly scientist intuition has made him as a constant oasis of ideas and passions in science, which exceptionally inspires and enrich my growth as a student, a researcher and a scientist want to be. I am indebted to him more than he knows.

My sincere thanks go to the members of the thesis committee for their time. And my pleasure to the CIS department for allowing me to work on the thesis.

Furthermore, I would thank my brother Naseer. He is the one who encouraged me to do my Master study, I am thankful that in the midst of all his helping

Words fail me to express my appreciation to my wife Houda whose dedication, love and persistent confidence in me, has taken the load off my shoulder. I owe her for being unselfishly let her intelligence, passions, and ambitions collide with mine.

Finally, I would like to thank everybody who participated in the realization of a successful thesis, as well as expressing my apology that I could not mention personally one by one.

This research is supported by a grant from the University of Michigan’s OVPR.
Abstract

Service-Oriented Architecture (SOA) utilizes services as the building blocks for developing software systems distributed within and across organizations. The most common realization of SOAs is based on Web services; these are loosely coupled entities that provide pre-defined capabilities via XML-based standards on the Web. The primary value of SOA is the ability to reuse existing services either as standalone or as part of composite services that perform more complex functions by combining pre-existing services (called participants).

SOAs in modern application domains face unavoidable changes during their lifetime. We identify two categories of changes in SOAs: failure-related and content-related changes. Failure-related changes refer to changes that may lead to failures in SOA services. Content-related changes refer to changes in the data returned by a service or policies associated to a service. Changes that occur in a Web service may have an impact on other services within an SOA. Hence, SOAs must rapidly manage changes to avoid run-time failures, maintain consistency, and take advantage of evolving business, market, and environment conditions.

The objective if this Thesis is to propose a framework for the automatic detection of change in SOAs. In particular, we introduce two families of change detection protocols: soft-state and notification. Soft-state protocols deal with failure-related changes in SOAs. We develop three soft-state protocols: pure, explicit removal, and reliable. Notification protocols deal with failure-related changes in SOAs. We introduce two notification approaches for SOAs: explicit and implicit. For each approach, we propose three protocols to deal with the issues of double notifications and notification loops: centralized, distributed, and header-based notifications. We implement three prototypes to illustrate the viability of the proposed framework. We also conduct experiment studies to compare and analyze the performances of the proposed protocols.
# Table of Contents

ACKNOWLEDGMENTS ......................................................................................... i

ABSTRACT ........................................................................................................ ii

LIST OF FIGURES .............................................................................................. vii

LIST OF TABLES ................................................................................................ xii

CHAPTERS

1. INTRODUCTION ............................................................................................. 1
   1.1 Motivation ...................................................................................................... 2
   1.2 Contributions ................................................................................................. 2
   1.3 Related work .................................................................................................. 4
      1.3.1 Techniques for Detecting Failure-Related Changes ................................. 4
      1.3.2 Techniques for Detecting Content-Related Changes ................................. 6
   1.4 Organization of the Thesis ............................................................................ 7

2. SERVICE-ORIENTED ARCHITECTURE: CONCEPTS AND
   TECHNOLOGIES ............................................................................................ 8
   2.1 Introduction .................................................................................................. 8
   2.2 What is Service-Oriented Architectures? ....................................................... 9
      2.2.1 Scenario ................................................................................................ 10
      2.2.2 Benefits of using Web Services ............................................................ 11
   2.3 Web Service Reference Model ...................................................................... 12
   2.4 Web Service Technologies ........................................................................... 14
2.4.1  XML ...............................................................14
   2.4.1.1 DTDs .........................................................16
   2.4.1.2 XML Schema ..................................................18
2.4.2  SOAP .............................................................20
2.4.3  WSDL .............................................................22
2.4.4  UDDI .............................................................24
2.5  Web Service Stack ...................................................28
2.6  Web Service Composition ...........................................29
   2.6.1  Why Web Service Compositions? ..................................29
   2.6.2  Orchestration vs. Choreography ....................................30
   2.6.3  Different Approaches for Web Service Composition .............32
      2.6.3.1 Process-based composition ....................................32
      2.6.3.2 Interactive composition .........................................34
      2.6.3.3 Planning-based composition ....................................34
3.  SOFT-STATE PROTOCOLS FOR BOTTOM-UP FAILURE DETECTION
    IN SERVICE-ORIENTED ARCHITECTURES .................................36
   3.1  Introduction ................................................................36
   3.2  The Service-Oriented Soft-State Model ................................37
      3.2.1  Motivating Scenario .................................................37
      3.2.2  Bottom-up Fault Taxonomy for Service-Oriented Application ....38
      3.2.3  State of a Participant Service ......................................40
      3.2.4  Soft-State Senders and Receivers ..................................41
   3.3  Pure Soft-State protocol ..................................................45
   3.4  Soft-State Protocol with Explicit Removal (Removal-SS) .............50
3.5 Reliable Soft-State Protocol (Reliable-SS) ................................................................. 56
3.6 Implementation ........................................................................................................... 65
3.7 Performance Evaluation ............................................................................................. 81

4. NOTIFICATION PROTOCOLS FOR SERVICE-ORIENTED ARCHITECTURE ................................................................. 86

4.1 Introduction ............................................................................................................... 86
4.2 The Notification Model ............................................................................................. 87
   4.2.1 Explicit and Implicit Notification ........................................................................ 87
   4.2.2 The Notification Ontology ................................................................................. 88
   4.2.3 Notifiers ............................................................................................................ 90
4.3 Notification Protocols ............................................................................................... 91
   4.3.1 Naive Notification ............................................................................................. 91
   4.3.2 Centralized Notification .................................................................................... 93
   4.3.3 Distributed Notification .................................................................................... 97
   4.3.4 Header-based Notification ............................................................................... 101
4.4 Implementation ......................................................................................................... 105
   4.4.1 Case Study 1: Disaster Recovery ..................................................................... 105
   4.4.2 Case Study 2: Agriculture-Emerald Ash Borer ............................................... 109
4.5 Experiments .............................................................................................................. 112
   4.5.1 Case Study 1 - Disaster Recovery (DR) ............................................................ 113
      4.5.1.1 Experiment DR 1 ...................................................................................... 113
      4.5.1.2 Experiment DR 2 ...................................................................................... 114
      4.5.1.3 Experiment DR 3 ...................................................................................... 115
      4.5.1.4 Experiment DR 4 ...................................................................................... 117
4.5.1.5 Experiment DR 5 .................................................................119
4.5.1.6 Experiment DR 6 .................................................................121
4.5.1.7 Experiment DR 7 .................................................................123
4.5.1.8 Experiment DR 8 .................................................................124
4.5.1.9 Experiment DR 9 .................................................................126
4.5.1.10 Experiment DR 10 ...............................................................127
4.5.1.11 Summary ...............................................................129

4.5.2 Case Study 2 – Emerald Ash Borer (EAB) .................................132
4.5.2.1 Experiment EAB 1 .................................................................132
4.5.2.2 Experiment EAB 2 .................................................................133
4.5.2.3 Experiment EAB 3 .................................................................134
4.5.2.4 Experiment EAB 4 .................................................................136
4.5.2.5 Experiment EAB 5 .................................................................138
4.5.2.6 Experiment EAB 6 .................................................................140
4.5.2.7 Experiment EAB 7 .................................................................142
4.5.2.8 Experiment EAB 8 .................................................................144
4.5.2.9 Experiment EAB 9 .................................................................146
4.5.2.10 Experiment EAB 10 .............................................................147
4.5.2.11 Summary .................................................................149

5. CONCLUSION ................................................................152

REFERENCES ................................................................154
# List of Figures

2.1 A travel agreement scenario...........................................................................................................10
2.2 Roles of Web Services .....................................................................................................................12
2.3 Steps to invoke Web services ........................................................................................................13
2.4 Web Service Technologies ............................................................................................................14
2.5 A bibliography entry (a) in HTML and (b) in XML ......................................................................15
2.6 Simple XML document..................................................................................................................16
2.7 A short bibliography in XML ..........................................................................................................17
2.8 A DTD for the bibliography example ............................................................................................18
2.9 The XML Schema for the bibliography in figure 2.7 .................................................................19
2.10 Sequence and Choices in (a) a DTD and (b) XML Schema ......................................................20
2.11 General Structure of SOAP Messages .........................................................................................21
2.12 SOAP Request and Respond .......................................................................................................22
2.13 General Structure of WSDL Document .....................................................................................23
2.14 WSDL Document ........................................................................................................................25
2.15 UDDI Data Model ........................................................................................................................26
2.16 UDDI Document ..........................................................................................................................27
2.17 Web Service Stack ........................................................................................................................29
2.18 General approach of e-business interaction ................................................................................31
2.19 Orchestration approach ................................................................................................................31
2.20 Choreography Approach .............................................................................................................32
3.1 Motivating Scenario .......................................................................................................................37
3.2 Fault Taxonomy for Service-Oriented Applications ......................................................................39
3.27 Impact of Pure and Soft State with Explicit Removal on Failure Detection Time........84
3.28 Consistency in Pure Soft State and Reliable Soft State .................................85
4.1 Example of Notification Ontology......................................................................89
4.2 Interactions among Web Services......................................................................90
4.3 Naive Implicit Notification – A Scenario ..........................................................92
4.4 Naive Explicit Notification – A Scenario .........................................................93
4.5 Centralized Implicit Notification – A Scenario ..................................................95
4.6 Centralized Explicit Notification – A Scenario ..................................................96
4.7 Distributed Implicit Notification – A Scenario .................................................98
4.8 Distributed Explicit Notification – A Scenario ..................................................100
4.9 Message Structure in Header-based Notifications ..............................................101
4.10 Header-based Implicit Notification – A Scenario.............................................103
4.11 Header-based Explicit Notification – A Scenario .............................................105
4.12 Disaster Info Taxonomy ....................................................................................106
4.13 Disaster Partner Taxonomy ................................................................................107
4.14 Generating Notifiers in the Disaster Prototype ..................................................108
4.15 Screenshots of Disaster Recovery Prototype ....................................................109
4.16 EAB Info Taxonomy .........................................................................................110
4.17 EAB Partners Taxonomy ...................................................................................111
4.18 Screenshots of EAB Prototype .........................................................................112
4.19 Implicit Notification of Experiment 1 .................................................................113
4.20 Explicit Notification of Experiment 1 .................................................................113
4.21 Implicit Notification of Experiment 2 .................................................................114
4.22 Explicit Notification of Experiment 2 .................................................................115
4.47 Implicit Notification of Experiment 4.................................................................137
4.48 Explicit Notification of Experiment 4...............................................................137
4.49 Implicit Notification of Experiment 5...............................................................139
4.50 Explicit Notification of Experiment 5...............................................................139
4.51 Implicit Notification of Experiment 6...............................................................141
4.52 Explicit Notification of Experiment 6...............................................................142
4.53 Implicit Notification of Experiment 7...............................................................143
4.54 Explicit Notification of Experiment 7...............................................................144
4.55 Implicit Notification of Experiment 8...............................................................145
4.56 Explicit Notification of Experiment 8...............................................................145
4.57 Implicit Notification of Experiment 9...............................................................146
4.58 Explicit Notification of Experiment 9...............................................................147
4.59 Implicit Notification of Experiment 10............................................................148
4.60 Explicit Notification of Experiment 10............................................................149
4.61 Implicit - Notification Time and Number of Notification in CN, DN, and HN ....150
4.62 Explicit - Notification Time and Number of Notification in CN, DN, and HN ....151
List of Tables

3.1 SS-S₁ Algorithm for Pure Soft State ................................................................. 46
3.2 SS-R₁ Algorithm for Pure Soft State ..................................................................... 49
3.3 SS-Sᵢ Algorithm for Soft State with Explicit Removal .......................................... 52
3.4 SS-Rᵢ Algorithm for Soft State with Explicit Removal .......................................... 55
3.5 SS-Sᵢ Algorithm for Reliable Soft State .............................................................. 62
3.6 Algorithm for Reliable Soft State .......................................................................... 64
4.1 Naive Implicit Notification Algorithm Executed by WS₁ ........................................ 91
4.2 Naive Explicit Notification Algorithm Executed by WS₁ ...................................... 92
4.3 Centralized Implicit Notification Algorithm Executed by WS₁ ............................... 94
4.4 Centralized Explicit Notification Algorithm Executed by WS₁ ............................... 96
4.5 Distributed Implicit Notification Algorithm Executed by WS₁ ............................... 98
4.6 Distributed Explicit Notification Algorithm Executed by WS₁ ............................... 99
4.7 Header-based Implicit Notification Algorithm Executed by WS₁ ......................... 102
4.8 Header-based Explicit Notification Algorithm Executed by WS₁ ......................... 104
4.9 The results of Experiment 1 - DR ........................................................................ 114
4.10 The results of Experiment 2 - DR ........................................................................ 115
4.11 The results of Experiment 3 - DR ........................................................................ 117
4.12 The results of Experiment 4 - DR ........................................................................ 119
4.13 The results of Experiment 5 - DR ........................................................................ 121
4.14 The results of Experiment 6 - DR ........................................................................ 123
4.15 The results of Experiment 7 - DR ........................................................................ 124
4.16 The results of Experiment 8 - DR ........................................................................ 126
Chapter 1

Introduction

Service-oriented architectures are slated to shape modern societies in vital areas such as healthcare, government, science, finance, and business [PV, H05, LR99]. They have recently emerged as a promising approach for application integration [SR06, KLS+07, PTD+06, BL06, F05, CS06]. Service-oriented architectures utilize services as the building blocks for developing software systems distributed within and across organizations [HS05, AC05]. The most common realization of service-oriented architectures is based on Web services; these are loosely coupled entities that provide pre-defined capabilities via XML-based standards on the Web [ACK+03, SH05]. Web services may wrap a variety of independently provided resources such as legacy applications, sensors, databases, storage devices, COTS products, and ERP packages [PV, HPD+05, CFF+04, SWM+06]. The primary value of service-orientation is the ability to reuse existing services either as standalone or as part of composite services that perform more complex functions by combining pre-existing services (called participants).

Standards are key enablers of Web services. Major industry players took a lead to set up those standards. This has greatly facilitated the adoption and deployment of Web services. Three key XML-based standards have been defined to support Web services: SOAP, WSDL, and UDDI. SOAP defines a communication protocol for Web services. WSDL (Web Service Description Language) enables service providers to describe the interface of their applications (e.g., operations, messages, and data). UDDI (Universal Description Discovery and Integration) offers a registry service that allows advertisement and discovery of Web services.
1.1 Motivation

Web services in modern application domains (e.g., grid computing [F01, FKN+02, LG03], disaster management [BCL04], supply chains [B03], and e-science [DH04, DJS05]) face unavoidable changes during their lifetime. We identify two categories of changes in Web services: failure-related and content-related changes. Failure-related changes (also called faults) refer to changes that may lead to failures in services (standalone or composite). Examples of failure-related changes include sever/network failures and shutdowns scheduled by providers for maintenance. Content-related changes refer to changes in the (i) data returned by a service (e.g., value of a stock price in a stock market service) or (ii) policies associated to a service (e.g., quality of service, security policy). We adopt a broad definition of policy, encompassing all requirements under which a service is consumed. Some changes may be considered as both failure-related and content-related. For instance, an update in the WSDL interface of service WS (e.g., changes in the data types of input and output messages) is content-related (i.e., update in a service interface policy). This change is also failure-related as a client’s invocation to WS may lead to run-time failures if that client is not aware of the change; the client may send messages with obsolete data types to WS.

Changes that occur in a Web service may have an impact on other services. For instance, the unavailability of a service may lead to run-time failures in composite services that use that service as participant; updates in the “wind speed” data returned by a weather service may trigger the execution of emergency services in a disaster management architecture; changes in a service interface may impact the way other services interact with that service. Hence, Web services must rapidly manage changes to avoid run-time failures, maintain consistency, and take advantage of evolving business, market, and environment conditions.

1.2 Contributions

One key requirement for managing changes is the ability to detect those changes. Our focus in this Thesis is on change detection in service-oriented architectures. In particular, we propose two families of protocols for detecting failure-related and content-related changes, respectively. In these protocols, changes are detected via a peer-to-peer network of administrative services
(fault coordinators for failure-related changes and notifiers for content-related changes). This has four major advantages. First, it concords with recent Web service trends (e.g., Open Grid Services Architecture [FK04]) which define administrative tasks as Web services. Second, it decouples the business logic of Web services from change detection logic. Third, it preserves the autonomy of participant services and distributes change management over Web services. Finally, service providers/composers can join the system by merely downloading the trusted codes of fault coordinators/notifiers, deploying them as Web services, and registering them in UDDI. Our main contributions in this Thesis are summarized below:

- **Soft-State Protocols for Web services**: We propose a family of soft-state protocols to deal with failure-related changes in Web services. Soft state denotes a type of protocols widely used by communication protocols in which nodes maintain state consistency through periodic refresh messages [JVD04, PZJ+07]. State which is not refreshed in time expires. We adopt this approach to Web services and propose three protocols: pure soft-state, soft-state with explicit removal, and reliable soft-state.

- **Notification Protocols for Web services**: We propose a family of notification protocols to deal with content-related changes in Web services. We introduce two notification approaches for Web services: explicit and implicit. In explicit notification (also known as publish-subscribe), information is provided to only those services that explicitly showed their interest in receiving it (via subscription). In implicit notification, information is provided to consumers that may benefit from receiving that information even if they did not explicitly request it via subscription. Implicit notification uses the concept of ontology [BHL01, F03] to model possible notification patterns that exist among Web services. For each approach, we propose three protocols to deal with double notifications and notification loops: centralized notification, distributed notification, and header-based notification.

- **Prototype Implementation and Performance Study**: We develop three (3) prototypes to illustrate the viability of the proposed change detection protocols. The first prototype implements the proposed soft-state protocols. It allows the generation of failure-related changes at the participant level and shows how these changes are detected by the corresponding composite services. The second and third prototypes implement the proposed notification protocols (implicit and explicit) in two application domains:
disaster recovery and agriculture (for controlling invasive species). We also conduct experiment studies to compare and analyze the performances of the proposed protocols.

1.3 Related Work

In this section, we overview major techniques proposed in the literature to deal with the detection of failure-related and content-related changes.

1.3.1 Techniques for Detecting Failure-Related Changes

Current techniques for managing changes in Web services allow developers to specify changes within their service policies (e.g., fault elements in SOAP and WSDL [ACK+03, SH05], exception handling in BPEL [KMW03]). Such techniques are static, ad hoc, time-consuming, and make service design complex [MMP06, VS05]. Current software engineering approaches for fault management have hard-coded, internal, and application-specific capabilities that limit their generalization and reuse [CGS05, GK05, YYX04, QTZ05]. They disperse the adaptation logic throughout the application, making it costly to modify and maintain [GS02, T04, S07]. Recent approaches for self-healing software systems (e.g., [GS02, T04, CGS05, GK05, YYX04, QTZ05, S07]) mostly focus on “traditional” applications not service-oriented. The peculiarity of faults in Web services, specificity of Web service interaction models (no assumption of shared memory), and the autonomy, distribution, and heterogeneity of services make these approaches difficult to apply in Web services.

The concept of soft state has been widely used for building robust and reliable systems. A number of network protocols such as RSVP [ZDE+93], SRM [FJL+97], PIM [DEF+96, DEJ+96, EFH+], SIP [HSS+99] and IGMP [D89] have us this concept. However, to the best of our knowledge, this work is the first to use soft-state in Web services.

[MSS+05] focuses on handling exceptional changes that can be raised inside workflow-driven Web applications is proposed. It first classifies these changes into behavioral (or user-generated), semantic (or application), and system exceptions. The behavior exceptions are driven by improper execution order of process activities. The semantic exceptions are driven by unsuccessful logical outcome of activities execution. The system exceptions are driven by the
malfunctioning of the workflow-based Web application, such as network failures and system breakdowns. The change management framework consists of three major components: capturing model, notifying model, and handling model. The capturing model captures events and stores the exceptions data in the workflow model. The notifying model propagates the occurred exceptions to the users. The handling model defines a set of recovery policy to resolve the exception. Different recovery policies apply to different types of exceptions.

[AMA03] focuses on managing bottom-up changes in Web services. It first presents a taxonomy that classifies bottom-up changes into categories. Changes occur either at the service level or business level: *triggering* changes occur at the service level and *reactive* changes occur at the business level in response to the triggering changes. A set of mapping rules are defined between triggering changes and reactive changes. These rules are used for propagating changes. A Petri-net based change model is proposed as a mechanism for automatically reacting changes. Ontology is used for locating services from an exploratory service space. Agents are employed to assist in detecting and managing changes.

The WS coordination framework (WS–Coordination [ACK+03]) provides a foundation layer for consensus between WS, where specific consensus protocols can be built upon, e.g. distributed transactions. Two particular specifications build upon WS-Coordination: WS-AtomicTransaction for short running ACID transactions and WS-BusinessActivity for long running transactions with weaker guarantees for atomicity and isolation. WS-Reliability [ACK+03] specifies a SOAP-based protocol for reliable message delivery with at-least-once, at-most-once, and exactly-once semantics. The first option guarantees that a message is delivered, the second one that duplicates are eliminated, and the third one combines both requirements. Moreover, an ordering constraint can also be imposed on a group of messages. That is, a sequence of messages sent is delivered in the same order at the receiver (FIFO ordering).

There is a large body of work on failure detectors for different types of distributed systems, from reliable process group communication to large scale grids and from real-time factory floor settings to multi-level failure detection for clusters [CT96, DGM02, FDG+99, FRT01, GCG01, RMY98, RBV03, SFK+98, V96]. However, these techniques are developed for traditional distributed systems not SOAs. WS-Membership is a framework based on WS-Coordination for
monitoring Web services and provision of membership information [WC03]. It uses a gossip-style failure detector not soft-state.

1.3.2 Techniques for Detecting Content-Related Changes

The publish-subscribe paradigm has received considerable attention over recent years [BR04]. It provides a loosely coupled form of interaction that is well suitable for large-scale distributed systems. Publish-subscribe systems are generally categorized as topic-based or content-based [EFG+03]. In topic-based systems, each event belongs to one of a fixed set of subjects (also called topics); publishers are required to label each event with a topic name. In content-based systems, events are no longer divided into different subjects. The subscriber defines a subscription condition according to the internal structure of events; all events that meet the condition will be sent to the subscriber. Our focus in this Thesis is on topic-based notifications. The work presented in this Thesis extends publish-subscribe to SOAs (explicit notification) while dealing with double notifications and notification loops. Besides, it proposes a new type of notifications called implicit notifications.

WS-Notification is a family of specifications that define a standard Web services approach to notification using a topic-based publish/subscribe systems [GNC+04, V04]. It includes a standard message exchanges to be implemented by service providers that wish to participate in Notifications, standard message exchanges for a notification broker service provider (allowing publication of messages from entities that are not themselves service providers), operational requirements expected of service providers and requestors that participate in notifications, and an XML model that describes topics. WS-Notification follows the explicit notification approach. Besides, it does not deal with issues of double notifications and notification loops.

Several techniques have adopted the concept of ontology in publish-subscribe systems. [WJL04] proposes an ontology-based matching algorithm for content-based publish-subscribe systems. [SC06] presents a publish-subscribe system which utilizes ontology to classify and query published data. [WJL+04] uses ontology to model events, topics, and subscriptions in content-based publish-subscribe systems. Our work is different in that it uses ontology to model notification patterns among producers and consumers.
Many frameworks for realizing adaptive workflows have been proposed \cite{VT02, CCB98, RRP04, SOM99, MA99}. They mainly focus on adjusting the process instances to a changed process schema. An important issue addressed by these frameworks is how to define correctness criteria, which evaluate the compliance of process instance with a changed process schema. Once such criteria are determined, mechanisms are proposed to propagate a process type change to an instance.

1.4 Organization of the Thesis

This Thesis is organized as follows. In Chapter 2, we review the major concepts, standards, and technologies for enabling Web services and service-oriented architectures. In Chapter 3, we describe the proposed soft-state protocols for detecting failure-related changes in service-oriented architectures; we also present our prototype implementation and discuss experiment results to assess the performance of the proposed protocols. In Chapter 4, we describe the proposed protocols for explicit and implicit notifications in service-oriented architectures; we also overview two prototype implementations for notification protocols (disaster recovery and controlling invasive species in agriculture) and analyze experiment results. In Chapter 5, we summarize our contributions and outline future work.
Chapter 2

Service-Oriented Architectures: Concepts and Technologies

2.1 Introduction

The Web is a distributed, dynamic, and large information repository. It has now evolved to encompass various information resources accessible worldwide. Organizations across all spectra have already moved their main operations to the Web, which has brought about a fast growth of various Web applications. This has dramatically increased the need to build a fundamental infrastructure for efficient deployment and access of the exponentially growing plethora of Web applications. The development of enabling technologies for such an infrastructure is expected to change the business paradigm on the Web. Web services have de facto become the most significant technological by-product. Simply put, a Web service is a piece of software application whose interface and binding can be defined, described, and discovered as XML artifacts [ACK+03]. It supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols. Examples of Web services include online reservation, ticket purchase, stock trading, and auction.

In this chapter, we describe the concepts, technologies, and architectures for enabling Web services. In Section 2.2, we define service-oriented architecture and Web services. In Section 2.3, we describe the Web service reference model. In Section 2.4, we overview the major Web
service technologies. In Section 2.5, we describe the Web service stack. Section 2.6 is devoted to Web service composition.

2.2 What is Service Oriented Architecture?

A service-oriented architecture is a collection of services that communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity. Some means of connecting services to each other is needed. If a service-oriented architecture is to be effective, we need a clear understanding of the term service. A service is a function that is well-defined, self-contained, and does not depend on the context or state of other services. The technology of Web services is the most likely connection technology of service-oriented architectures.

Many different of definitions are adopted about Web services. For example, the W3C consortium defines a Web service as “a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards” [WSA03]. IBM defines Web services as “self-describing, self-contained, modular applications that can be mixed and matched with other Web services to create innovative products, processes, and value chains. Web services are Internet applications that fulfill a specific task or a set of tasks that work with many other Web services in a manner to carry out their part of a complex workflow or a business transaction”. According to Microsoft, “A Web Service is a unit of application logic providing data and services to other applications. Applications access Web Services via ubiquitous Web protocols and data formats, such as HTTP, XML, and SOAP, with no need to worry about how each Web Service is implemented”. HP defines Web services as “modular and reusable software components that are created by wrapping a business application inside a Web service interface. Web services communicate directly with other Web services via Standards-based technologies”. SUN perceives a Web service as an “application functionality made available on the World Wide Web. A Web service consists of a network-accessible service, plus a formal description of how to connect to and use the service”.

9
2.2.1 Scenario

We consider a travel agency, named TravelAgency, providing the travel arrangement (e.g., transportation, itinerary, and accommodations) for its clients (Figure 2.1).

Assume a university professor, Joan, wants to spend her vacation in Sydney, Australia. Typical services needed by Joan might include airlines, ground transportation (e.g., taxi and car rental), accommodation (e.g., hotels and inns), and other entertainment services (e.g., restaurant and opera house). Joan needs to find travel agency that provides travel packages. We assume that the TravelAgency is located by Joan. Joan would then call the TravelAgency. To offer a complete tour package, the TravelAgency needs support from its business partners (e.g., Air Company, Hotel, Restaurant, and Car Rental) to arrange flights, hotels, cars, and other entertainment facilities. These companies all define the service description for their Web services and publish them on a well-known service registry, whereby the TravelAgency can search and locate them. The TravelAgency needs to outsource services from these business partners to provide the entire travel package. Since Web services with similar functionalities might be provided by competing business partners, there is a need to optimize access to those Web services or composition thereof with respect to the expected quality. In addition, the Web services need to be accessed in a reliable and secure manner. This example
articulates a typical Web service usage scenario. It will serve as a running example to illustrate the various Web service concepts.

2.2.2 Benefits of using Web Services

1. Endorses interoperability via reducing the necessities for common understanding. XML-based interface definition language, an XML-based service description (WDS) and a protocol of collaboration and negotiation are the solitary necessities for shared understanding involving a service provider and a service requester.

2. Allows just-in-time affiliation. Associations in Web services are constrained at prolongation. A service requester explains the competence of the service that is essential as well as makes use of the WSDL to get a suitable service. Once a service with the mandatory potential is initiated, the information from the service's NASSL document is used to attach to it. Dynamic service discovery and invocation (publish, find, bind) and message-oriented collaboration acquiesce applications with looser pairing, permitting just-in-time incorporation of up-to-the-minute applications and services. This consecutively yields systems that are self-configuring, adaptive and forceful with less distinct points of malfunction.

3. Decreases complication by summing up. Each and every part in Web Services is a service. The type of behavior a service provides is what is vital, not how the service is put into action. The method of describing the behavior encapsulated by a service is known as the WESL document.

4. Allows interoperability of inheritance requests. Web services architecture effortlessly allows new interoperability between these applications by enabling legacy applications to be wrappered in WDSL documents as well as exposed as services. In addition, safety measures, middleware and communications skills can be wrappered to contribute in a Web service as ecological fundamentals. Illustrations of this can be found in amalgamate circumstances, where the consequential project should incorporate dissimilar IT systems and industry procedures. A service-oriented structural design would significantly smooth the progress of a flawless assimilation between these systems. Another case can be found
in the combination of the travel business with persistent computing, whilst principally mainframe-based travel applications can be shown as services during wrappering and made vacant for use by an assortment of devices in a service-oriented environment. New services can be created and dynamically published and discovered without disturbing the existing environment.

2.3 Web Service Reference Model

Participants in a Web services model are categorized into three types (Figure 2.2):

- **Service provider** is the owner of the Web services. It holds the implementation of the service application and makes it accessible via the Web.
- **Service client** represents a human or a software agent that intends to make use of some services to achieve a certain goal.
- **Service registry** is a searchable registry providing service descriptions. It implements a set of mechanisms to facilitate service providers to publish their service descriptions. Meanwhile, it also enables service clients to locate services and get the binding information.

![Figure 2.2. Roles of Web Services](image)

Interactions with a Web service take place in three modes:
- Service publication is to make the service description available in the registry so that the service client can find it.
- Service lookup is to query the registry for a certain type of service and then retrieve the service description.
- Service binding is to locate, contact, and invoke the service based on the binding information in the service description. The steps for binding a service are given below (Figure 2.3):

1- The provider needs to define the service in WSDL.
2- The provider registers the service in UDDI.
3- The requester is interested to find a certain service, he can benefit from UDDI to get all the Web service that’s published by the providers. The UDDI links the requester to UDDI document that has all the locations of the Web services.
4- The requester searches in the UDDI document for the desired service, and then invokes the service to get the description of the service.
5- The provider links the requester to the WSDL file for that service.
6- After looking at the WSDL, if the requester is interested in that service, then the requester invokes that service.
7- The provider then responds by sending the result to the requester.

---

**Figure 2.3.** Steps to invoke Web service
2.4  Web Service Technologies

The basic Web service technologies include XML, SOAP, WSDL, and UDDI (Figure 2.4). In the rest of this section, we overview each of this technologies.

![Figure 2.4. Web Service Technologies](image)

2.4.1  XML

XML is a markup language for documents containing structured information. The XML specification defines a standard way to add markup to documents. XML specifies neither semantics nor a tag set like in HTML. In fact XML is really a meta-language for describing markup languages. In other words, XML provides a facility to define tags and the structural relationships between them. Since there's no predefined tag set, there can't be any preconceived semantics. All of the semantics of an XML document will either be defined by the applications that process them or by style sheets.

It is important to understand why XML was created. XML was created so that richly structured documents could be used over the Web and XML is being designed to deliver structured content over the Web. The only viable alternatives, HTML, are not practical for this purpose. In other words XML was designed to describe data and its structure, and to focus on what data is, while HTML was designed to display data and to focus on how data looks (Figure 2.5).
Because of the features of XML, Web services used XML scheme to communicate with the participants over internet. XML Web services are the fundamental building blocks in the move to distributed computing on the Internet. Open standards and the focus on communication and collaboration among people and applications have created an environment where XML Web services are becoming the platform for application integration. Applications are constructed using multiple XML Web services from various sources that work together regardless of where they reside or how they were implemented. In Web services, XML is being used to publish data from database systems to the Web by providing input to content generators for Web pages, and database systems are increasingly used to store and query XML data, often by handling queries issued over the Internet. Figure 2.6 illustrate simple XML document, it has one root and it encapsulates the book element and other sub element (author, firstname, lastname, etc)
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- This is an XML and HTML comment -->
<book>
    <author>
        <lastname> Gray </lastname>
        <firstname> Jim </firstname>
    </author>
    <title> Principles of Transaction Processing </title>
    <chapter number="1">
        <chaptitle> Introduction </chaptitle>
    </chapter>
</book>

Figure 2.6. Simple XML document

2.4.1.1 DTDs

DTDs, which are also used in SGML, define the structure of XML document. It’s easiest to think of a DTD as a context-free grammar. In particular, DTDs let users specify the set of tags, the order of tags, and the attributes associated with each. A well-formed XML document that conforms to its DTD is called valid. Figure 2.8 shows a simple DTD for the bibliography example in figure 2.7. A DTD is declared in the XML document’s prolog using the !DOCTYPE tag. The DTD can be included within the XML document, or it can be contained in a separate file. If the DTD is in a separate file, say document.dtd, the XML document includes the statement:

    <!DOCTYPE Document SYSTEM “document.dtd”>

We can also refer to an external DTD through a URI.

*DTD elements* can be either nonterminal or terminal. Nonterminal elements (BIB and BOOK in figure 2.8) contain sub elements, which can be grouped as sequence or choice. A sequence defines the order in which sub elements must appear. A choice gives a list of alternatives for sub elements. Sequence and choice can contain each other. In figure 2.8, both BIB and BOOK are sequences. A BOOK element has to have least one AUTHOR (indicated by +), followed by a TITLE and, optionally, by a PUBLISHER and a YEAR (indicate by ?). A choice is indicated by
the logical operator |. The example below shows a choice in which a SECTION can either be a TITLE followed by at least one PARAGRAPHS, or a TITLE followed by zero or more PARAGRAPHS (indicate by the wildcard *) and at least one SUBSECTION.

```xml
<!ELEMENT SECTION ((TITLE, (PARAGRAPH+)) | (TITLE, (PARAGRAPH*), (SUBSECTION+)))>
```

Terminal elements can be declared as parsed character data (#PCDATA, as in figure 2.8) or as EMPTY. Elements can also be declared as ANY. An element declared as ANY is a terminal element is the grammar, but it can contain sub elements of any declared type, as well as character data.

```xml
<?xml version="1.0" standard="yes">
<!- This is an example bibliography. ->
<BIB>
   <BOOK nickname="Dragon book">
   ...      <TITLE>Principles of Database and knowledge-Base System, Vol.
                    1</TITLE>
   </BOOK>
…
</BIB>
```

**Figure 2.7.** A short bibliography in XML

Elements can have zero or more attributes, which are declared using the !ATTLIST tag. Unlike element definition, attribute definitions do not impose order on when the attributes occurs. Furthermore, if several attributes are declared for the same element type, they can be declared using multiple !ATTLIST tags. Attributes can be optional (#IMPLIED), required (#REQUIRED), or fixed (#FIXED). Optional attributes can, and fixed attributes must, have a default value:

```xml
<!ATTLIST PROJECT url CDATA "http://www.myserver.net/index.html">
```
Attribute can have different data types. Character data (CDATA) is the most common. With type id, idref, and idrefs, element identifiers can be declared and referenced. Every value of referencing attribute must be a value of another attribute of type id. Attributes can also be enumerated as user-define types:

```xml
<!ATTLIST PUBLICATION format (html | pdf | ps) #REQUIRED>
```

Popular application of DTDs include XHTML (www.w3.org/TR/xhtml/) and the chemical Markup Language (CML, http://www.xmlcml.org/)

```xml
<!DOCTYPE bib [
   <!ELEMENT BIB (BOOK+)>
   <!ELEMENT BOOK (AUTHOR+, TITLE, PUBLISHER?, YEAR?)>
   <!ATTLIST BOOK isbn CDATA #IMPLIED
      nicknam CDATA #IMPLIED>
   <!ELEMENT AUTHOR (#PCDATA)>
   <!ATTLIST AUTHOR id ID #IMPLIED
      idref IDREF #IMPLIED>
   <!ELEMENT TITLE (#PCDATA)>
   <!ELEMENT PUBLISHER (#PCDATA)>
   <!ELEMENT YEAR (#PCDATA)>
]>`

**Figure 2.8.** A DTD for the bibliography example

### 2.4.1.2 XML Schema

Although DTDs were the first proposal to provide for a standardized data exchange between users, they have disadvantages. Their expressive power seems limited, and their syntax is not XML. Several approaches address these disadvantages by defining a schema language (rather than a grammar) for XML document:

- Document definition markup language (DDML), formerly known as XSchema
- Document content description (DCD)
- Schema for object-oriented XML (SOX), and
- XML-Data (replace by DCD)

The W3C’s XML Schema activity takes these four proposal into consideration.
XML Schema is well-formed XML that allows the user to define datatype for BOOK elements describes by a DTD in figure 2.8. Again, XML Schema uses its own namespace. The ComplexType element indicates a complex data-type associated with the non terminal element BOOK and consisting of other elements and attributes (Figure 2.9).

XML Schema supports a variety of atomic datatypes, such as string, decimal, and date. Datatypes that have been defined to retain compatibility with DTDs, such as id and idref, should only be used for attributes. The user can simulate the +, *, and ? of DTDs or regular expression using the minOccurs and maxOccurs attributes. figure 2.10 shows the definition of a SECTION element written (a) as a DTD and (b) in XML Schema. XML Schema also distinguishes between sequences and choices, which are instances of the group element. Groups can be defined outside of a type and referenced using the ref attribute.

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   <xsd:element name="BOOK" type="BOOKTYPE">
      <xsd:complexType name="BOOK_TYPE">
         <xsd:element name="AUTHOR" type="xsd:string">
            minOccurs="1" maxOccurs="unbounded"/>
         <xsd:element name="TITLE" type="xsd:string"/>
         <xsd:element name="PUBLISHER" type="xsd:string"/>
         <xsd:element name="YEAR" type="xsd:decimal"/>
         <xsd:attribute name="isbn" type="xsd:string"/>
         <xsd:attribute name="nickname" type="xsd:string"/>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

**Figure 2.9.** The XML Schema for the bibliography in figure 2.7

XML Schema also supports inheritance as part of the more general concept of derivation. A type definition can be accompanied by a base (the name of the base type) and a derivedBy (the kind of derivation) attribute. Possible values for the derivedBy attribute include extension and restriction. As a further extension, a list element lets you define a list of elements of some base type. The ultimate goal of XML Schema can be only to replace DTDs. Currently; its chances of doing so remain unclear. For more information about XML Schema, see [http://www.w3.org.XML/Schema.html](http://www.w3.org.XML/Schema.html)
Figure 2.10. Sequence and Choices in (a) a DTD and (b) XML Schema

2.4.2 SOAP

SOAP [SOAP03] is a WS messaging standard that enables communication among Web services. It provides a lightweight messaging framework for exchanging XML-based messages. SOAP is independent of languages and platforms. It supports two types of communications: messaging and RPCs. It is designed to work with a great variety of transport protocols. The predominant protocol that is combined with SOAP is HTTP, which helps achieve the synchronous communication. A SOAP message contains one XML element (Envelope) and two child elements (Header and Body) (figure 2.11). The Envelope defines the namespaces for the remaining content of a SOAP message. The Header is an optional element. It can carry auxiliary information in a SOAP message for supporting authentication, transaction, and
payment. The **Header** enables a SOAP message to be extended in an application-specific manner. The **Body** is the mandatory part of a SOAP message. It specifies the information to be carried from the initial message sender to the ultimate message receiver. For message exchanging, the **Body** part encompasses the exchanged information, while for RPCs, **Body** includes the remote method name, its associated address, and the arguments thereof. Figure 2.12 depicts SOAP request and respond. In SOAP request, Envelope has tow elements, Header and Body, it request for adding tow number. In SOAP respond, it also has Header and Body inside the envelope, it will respond by add the tow numbers that passed to the service and it will return the result of adding the numbers.

![Figure 2.11. General Structure of SOAP Messages](image)

Since composite Web services have many independent Web services, SOAP Asynchronous can be used in B2B interaction. SMTP is one way to achieve this technique. A synchronous approach blocks a process until the operation completes, it means the next line of execution will be executed when the operation is completed from the last line of code. In Asynchronous approach, when a Web service is invoked, it did not wait to get the response from it, another Web service can be invoked while the first Web service is in invoking. This technique solved a
problem of waiting long time to get the response of invoking many Web services such as the previous example of travel agency.

![SOAP Request](image)

![SOAP Respond](image)

**Figure 2.12.** SOAP Request and Respond

### 2.4.3 WSDL

WSDL [WSDL03] is the current industry standard for WS description. It goes beyond IDLs in the conventional middleware in two major ways. First, WSDL specifies the mechanisms to access a Web service in addition to the service name and signatures. Second, WSDL defines the location to invoke a Web service, whereby the service requestor can locate the service and interact with it using SOAP. WSDL mainly focuses on the syntactic description of Web services using XML. It does not allow the specification of Web service semantic features. For example, no constructs are defined to describe document types (e.g., whether an operation is a request for quotation or a purchase order). A WSDL document describes programming interfaces and
accessing formats of a Web service. It defines a Web service at two levels: the application level and the concrete level. WSDL helps service requestors access a Web service through a clear separation of the abstract and concrete descriptions of a Web service.

**Figure 2.13. General Structure of WSDL Document**

At the abstract level, the WSDL description includes three basic elements: Type, Message, and PortType (figure 2.13). Types define the type of the data that are relevant for the information exchange. Although WSDL can support any type system, it adopts XML Schema Definition (XSD) as the canonical type system to achieve maximum interoperability and platform neutrality. Message represents an abstract definition of the transferred data. A message consists of one or more logical parts. Each part associates with a type of XSD or other type systems. These logic parts provide a flexible way to describe the abstract content of messages. PortType is a set of abstract operations provided by an endpoint of a Web service. An endpoint of a service can support four transmission primitives: one-way, request–response, solicit–response, and notification. At the concrete level, the WSDL description provides information of binding a concrete service endpoint. It specifies three aspects of binding information: communication protocols, data format specifications, and network addresses.
Binding describes the first two aspects. It specifies the communication protocol, such as SOAP and HTTP. It also specifies the data format of the operations and messages. Port and Service elements describe the network addresses. A Port specifies a single address for binding a service endpoint. A Service, on the other hand, aggregates a set of related ports. The WSDL documents describe primary information for accessing Web services. Figure 2.14 shows a general WSDL document, it has five main elements type, message, part, portType, binding, and service.

2.4.4 UDDI

UDDI [UDDI03] defines a standard to publish and discover Web services. The core component of UDDI is a service registry. The registry stores the general information of Web services. The information can facilitate service requestors in querying different Web services. UDDI provides a service query API to locate appropriate Web services. It also defines a service publication API for service providers, who can use it to advertise their services. UDDI encodes three types of information about services: white pages, yellow pages, and green pages. The white pages contain the contact information of business entities and the Web services they provide. UDDI uses two elements to express white pages information, including businessEntity and businessService (figure 2.15). The businessEntity describes a business organization that provides Web services. It specifies the information of service providers, including names, brief description, and contact details. Each businessEntity may include one or more businessService elements. The businessService describes a collection of related Web services offered by an organization specified by a businessEntity. The yellow pages contain the classification information of businesses and services. Different business or services are classified based on their functionalities. Examples of such categories include restaurants, hotel, etc. The yellow pages specify the category to which the businesses or services belong. The green pages contain the information about technical details of Web services, including bindingTemplate and tModel. The bindingTemplate describes the information about concretely binding an endpoint of a service. The tModel describes the technical specification of a Web service, such as a Web service type, a protocol used by Web services, or a category system.
Figure 2.14. WSDL Document
UDDI provides well-defined programming interfaces for users to inquire and update the service registry data. The inquiry API and publication API are both based on SOAP messages. The inquiry API enables service requesters to find and get details about Web services. The publication API enables service providers to publish services, update services information, and update the security mechanism. The major usage mode of current Web service discovery technologies is design time discovery. Users query the service registry, retrieve the service description, and write the client code to invoke the service. To accessing Web services at large, it requires WS discovery to support dynamic binding. However, a significant challenge for dynamic binding is to unambiguously describe Web service using machine interpretable languages. In addition, issues such as fault-tolerance and load-balancing also need to be addressed. Figure 2.16 shows an XML of UDDI document about the information of category bag, business entity, binding template, and tModel of a specific Web service.

**Figure 2.15.** UDDI Data Model
Figure 2.16. UDDI Document
2.5 Web Service Stack

The Web service stack contains five key layers: communications, messaging, descriptions, discovery, and processes, which are shown along the vertical direction in figure 2.17. It is an extension of the W3C service stack [WSA03]. Similar to the W3C service stack, each stack layer provides certain functionality to support interoperation between Web services and service clients or among Web services. However, we categorize interoperability into two dimensions: syntactic and semantic. Therefore, our service stack is distinguished from the W3C stack by further identifying the syntactic and semantic interoperability offered by all layers above the messaging layer.

- **Communications**: The underpinning of the Web services stack is the network, where the underlying communications take place. A set of network protocols helps realize the network accessibility of Web services. The wide adoption of HTTP makes it the first choice of standard network protocol for Internet available Web services. Other network protocols could also be supported, such as SMTP.

- **Messaging**: The messaging layer provides a document-based messaging model for interaction with Web services. The messaging model works with a wide variety of network protocols. For example, the messaging model can be combined with HTTP to traverse firewalls. In another case, combination with SMTP enables the interaction with Web services that support asynchronous message exchanges.

- **Description**: The description (or representation) layer is for describing Web services. It wraps Web services and specifies their functionalities, operations, data types, and binding information using a service interface. The WS discovery will rely on the WS representation to locate appropriate Web services.

- **Discovery**: The discovery layer is for locating and publishing Web services. It enables the usage of Web services in a much wider scale. The service providers can store the service descriptions in a service registry via the publication functionalities provided by the WS discovery. Meanwhile, service requestors can query the service registry and look for interested services based on the stored service descriptions.

- **Processes**: The processes layer supports more complex interactions between Web services, which enable Web service interoperation. It relies on the basic interaction functionalities provided by the technologies at lower layers in the Web service stack. For example, it needs
Web service discovery and representation for querying and locating Web services based on their descriptions. The selected Web services are used to construct the process, which consists of a sequence of coordinated Web services.

![Web Service Stack](image)

**Figure 2.17.** Web Service Stack

### 2.6 Web Service Composition

Composite Web services, also called Service-Oriented Applications (SOAs), are application defined by combining outsourced Web services to offer value-added services. We use the terms SOA and composite services interchangeably. The process of defining SOAs is called Web service composition. Web service composition is different from traditional application integration, where applications are tightly coupled and physically combined. Web services adopt a document-based messaging model, which supports the integration of loosely coupled applications that are across multiple organizations.

#### 2.6.1 Why Web Service Composition?

*Web service composition* refers to the process of combining several Web services to provide a *value-added service* [CGS01, TAA+01]. It is emerging as the technology of choice for building cross-organizational applications on the Web [ACK+03, MBB+03]. This is mainly motivated by
three factors. First, the adoption of XML-based messaging over well-established and ubiquitous protocols (e.g., HTTP) enables communication among disparate systems. Indeed, major existing environments are able to communicate via HTTP and parse XML documents. Second, the use of a document-based messaging model in Web services caters for loosely coupled relationships among organizations' applications. This is in contrast with other technologies (e.g., software components [S02]) which generally use object-based communication, thereby yielding systems where the coupling between applications is tight. Third, tomorrow's Web is expected to be highly populated by Web services [CS01]. Almost every “asset" would be turned into a Web service to drive new revenue streams and create new efficiencies. We identify two types of Web services: simple and composite. Simple services are Internet-based applications that do not rely on other Web services to fulfill consumers’ requests. A composite service is defined as a conglomerate of outsourced Web services (called participant services) working in tandem to offer a value-added service. Tax Preparator is an example of composite service used by citizens to file their taxes. It combines simple Web services such as financial services at citizens' companies to get W2 information, banks' and investment companies' services to retrieve investment information, and electronic tax filing services provided by state and federal revenue agencies. From a business perspective, Web service composition offers several advantages [TSG01]. First, composite services allow organizations to minimize the amount of work required to develop applications, ensuring a rapid time-to-market. Second, application development based on Web services reduces business risks since reusing existing services avoids the introduction of new errors. Third, composing Web services enables the reduction of skills and effort requirements for developing applications. Finally, the possibility of outsourcing the “best-in-their-class” services allows companies to increase their revenues.

2.6.2 Orchestration vs. Choreography

Web services have ability to enable interactions between organizations. There are two ways to make such interaction happen. Figure 2.18 illustrates the general idea of how organizations communicate between each others.
Orchestration approach refers to process of a specific Web service, how it work, including the execution order of interaction under control of a single endpoint. It shows BPEL workflow steps of a business in execution time. Figure 2.19 depicts the execution order of a client request, ERP requests for a purchase order, ERP will receive the purchase order acknowledge, and then ERP will receive invoice from the provider. These execution steps are made by a client.

On the other hand, Choreography shows the execution order in the involving Web services, how many of Web services are interacting between each others. Choreography keeps tacking of the sequence of messages that involves many Web services. Choreography is more collaborative
than Orchestration approach. Figure 2.20 shows how two Web services are communicating to each other; it shows how client and the provider integrate their business processes.

![Figure 2.20. Choreography Approach](image)

### 2.6.3 Different Approaches for Web Service Composition

Web service composition can be conducted in three different fashions: process/programming, interaction, and planning.

#### 2.6.3.1 Process-based composition

Most existing Web service composition techniques require programming to some extent for constructing the orchestration model [C02, BSD03, CIJ+00]. Composers first need to study the component services that are described using WSDL or some ontology languages and understand the functionalities of the services and the supported operations. A further step analysis requires identifying the way operations are interconnected, services are invoked, and messages are mapped to one another. The process-based composition scheme makes the process of composing service demanding for composers. Composers need to be domain experts who are familiar with the service description language, the service orchestration algebra, and the corresponding
programming skills. Since common users cannot act as a service composer, the programming based scheme hinders common users from composing Web services at large.

BPEL4WS is a process-based composition approach. It combines the key features of XLANG and WSFL [P03]. BEPL4WS models the behavior of a business process based on the interactions with the involved business partners. It establishes grammar guidelines to describe the business process based on XML, including its control logic and message format. It uses WSDL to model the services in the process flow. It also depends on WSDL to describe the external services that are needed by the process. A major design goal of BPEL4WS is to separate the public aspects of business process behavior from the internal ones. The separation helps businesses conceal their internal decisions from their business partners. Moreover, internal changes of the process implementation no longer affect the public business protocol. Therefore, BPEL4WS has both abstract and executable business processes to support the separation. An abstract process also refers to the business protocol. It specifies the public aspects in the business interactions. Specially, an abstract process only deals with the exchanges of public messages between business partners. It is isolated from the execution of a process flow. Therefore, an abstract process will not release any internal details of the process to its partners. An execution process contains the logic and state of a process. It specifies the sequence of the Web Service interactions conducted in the business process of each business partner. A business process consists of several steps called activities. BPEL4WS supports two types of activities, including basic activities and structured activities. Basic activities manage the interactions of the process with its external resources. Its major task is to receive, reply, and invoke Web services. Structured activities control the overall process flow. They specify the sequence of Web services in the flow. BPEL4WS also defines a set of control constructs to enable the sequencing mechanism, which is similar to XLANG. BPEL4WS provides a set of mechanisms to support transactions and handle exceptions. It uses a scope element to aggregate several activities in a transaction. When an error occurs, a compensation procedure will be invoked, which is also similar to XLANG.
2.6.3.2 Interactive composition

The interactive composition scheme blurs the distinction between composers and common users. Composers are required to have a clear goal and know the tasks that need to be performed to accomplish the composition. Common users can be guided through a set of steps to finish a composer’s task. The composition scheme will work interactively with the common users to help them achieve the orchestration model. The orchestration process can start from users’ goals and work backward by chaining all related services. It can also start from some initial states and achieve the users’ goals by adding services in the forward direction. At each step, the scheme will choose a new service based on the task specified by the users. The interactive scheme can also capture the constraints and preferences during the interaction process. The constraints and preferences can serve as additional criteria to select services for the composition.

An interactive composition approach is proposed in [SPH04]. It adopts the OWL ontologies to model the component services. The service model specifies the input, output, precondition, and effects (IOPE) of services. The proposed approach also implements a tool for automatic translation from WSDL to OWL-S, which enables the support of WSDL-based component services. The data types are defined by XML-schema and message exchange between component services relies on the data flow approach. The composed services are specified using OWL-S. The interactive composition can be performed by chaining component services in either the forward or the backward directions. At each step, the composition scheme adds a new service based on the users’ selection. Existing component service in the orchestration model can serve as a criterion to filter candidate services. Only the services that match the IOPE properties of existing services can be selected by the system and presented to the users.

2.6.3.3 Planning-based composition

The planning-based composition scheme aims to relieve users from the composition processes as much as possible. It relies on AI planning techniques for automatic service composition. In this context, users are allowed to submit a declarative query specifying the goal he/she wants the composite service to achieve together with some of the constraints and preferences that need to be satisfied. Based on the user’s query, the composition scheme can derive a corresponding orchestration model with all constraints and preferences satisfied. The planning scheme regards
services as actions that are applicable in states. State transitions are specified using the preconditions of some actions. A transition will lead to some new states, in which the effects of some actions are valid. Based on this, the composition scheme recursively adds new services until users’ goals have been achieved. The states of existing service in the orchestration will determine the selection of the new services. For example, the preconditions of the new services should be satisfied via the effect of some existing services.

A representative planning-based composition approach is presented in [MSZ01]. It is based on situation calculus to compose Web services. More specifically, it adopts Golog, which is a logic programming language, and makes some extensions to adapt it to Web services. The situation calculus enables software agents to reason about Web services. Web services are modeled as actions, which is similar to the classical AI planning problem. Web services are associated with some preconditions and generate some effect under these preconditions. Simple Web services are categorized into two groups. The Web services in the first group perform information collection actions. The Web services in the second group perform world-altering actions. Composite Web services perform complex actions by composing simple Web services from both groups. Users can specify their requests and constraints, which can be transformed using situation calculus. Users’ constraints can be used to customize the predefined generic composition templates. This helps generate the specific composition plans that fulfill users’ requirements.
Chapter 3

Soft-State Protocols for Bottom-up Failure Detection in Service-Oriented Architectures

3.1 Introduction

In this Chapter, we propose a family of protocols, called soft-state protocols, to detect failure-related changes. Failure-related changes (also called faults) refer to changes that may lead to failures in Web services (standalone or composite). Examples of failure-related changes include sever/network failures and shutdowns scheduled by providers for maintenance.

Soft state denotes a type of protocols widely used by communication protocols in which nodes maintain state consistency through periodic refresh messages. State which is not refreshed in time expires. Advantages of soft-state include implicit error recovery. Hard-state signaling takes the converse approach to soft state - installed state remains installed unless explicitly removed by the receipt of a state-teardown message from the state-installer.

The rest of this chapter is organized as follows. In Section 3.2, we introduce the soft-state model for service-oriented architectures. In Section 3.3, we describe the pure soft-state protocol. In Section 3.4, we propose the soft-state protocol with explicit removal. In Section 3.5, we present the reliable soft-state protocol. In Section 3.6, we describe the prototype implementation. Section 3.7 is devoted to experiments and performance study.
3.2 **The Service-Oriented Soft-State Model**

In this section, we present the proposed model for failure detection in SOAs. We first describe a motivating scenario. Next, we introduce a taxonomy for bottom-up faults in SOAs. Finally, we define the notions of state, soft-state senders, and soft-state receivers.

3.2.1 **Motivating Scenario**

Figure 3.1 depicts two service-oriented applications SOA\(_1\) and SOA\(_2\) that translate words from English to Spanish and German to Italian, respectively. We assume that SOA\(_1\) and SOA\(_2\) outsource from external Web services to perform their translations. Participants and ascendants relationships are defined as follows:

- Participants(SOA\(_1\)) = \{WS\(_1\), WS\(_2\), WS\(_3\)\}; Participants(SOA\(_2\)) = \{WS\(_3\), WS\(_4\)\}.
- Ascendants(WS\(_1\)) = \{SOA\(_1\)\}; Ascendants(WS\(_2\)) = \{SOA\(_1\)\}; Ascendants(WS\(_4\)) = \{SOA\(_2\)\}; Ascendants(WS\(_3\)) = \{SOA\(_1\), SOA\(_2\)\}.
- At the reception of an English word (step 1), SOA\(_1\) invokes WS\(_1\) to get its French translation (step 2). Then, it invokes WS\(_2\) to get the German translation \(y\) of the French word (step 3). Finally, it invokes WS\(_3\) to get the final result (step 4) and returns it to the

![Figure 3.1. Motivating Scenario](image-url)
user (step 5). SOA$_2$ receives a German word from the user (step a). It first invokes WS$_3$ to get the Spanish translation (step b). Then, it invokes WS$_4$ to get the Italian translation (step c). Finally, it returns the result to the user (step d).

Let us now consider the following two scenarios. First, we assume a server failure in WS$_1$. In this case, SOA$_1$ invocation to WS$_1$ may result in a run-time failure. Second, we assume that WS$_3$ provider changed the SOAP binding (in the WSDL specification) for WS$_3$ from SOAP/HTTP to SOAP/SMTP. In this case, SOA$_1$ and SOA$_2$ invocations to WS$_3$ may lead to run-time failures. This calls for techniques that allow SOA to detect changes in their participants. We will introduce such techniques in the rest of this chapter.

### 3.2.2 Bottom-up Fault Taxonomy for Service-Oriented Applications

A fault detection approach must refer to a taxonomy that describes the different types of faults that SOAs are expected to be able to detect. Figure 3.2 depicts the proposed taxonomy for bottom-up faults in SOAs. A bottom-up fault is defined as an abnormal condition or defect at a participant that may lead to a failure in its ascendant SOAs. We identify two types of bottom-up faults: physical and logical. **Physical faults** are related to the infrastructure that supports Web service. They are categorized as node and transmission faults. **Node faults** occur if the servers (e.g., application server, Web server) hosting a participant are out of action. **Transmission faults** occur if the network connection to a participant is down. **Logical faults** are initiated by service providers; this is in contrast to physical faults which are out of service providers’ control. They include status change, participation refusal, and policy change. A **status change** occurs if the service provider explicitly modifies the status of its service from available to temporarily/permanently unavailable. A service status may be changed through a freeze or stop fault. In the **freeze** fault, the provider shuts down its services for a limited time period (e.g., for maintenance or upgrade). In the stop fault, the provider makes its service permanently unavailable (i.e., the provider is no longer offering the service).

Before an SOA outsources from a Web service, it first needs to get a participation approval from the service. A **participation refusal** fault occurs if a service is not willing to participate in an SOA. The way participation decision is made varies from a Web service to another. For instance, a service may assess the reputation of the SOA. It may also check that its workload
A policy change fault occurs if the provider updates one of its service policies. We adopt a broad definition of policy, encompassing all requirements under which a service may be consumed. Policies are specified in XML-based Web service languages/standards (e.g., WSDL, WS-Security). A change in a participant’s policy may impact the way ascendant SOAs interact with that participant. Hence, they should be considered as a logical fault. For instance, let us assume that WS1 provider updated the signature of its input message (e.g., a new input parameter is required). SOA1 invocation to WS1 may result in a failure if SOA1 does not adapt its application to this change.

A policy change is defined by two properties: category of the policy and scope of the change. These two properties allow SOAs to capture the details about a policy change, and hence react accordingly to that change. We categorize policies into functional, non-functional, value-added, and specialized policies. Functional policies describe the operational features of a Web service (e.g., in WSDL). Non-functional policies include parameters that measure the quality of the service (e.g., response time, price). Value-added policies provide “better” environments for Web service interactions. They refer to a set of specifications for supporting optional (but important) requirements for the service (e.g., security, privacy, collaboration policies). A specialized policy defines requirements that are specific to an application domain. Shipping and billing are examples of specialized policies in business-to-business e-commerce. The scope of a change defines the subject to which that change was applied. It includes details about (i) the location of
the modified policy specification and (ii) the element that has been updated within that specification. The specification location is given by the URI of the XML file that stores the specification. The updated element is identified by the XPath query of that element within the specification. We give below a definition of policy change.

**Definition 3.1**

A policy change is defined by \((C,S)\) where
- \(C \in \{\text{“functional”}, \text{“non-functional”}, \text{“value-added”}, \text{“specialized”}\}\) is the policy category.
- \(S\) is defined by \((U,Q)\) where \(U\) is the URI of the XML policy document and \(Q\) is the XPath query of the element that has been changed within that document.

**3.2.3 State of a Participant Service**

In service-oriented settings, each SOA relies on participant services to perform users’ requests; a request submitted to an SOA is executed by invoking all or some of its participants. For instance, SOA\(_1\) relies on WS\(_1\), WS\(_2\), and WS\(_3\) to perform translation from English to Spanish (Figure 3.1). Since policy changes\(^1\) in a participant may impact the way SOAs interact with that participant, there is need to notify SOAs about such changes. For example, assume that WS\(_1\) provider modifies the port element of WS\(_1\)’s WSDL document (e.g., changing the SOAP/HTTP address). SOA\(_1\) should be made aware of this change to be able to interact with WS\(_1\). To keep track of policy changes, each participant WS\(_i\) has a data structure called State\(_i\) (Figure 3.3). The content of State\(_i\) is periodically communicated to all ascendant SOAs. If SOA\(_j\) does not receive State\(_i\) after a certain period of time, then it assumes a physical or service unavailability failure in WS\(_i\). Otherwise, SOA\(_j\) reads State\(_i\) to find out about the changes made to WS\(_i\).

State\(_i\) is composed of two attributes: ChangeStatus and ChangeDetails. ChangeStatus is equal to True if and only if changes have been made to WS\(_i\). Several changes may occur in WS\(_i\) during a time period; details about these changes are stored in the ChangeDetails set. Each element of this set is a couple \((C,S)\) where \(C\) is the category and \(S\) is the scope of a change. ChangeStatus

\(^1\) In the rest of this paper, we will use “policy change” and “change” interchangeably.
and ChangeDetails are initialized to False and $\emptyset$, respectively. If a change $(C,S)$ is detected on $WS_i$, $State_i.ChangeStatus$ is set to True and $(C,S)$ is added to $State_i.ChangeDetails$. The following definition summarizes $State_i$ properties.

```
<table>
<thead>
<tr>
<th>State$_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChangeStatus</td>
</tr>
<tr>
<td>True/False</td>
</tr>
</tbody>
</table>

<p>| ChangeDetails |</p>
<table>
<thead>
<tr>
<th>Category</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.3 State of a Participant Service $WS_i$

**Definition 3.2**

Let $WS_i$ be a participant service. The state $State_i$ of $WS_i$ is defined by $(ChangeStatus, ChangeDetails)$ where $ChangeStatus$ and $ChangeDetails$ are defined as follows:

- $ChangeStatus = True \iff$ changes have been made to $WS_i$.
- $ChangeDetails = \{(C,S) / C$ and $S$ are the category and scope of a change in $WS_i\}$.
- Initially do: $State_i.ChangeStatus = False$ and $State_i.ChangeDetails = \emptyset$.
- At the occurrence of a change $(C,S)$ in $WS_i$ do: $State_i.ChangeStatus = True$ and $State_i.ChangeDetails = State_i.ChangeDetails \cup \{(C,S)\}$.

3.2.4 Soft-State Senders and Receivers

Bottom-up failure detection is a collaborative process between participants and their ascendants. In the proposed approach, we externalize failure detection protocols from participants and SOAs. This has three major advantages. First, it allows a clear separation between the business logic of participants/SOAs and failure detection tasks. Second, it facilitates the integration of fault detection mechanisms in legacy services/SOAs (i.e., services/SOAs designed with no failure detection capabilities). Third, it preserves the autonomy of Web services; no modification needs to be performed on Web services to participate in failure detection protocols. Failure detection is facilitated through external Web services called fault coordinators attached to participants/SOAs.
Providing coordinators as Web services has three major advantages. First, it concords with recent trends in service-oriented architectures (e.g., Open Grid Services Architecture) where all management tasks are provided as Web services. Second, it facilitates the integration of coordinators with existing Web service engines; an SOA can be seen as a composition of regular business services and coordinators. Third, participant providers and SOA developers can integrate coordinators with their services/applications by merely discovering those coordinators from a service registry, downloading their trusted code, and deploying them as Web services. Coordinators are deployed and run on the same servers as their attached participants/SOAs. This means that if a physical fault occurs in a participant/SOA, it also occurs in the attached coordinator.

We define two types of fault coordinators: soft-state senders and soft-state receivers. Each participant (resp., SOA) has a soft-state sender (resp., receiver) attached to it. Figure 3.4 depicts a network of soft-state sender and receivers. Each soft-state sender SS-\(S_i\) maintains the State\(_i\) data structure. The content of State\(_i\) (i.e., ChangeStatus and ChangeDetails attributes) is communicated by SS-\(S_i\) to its receivers via Refresh() messages. To keep track of its receivers, SS-\(S_i\) maintains a Receivers(SS-\(S_i\)) data structure. Receivers(SS-\(S_i\)) is the set of receivers to which SS-\(S_i\) needs to send Refresh() messages. If WS\(_i\) (associated to SS-\(S_i\)) participates in SOA\(_j\) (associated to SS-\(R_j\)) then SS-\(R_j\) \(\in\) Receivers(SS-\(S_i\)). SS-\(S_i\) periodically sends Refresh() messages to its receivers. The duration of this period is determined by the \(\tau_{SSS}\) timer maintained by SS-\(S_i\). At the end of \(\tau_{SSS}\), (\(\tau_{SSS}\) becomes equal to 0) SS-\(S_i\) sends Refresh() to all receivers in Receivers(SS-\(S_i\)). Each soft-state receiver SS-\(R_j\) maintains two data structures: Sender(SS-\(R_j\)) and \(\tau_{SSR}\). Sender(SS-\(R_j\)) is the set of senders from which SS-\(R_j\) expects to receive Refresh() messages. If WS\(_i\) (associated to SS-\(S_i\)) participates in SOA\(_j\) (associated to SS-\(R_j\)) then SS-\(S_i\) \(\in\) Senders(SS-\(R_j\)). \(\tau_{SSR}\) is a timer used by SS-\(R_j\) to process Refresh() messages received from soft-state senders. At the end of \(\tau_{SSR}\) (\(\tau_{SSS}\) becomes equal to 0), SS-\(R_j\) checks whether it received a Refresh() message from each SS-\(S_i\) that belongs to Senders(SS-\(R_j\)).
The main issues addressed by SS-S_i and SS-R_i are (a) detecting logical faults in WS_i, (b) communicating those faults to SS-R_i, (c) detecting logical and physical faults in SS-S_i, and (d) reacting to those faults. Our main focus in this thesis is on interactions between soft-state senders and receivers (issue (b)) and bottom-up fault detection (issue (c)). The way SS-S_i is made aware of logical faults in WS_i is out of the scope of this research (issue (a)). For example, several solutions could be adopted by SS-S_i to detect policy changes in WS_i. For instance, SS-S_i may use XML version control algorithms to detect changes in WS_i policy specifications. Another solution is to provide an interface in SS-S_i through which WS_i provider submits all policy changes. A third solution is to use the publish/subscribe model where SS-S_i subscribes with WS_i on policy changes. Additionally, the way SS-R_i reacts to faults (issue (d)) is out of the scope of this research. We assume the existence of utility functions and procedures to deal with this issue. The way these utility procedures and functions are defined is out of the scope of this thesis:

- **Agreed2Join**: SS-S_i uses this utility function to decide whether WS_i would participate in SOA_j or not.
- **Process-Refusal**: SS-R_j calls this utility procedure to deal with SS-S_i participation refusal faults.
- **Process-Changes()**: SS-R<sub>i</sub> uses this utility procedure to process all changes that may have occurred in SS-S<sub>j</sub> during the last SS-R<sub>i</sub> cycle.
- **Process-Shutdown()**: SS-R<sub>i</sub> uses this utility procedure to handle freeze faults scheduled by participant providers.
- **Process-Stop()**: SS-R<sub>i</sub> uses this utility procedure to handle stop faults scheduled by participant providers.
- **Process-No-Refresh()**: If SS-R<sub>i</sub> does not receive Refresh() messages from SS-S<sub>j</sub>, it assumes a failure in SS-S<sub>j</sub>. SS-R<sub>i</sub> copes with this failure by calling the **Process-No-Refresh()** utility procedure.

**Example 3.1**

Figure 3.5 shows the soft-state senders and receive of the SOAs and participant services described in our motivating scenario (Section 3.2.1). The example includes four (4) soft-state senders SS-S<sub>1</sub>, SS-S<sub>2</sub>, SS-S<sub>3</sub>, and SS-S<sub>4</sub> associated to participants WS<sub>1</sub>, WS<sub>2</sub>, WS<sub>3</sub>, and WS<sub>4</sub> respectively. SOA<sub>1</sub> and SOA<sub>2</sub> have SS-R<sub>1</sub> and SS-R<sub>2</sub> attached to them. In the examples given in the rest of this chapter, we focus on the Refresh() messages sent by SS-S<sub>3</sub> to SS-R<sub>1</sub> and SS-R<sub>2</sub>. We assume that \( \tau_{SSR1} = \tau_{SSR2} = 2 \times \tau_{SSS3} = 4 \times \tau_{ACK} \).
3.3 Pure Soft-State Protocol (Pure-SS)

In the pure soft-state protocol, each SS-S\textsubscript{i} sends Refresh() messages to its receivers at the end of a \( \tau_{SSS} \) cycle. The categories and scopes of all changes that occurred during that cycle are embedded in those Refresh() messages. At the end of its \( \tau_{SSR} \) cycle, each SS-R\textsubscript{j} checks whether it received Refresh() messages from senders. It also processes all changes communicated by senders through those messages.

Table 3.1 includes the Pure-SS algorithm executed by a soft-state sender SS-S\textsubscript{i}. The first message to received by SS-S\textsubscript{i} from SS-R\textsubscript{j} is Join(); this message invites WS\textsubscript{i} to participate in SOA\textsubscript{j} (lines 1-12). SS-S\textsubscript{i} executes the utility function Agreed2Join() to figure out whether WS\textsubscript{i} is willing to participate in SOA\textsubscript{j}. If Agree2Join() returns False, SS-S\textsubscript{i} sends the Decision2Join(SS-S\textsubscript{i},False) message to SS-R\textsubscript{j}. Otherwise, SS-S\textsubscript{i} adds SS-R\textsubscript{j} to its receivers. If SS-R\textsubscript{j} is the first receiver of SS-S\textsubscript{i}, SS-S\textsubscript{i} initializes State\textsubscript{i} and starts its \( \tau_{SSS} \) timer. Finally, SS-S\textsubscript{i} sends its decision to SS-R\textsubscript{j} through the Decision2Join(SS-S\textsubscript{i},True) message. At any time, SS-S\textsubscript{i} may receive a Leave() message from SS-R\textsubscript{j} (lines 13-15). This message indicates that SOA\textsubscript{j} is no longer using WS\textsubscript{i} as a participant. Hence, SS-S\textsubscript{i} removes SS-R\textsubscript{j} from its receivers. At the occurrence of a change (with a category C and scope S) in WS\textsubscript{i} (lines 16-21), SS-S\textsubscript{i} sets State\textsubscript{i}.ChangeStatus to True. SS-S\textsubscript{i} keeps track of that change by inserting (C,S) in State\textsubscript{i}.ChangeDetails. In this way, the state of SS-S\textsubscript{i} to be sent to receivers at the end of \( \tau_{SSS} \) cycle includes all changes that have occurred during that cycle. At the end of \( \tau_{SSS} \) timer (lines 22-29), SS-S\textsubscript{i} sends a Refresh() message to each SS-S\textsubscript{i} receiver. This message includes State\textsubscript{i} attributes as parameters. SS-S\textsubscript{i} then reinitializes State\textsubscript{i} and restarts the \( \tau_{SSS} \) timer.

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>At Reception of Join(SS-R\textsubscript{j}) from SS-R\textsubscript{j}, Do</td>
</tr>
<tr>
<td>02</td>
<td>If Agreed2Join(SS-R\textsubscript{j}) = True</td>
</tr>
<tr>
<td>03</td>
<td>Then Receivers(SS-S\textsubscript{i}) = Receivers(SS-S\textsubscript{i}) ( \cup ) {SS-R\textsubscript{j}};</td>
</tr>
<tr>
<td>04</td>
<td>If</td>
</tr>
<tr>
<td>05</td>
<td>Then State\textsubscript{i}.ChangeStatus = False;</td>
</tr>
<tr>
<td>06</td>
<td>State\textsubscript{i}.ChangeDetails = ( \emptyset );</td>
</tr>
<tr>
<td>07</td>
<td>Start ( \tau_{SSS} ) timer of SS-S\textsubscript{i};</td>
</tr>
<tr>
<td>08</td>
<td>EndIf</td>
</tr>
<tr>
<td>09</td>
<td>Send Decision2Join(SS-S\textsubscript{i},True) to SS-R\textsubscript{j}</td>
</tr>
</tbody>
</table>
else send decision2join(ss-s_i,false) to ss-r_j
end

at reception of leave(ss-r_j) from ss-r_j do
receivers(ss-s_i) = receivers(ss-s_i) - {ss-r_j};
end

at the occurrence of change(c,s) in ws_i do
    state_i.changestatus = true;
    state_i.changedetails = state_i.changedetails \cup \{(c,s)\}; // insert the old and new
    // categories/scopes in case of several changes per cycle
end

at the end of \(\tau_{SSS}\) timer of ss-s_i do
for each ss-r_j / ss-r_j \in receivers(ss-s_i) do
    send refresh(state_i.changestatus, state_i.changedetails) to ss-r_j;
endfor
state_i.changestatus = false;
state_i.changedetails = \ø;
re-start \(\tau_{SSS}\) timer of ss-s_i;
end

Table 3.1. SS-s_i Algorithm for Pure Soft State

Table 3.2 gives the algorithm executed by a soft-state receiver ss-r_i. In this algorithm, ss-r_i maintains a variable (positive integer) called max-retry_i: if ss-r_i does not receive refresh() messages from ss-s_j during max-retry_i consecutive \(\tau_{SSR}\) cycles, it considers ws_j as failed. For instance, if max-retry_i is equal to three, then ss-r_i will consider ws_j as failed if it does not receive a refresh() from ss-s_j during three consecutive cycle. The value of max-retry_i is set by soa_i developer and may vary from an application to another. The smaller is max-retry_i, the more pessimistic is ss-r_i about ss-s_j failure. To keep track of refresh() messages transmitted by senders to ss-r_i, ss-r_i maintains a table called sr-table_i. This table contains an entry for each ss-s_j \in sender(ss-r_i). Each entry contains two columns:
- **Refreshed**: \( SR-Table_i[SS-S_j,\text{Refreshed}] \) equals True iff \( SS-R_i \) received a \( \text{Refresh()} \) from \( SS-S_j \) in the current \( \tau_{SSR} \) cycle.

- **Retry**: \( SR-Table_i[SS-S_j,\text{Retry}] \) contains the number of consecutive cycles during which \( SS-R_i \) did not receive \( \text{Refresh()} \) messages from \( SS-S_j \). \( SR-Table_i[SS-S_j,\text{Retry}] \) should always be less than or equal to \( \text{Max-Retry}_i \).

Whenever a new participant \( WS_j \) is added to SOA\(_i\), \( SS-R_i \) sends a \( \text{Join()} \) message to \( SS-S_j \) (lines 1-3). At the deletion of \( WS_j \) from SOA\(_i\), \( SS-R_i \) sends a \( \text{Leave()} \) message to \( SS-S_i \) and deletes \( SS-S_j \) entry from \( SR-Table_i \) (lines 4-7). At the reception of a positive \( \text{Decision2Join()} \) message from \( SS-S_j \) (lines 8-19), \( SS-R_i \) adds \( SS-S_j \) to its senders. It also creates a new entry for \( SS-S_j \) in \( SR-Table_i \) and initializes the \( \text{Refreshed} \) and \( \text{Retry} \) columns of that entry to False and 0, respectively. Finally, \( SS-R_i \) starts its \( \tau_{SSR} \) timer in the case where \( SS-S_j \) is the first \( SS-R_i \) sender (lines 14-15). If the \( \text{Decision2Join()} \) message received from \( SS-S_j \) is negative (line 17), \( SS-R_i \) calls the utility procedure \( \text{Process-Refusal()} \) to handle this situation (i.e., \( WS_j \) is not willing to join SOA\(_i\)).

At the reception of a \( \text{Refresh()} \) message from \( SS-S_j \) (lines 20-25), \( SS-R_i \) sets the \( \text{Refreshed} \) column of \( SS-S_j \) entry in \( SR-Table_i \) to True. If the \( \text{ChangeStatus} \) parameter submitted with \( \text{Refresh()} \) is True, \( SS-R_i \) processes all changes that may have occurred in \( SS-S_j \) during the last \( \tau_{SSR} \) cycle; this is done by calling the utility procedure \( \text{Process-Changes()} \). At the end of \( \tau_{SSR} \) timer (lines 26-38), \( SS-R_i \) checks if it received a \( \text{Refresh()} \) message from each participant’s sender. If \( SS-R_i \) received \( \text{Refresh()} \) from \( SS-S_j \), it re-initializes the \( \text{Refreshed} \) and \( \text{Retry} \) columns of \( SS-S_j \) entry in \( SR-Table_i \) to False and 0, respectively. Otherwise (i.e., refresh not received from \( SS-S_j \)), \( SS-R_i \) increments the \( \text{Retry} \) column of \( SS-S_j \) entry in \( SR-Table_i \). If \( SS-S_j \)’s \( \text{Retry} \) equals \( \text{Max-Retry}_i \) (i.e., \( SS-R_i \) did not receive refresh messages from \( SS-S_j \) during \( \text{Max-Retry}_i \) consecutive \( SS-R_i \) cycles), \( SS-R_i \) processes \( WS_j \) failure by calling the utility procedure \( \text{Process-No-Refresh()} \). \( SS-R_i \) finally restarts its \( \tau_{SSR} \) timer.

```
(01) At addition of \( WS_j \) to SOA\(_i\), Do
(02)   Send \( \text{Join}(SS-R_i) \) to \( SS-S_j \);
(03) End

(04) At deletion of \( WS_j \) from SOA\(_i\), Do
(05)   Send \( \text{Leave}(SS-R_i) \) to \( SS-S_j \);
```
Delete SS-S<sub>j</sub> entry from SR-Table<sub>i</sub>;

End

At Reception of Decision2Join(SS-S<sub>j</sub>,decision) From SS-S<sub>j</sub> Do

If decision = True
    Then Senders(SS-R<sub>i</sub>) = Senders(SS-R<sub>i</sub>) ∪ {SS-S<sub>j</sub>};
    Create an entry for SS-S<sub>j</sub> in SR-Table<sub>i</sub>;
    SR-Table<sub>i</sub>[SS-S<sub>j</sub>,Refreshed] = False;
    SR-Table<sub>i</sub>[SS-S<sub>j</sub>,Retry] = 0;
    If |Senders(SS-R<sub>i</sub>)| = 1
       Then Start \( \tau_{\text{SSR}} \) timer of SS-R<sub>i</sub>;
    EndIf
Else Process-Refusal(SS-S<sub>j</sub>);
EndIf
End

At Reception of Refresh(ChangeStatus, ChangeDetails) From SS-S<sub>j</sub> Do

SR-Table<sub>i</sub>[SS-S<sub>j</sub>, Refreshed] = True;
If ChangeStatus = True // A Change occurred in SS-S<sub>j</sub> participant
    Then Process-Changes(SS-S<sub>j</sub>, ChangeDetails);
EndIf
End

At the end of \( \tau_{\text{SSR}} \) timer of SS-R<sub>i</sub> Do

For each SS-S<sub>j</sub> / SS-S<sub>j</sub> ∈ Senders(SS-R<sub>i</sub>) Do
    If SR-Table<sub>i</sub>[SS-S<sub>j</sub>, Refreshed] = True
       Then SR-Table<sub>i</sub>[SS-S<sub>j</sub>, Refreshed] = False;
       SR-Table<sub>i</sub>[SS-S<sub>j</sub>, Retry] = 0;
    Else SR-Table<sub>i</sub>[SS-S<sub>j</sub>, Retry]++ // Refreshed Not Received
       If SR-Table<sub>i</sub>[SS-S<sub>j</sub>, Retry] = Max-Retry<sub>i</sub>
          Then Process-No-Refresh(SS-S<sub>j</sub>); // Process the Failure
EndIf
EndIf
EndFor
Re-start \( \tau_{\text{SSR}} \) timer of SS-R<sub>i</sub>;
Example 3.2

Let us consider the scenario shown in Figure 3.6. We assume that at time $t_{31}$, SS-S$_3$ detects a change (with category $C_1$ and scope $S_1$) in WS$_3$. This change is communicated to SS-R$_1$ and SS-R$_2$ via the $\text{Refresh}(\text{True},\{(C_1,S_1)\})$ messages sent at time $t_2$. SS-R$_1$ and SS-R$_2$ process those changes at times $t_{11}$ and $t_{21}$, respectively.

![Diagram](image)

**Figure 3.6.** Pure Soft-State Protocol: An Example

At time $t_3$, SS-R$_1$ and SS-R$_2$ note the reception of $\text{Refresh}()$ message from SS-S$_3$. At this time, SS-S$_3$ sends new $\text{Refresh}()$ messages to both receivers with the parameters $(\text{False},\emptyset)$ since no changes have been detected in the second SS-S$_3$ cycle. In the third SS-S$_3$ cycle, SS-S$_3$ sends
Refresh() messages to SS-R_1 and SS-R_2 with the parameters \((\text{True},\{(C_1,S_1),(C_2,S_2)\})\) since two changes are detected by SS-S_3 in this cycle. SS-R_1 and SS-R_2 process those changes at times \(t_{12}\) and \(t_{22}\), respectively. At time \(t_5\), SS-R_1 and SS-R_2 note the reception of Refresh() messages from SS-S_3. At times \(t_5\), \(t_6\), \(t_7\), \(t_8\), \(t_9\), and \(t_{10}\), SS-S_3 sends new Refresh() messages to SS-R_1 and SS-R_2 with the parameters \((\text{False},\emptyset)\) since no changes have been detected in the corresponding SS-S_3 cycle. We assume that the Refresh() messages sent to SS-R_2 starting from \(t_7\) are lost. At time \(t_7\), SS-R_1 and SS-R_2 note the reception of Refresh(). At time \(t_9\), SS-R_2 finds out that it did not receive Refresh() messages from SS-S_3 during the last SS-R_2 cycle. If \(\text{Max-Retry}_2\) is equal to 1, then SS-R_2 concludes that SS-S_3 failed and hence calls the Process-No-Refresh() utility procedure.

### 3.4 Soft-State Protocol with Explicit Removal (Removal-SS)

In the Pure-SS protocol, SS-R_j assumes a failure in SS-S_i if it does not receive Refresh() messages from SS-S_i after \(\text{Max-Retry}_i\) SS-R_j cycles. This could happen because of one of a physical fault (node or transmission) or status change (logical fault) in SS-S_i. While physical faults are out of SS-S_i control, status changes are scheduled by service providers and hence, can explicitly be notified to SS-R_j. This would have two major advantages. First, SS-R_j will be able to detect faults due to status change as soon as they are communicated by SS-S_i, instead of waiting the end of \(\text{Max-Retry}_i\) SS-R_j cycles. Second, SS-R_j may differentiate between logical and physical faults and hence, process them appropriately.

The soft-state protocol with explicit removal (Removal-SS) extends Pure-SS with explicit removal messages; these messages announce future status change faults in participants. Table 3.3 shows the steps of the Removal-SS algorithm executed by the soft-state sender SS-S_i. The statements in lines 1-29 are common to Pure-SS and Removal-SS protocols. If WS_i provider is scheduling a shut-down of its service (lines 30-34), SS-S_i sends an explicit Shutdown() message along with the down and up times to each SS-R_j \(\in\text{Senders}(\text{SS-S}_i)\). In this case, SS-S_i will not send Refresh() messages during the period [DownFrom,UpTo]. If the service provider decides to re-start WS_i (lines 35-40), SS-S_i sends Awake() messages to all receivers in Receivers(\text{SS-S}_i) and re-starts its \(\tau_{\text{SSS}}\) timer. If WS_i provider decides to stop its service with \(\text{SOA}_j\) (lines 41-44),
SS-S<sub>i</sub> removes SS-R<sub>j</sub> from its receivers and sends an explicit removal message Stop() to SS-R<sub>j</sub>. SS-S<sub>i</sub> will no longer send Refresh() messages to SS-R<sub>j</sub>.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>At Reception of Join(SS-R&lt;sub&gt;j&lt;/sub&gt;) from SS-R&lt;sub&gt;j&lt;/sub&gt; Do</td>
</tr>
<tr>
<td>02</td>
<td>If Agreed2Join(SS-R&lt;sub&gt;j&lt;/sub&gt;) = True</td>
</tr>
<tr>
<td>03</td>
<td>Then Receivers(SS-S&lt;sub&gt;i&lt;/sub&gt;) = Receivers(SS-S&lt;sub&gt;i&lt;/sub&gt;) ∪ {SS-R&lt;sub&gt;j&lt;/sub&gt;};</td>
</tr>
<tr>
<td>04</td>
<td>If</td>
</tr>
<tr>
<td>05</td>
<td>Then State&lt;sub&gt;i&lt;/sub&gt;.ChangeStatus = False;</td>
</tr>
<tr>
<td>06</td>
<td>State&lt;sub&gt;i&lt;/sub&gt;.ChangeDetails = {};</td>
</tr>
<tr>
<td>07</td>
<td>Start τ&lt;sub&gt;SSS&lt;/sub&gt; timer of SS-S&lt;sub&gt;i&lt;/sub&gt;;</td>
</tr>
<tr>
<td>08</td>
<td>EndIf</td>
</tr>
<tr>
<td>09</td>
<td>Send Decision2Join(SS-S&lt;sub&gt;i&lt;/sub&gt;,True) to SS-R&lt;sub&gt;j&lt;/sub&gt;</td>
</tr>
<tr>
<td>10</td>
<td>Else Send Decision2Join(SS-S&lt;sub&gt;i&lt;/sub&gt;,False) to SS-R&lt;sub&gt;j&lt;/sub&gt;</td>
</tr>
<tr>
<td>11</td>
<td>EndIf</td>
</tr>
<tr>
<td>12</td>
<td>End</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>At Reception of Leave(SS-R&lt;sub&gt;j&lt;/sub&gt;) from SS-R&lt;sub&gt;j&lt;/sub&gt; Do</td>
</tr>
<tr>
<td>14</td>
<td>Receivers(SS-S&lt;sub&gt;i&lt;/sub&gt;) = Receivers(SS-S&lt;sub&gt;i&lt;/sub&gt;) – {SS-R&lt;sub&gt;j&lt;/sub&gt;};</td>
</tr>
<tr>
<td>15</td>
<td>End</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>At the occurrence of Change(C,S) in WS, Do</td>
</tr>
<tr>
<td>17</td>
<td>// C and S are the detected change category and scope, respectively</td>
</tr>
<tr>
<td>18</td>
<td>State&lt;sub&gt;i&lt;/sub&gt;.ChangeStatus = True;</td>
</tr>
<tr>
<td>19</td>
<td>State&lt;sub&gt;i&lt;/sub&gt;.ChangeDetails = State&lt;sub&gt;i&lt;/sub&gt;.ChangeDetails ∪ {(C,S)}; // insert the old and new</td>
</tr>
<tr>
<td>20</td>
<td>// categories/scopes in case of several changes per cycle</td>
</tr>
<tr>
<td>21</td>
<td>End</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>At the end of τ&lt;sub&gt;SSS&lt;/sub&gt; timer of SS-S&lt;sub&gt;i&lt;/sub&gt; Do</td>
</tr>
<tr>
<td>23</td>
<td>For each SS-R&lt;sub&gt;j&lt;/sub&gt; / SS-R&lt;sub&gt;j&lt;/sub&gt; ∈ Receivers(SS-S&lt;sub&gt;i&lt;/sub&gt;) Do</td>
</tr>
<tr>
<td>24</td>
<td>Send Refresh(State&lt;sub&gt;i&lt;/sub&gt;.ChangeStatus, State&lt;sub&gt;i&lt;/sub&gt;.ChangeDetails) To SS-R&lt;sub&gt;j&lt;/sub&gt;;</td>
</tr>
<tr>
<td>25</td>
<td>EndFor</td>
</tr>
<tr>
<td>26</td>
<td>State&lt;sub&gt;i&lt;/sub&gt;.ChangeStatus = False;</td>
</tr>
<tr>
<td>27</td>
<td>State&lt;sub&gt;i&lt;/sub&gt;.ChangeDetails = {};</td>
</tr>
<tr>
<td>28</td>
<td>Re-start τ&lt;sub&gt;SSS&lt;/sub&gt; timer of SS-S&lt;sub&gt;i&lt;/sub&gt;;</td>
</tr>
<tr>
<td>29</td>
<td>End</td>
</tr>
</tbody>
</table>
Table 3.3. SS-S_i Algorithm for Soft State with Explicit Removal

Table 3.4 describes the algorithm executed by the soft-state receiver SS-R_i. The statements in lines 1-25 are similar to the ones given included in Pure-SS algorithm. To handle freeze faults, SR-Table_i entries are extended with DownTime and UpTime attributes. SR-Table_i[SS-S_j,DownTime] and SR-Table_i[SS-S_j,UpTime] contain the down and up times of the shutdown schedule by WS_i provider, respectively. At the creation of an entry for SS-S_j in SR-Table_i, SR-Table_i[SS-S_j,DownTime] and SR-Table_i[SS-S_j,UpTime] are initialized to 0 (lines 14-15).

At the end of $\tau_{SSR}$ timer (lines 28-43), SS-R_i checks if it received a Refresh() message from senders. If a sender SS-S_j is in the frozen status (lines 30-32), SS-R_i does not expect to receive a Refresh() from SS-S_j and hence skips SS-S_j. Otherwise, SS-R_i handles SS-S_j as in the case of Pure-SS protocol (lines 33-43). At the reception of a Shutdown() message from SS-S_j (lines 44-48), SS-R_i processes the shutdown scheduled by SS-S_j by calling the utility procedure Process-Shutdown(). At the reception of a Awake() message from SS-S_j (lines 49-55), SS-R_i checks if SS-S_j still belongs to Senders(SS-S_j). SS-S_j could have been removed from Senders(SS-S_j) as part of the Process-Shutdown() utility procedure. If SS-S_j $\notin$ Senders(SS-S_j), SS-R_i sends a
Leave() message to SS-S\textsubscript{j}. Otherwise, SS-R\textsubscript{i} reinitializes SR-Table\textsubscript{i}[SS-S\textsubscript{j},DownTime] and SR-Table\textsubscript{i}[SS-S\textsubscript{j},UpTime] with 0. At the reception of a Stop() message from SS-S\textsubscript{j} (lines 56-60), SS-R\textsubscript{i} deletes SS-S\textsubscript{j} entry in SR-Table and removes SS-S\textsubscript{j} from Senders(SS-R\textsubscript{i}). Finally, it processes the stop notified by SS-S\textsubscript{j} by calling the utility procedure Process-Stop().

(01) At addition of WS\textsubscript{j} to SOA, Do
(02)     Send Join(SS-R\textsubscript{i}) to SS-S\textsubscript{j};
(03) End

(04) At deletion of WS\textsubscript{j} from SOA, Do
(05)     Send Leave(SS-R\textsubscript{i}) to SS-S\textsubscript{j};
(06)     Delete SS-S\textsubscript{j} entry from SR-Table\textsubscript{i};
(07) End

(08) At Reception of Decision2Join(SS-S\textsubscript{j},decision) From SS-S\textsubscript{j}, Do
(09)     If decision = True
(10)         Then Senders(SS-R\textsubscript{i}) = Senders(SS-R\textsubscript{i}) \cup \{SS-S\textsubscript{j}\};
(11)         Create an entry for SS-S\textsubscript{j} in SR-Table\textsubscript{i};
(12)         SR-Table\textsubscript{i}[SS-S\textsubscript{j},Refreshed] = False;
(13)         SR-Table\textsubscript{i}[SS-S\textsubscript{j},Retry] = 0;
(14)         SR-Table\textsubscript{i}[SS-S\textsubscript{j},DownTime] = 0;
(15)         SR-Table\textsubscript{i}[SS-S\textsubscript{j},UpTime] = 0;
(16)         If |Senders(SS-R\textsubscript{i})| = 1
(17)             Then Start \(\tau\textsubscript{SSR}\) timer of SS-R\textsubscript{i};
(18)         EndIf
(19)     Else Process-Refusal(SS-S\textsubscript{j});
(20)     EndIf
(21) End

(22) At Reception of Refresh(ChangeStatus,ChangeDetails) From SS-S\textsubscript{j}, Do
(23)     SR-Table\textsubscript{i}[SS-S\textsubscript{j},Refreshed] = True;
(24)     If ChangeStatus = True // A Change occurred in SS-S\textsubscript{j} participant
(25)         Then Process-Changes(SS-S\textsubscript{j},ChangeDetails);
(26)     EndIf
(27) End
(28) **At the end of** $\tau_{SSR}$ **timer of** SS-$R_i$ **Do**

(29) **For** each SS-$S_j$ / SS-$S_j \in$ Senders(SS-$R_i$) **Do**

(30) **If** Time $\geq$ SR-Table$_i$[SS-$S_j$, DownTime] $\wedge$ Time $<$ SR-Table$_i$[SS-$S_j$, UpTime]

(31) **Then** Continue;

(32) **EndIf**

(33) **If** SR-Table$_i$[SS-$S_j$, Refreshed] = True

(34) **Then** SR-Table$_i$[SS-$S_j$, Refreshed] = False;

(35) SR-Table$_i$[SS-$S_j$, Retry] = 0;

(36) **Else** SR-Table$_i$[SS-$S_j$, Retry]++ // Refreshed Not Received

(37) **If** SR-Table$_i$[SS-$S_j$, Retry] = Max-Retry$_i$

(38) **Then** Process-No-Refresh(SS-$S_j$); // Process the Failure

(39) **EndIf**

(40) **EndIf**

(41) **EndFor**

(42) **Re-start** $\tau_{SSR}$ **timer of** SS-$R_i$;

(43) **End**

(44) **At** Reception of Shutdown(DownFrom, UpTo) **From** SS-$S_j$ **Do**

(45) SR-Table$_i$[SS-$S_j$, DownTime] = DownFrom;

(46) SR-Table$_i$[SS-$S_j$, UpTime] = UpTo;

(47) Process-Shutdown(SS-$S_j$);

(48) **End**

(49) **At** Reception of Awake(SS-$S_j$) **From** SS-$S_j$ **Do**

(50) **If** SS-$S_j$ $\notin$ Senders(SS-$R_i$)

(51) **Then** Send Leave(SS-$R_i$) to SS-$S_j$;

(52) **Else** SR-Table$_i$[SS-$S_j$, DownTime] = 0;

(53) SR-Table$_i$[SS-$S_j$, UpTime] = 0;

(54) **EndIf**

(55) **End**

(56) **At** Reception of Stop(SS-$S_j$) **From** SS-$S_j$ **Do**

(57) Remove SS-$S_j$ **Entry** From SR-Table$_i$;

(58) Senders(SS-$R_i$) = Senders(SS-$R_i$) $-$ { SS-$S_j$};
Table 3.4. SS-R_i Algorithm for Soft State with Explicit Removal

Example 3.3

Let us consider the scenario shown in Figure 3.7. We assume that at time t_{31}, WS_3 provider schedules a shutdown during the period [t_{31},t_{5}]. This explicit removal is communicated to SS-R_1 and SS-R_2 via Shutdown() messages. SS-R_1 and SS-R_2 receive those messages at times t_{11} and t_{21}, respectively, and call Process-Shutdown() procedure. Hence, they do not expect to receive Refresh() messages at t_{5}.

![Diagram of Soft-State Protocol with Explicit Removal Message: An Example](image)

**Figure 3.7.** Soft-State Protocol with Explicit Removal Message: An Example
At time $t_5$, WS$_3$ is reinstated; SS-$S_3$ sends Awake() message to SS-R$_1$ and SS-R$_2$. SS-$S_3$ resumes sending Refresh() at time $t_6$. Assume that at time $t_{32}$, WS$_3$ provider would like to stop its service with SS-R$_2$. For that purpose, SS-$S_3$ sends a Stop() message to SS-R$_2$. SS-R$_2$ receives this message at time $t_{22}$, deletes SS-$S_3$ entry in SR-Table$_3$, removes SS-$S_3$ from Senders(SS-R$_2$), and calls Process-Stop() procedure. At times $t_7$ and $t_9$, SS-R$_1$ notes the reception of Refresh() messages. However, SS-R$_2$ does not expect the reception of such messages since WS$_3$ is no longer a participant in SOA$_2$.

### 3.5 Reliable Soft-State Protocol (Reliable-SS)

In the Pure-SS and Removal-SS protocols, SS-$S_i$ sends best effort (unreliable) Refresh() messages to SS-R$_j$. However, Refresh() messages may be lost and SS-$S_i$ has no way to figure out whether SS-R$_j$ received those messages. To deal with this issue, we propose the reliable soft-state protocol (Reliable-SS) which extends Pure-SS with the following two features. First, Refresh() messages are transmitted reliably. Each time a Refresh() is sent, SS-$S_i$ starts a $\tau_{ACK}$ timer waiting for an ACK() message from SS-R$_j$. On receiving a Refresh() message, SS-R$_j$ sends an acknowledgment (ACK) to the SS-$S_i$. If no ACK() message is received before $\tau_{ACK}$ timer expires, SS-$S_i$ re-sends Refresh() to SS-R$_j$. Second, SS-R$_j$ employs a notification mechanism through which it informs SS-$S_i$ about SS-R$_j$'s view on SS-$S_i$ (in terms of failure) at the end of $\tau_{SSR}$ timer. This allows SS-$S_i$ to recover from false failure detection by re-sending Refresh() to SS-R$_j$. To co-relate Refresh() and ACK() messages (i.e., associate each ACK() to the right Refresh()), SS-$S_i$ and SS-R$_j$ timestamp Refresh() and ACK() messages. The timestamp of each ACK() should be equal to the timestamp of the corresponding Refresh().

To implement the Reliable-SS protocol, each SS-$S_i$ maintains a table called SS-$Table$; this table contains an entry for each SS-R$_j \in$ Receivers(SS-$S_i$). The entry for SS-R$_j$ contains the following columns:

- **ACK**: SS-$Table$[SS-R$_j$,ACK] equals True iff SS-$S_i$ received an ACK() for the latest Refresh() sent to SS-R$_j$. The initial value for this column is False.
- **Timestamp**: $SS$-$Table_i[SS-R_j, Timestamp]$ contains the timestamp of the latest Refresh() sent to SS-R$_j$.

- **ChangeStatus**: $SS$-$Table_i[SS-R_j, ChangeStatus]$ equals True iff changes have been made to WS$_i$ since the last ACK() was processed. The initial value for this column is False.

- **ChangeDetails**: $SS$-$Table_i[SS-R_j, ChangeDetails]$ contain the categories and scopes of changes made to WS$_i$ since the last ACK() was processed. The initial value for this column is $\emptyset$.

In Reliable SS-S, SS-$S_i$ may send several Refresh() to SS-$R_j$ during a given cycle. If an ACK() sent back by SS-$R_j$ is delayed, then SS-$S_i$ may receive several ACK() during that cycle. Hence, SS-$S_i$ needs to determine whether an ACK() received from SS-$R_j$ corresponds to the latest Refresh() sent by SS-$S_i$ to SS-$R_j$. This is done by comparing the ACK() timestamp with the timestamp of the latest Refresh() message sent by SS-$S_i$ to SS-$R_j$ (stored in $SS$-$Table_i[SS-R_j, Timestamp]$). If the timestamps are different, SS-$S_i$ considers the ACK() message as obsolete and hence ignores it.

**Definition 3.3**

Let ACK(t) be a message sent by SS-$R_j$ to SS-$S_i$. We say that ACK(t) is relevant if $t = SS$-$Table_i[SS-R_j, Timestamp]$. Otherwise, ACK(t) is obsolete.

Table 3.5 shows the steps of the Reliable-SS algorithm executed by the soft-state sender SS-$S_i$. At the reception of Join() from SS-$R_j$ (lines 1-16), SS-$S_i$ executes the utility function Agreed2Join() to decide whether it is interested in participating in SOA$_j$ or not. If Agreed2Join() returns False, then SS-$S_i$ sends the Decision2Join(SS-$S_i$, False) message to SS-$R_j$. Otherwise, SS-$S_i$ adds SS-$R_j$ to its receivers. If SS-$R_j$ is the first SS-$S_i$ receiver, SS-$S_i$ initializes State$_i$ creates an entry for SS-$R_j$ in $SS$-$Table_i$, initializes that entry, and starts its $\tau_{SSS}$ timer. Finally, SS-$S_i$ sends its decision to SS-$R_j$ through the Decision2Join(SS-$S_i$, True) message. The reception of Leave() messages and change occurrences are handled by Reliable-SS in the same way as in Pure-SS (lines 17-26).
At the end of $\tau_{\text{SSS}}$ timer (lines 27-49), SS-$S_i$ sends `Refresh()` messages to its receivers. In Reliable-SS, the state to be communicated by SS-$S_i$ may vary from a receiver to another. For instance, assume that SS-$S_i$ did not receive an ACK() from SS-$R_j$ regarding a `Refresh()` sent by SS-$S_i$ at time $t_1$ (`Refresh()` or ACK() may have been lost). At time $t_2$, SS-$S_i$ needs to re-send the content of State$_i$ (as of times $t_1$ and $t_2$) to SS-$R_j$. We refer to the `Refresh()` message sent at time $t_1$ as pending.

**Definition 3.4**

Let us consider a `Refresh(True,D,t)` message sent by SS-$S_i$ to SS-$R_j$ where D contains the change details and t is the message timestamp. We say that `Refresh(True,D,t)` is *pending* if $t = \text{Table}_i[SS-R_j,\text{Timestamp}]$ (i.e., `Refresh(True,D,t)` is the latest refresh sent to SS-$R_j$) and Table$_i[SS-R_j,\text{ACK}] = \text{False}$ (i.e., SS-$S_i$ did not receive an ACK($t$) from SS-$R_j$). We refer to the changes defined by D as *pending changes*.

One important task of SS-$S_i$ is hence to compute the state to be submitted to each receiver. The computed state will be stored in intermediary variables MyChangeStatus and MyChangeDetails. For each receiver SS-$R_j$, SS-$S_i$ checks if it received ACK() messages from SS-$R_j$ acknowledging `Refresh()` messages sent during the last SS-$S_i$ cycle. This is done by accessing the ACK column of SS-$R_j$ entry in SS-Table (line 23). We identify the following two cases:

- **Case 1**: If SS-Table$_i[SS-R_j,\text{ACK}] = \text{False}$, SS-$R_j$ may have missed `Refresh()` messages sent by SS-$S_i$ during the latest cycle (lines 29-31). In this case, SS-$S_i$ adds the pending changes (stored in SS-Table$_i[SS-R_j,\text{ChangeStatus}]$ and SS-Table$_i[SS-R_j,\text{ChangeDetails}]$) to the current ones (stored in State$_i$). The new value of SS-$S_i$ state is computed as follows. MyChangeStatus is set to True if State$_i$.ChangeStatus is True (i.e., there have been changes since the last `Refresh()` sent to SS-$R_j$) or if there are pending changes (i.e., SS-Table$_i[SS-R_j,\text{ChangeStatus}]$ is True). MyChangeDetails includes the current change details (stored in State$_i$.ChangeDetails) and the pending change details (saved in SS-Table$_i[SS-R_j,\text{ChangeDetails}]$).
Case 2: If SS-Table$_i$[SS-R$_j$.ACK] = True, SS-$S_i$ does not have pending changes for SS-$R_j$ (lines 34-35). The MyChangeStatus and MyChangeDetails will include current changes only (stored in State$_i$).

Once SS-$S_i$ has computed the state to be communicated to SS-$R_j$, it sends Refresh() to SS-$R_j$ (lines 37-38). SS-$S_i$ then stores the timestamp in SS-Table$_i$[SS-$R_j$.Timestamp] to keep track of the most recent Refresh() timestamp. It also sets SS-Table$_i$[SS-$R_j$.ACK] to false (since the ACK() has not been received yet) and saves MyChangeStatus/MychangeDetails in SS-$R_j$ entry within SS-Table$_i$. After sending Refresh() to all its receivers, SS-$S_i$ reinitializes State$_i$, restarts $\tau_{ACK}$ (waiting for ACK() messages to be sent by receivers) and $\tau_{SSS}$ (starting a new cycle) timers.

At the reception of an ACK() message from SS-$R_j$ (lines 50-57), SS-$S_i$ checks if the ACK() is relevant. If yes, SS-$S_i$ sets the ACK column associated to SS-$R_j$ entry in SS-Table$_i$ to True. It also reinitializes the ChangeStatus and ChangeDetails columns associated to SS-$R_j$ entry in SS-Table$_i$ to False and $\emptyset$, respectively. This allows SS-$S_i$ to delete the saved changes that have been received by SS-$R_j$. Otherwise (the ACK() is obsolete), SS-$S_i$ ignores the ACK() message.

At the end of $\tau_{ACK}$ timer (lines 50-67), SS-$S_i$ checks if it received an ACK() message from its receivers. This is done by accessing the ACK column in SS-Table of each receiver SS-$R_j$. If SS-Table$_i$[SS-$R_j$.ACK] = False, SS-$R_j$ may have missed Refresh() messages sent by SS-$S_i$ during the last cycle. SS-$S_i$ Computes the new state to be communicated to SS-$R_j$ as explained in Case 1 above, sends a timestamped Refresh() message to SS-$R_j$, and saves the timestamp along with the newly computed state in SS-$R_j$ entry within SS-Table$_i$. After processing all receivers, SS-$S_i$ restarts $\tau_{ACK}$ timer. SS-$S_i$ may receive a Refresh-Lost() message from a SS-$R_j$ (lines 75-88). This message indicates that SS-$R_j$ did not received any Refresh() from SS-$S_i$ during the last $\tau_{SSR}$ cycle. SS-$S_i$ computes the new state to be communicated to SS-$R_j$ as explained in Case 1 above, sends a timestamped Refresh() message to SS-$R_j$, saves the timestamp and state in SS-$R_j$ entry within SS-Table$_i$, and restarts its $\tau_{ACK}$ timer.

```plaintext
(01) At Reception of Join(SS-$R_j$) from SS-$R_j$ Do
(02)   If Agreed2Join(SS-$R_j$) = True
(03)      Then Receivers(SS-$S_i$) = Receivers(SS-$S_i$) \cup \{SS-$R_j$\};
(04)      If |Receivers(SS-$S_i$)| = 1
```
Then State\textsubscript{i}.ChangeStatus = False;

State\textsubscript{i}.ChangeDetails = ∅;

Create an entry for SS-R\textsubscript{j} in SS-Table\textsubscript{i};

SS-Table\textsubscript{i}[SS-R\textsubscript{j},ACK] = False;

SS-Table\textsubscript{i}[SS-R\textsubscript{j},ChangeStatus] = False;

SS-Table\textsubscript{i}[SS-R\textsubscript{j},ChangeDetails] = ∅;

Start \( \tau_{SSS} \) timer of SS-S\textsubscript{i};

EndIf

Send Decision2Join(SS-S\textsubscript{i},True) to SS-R\textsubscript{j}

Else Send Decision2Join(SS-S\textsubscript{i},False) to SS-R\textsubscript{j}

EndIf

End

At Reception of Leave(SS-R\textsubscript{j}) from SS-R\textsubscript{j} Do

Receipts(SS-S\textsubscript{i}) = Receipts(SS-S\textsubscript{i}) \{SS-R\textsubscript{j}\};

Delete SS-R\textsubscript{j} entry from SS-Table\textsubscript{i};

End

At the occurrence of Change(C,S) in WS\textsubscript{i} Do

// C and S are the detected change category and scope, respectively

State\textsubscript{i}.ChangeStatus = True;

State\textsubscript{i}.ChangeDetails = State\textsubscript{i}.ChangeDetails \cup \{(C,S)\}; // insert the old and new categories/scopes in case of several changes per cycle

End

At the end of \( \tau_{SSS} \) timer of SS-S\textsubscript{i} Do

For each SS-R\textsubscript{j} / SS-R\textsubscript{j} \in Receipts(SS-S\textsubscript{i}) Do

If SS-Table\textsubscript{i}[SS-R\textsubscript{j},ACK] = False

Then MyChangeStatus = State\textsubscript{i}.ChangeStatus \lor SS-Table\textsubscript{i}[SS-R\textsubscript{j},ChangeStatus];

MyChangeDetails = State\textsubscript{i}.ChangeDetails \lor SS-Table\textsubscript{i}[SS-R\textsubscript{j},ChangeDetails];

// if SS-S\textsubscript{i} did not receive ACK from SS-R\textsubscript{j}, then it should re-send

// Refresh to SS-R\textsubscript{j} with the new and pending changes

Else MyChangeStatus = State\textsubscript{i}.ChangeStatus;

MyChangeDetails = State\textsubscript{i}.ChangeDetails;

EndIf

T = GetTimestamp();
Send Refresh(MyChangeStatus,MyChangeDetails,T) To SS-R_j

SS-Table[SS-R_j,ACK] = False;

SS-Table[SS-R_j,Timestamp] = T; // Save timestamp of the latest refresh

SS-Table[SS-R_j,ChangeStatus] = MyChangeStatus;

SS-Table[SS-R_j,ChangeDetails] = MyChangeDetails;

// save the State sent to SS-R_j in SS-Table_i

Re-start ACK timer $\tau_{ACK}$ of SS-S_i;

EndFor

State_i.ChangeStatus = False;

State_i.ChangeDetails = $\emptyset$;

Re-start $\tau_{SSS}$ timer of SS-S_i;

End

At the reception of ACK(T) From SS-R_j Do

If T = SS-Table[SS-R_j,Timestamp] // Test if this is a relevant ACK

Then SS-Table[SS-R_j,ACK] = True;

SS-Table[SS-R_j,ChangeStatus] = False;

SS-Table[SS-R_j,ChangeDetails] = $\emptyset$; // delete previously saved changes

Else Ignore ACK Message // ACK of an obsolete Refresh message

Endif

End

At the end of $\tau_{ACK}$ timer of SS-S_i Do

For each SS-R_j / SS-R_j $\in$ Receivers(SS-S_i) Do

If SS-Table[SS-R_j,ACK] = False

Then MyChangeStatus = State_i.ChangeStatus $\lor$ SS-Table[SS-R_j,ChangeStatus];

MyChangeDetails = State_i.ChangeDetails $\cup$ SS-Table[SS-R_j,ChangeDetails];

// if SS-S_i did not receive ACK from SS-R_j, then it should re-send

// Refresh to SS-R_j with the new and pending changes

T= GetTimestamp();

Send Refresh(MyChangeStatus,MyChangeDetails,T) To SS-R_j

SS-Table[SS-R_j,Timestamp] = T; // Save timestamp of the latest refresh

SS-Table[SS-R_j,ChangeStatus] = MyChangeStatus;

SS-Table[SS-R_j,ChangeDetails] = MyChangeDetails;

// save the State sent to SS-R_j in SS-Table_i

EndIf
At the Reception of Refresh-Lost(SS-Rj) From SS-Rj Do

SS-Table[i][SS-Rj,ACK] = False;

MyChangeStatus = State.i,ChangeStatus ∨ SS-Table[i][SS-Rj,ChangeStatus];

MyChangeDetails = State.i,ChangeDetails ∪ SS-Table[i][SS-Rj,ChangeDetails];

// if SS-Si should re-send Refresh to SS-Rj with
// the new and pending changes

T = GetTimestamp();

Send Refresh(MyChangeStatus,MyChangeDetails,T) To SS-Rj

SS-Table[i][SS-Rj,Timestamp] = T; // Save timestamp of the latest refresh

SS-Table[i][SS-Rj,ChangeStatus] = MyChangeStatus;

SS-Table[i][SS-Rj,ChangeDetails] = MyChangeDetails;

// save the State sent to SS-Rj in SS-Table

Re-start ACK timer τACK of SS-Si;

End

Table 3.5. SS-Si Algorithm for Reliable Soft State

Table 3.6 gives the algorithm executed by the soft-state receiver SS-Ri. Lines 1-18 are the same as in Pure-SS. At the reception of a Refresh(ChangeStatus,ChangeDetails,T) message from SS-Sj (lines 19-25), SS-Ri sets the Refreshed column of SS-Sj entry in SR-Table to True and sends an ACK(T) message to SS-Sj. The use of the same timestamp t in both messages allows SS-Sj to correlate Refresh() with ACK() messages. SS-Ri processes all changes that may have occurred in SS-Sj during the last SS-Ri cycle (if ChangeStatus is True) by calling the utility procedure Process-Changes().

At the end of τSSR timer (lines 26-39), SS-Ri checks if it received a Refresh() message from each SS-Sj that belongs to Senders(SS-Ri). If so, SS-Ri re-initializes the Refreshed and Retry columns of SS-Sj entry in SR-Tablei. Otherwise (i.e., refresh not received from SS-Sj), SS-Ri increments the Retry column of SS-Sj entry in SR-Tablei and sends a Refresh-Lost() message to SS-Sj. If SS-Sj’s Retry equals Max-Retryi (i.e., SS-Ri did not receive refresh messages from SS-Sj during
Max-Retry_i consecutive SS-R_i cycles), SS-R_i processes WS_j failure by calling the utility procedure Process-No-Refresh(). SS-R_i finally restarts τ_{SSR} timer.

\[
\begin{align*}
(01) \textbf{At} \text{ addition of } WS_j \text{ to } SOA_i \textbf{ Do} \\
(02) \quad & \text{ Send } \text{Join}(SS-R_i) \text{ to } SS-S_j; \\
(03) \quad \textbf{End} \\
(04) \textbf{At} \text{ deletion of } WS_j \text{ from } SOA_i \textbf{ Do} \\
(05) \quad & \text{ Send } \text{Leave}(SS-R_i) \text{ to } SS-S_j; \\
(06) \quad & \text{ Delete } SS-S_j \text{ entry from } SR-Table_i; \\
(07) \quad \textbf{End} \\
(08) \textbf{At} \text{ Reception of Decision2Join}(SS-S_j,\text{decision}) \text{ From } SS-S_j \textbf{ Do} \\
(09) \quad & \text{ If } \text{decision} = \text{True} \\
(10) \quad \quad & \text{ Then } \text{Senders}(SS-R_i) = \text{Senders}(SS-R_i) \cup \{SS-S_j\}; \\
(11) \quad \quad & \quad \text{SR-Table}[SS-S_j,\text{Refreshed}] = \text{False}; \\
(12) \quad \quad & \quad \text{SR-Table}[SS-S_j,\text{Retry}] = 0; \\
(13) \quad \quad & \quad \text{If } |\text{Senders}(SS-R_i)| = 1 \\
(14) \quad \quad \quad & \quad \text{ Then } \text{Start } \tau_{SSR} \text{ timer of } SS-R_i; \\
(15) \quad \quad \text{ Endif} \\
(16) \quad \quad & \quad \text{Else } \text{Process-Refusal}(SS-S_j); \\
(17) \quad \quad \text{ Endif} \\
(18) \quad \text{ End} \\
(19) \textbf{At} \text{ Reception of } \text{Refresh}(\text{ChangeStatus}, \text{ChangeDetails},T) \text{ From } SS-S_j \textbf{ Do} \\
(20) \quad & \quad \text{SR-Table}[SS-S_j,\text{Refreshed}] = \text{True}; \\
(21) \quad & \quad \text{Send } \text{ACK}(T) \text{ To } SS-S_j; \\
(22) \quad & \quad \text{If } \text{ChangeStatus} = \text{True} \\
(23) \quad \quad & \quad \text{Then } \text{Process-Changes}(SS-S_j,\text{ChangeDetails}); \\
(24) \quad \quad \text{ Endif} \\
(25) \quad \text{ End} \\
(26) \textbf{At} \text{ the end of } \tau_{SSR} \text{ timer of } SS-R_i \textbf{ Do} \\
(27) \quad & \quad \text{For each } SS-S_j / SS-S_j \in \text{Senders}(SS-R_i) \textbf{ Do} \\
(28) \quad \quad & \quad \text{If } \text{SR-Table}[SS-S_j,\text{Refreshed}] = \text{True} \\
(29) \quad \quad \quad & \quad \text{Then } \text{SR-Table}[SS-S_j,\text{Refreshed}] = \text{False}; \\
(30) \quad \quad & \quad \quad \text{SR-Table}[SS-S_j,\text{Retry}] = 0;
\end{align*}
\]
Else SR-Table[SS-Sj, Retry]++; // Refreshed Not Received

Send Refresh-Lost(SS-Ri) To SS-Sj;

If SR-Table[SS-Sj,Retry] = Max-Retryi

Then Process-No-Refresh(SS-Sj); //Process the Failure

EndIf

EndFor

Re-start \( \tau_{SSR} \) timer of SS-Ri;

Table 3.6 SS-Ri Algorithm for Reliable Soft State

Example 3.4

In the scenario shown in Figure 3.8, we assume that Max-Retry1 and Max-Retry2 are equal to 3. We also assume that no changes occur in WS3. At time \( t_1 \), SS-S3 sends Refresh\((t_1)\) messages to SS-R1 and SS-R2. Let us assume that the Refresh\((t_1)\) sent to SS-R2 is lost. SS-R1 responds by sending ACK\((t_1)\) to SS-S3 at time \( t_{11} \). At the end of \( \tau_{ACK} \) timer (time \( t_2 \)), SS-S3 re-sends Refresh\((t_2)\) to SS-R2 which has also been lost. At the end of \( \tau_{SSS} \) timer (time \( t_3 \)), SS-S3 sends Refresh\((t_3)\) messages to SS-R1 and SS-R2. We assume that Refresh\((t_3)\) sent to SS-R2 is lost again. SS-R1 responds by sending ACK\((t_3)\) to SS-S3 at time \( t_{12} \). At the end of \( \tau_{ACK} \) timer (time \( t_4 \)), SS-S3 re-sends Refresh\((t_4)\) to SS-R2. Let us assume that this message is delayed. At the end of \( \tau_{SSR} \) timer (time \( t_5 \)), SS-R2 informs SS-S3 about the non-reception of Refresh() messages. At the same time (\( t_5 \)), SS-S3 sends Refresh\((t_5)\) messages to SS-R1 and SS-R2. At times \( t_{21} \) and \( t_{22} \), SS-R2 receives Refresh\((t_4)\) and Refresh\((t_5)\) and hence sends back ACK\((t_4)\) and ACK\((t_5)\) to SS-S3. SS-S3 ignores ACK\((t_4)\) since it is obsolete. Let us now assume that Refresh\((t_5)\) sent to SS-R1 is lost. At the end of \( \tau_{SSS} \) timer (time \( t_6 \)), SS-S3 re-sends Refresh\((t_6)\) to SS-R1. At time \( t_{13} \), SS-R1 receives Refresh\((t_6)\) and sends ACK\((t_6)\) to SS-S3.

\[ \Box \]
3.6 Implementation

As a proof of concept, we provide prototype implementations of the soft-state protocols. We used Microsoft Windows Server 2003 (operating system), Microsoft Visual Studio 8 (development kit), UDDI Server, IIS Server, and SQL Server (for history tables). We deployed ten (10) receivers and twenty (20) senders (implemented in C#) and registered them in UDDI. In the current prototype, all soft-state senders and receivers are deployed in the same machine (Intel(R) processor, 1500MHz, and 512MB of RAM).
Figure 3.9. Generating Soft-State Senders

Figure 3.9 depicts how to generate a new sender in the system. SS_S10 has been created under ParticipantService category (the top left screenshot). SS_S10.cs and SS_S10.asmx have been added automatically into the solution explorer of the project (the top right screenshot). Also, SS_S10 has been registered in UDDI service with the access point of the sender (the bottom screenshot).

Figure 3.10 depicts the soft state sender and soft state receiver in pure soft state. The top left screenshot shows “Start Service” where SS-R starts, “Get Update” where SS-S sends updates and SS-R can review them, “Remove Service” From SS-R when SS-R wants to stop a service from its system, “Add Service” to SS-R when SS-R wants to add a new service to its system, “Failure Simulation” and “Generate Failure” have been used for the experiments, “Refresh Log”
and “Failure Log” used to display the state of SS-S (i.e., whether they are alive or failures occur in their services). The right top screenshot shows that there are two receivers, SS_R1 and SS_R2, and there are senders participating with them. The bottom screenshots shows how to start senders and communicate changes to receivers.

Figure 3.10. Sender and Receiver in Pure Soft State Protocol

SS_R1 and SS_S1 have been started in figure 3.11. SS_S1 and SS_R1 are started in the top left and right screenshots respectively. SS_R1 detects failure in SS_S2 and SS_S3, while SS_R1 received refresh messages from SS_S1 in the bottom screenshots.
Figure 3.11. SS_S1 and SS_R1 have been started

SS_S2 is started in figure 3.12 (the top left screenshot). SS_R1 received messages from SS_S1 and SS_S2, but SS_R1 still detects failure in SS_S3 in the top right screenshot. In the bottom left screenshot, SS_S3 is started. Finally, SS_R1 received messages from all its senders, and hence, no failure detection by SS_R1 in the bottom right screenshot.
Figure 3.12. SS_S2 and SS_S3 are started

Figure 3.13 depicts update occur in Web service 1. SS_S1 detects the changes; it will forward the changes to its receiver via refresh message (the top left screenshot). The changes are in the access point of the sender in the WSDL document. SS_R1 received these changes, it will process them and these changes can be displayed in SS_R1 home page (the top left and bottom screenshots)
Figure 3.13. Changes detected by SS_S1

Figure 3.14 depicts senders and receivers of soft state with explicit removal message. It has all the functionality of pure soft state. In addition, it has “Get Outage” where SS_S schedules to shutdown the service for a certain amount of time, and “Display Removal Message” when a sender wants to stop the service, the sender sends removal message to its receiver about stopping the service (the top left and right screenshot). Also, SS_S has the same functionality of pure soft state with the addition of two things, the first one is when a sender wants to stop its service, it is able to select the receiver from the dropdown list and send the reason why the sender wants to stop the service. Another thing is “Schedule Outage”, when a sender wants to shutdown the service for a certain amount of time, it is able to send down time and up time to its receiver about the schedule outage (the bottom left and right screenshots)
Figure 3.14. SS_R and SS_S of soft state with explicit removal message

Figure 3.15 shows SS_S1 and SS_SR1 have been started (the top left and right screenshots). SS_R1 received messages from SS_S1 while SS_R1 detects failure in SS_S2 and SS_S3 (the bottom left and right screenshots)
Figure 3.15. SS_S1 and SS_R1 are started

Figure 3.16 shows SS_S2 and SS_S3 have been started (the top left and bottom left screenshots). SS_R1 received messages from all its senders, hence no failure detection by SS_R1 (the bottom right screenshot)
Figure 3.16. SS_S2 and SS_S3 are started

Figure 3.17 depicts SS_S2 wants to stop providing the service to SS_R1 (the top left screenshot). SS_S2 sends explicit message why it wants to stop participating with SS_R1 (“not willing to participate with SS_R1). SS_R1 received the message (the top left and bottom screenshots) and it will process stop SS_S2.
Figure 3.17. SS_S2 stops providing the service to SS_R1

Figure 3.18 depicts SS_R1 received refresh messages from all its senders (the top left screenshot). SS_S1 scheduled to stop its service from 06/27/2008 to 06/30/2008 for maintaining the service (the top right screenshot). SS_R1 received the message, and then SS_R1 is not expecting to receive refresh messages from SS_S1 in this period of time since it is out of service (the bottom screenshots)
Figure 3.18. SS_S1 stops the service for a period of time

Figure 3.19 depicts senders and receivers of reliable soft state. SS_R is the same as in soft state with explicit removal (the top screenshots). SS_S is the same as in pure soft state with extra one thing, which is “Inform Notification” (the bottom screenshots). If SS_R did not receive refresh message within its time cycle, it will send notification to SS_S about the lost message. SS_S can display the notification by hitting Get Notification button (bottom left and right).
Figure 3.19. SS_R and SS_R of reliable soft state

SS_S1 and SS_R1 have been started in figure 3.20 (the top left and right screenshots). SS_R1 detects failure in SS_S2 and SS_S3 (the bottom left and right screenshots)
Figure 3.20. SS_S1 and SS_R1 are started

SS_S2 and SS_S3 have been started in figure 3.21 (the top left and bottom left screenshots).
Hence, no failure detected by SS_R1 (the bottom right screenshot)
SS_R1 has been started in figure 3.22 (the top left screen). It detects failure in SS_S1, SS_S2, and SS_S3. SS_R1 then sends notification to them about the lost messages (the rest of screenshots).

**Figure 3.21.** SS_S2 and SS_S3 are started
Figure 3.22. SS_R1 sends notifications to its senders about the lost messages
SS_R1 received messages from SS_S1 in figure 3.23 (the top screenshots), SS_R1 does not send notification to SS_S1, but SS_R1 keeps sending notification to SS_S2 and SS_S3 since SS_R1 does not receive messages from them (the bottom screenshots)

Figure 3.23. SS_R1 sends notification to SS_S2 and SS_S3
3.7 Performance Evaluation

We conducted experiments to assess the different parameters that may impact the performance of the proposed protocol. We created twenty (20) receivers and fifty (50) senders, and registered them in UDDI. We randomly associate (as participants) ten (10) senders to each receiver. The aim of our performance evaluation is to analyze the relationship between \( \tau_{SSR}/\tau_{SSS} \) values and the following three parameters: failure detection time, consistency, and false failures.

Failure detection time is the time it takes for a receiver to detect a failure in its senders. Let us assume that a fault occurred in a sender at time \( t_1 \) and has been detected by a receiver at time \( t_2 \). The failure detection time is equal to \( t_2 - t_1 \). Figure 3.24 compares the average failure detection time for various \( \tau_{SSR} \) timer values (30s, 60s, 90s, 120s, 150s, and 180s), a fixed value of \( \tau_{SSS} \) timer (\( \tau_{SSS}=15s \)), and different (physical) fault ratios. For instance, a fault ratio of 10% means that 1 out of 10 participants within a composite service failed. We create node faults by physically stopping the services corresponding to faulty senders (selected randomly). We simulate transmission faults by randomly setting the entries of a fault flag table to True/False.

As showed by Figure 3.24.a (left side), the smaller is \( \tau_{SSR} \), the shorter is failure detection time. The example depicted in Figure 3.24.b (right side) explains this result. It assumes that all Refresh() messages sent by SS-S (\( \tau_{SSS}=30s \)) are lost (fault ratio = 100%). The fault that occurred at time \( t_2 \) in SS-S is detected by SS-R_1 at time \( t_3 \), SS-R_2 at time \( t_4 \), and SS-R_3 at time \( t_5 \). Failure detection times for SS-R_1, SS-R_2, and SS-R_3 are 60s, 90s, and 120s, respectively.

![Figure 3.24. Impact of SS_R timer on Failure Detection Time](image)
Consistency is the second performance parameter we consider in our study. SS-R and its sender SS-S are consistent at time t if at least one of the Refresh() messages sent by SS-S to SS-R during the last \( \tau_{SSR} \) cycle has been received by SS-R (as stated in lines 28-35 of Table 3.2). For the sake of simplicity, we assume that only transmission faults may occur. Figure 3.25 compares consistency for various \( \tau_{SSS} \) timer values (5s, 10s, 15s, 20s, and 25s) and a fixed value of \( \tau_{SSR} \) timer (\( \tau_{SSR} = 40s \)) for various transmission failure ratios. We measure consistency at the end of each \( \tau_{SSR} \) cycle. As showed by Figure 3.25.a (left side), the smaller is \( \tau_{SSS} \), the higher is consistency. The reason is that a smaller value of \( \tau_{SSS} \) gives receivers more chances to recover from transmission faults. The example depicted in Figure 3.25.b (right side) explains this result. It assumes a transmission failure ratio of 75% (3 messages lost out of 4). For \( \tau_{SSS} = 5s \), there is consistency at times \( t_9 \) and \( t_{17} \) (2 over 2). For \( \tau_{SSS} = 10s \), there is consistency at \( t_7 \) but not at \( t_{13} \) (1 over 2). Finally, for \( \tau_{SSS} = 15s \), there is no consistency at \( t_7 \) and \( t_{13} \) (0 over 2).

False failure is a situation where receivers assume failures that did not occur in their senders. Figure 3.26 analyzes the relationship between false failures and timer difference (i.e., \( \tau_{SSS} - \tau_{SSR} \)). We set \( \tau_{SSR} \) to 20s and vary \( \tau_{SSS} \) from 20s to 25s, 30s, 35s, 40s, etc. Hence, the timer difference takes the following values: 0, 5s, 10s, 15s, 20s, etc. Figure 3.26.a (left side) shows that false failures occur if \( \tau_{SSS} - \tau_{SSR} \geq 0 \) (i.e., \( \tau_{SSS} \geq \tau_{SSR} \)). In addition, the bigger is \( \tau_{SSS} \) (compared to \( \tau_{SSR} \)), the bigger is the number of false failures. These false failures correspond to cases where
Refresh() are sent after the end of the corresponding $\tau_{SSR}$ cycles. For instance (Figure 3.26.a, left side), if $\tau_{SSS}=\tau_{SSR}$, SS-R detects a false failure at $t_2$ (end of $\tau_{SSR}$ cycle) because Refresh() is sent by SS-S at the same time. This Refresh() is processed by SS-R at time $t_3$. If $\tau_{SSS}=2\times\tau_{SSR}$, SS-R detects false failures at $t_2$ and $t_3$ since Refresh() is sent at $t_3$. This Refresh() is processed by SS-R at time $t_4$. At time $t_5$, SS-R detects another false failure since it did not receive Refresh() during the last $\tau_{SSR}$ cycle. If $\tau_{SSS}=5\times\tau_{SSR}$, SS-R detects false failures at $t_2$, $t_3$, $t_4$, $t_5$, and $t_6$ since Refresh() is sent at $t_6$.

![Diagram](a)

**Figure 3.26.** Relationship between SS_S and SS_R Timers

Another experiment was run to compare between pure soft state and soft state with explicit removal in failure detection time. Figure 3.27 compares the average failure detection time for various percentage of physical and freeze fault of pure soft state and soft state with explicit removal. In figure 3.27.a, 100% physical means that all Refresh() messages sent by SS_S are lost. 50% physical and 50% freeze means that 50% Refresh() messages sent by SS_S are lost, and 50% explicit messages sent by SS_S for stopping the service for a certain amount of time. 20% physical and 80% freeze means that 20% Refresh() messages sent by SS_S are lost, and 80% explicit messages sent by SS_S for stopping the service for a certain amount of time. The example depicted in Figure 3.27.b (right side) explains this result, the bigger is physical fault, the longer is failure detection time, and the smaller is freeze fault, the shorter is failure detection time.

83
The last experiment was run to compare the consistency between pure soft state and reliable soft state. Figure 3.28.a shows 100% failure occurred in pure soft state and reliable soft state, it means that all Refresh() messages sent by SS_S are lost. In pure soft state, SS_S is not aware of losing message, and then, the state of losing messages in SS_S and SS_R is different. While, in reliable soft state, SS_S is aware of losing messages, since SS_S will receive ACK when SS_R received Refresh() messages, and SS_S will not receive ACK when SS_R did not receive Refresh() messages. Hence the state of losing messages in SS_S and SS_R in reliable soft state is similar in all cases. Figure 3.28.b (right side) explains this result, consistency in reliable is higher than in pure soft state. Also, the higher is failure, the lower is consistency, and the lower is failure, the higher in consistency in pure soft state.
Figure 3.28. Consistency in Pure Soft State and Reliable Soft State
Chapter 4.

Notification Protocols for
Service-Oriented Architectures

4.1 Introduction

In this chapter, we propose a family of notification protocols to deal with content-related changes in. We introduce two notification approaches for service-oriented architectures: explicit and implicit. In explicit notification (also known as publish-subscribe), information is provided to only those services that explicitly showed their interest in receiving it (via subscription). In implicit notification, information is provided to consumers that may benefit from receiving that information even if they did not explicitly request it via subscription. Implicit notification uses the concept of ontology [BHL01, F03] to model possible notification patterns that exist among Web services. For each approach, we propose three protocols to deal with double notifications and notification loops: centralized notification, distributed notification, and header-based notification.

The remainder of this Chapter is organized as follows. In Section 4.2, we describe our notification model for service-oriented architectures. In Section 4.3, we introduce four protocols for implicit and explicit notifications: naïve, centralized, distributed, and header-based. In Section 4.4, we describe two prototype implementations: disaster recovery and controlling invasive species in agriculture. Section 4.5 is devoted to experiments and performance analysis.
4.2 The Notification Model

In this section, we propose the notification model for service-oriented architectures. We first define explicit and implicit notifications. Then, we introduce the concept of notification ontology to model interaction patterns among Web services. Finally, we illustrate the way Web services exchange messages through administrative services called notifiers.

4.2.1 Explicit and Implicit Notifications

The ultimate goal of Web service is to automate application to application interactions on the Web. Web Service interactions can be either push- or pull-based. In pull-based interactions (or on demand), information is communicated only when it is requested. For example, when users make credit card payments for flight reservations on the Web, a Web Service could be invoked in the background to validate and process such payments. In push-based interactions (also called notifications), information is communicated at the occurrence of an event. For example, a Hurricane Research Center might want to notify all Life Support organizations as soon as it predicts the occurrence of a Hurricane; an airline might want to notify all Travel Agencies about flight deals as soon as the deal is launched. While pull-based interactions are generally one-to-one, notifications may be one-to-many. In one-to-many interactions, a Web service (e.g., the Web Service hosted by the Hurricane Research Center) notifies multiple Web services (e.g., Web Services hosted by all Life Support organizations across a country) about the occurrence of an event (e.g., hurricane prediction).

We identify two types of notifications: implicit and explicit. In explicit notification (also known as publish-subscribe), information is provided to only those consumers that explicitly showed their interest in receiving it (via subscription). The publish-subscribe paradigm has received considerable attention over recent years \[BR04\]. It provides a loosely coupled form of interaction that is well suitable for large-scale distributed systems. Publish-subscribe systems are generally categorized as topic-based or content-based \[EFG+03\]. In topic-based systems, each event belongs to one of a fixed set of subjects (also called topics); publishers are required to label each event with a topic name. In content-based systems, events are no longer divided into different subjects. The subscriber defines a subscription condition according to the internal structure of events; all events that meet the condition will be sent to the subscriber. Our focus in
this chapter is on topic-based notifications. In implicit notification, information is provided to consumers that may benefit from receiving that information even if they did not explicitly request it via subscription. In the rest of this section, we describe the proposed model for enabling implicit notifications in service-oriented architecture.

4.2.2 The Notification Ontology

We model the different patterns through which Web services exchange information through the notification ontology. The concept of ontology [BHL01, F03] was initially developed in Artificial Intelligence to facilitate knowledge sharing and reuse. It has since then been recognized as a popular research topic in various research communities such as knowledge engineering, e-commerce, natural language processing, cooperative information systems, and information integration. Ontology is a formal and explicit specification of a shared conceptualization. “Conceptualization” refers to an abstraction of a domain that identifies the relevant concepts in that domain. “Shared” means that the ontology captures consensual knowledge. “Explicit” means that the concepts used in the ontology and the constraints on their use are explicitly defined. “Formal” intends that the ontology should be machine understandable and described using a well-defined model or language called ontology language (such as RDF - Resource Description Framework [F03]).

We model the notification ontology as an RDF graph. RDF data model consists of statements about resources (i.e., all things that have an identifier on the Web), encoded as object-attribute-value characteristics [F03]. The objects are resources, attributes are properties, and values are resources or strings. At an abstract level, an RDF graph can be seen as a labeled directed graph where nodes represent concepts (i.e., objects and values) and labeled edges represent relationships between concepts (i.e., attributes). In the notification ontology, concepts refer to different service categories (i.e., domain of interest). An edge from C₁ to C₂ labeled with T means that partners that belong to category C₁ shares information of topic T with partners that belong to category C₂.

The categories (i.e., nodes) and topics (edges) of the notification ontology are defined according to two taxonomies: Info Taxonomy and Service Taxonomy. The Info taxonomy gives the various topics of messages that may be exchanged among Web services. A message M is defined by the
couple \((T,D)\) where \(T\) is a topic from the Info Taxonomy and \(D\) is the actual data to be sent. Topics may be recursively organized into subtopics. We use the notation \(\text{Subtopics}(T)\) to refer to all subtopics under \(T\) (children and descendents) within the Info taxonomy. The service taxonomy gives the categories of services that may need to exchange messages. A service may belong to more than one category. A category may be recursively organized into subcategories. We use the function \(\text{Subcategories}(C)\) to refer to all subcategories under \(C\) (children and descendents).

Figure 4.1. Example of Notification Ontology.

Figure 4.1 gives an example of notification ontology. It depicts five service categories: \(C_1\), \(C_2\), \(C_3\), \(C_4\), and \(C_5\). It states that services of category \(C_1\) notify services of categories \(C_2\) and \(C_3\) about messages of topic \(T_1\) and services of categories \(C_4\) and \(C_5\) about messages of topic \(T_2\). Services of category \(C_3\) notify services of categories \(C_6\) and \(C_7\) about messages of topic \(T_1\).

The notification ontology is distributed over the different notifiers in the system. Each notifier is only aware of the part of the ontology related to its category; if a notifier has \(C_i\) as a category, then it is only aware of the relationships (or edges) that have \(C_i\) as a source or target. For instance, a notifier of category \(C_1\) (Figure 4.1) knows that it should send messages of topic \(T_1\) to notifiers of categories \(C_2\) and \(C_3\). However, it is not aware that notifiers of category \(C_3\) will forward those messages to notifiers of categories \(C_6\) and \(C_7\).
4.2.3 Notifiers

One important feature of the proposed notification protocol is the automatic interaction among services to share topic-based messages. We propose a service-oriented architecture (SOA) to enable such interactions. Each Web service has an administrative service, called notifier, associated to it (Figure 4.2). All notifiers are registered in the service registry (UDDI in our case); they are categorized according to the type of service they are associate to (as defined in the service taxonomy).

![Figure 4.2. Interactions among Web Services](image)

Notifiers define a peer-to-peer network for sharing information among services. Each provider publishes information via its associated notifier. The notifier formats the published information into a message and sends it to other peers using one of the protocols described in Section 4.3. At the reception of a message, a notifier applies the same protocol to forward the message to other notifiers. The WSDL document of each notifier WS\textsubscript{i} includes two operations: Publish() and Notify(). Publish() allows partner-to-notifier interactions. It is invoked by the provider of the service corresponding to WS\textsubscript{i} to disseminate a published info. We say that WS\textsubscript{i} is the root notifier of that message. Notify() allows notifier-to-notifier interactions. It enables the exchange of messages among notifiers.
4.3 Notification Protocols

In this section, we present four protocols for both implicit and explicit notifications: naïve notification, centralized notification, distributed notification, and header-based notification.

4.3.1 Naive Notification

Table 4.1 depicts the Naive Implicit Notification (NIN) algorithm executed by a notifier \( WS_i \). At the invocation of \( Publish(T,D) \) or \( Notify(T,D) \), \( WS_i \) calls a utility procedure to process the message. The way this procedure is defined depends on the internal business logic of the service and is out of the scope of this paper. \( WS_i \) then identifies the services that need to be notified about \( M(T,D) \). For that purpose, \( WS_i \) first gets its category \( C_p \); this is done by accessing \( WS_i \) entry in UDDI. \( WS_i \) then determines all edges \( C_p \to C_q \) labeled with \( T \) or a subtopic of \( T \). The reason behind including subtopics is the following: if a service is interested in a topic \( T \), then it is interested in any subtopic of \( T \). \( WS_i \) finally sends \( M(T,D) \) to each notifier \( WS_k \) that belongs to category \( C_q \). \( M(T,D) \) is also forwarded to services that belong to subcategories of \( C_q \).

```plaintext
// T: is the message topic, and D: is the message data

(1) At Invocation of Publish(T,D) or Notify(T,D)
(2) Process(T,D)
(3) Determine Category \( C_p \) of \( WS_i \)
(4) \( \tau = \{T\} \cup \text{Subtopics}(T) \)
(5) For each link \( C_p \to C_q \) labeled with a topic from \( \tau \) Do
(6) \( C = \{C_q\} \cup \text{Subcategories}(C_q) \)
(7) For each \( WS_k \) of a category from \( C \) Do
(8) Invoke Notify(T,D) of \( WS_k \)
(9) End For
(10) End For
(11) End
```

| Table 4.1. Naive Implicit Notification Algorithm Executed by \( WS_i \) |

The NIN protocol has two major drawbacks. First, a message may be sent indefinitely in the system (notification loops). Second, a message may be received repeatedly by a notifier (double notifications). To illustrate these drawbacks, let us consider the scenario depicted in Figure 4.3. \( WS_1 \) is registered under \( C_1 \) and \( C_3 \) categories; \( WS_2 \) is registered under \( C_2 \); \( WS_3 \) and \( WS_4 \) are registered under \( C_3 \). Let us assume that \( M(T,D) \) is published to \( WS_1 \). Based on the notification
ontology graph, WS\textsubscript{1} (under C\textsubscript{1}) sends M(T,D) to WS\textsubscript{2} (under C\textsubscript{2}). Then, WS\textsubscript{2} send M(T,D) to services under C\textsubscript{3} (i.e., WS\textsubscript{3}, WS\textsubscript{4}, and WS\textsubscript{1}). WS\textsubscript{1} will again forward M(T,D) to WS\textsubscript{2}, hence creating an infinite notification loop. In the rest of this section, we propose three protocols that deal with notification loops and double notifications.

![Diagram of notification ontology and trace]

**Figure 4.3.** Naive Implicit Notification – A Scenario.

Table 4.2 depicts the explicit version of the Naïve protocol executed by WS\textsubscript{i}. We refer to this protocol as Naïve Explicit Notification (NEN). We will focus on the Notify() operation only. At the invocation of Publish(T,D) or Notify(T,D), WS\textsubscript{i} calls a utility procedure to process the message. WS\textsubscript{i} forwards M to each notifier WS\textsubscript{k} that subscribed with WS\textsubscript{i} on topic T.

```plaintext
// T: is the message topic, and D: is the message data

(1) At Invocation of Notify(T,D)
(2) Process(T,D)
(3) \( \tau = \{T\} \cup \text{Subtopics}(T) \)
(4) For each WS\textsubscript{k} \in \text{Subscriber}(t) | t \in \tau Do
(5) Invoke Notify(T,D) of WS\textsubscript{k}
(6) EndFor
(7) End

Table 4.2. Naive Explicit Notification Algorithm Executed by WS\textsubscript{i}
```

Let us illustrate the NEN protocol through the scenario depicted in Figure 4.4. Since WS\textsubscript{1} is the first to receive the message M(T,D), it forwards M to its subscribers WS\textsubscript{2} and WS\textsubscript{3}. WS\textsubscript{2} has
WS₄ and WS₁ as subscribers on topic T. It forwards M to WS₄ and WS₁. Since there is no control to stop sending M to the subscribers, it will go to infinite loop.

![Diagram](image)

**Figure 4.4.** Naive Explicit Notification – A Scenario

### 4.3.2 Centralized Notification

The Centralized Implicit Notification (CIN) protocol uses a centralized history to keep track of all messages along with their consumers and producers. The use of a centralized history ensures that each notifier receives a message at most once. This allows avoiding double notifications and notification loops.

Table 4.3 depicts the CIN algorithm executed by a notifier WSᵢ. At the invocation of Publish(ID,T,D) or Notify(ID,T,D), WSᵢ calls a utility procedure to process the message. If the operation invoked is Publish(ID,T,D), then WSᵢ (the root notifier) generates a unique message ID. We use UUIDs (Universal Unique Identifier) as message IDs; UUIDs are 128-bit numbers used to uniquely identify an object or entity on the Internet. IDs are carried by messages throughout the notification process. Since WSᵢ is the producer of M(ID,T,D), it stores (ID,*, WSᵢ) in the history. WSᵢ then determines all services that need to be notified about M(ID,T,D). The same statements are executed by WSᵢ at the invocation of Notify(ID,T,D). WSᵢ first gets all edges Cᵢ → Cᵢ labeled with T or a subtopic of T (Cᵢ is the category of WSᵢ) from the notification ontology graph. WSᵢ then determines the list of all services with Cᵢ or a subcategory of Cᵢ as a category. WSᵢ forwards M(ID,T,D) to WS_k (by calling the operation WS_k.Notify(ID,T,D)) only if WS_k has not received this message yet; this is done by checking...
that neither \((\text{ID}, *, \text{WS}_k)\) nor \((\text{ID}, \text{WS}_k, *)\) belongs to the history. \text{WS}_i\ finally inserts \((\text{ID}, \text{WS}_k, *)\) in the history.

// ID: is a unique message ID, T: is the message topic, and D: is the message data

(1) \textbf{At} Invocation of Publish(\text{ID},T,D) or Notify(\text{ID},T,D)
(2) \textbf{Process}(\text{ID},T,D)
(3) \textbf{If} the operation invoked is Publish(\text{ID},T,D)
(4) \quad \textbf{Then} Generate a unique message ID \quad \quad // \text{WS}_i\ is the producer of M
(5) \quad \quad \quad \quad \textbf{Insert} (\text{ID},*,\text{WS}_i) \textbf{in History}
(6) \textbf{End If}
(7) \textbf{Determine} Category \text{C}_p\ of \text{WS}_i
(8) \quad \tau = \{T\} \cup \text{Subtopics}(T)
(9) \quad \quad \textbf{For each link} \text{C}_p \rightarrow \text{C}_q\ labeled with a topic from \tau \textbf{Do}
(10) \quad \quad \quad \text{C} = \{\text{C}_q\} \cup \text{Subcategories}(\text{C}_q)
(11) \quad \quad \quad \quad \textbf{For each} \text{WS}_k\ of a category from \text{C} \textbf{Do}
(12) \quad \quad \quad \quad \quad \textbf{If} (\text{ID},\text{WS}_k,*) \in \text{History} \quad (\text{ID},*,\text{WS}_k) \in \text{History}
(13) \quad \quad \quad \quad \quad \quad \textbf{Then} Ignore \text{WS}_k \quad // \text{WS}_k\ has already been producer or consumer of M
(14) \quad \quad \quad \quad \quad \quad \quad \textbf{Else}
(15) \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{Insert} (\text{ID},\text{WS}_k,*) \textbf{in History}
(16) \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{Invoke Notify}(\text{ID},T,D)\ of \text{WS}_k
(17) \quad \quad \quad \quad \quad \textbf{End If}
(18) \quad \quad \quad \quad \textbf{End For}
(19) \quad \textbf{End For}
(20) \textbf{End}

\begin{table}[h]
\centering
\begin{tabular}{|c|}
\hline
\textbf{Table 4.3. Centralized Implicit Notification Algorithm Executed by WS}_i \\
\hline
\end{tabular}
\end{table}

To illustrate CIN protocol, let us consider the scenario depicted in Figure 4.5. \text{WS}_1\ is registered under \text{C}_1\ category; \text{WS}_2\ is registered under \text{C}_1\ and \text{C}_2; \text{WS}_3\ is registered under \text{C}_2; \text{WS}_4\ and \text{WS}_5\ are registered under \text{C}_3.\ Let us assume that the information \((T,D)\) is published to \text{WS}_1. \text{WS}_1\ generates a unique message ID equals to 1 and inserts \((1, *, \text{WS}_1)\) in the history. Based on the notification ontology, \text{WS}_1\ figures out that \text{WS}_1, \text{WS}_2, \text{WS}_3, \text{WS}_4,\ and \text{WS}_5\ are candidates to receive \text{M}(1,T,D).\ However, the history shows that \text{WS}_1\ has already received the message as a producer. Hence, \text{WS}_1\ sends \text{M} to \text{WS}_2, \text{WS}_3, \text{WS}_4,\ and \text{WS}_5\ only. It also inserts \((1, \text{WS}_2, *)\), \((1, \text{WS}_3, *)\), \((1, \text{WS}_4, *)\), and \((1, \text{WS}_5, *)\) in the History.
Table 4.4 depicts the Centralized Explicit Notification (CEN) algorithm executed by WS_i. We will focus on the Notify() operation only. At the invocation of Publish(ID,T,D) or Notify(ID,T,D), WS_i calls a utility procedure to process the message. If the operation invoked is Publish(ID,T,D), then WS_i (the root notifier) generates a unique message ID, and, WS_i inserts (ID, *, WS_i) in History. For each subscriber WS_k with WS_i on τ, WS_i checks if WS_k exists in History as a consumer or producer (i.e., WS_k already received M). If yes, WS_i skips WS_k. Otherwise, WS_i inserts (ID, WS_k, *) in the History (i.e., WS_k is a consumer of M) and forwards M to WS_k.

---

### Figure 4.5. Centralized Implicit Notification – A Scenario.

---

<table>
<thead>
<tr>
<th>History</th>
<th>Message</th>
<th>Consumer</th>
<th>Producer</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>WS_i</td>
<td>*</td>
<td>WS_i</td>
</tr>
<tr>
<td>1</td>
<td>WS_i</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>WS_i</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>WS_k</td>
<td>*</td>
<td></td>
</tr>
</tbody>
</table>

---

// ID: is a unique message ID, T: is the message topic, and D: is the message data

(1) **At** Invocation of Publish(ID,T,D) or Notify(ID,T,D)
(2) **Process** (ID,T,D)
(3) **If** the operation invoked is Publish(ID,T,D)
(4) **Then** Generate a unique message ID // WS_i is the producer of M
(5) **Insert** (ID,*,WS_i) in History
(6) **End If**
(7) τ = {T} \cup Subtopics(T)
(8) **For** each WS_k ∈ Subscriber(t) | t ∈ τ **Do**
(9) **If** (ID,WS_k,*) ∈ History V (ID,*,WS_k) ∈ History
(10) **Then** Ignore WS_k // WS_k has already been producer or consumer of M
(11) **Else**
(12) **Insert** (ID,WS_k,*) into History
(13) **Invoke** Notify(ID,T,D) of WS_k
(14) **EndIf**
(15) **EndFor**
Let us illustrate CEN protocol through the scenario depicted in Figure 4.6. We assume that ID is equal to 1.

**Table 4.4. Centralized Explicit Notification Algorithm Executed by WS$_i$**

Since WS$_1$ is the first to receive the message M(ID, T, D), WS$_1$ does not exist in History. WS$_1$ inserts (ID, *, WS$_1$) into History. WS$_1$ has WS$_2$ as a subscriber to topic T. Since WS$_2$ does not...
exist in History as a consumer or producer, WS₁ inserts (ID, WS₂, *) and forwards M to WS₂. WS₂ has WS₅ and WS₃ as subscribers to topic T. Since WS₅ and WS₃ do not exist in History as a consumer or producer, WS₂ inserts (ID, WS₅, *) and (ID, WS₃, *) into History and forwards M to them (i.e., WS₃ and WS₅). WS₅ has WS₆ as a subscriber to topic T. Since WS₆ does not exist in History as consumer or producer, WS₂ inserts (ID, WS₅, *) into History and forwards M to WS₆. WS₃ has WS₂, WS₄, and WS₆ as subscribers to topic T. Since WS₂ and WS₆ exist in History as consumers, WS₃ does not forward M to WS₂ and WS₆. Since WS₄ does not exist in History, WS₃ inserts (ID, WS₄, *) and forwards M to WS₄. WS₄ does not have subscribers for topic T. This ends the notification process.

4.3.3 Distributed Notification

The use of a centralized history in CIN suffers from two major drawbacks. First the history table node constitutes a single point of failure. Second, each notifier needs to remotely access the history table; this may increase the time required to send notifications. To address these problems, we propose a distributed version of the notification protocols (Table 4.5). We refer to this protocol as Distributed Implicit Notification (DIN). Each notifier WSᵢ maintains a local history called Historyᵢ. Historyᵢ contains the IDs of all messages received by WSᵢ (as a producer or consumer). At the invocation of Publish(ID,T,D) or Notify(ID,T,D), WSᵢ calls a utility procedure to process the message. If the operation invoked is Publish(ID,T,D), then WSᵢ (the root notifier) generates a unique message ID. WSᵢ checks if ID belongs to Historyᵢ. If so, WSᵢ ignores M(ID,T,D). Otherwise, WSᵢ inserts ID in Historyᵢ and forwards M(ID,T,D) using the same algorithm as CIN (lines 10-17).

```plaintext
// ID: is a unique message ID, T: is the message topic, and D: is the message data
(1) At Invocation of Publish(ID,T,D) or Notify(ID,T,D)
(2) Process(ID,T,D)
(3) If the operation invoked is Publish(ID,T,D)
(4) Then Generate a unique message ID         // WSᵢ is the producer of M
(5) End If
(6) If ID ∈ Historyᵢ
g
(7) Then ignore M(ID,T,D)
(8) Else Insert(ID) into Historyᵢ
(9) Determine Category Cᵢ of WSᵢ
```
To illustrate DIN protocol, let us consider the scenario depicted in Figure 4.7. This scenario shows that repeated notifications are still possible in DIN.
However, such repetitions are detected by notifiers; this allows avoiding notification loops. Since WS\textsubscript{1} is the first to receive the information defined by T and D, it generated a unique ID (say 1), inserts ID in History\textsubscript{1} and forwards M(1, T, D) to services that belong to C\textsubscript{1} (i.e., WS\textsubscript{1}) and C\textsubscript{3} (i.e., WS\textsubscript{3}, WS\textsubscript{4}, and WS\textsubscript{5}). At the reception of M, WS\textsubscript{3} inserts ID into History\textsubscript{3} since ID \( \notin \) History\textsubscript{3}. Then, it forwards M to services that belong to C\textsubscript{2} (i.e., WS\textsubscript{1} and WS\textsubscript{2}). Since ID \( \in \) History\textsubscript{1}, WS\textsubscript{1} considers M as a repeated notification and hence ignores the message. Since ID \( \notin \) History\textsubscript{2}, WS\textsubscript{2} inserts ID into History\textsubscript{2}. At the reception of M, WS\textsubscript{4} inserts ID into History\textsubscript{4} since ID \( \notin \) History\textsubscript{4}. Then, it forwards M to services that belong to C\textsubscript{2} (i.e., WS\textsubscript{1} and WS\textsubscript{2}). Since ID \( \in \) History\textsubscript{1} and ID \( \in \) History\textsubscript{2}, WS\textsubscript{1} and WS\textsubscript{2} ignore M. At the reception of M, WS\textsubscript{5} forwards M to WS\textsubscript{1} and WS\textsubscript{2}. Similarly, the message is ignored by both WS\textsubscript{1} and WS\textsubscript{2}.

Table 4.6 depicts the Distributed Explicit Notification (DEN) algorithm executed by WS\textsubscript{i}. At the invocation of Publish(ID,T,D) or Notify(ID,T,D), WS\textsubscript{i} calls a utility procedure to process the message. If the operation invoked is Publish(ID,T,D), then WS\textsubscript{i} (the root notifier) generates a unique message ID. WS\textsubscript{i} checks if ID does not exist in the History\textsubscript{i}, if so then for each subscriber WS\textsubscript{k} with WS\textsubscript{i} on topic T, WS\textsubscript{i} inserts ID in History\textsubscript{i}, and forwards the message to WS\textsubscript{k}. Otherwise ignore the message (i.e. ID already exists in the History\textsubscript{i}, thus it received M) and do not forward the message to its subscribers.

```
// ID: is a unique message ID, T: is the message topic, and D: is the message data

(1) At Invocation of Publish(ID,T,D) or Notify(ID,T,D)
(2) Process(ID,T,D)
(3) If the operation invoked is Publish(ID,T,D)
(4) Then Generate a unique message ID      // WS\textsubscript{i} is the producer of M
(5) End If
(6) If ID \( \in \) History\textsubscript{i}
(7) Then
(8) Insert(ID) into History\textsubscript{i}
(9) \( \tau = \{T\} \cup \text{Subtopics(T)}\)
(10) For each WS\textsubscript{k} \( \in \) Subscriber\textsubscript{i}(t) \( | \) t \( \in \) \( \tau \) Do
(11) Invoke Notify(ID,T,D) of WS\textsubscript{k}
(12) EndFor
(13) Else
(14) Ignore M(ID,T,D)
(15) End If
(16) End
```

Table 4.6. Distributed Explicit Notification Algorithm Executed by WS\textsubscript{i}
Let us illustrate DEN algorithm via the scenario depicted in Figure 4.8. Let us assume that ID is equal to 1. Since WS₁ is the first to receive the message \( M(\text{ID,T,D}) \), History₁ is empty. WS₁ inserts ID into History₁ and forwards M to WS₂ and WS₃. Since History₂ and History₃ are empty, WS₂ and WS₃ insert ID into History₂ and History₃, respectively. WS₂ has two subscribers WS₄ and WS₅ and hence forwards M to WS₄ and WS₅. Since History₄ and History₅ are empty, WS₄ and WS₅ insert ID into History₄ and History₅, respectively. WS₄ has one subscriber WS₅ and hence forwards M to WS₅. Since ID exists in History₅, WS₅ ignores the message. The double notification is detected at this time by WS₅. WS₃ has two subscribers WS₂ and WS₄. It forwards M to WS₂ and WS₄. Since ID exists in History₂ and History₄, ID will not be inserted into History₂ and History₄. The double notifications are detected at this time by WS₂ and WS₄.

**Figure 4.8.** Distributed Explicit Notification – A Scenario
4.3.4 Header-based Notification

The major drawback of distributed notification protocols (implicit and explicit) is the possibility of double notifications. In Header-based Notification, SOAP message header is used to store the names of Web services that received the message. Saving the name of Web services in header prevents repeated notification. A header is composed of two parts (Figure 4.9): Root and Notified. Root contains the name of the root notifier. Notified is a list of all Web services (except the root) that have been notified about the message. Each Web service WS1 willing to send a message to another service WS2 checks whether WS2 received the message or not. If not, WS2 will be inserted in the message header and WS1 forwards the message to WS2. Otherwise, WS1 ignores WS2.

![Diagram of Header-based Notification](image)

**Figure 4.9.** Message Structure in Header-based Notifications

Table 4.7 depicts the Header-based Implicit Notification (HIN) algorithm executed by WSi. At the invocation of Publish(Header,T,D) or Notify(Header,T,D), WSi calls a utility procedure to process the message. If the operation invoked is Publish(Header,T,D), WSi creates a new Header and assigns WSi to the Root of Header.

WSi first gets all edges $C_p \rightarrow C_q$ labeled with T or a subtopic of T ($C_p$ is the category of WSi) from the notification ontology graph. WSi then determines the list of all services (called ServiceList) with $C_q$ or a subcategory of $C_q$ as a category. WSi saves the list of services that
have previously been notified in OldNotified. WSi considers the following two cases for each service WSj in ServiceList, If WSj is the Root or belongs to Hader.Notifier, then WSi ignores WSj. Otherwise, WSi adds WSj to Header.Notified. Finally, WSi forwards the message with the updated header to all services that belong to ServiceList but not to OldNotified ∪ {Root}.

// Header: is the message header, T: is the message topic, and D: is the message data

(1) At Invocation of Publish(Header,T,D) or Notify(Header,T,D)
(2) Process(Header,T,D)
(3) If the operation invoked is Publish(Header,T,D)
(4) Then Create M.Header
(5) M.Header.Root = WSi
(6) End If
(7) ServiceList = ∅
(8) Determine Category C_p of WSi
(9) τ = {T} ∪ Subtopics(T)
(10) For each link C_p → C_q labeled with a topic from τ Do
(11) CategoryFamily = {C_q} ∪ Subcategories(C_q)
(12) For each Cat in CategoryFamily Do
(13) S = get services of category Cat from registry
(14) ServiceList = ServiceList ∪ S
(15) EndFor
(16) EndFor
(17) OldNotified = M.Header.Notified
(18) M.Header.Notified = M.Header.Notified ∪ ServiceList
(19) For each WSj ∈ ServiceList Do
(20) If (WSj ∉ OldNotified) ∧ (WSj ≠ M.Header.Root) Then Invoke Notify(Header,T,D) of WSj Else Ignore WSj
(21) EndIf
(22) EndFor
(23) End

Table 4.7. Header-based Implicit Notification Algorithm Executed by WSi

Let us consider the scenario depicted in Figure 4.10. C1 has WS1, WS2, and WS3. C2 has WS4. And C3 has WS2, WS5, and WS6. Since WS3 is the first to receive the message M(Header,T,D), it initializes M.Header.Root with {WS3}. It also initializes M.Header.Notified with {WS1, WS2, WS4}, and forwards the message and the header to WS1, WS2, and WS4. Since {WS1, WS2, WS4} ∈ M.Header.Notified, WS1 and WS2 do not forward the message to {WS2, WS3, WS4}. WS4
forwards M to WS₅ and WS₆ with the updated M.Header.Notified = {WS₁, WS₂, WS₄, WS₅, WS₆}. However, WS₄ does not forward M to WS₂ because it is already in M.Header.Notified.

Figure 4.10. Header-based Implicit Notification – A Scenario

Table 4.8 depicts the Header-based Explicit Notification (HEN) algorithm executed by WSᵢ. At the invocation of Publish(Header,T,D) or Notify(Header,T,D), WSᵢ calls a utility procedure to process the message. If the operation invoked is Publish(Header,T,D), then WSᵢ creates new Header and assigns WSᵢ to the Root of Header. For each subscriber WSⱼ ∈ Subscriber, on T or a
subtopic of T, WS$_i$ checks if WS$_j$ is not the Root of Header and does not belong to Header.Notified. If so, WS$_i$ adds WS$_j$ to ServiceList (services to be notified). Otherwise, WS$_i$ ignores WS$_j$. WS$_i$ updates Header.Notified by concatenating ServiceList to Header.Notified. Finally, WS$_i$ forwards the message (with the updated header) to each service in ServiceList.

```plaintext
// Header: is the message header, T: is the message topic, and D: is the message data

(1) At Invocation of Publish(Header,T,D) or Notify(Header,T,D)
(2) Process(Header,T,D)
(3) If the operation invoked is Publish(Header,T,D)
(4) Then Create M.Header
(5) M.Header.Root = WS$_i$;
(6) End If
(7) ServiceList = ∅
(8) τ = {T} ∪ Subtopics(T)
(9) For each WS$_j$ ∈ Subscriber$_i$(t) | t ∈ τ Do
(10) If (WS$_j$ ∉ M.Header.Notified) ∧ (WS$_j$ ≠ M.Header.Root)
(11) Then ServiceList = ServiceList ∪ {WS$_j$}
(12) Else Ignore WS$_j$
(13) EndIf
(14) EndFor
(15) M.Header.Notified = M.Header.Notified ∪ ServiceList
(16) For each WS$_j$ ∈ ServiceList Do
(17) Invoke Notify(Header,T,D) of WS$_j$
(18) EndFor
(19) End
```

**Table 4.8.** Header-based Explicit Notification Algorithm Executed by WS$_i$

Let us illustrate HEN protocol through the scenario depicted in Figure 4.11. WS$_1$ has three subscribers to topic T: WS$_2$, WS$_3$, and WS$_4$. Since WS$_1$ is the first to receive the message M(Header,T,D), it initializes M.Header.Root with {WS$_1$}. It also initializes M.Header.Notified with {WS$_2$, WS$_3$, WS$_4$} and forwards the message to WS$_2$, WS$_3$, and WS$_4$. WS$_2$ has WS$_1$ and WS$_5$ as subscribers to topic T. Since {WS$_1$} = M.Header.Root, WS$_2$ forwards M to WS$_5$ only, with the updated M.Header.Notified = {WS$_2$, WS$_3$, WS$_4$, WS$_5$}. WS$_3$ has WS$_4$ as a subscriber to topic T. Since WS$_4$ ∈ M.Header.Notified, WS$_3$ does not forward M to WS$_4$. WS$_5$ has WS$_3$ and WS$_6$ as subscribers to topic T. Since WS$_3$ ∈ M.Header.Notified, WS$_5$ forwards the message to WS$_6$ only, with the updated M.Header.Notified = {WS$_2$, WS$_3$, WS$_5$, WS$_6$}.
4.4 Implementation

As a proof of concept, we provide two prototype implementations of the proposed protocols: disaster recovery and agriculture. We used Microsoft Windows Server 2003 (operating system), Microsoft Visual Studio 8 (development kit), UDDI Server, IIS Server, and SQL Server (for history tables). We deployed twenty (20) representative notifiers (implemented in C#) and registered them in UDDI under the corresponding categories. In the current prototypes, all notifiers are deployed in the same machine (Intel(R) processor, 1500MHz, and 512MB of RAM).

4.4.1 Case Study 1: Disaster Recovery

Disaster recovery system consists of a network of Web Services hosted by various Organizations, such as US Geological Survey Group (USGS), Pacific Earthquake Engineering...

Figure 4.11. Header-based Explicit Notification – A Scenario
Research Center (PEER), American Red Cross, US Coast Guard - Aviation Supplies, etc. Each of these organizations registers its business and Web Services in UDDI. It also publishes news on a variety of topics, such as Information about disaster predictions, Supplier Updates about resource availability, results of Geological research studies, press release information, etc.

We defined two taxonomies, namely *Disaster Info* (Figure 4.12) and *Disaster Partners* (Figure 4.13) taxonomies, to enable the specification of the Disaster ontology. The Disaster Info taxonomy gives the various types of messages that may be exchanged among Disaster partners. The following are the various types of information on Disaster (Figure 4.12): “Disaster Supplier Info”, “Disaster Info”, “Research Publication”, and “Press Release”. Each entry in the taxonomy corresponds to a type of Disaster message and has a unique ID (e.g., 1, 2, 1.1, 2.1.2). A Disaster message M is defined by (T,D) where T is an ID from the Disaster Info Taxonomy and D is the actual data to be sent.

![Disaster Info Taxonomy](image)

**Figure 4.12.** Disaster Info Taxonomy

The Disaster Partners taxonomy gives the categories of partners that may need to exchange Disaster-related information (Figure 4.13). We identify Nine (9) categories of partners: “Global Orgs”, “National Orgs”, “Aviation Supplies”, “Central Data Bank”, “Gov Orgs”, “Medicine Food Supplies”, “News Media”, “Volunteer Orgs”, and “Research Orgs”. Each Disaster partner has a category and is exposed as a Web service.
Any of the organizations participating in this notification system can act as a producer or consumer of information on a particular topic. For example, American Red Cross can publish information about the availability of its Food and Medical facilities in which case it acts as a “Producer”. On the other hand, it can receive information about disaster predictions from PEER in which case it acts as a “Consumer”.

Figure 4.14 depicts screenshots for creating a new notifier named Detroit TV. The system will automatically generate a “DetroitTV.cs” class and “DetroitTV.asmx” file (Figure 4.14, right). It also automatically registers DetroitTV notifier in UDDI along with the access point for invoking the notifier (Figure 4.14, bottom). The notifier is registered under the category specified by the administratot (Figure 4.14, top).
Figure 4.14. Generating Notifiers in the Disaster Prototype

Figure 4.15 depicts screenshots of the DR prototype. The first screenshot (top right) shows the default page displayed by the prototype. It includes a text description of the system and prompts users to select the protocol they are interested in using (Header-based Explicit in this case). Users are then taken to the next page (screenshot in the middle) to select the message topic (Disaster Supplies Info), the name of the root notifier (BlueSkyHauling), and the actual data to be published (“need more supplies in California”). Clicking on the Send button will invoke the Publish() method of BlueSkyHauling notifier. Users will then be taken to the result page (screenshot in the bottom). This page displays the name of the root notifier, the topic and actual data of the message, and the list of services that have been automatically notified (i.e., BBCNews, CDBNews, USPleasantHillGrain, MSNBCNews).
4.4.2 Case Study 2: Agriculture – Emerald Ash Borer

The Emerald Ash Borer (EAB) has killed or infested millions of ash trees in Michigan and is fast spreading to neighboring states. The US Department of Agriculture (USDA) estimates that if EAB went unchecked in the rest of the country, the loss to the nation could range from $20 billion to $60 billion. Ash wood is used for all traditional applications of hardwood from flooring and cabinets to baseball bats. In addition, ash trees are beautiful shade giving trees and one of the commonly used landscaping trees in most cities of North America. EAB has already killed or infested ten million ash trees in Michigan alone and fast spreading to adjoining states. One key requirement for the success of EAB containment programs is the underlying information sharing infrastructure. EAB partners are maintaining Web sites to publish...
information about the borer. However, this approach for sharing information is ad hoc and requires intensive human intervention. In this case study, we applied implicit and explicit notification protocols to facilitate the sharing of information about EAB interested parties. We defined two taxonomies, namely EAB Info (Figure 4.16) and EAB Partners (Figure 4.17) taxonomies, to enable the specification of the EAB ontology. The EAB Info taxonomy gives the various types of messages that may be exchanged among EAB partners. The following are the various types of information on EAB (Figure 4.16): “Potential Damage- Ecological Impact”, “Surveying an Area-GIP”, “Original/History-Infested Area”, “Original/History-Native Area”, “Regular/Quarantine Info-Need of Regulation”, “Biological Info-Physiology”, “Surveying Area-Isolation”, “Regular/Quarantine-Need of Regulation”, “Biological Info-Life Cycle”, and “Potential Damage-Economic”. Each entry in the taxonomy corresponds to a type of EAB message and has a unique ID (e.g., 1, 2, 1.1, 2.1.2). An EAB message M is define by (T,D) where T is an ID from the EAB Info Taxonomy and D is the actual data to be sent.

The EAB Partners taxonomy gives the categories of partners that may need to exchange EAB-related information (Figure 4.17). We identify twelve (12) categories of partners: “Federal Government”, “State Government”, “Local Government”, “University”, “Industry”, “News Paper”, “TV”, “Radio”, “Law Enforcement”, “Law Makers”, “Farmers and Nurseries”, and “Association”. Each EAB partner has a category and is exposed as a Web service.
The EAB ontology can be depicted as a labeled directed graph; nodes represent concepts and labeled edges represent relationships between concepts. Concepts refer to different categories of EAB partners as defined in the EAB Partners taxonomy. An edge from E1 to E2 labeled with TI means that partners that belong to category E1 shares information of type TI (as defined in the EAB Info taxonomy) with partners that belong to category E2.

Figure 4.18 depicts screenshots of the EAB prototype. The first screenshot (top right) shows the default page displayed by the prototype. It includes a text description of the system and prompts users to select the protocol they are interested in using (Header-based Implicit in this case). Users are then taken to the next page (screenshot in the middle) to select the message topic (BiologicalInformation-Physiology), the name of the root notifier (Michigan State University), and the actual data to be published (New 1200 Ash trees have been infected in Canton city). Clicking on the Send button will invoke the Publish() method of Michigan State University notifier. Users will then be taken to the result page (screenshot in the bottom). This page displays the name of the root notifier, the topic and actual data of the message, and the list of...
services that have been automatically notified (i.e., IllinoisDepartmentofAgriculture, IndianaStateDepartment ofAgriculture, Michigan DepartmentofAgriculture).

Figure 4.18. Screenshots of EAB Prototype.

4.5 Experiments

For each case study, we designed ten (10) experiments that represent ten typical scenarios. We ran each experiment for the explicit and implicit versions of three proposed protocols: CN, DN, and HN. We consider both synchronous and asynchronous method invocation calls.
4.5.1 Case Study 1 – Disaster Recovery (DR)

4.5.1.1 Experiment DR 1

In the implicit notification; the topic is Research Publication. IDER behaves as a producer and consumer (figure 4.19). In the explicit notification; the topic is Disaster Info. IDER behaves as a Producer and consumer (figure 4.20). Table 4.9 summarizes the results obtained for experiment 1

![Diagram](image-url)  
**Figure 4.19.** Implicit Notification of Experiment 1

![Diagram](image-url)  
**Figure 4.20.** Explicit Notification of Experiment 1
4.5.1.2 Experiment DR 2

In the implicit notification; the topic is Disaster Info. IDER behaves as a producer and consumer (figure 4.21). In the explicit notification, the topic is Disaster Info. TertaPak behaves as a producer and consumer (figure 4.22). Table 4.10 summarizes the results obtained for experiment 2.

<table>
<thead>
<tr>
<th>Method</th>
<th>Technique</th>
<th>Notification Time</th>
<th>Number of Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Notification CN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>2.5 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>4.0 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.5 s</td>
</tr>
<tr>
<td>Distributed Notification DN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>1.7 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.6 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>2.5 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.9 s</td>
</tr>
<tr>
<td>Header-based Notification HN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>1.4 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.0 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>1.1 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.7 s</td>
</tr>
</tbody>
</table>

Table 4.9. The result of the experiment 1
In the implicit notification, the topic is Disaster Supply Info. Blue Sky Hauling behaves as a producer and consumer (figure 4.23). In the explicit notification, the topic is Disaster Info. CBSNews behaves as a producer and consumer (figure 4.24). Table 4.11 summarizes the results obtained for experiment 3.
Figure 4.23. Implicit Notification of Experiment 3
Table 4.11. The result of the experiment 3

<table>
<thead>
<tr>
<th>Method</th>
<th>Technique</th>
<th>Notification Time</th>
<th>Number of Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Notification CN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>4.8 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.2 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>9.9 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>2.4 s</td>
</tr>
<tr>
<td>Distributed Notification DN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>3.5 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.6 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>7.6 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.2 s</td>
</tr>
<tr>
<td>Header-based Notification HN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>4.1 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.0 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>6.1 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.2 s</td>
</tr>
</tbody>
</table>

4.5.1.4 Experiment DR 4

In the implicit notification, the topic is Disaster Info. USCoastGaurd behaves as a producer and consumer (figure 4.25). In the explicit notification, the topic is Press Release. IDER behaves as a producer and consumer (figure 4.26). Table 4.12 summarizes the results obtained for experiment 4.
Figure 4.25. Implicit Notification of Experiment 4
**Figure 4.26.** Explicit Notification of Experiment 4

**Table 4.12.** The result of the experiment 4

<table>
<thead>
<tr>
<th>Method</th>
<th>Technique</th>
<th>Notification Time</th>
<th>Number of Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Notification CN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>5.7 s</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>13.2 s</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>2.3 s</td>
<td></td>
</tr>
<tr>
<td>Distributed Notification DN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>3.2 s</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>0.6 s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>10.5 s</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.1 s</td>
<td></td>
</tr>
<tr>
<td>Header-based Notification HN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>3.1 s</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>0.7 s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>9.7 s</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>2.2 s</td>
<td></td>
</tr>
</tbody>
</table>

**4.5.1.5 Experiment DR 5**

In the implicit notification; the topic is Press Release. IDER behaves as a producer and consumer (figure 4.27). In the explicit notification, the topic is Research Publication. IDER behaves as a producer and consumer (figure 4.28). Table 4.13 summarizes the results obtained for experiment 5.
Figure 4.27. Implicit Notification of Experiment 5

Figure 4.28. Explicit Notification of Experiment 5
Table 4.13. The result of the experiment 5

### 4.5.1.6 Experiment DR 6

In the implicit notification; the topic is Press Release. USVolunteerAssoc behaves as a producer and consumer (figure 4.29). In the explicit notification, the topic is Press Release. AmericanRedCross behaves as a producer and consumer (figure 4.30). Table 4.14 summarizes the results obtained for experiment 6.
**Figure 4.29.** Implicit Notification of Experiment 6

**Figure 4.30.** Explicit Notification of Experiment 6
Table 4.14. The result of the experiment 6

<table>
<thead>
<tr>
<th>Method</th>
<th>Technique</th>
<th>Notification Time</th>
<th>Number of Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Notification CN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>5.6 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.7 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>13.2 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>2.3 s</td>
</tr>
<tr>
<td>Distributed Notification DN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>2.8 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.9 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>10.7 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.2 s</td>
</tr>
<tr>
<td>Header-based Notification HN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>2.8 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.1 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>8.3 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.4 s</td>
</tr>
</tbody>
</table>

4.5.1.7 Experiment DR 7

In the implicit notification; the topic is Research Publication. USGS behaves as a producer and consumer (figure 4.31). In the explicit notification, the topic is Disaster Supply Info. AtlantaCDCP behaves as a producer and consumer (figure 4.32). Table 4.15 summarizes the results obtained for experiment 7

Figure 4.31. Implicit Notification of Experiment 7
Table 4.15. The result of the experiment 7

<table>
<thead>
<tr>
<th>Method</th>
<th>Technique</th>
<th>Notification Time</th>
<th>Number of Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Notification CN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>3.3 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>2.9 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.5 s</td>
</tr>
<tr>
<td>Distributed Notification DN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>1.9 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.8 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>2.4 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.9 s</td>
</tr>
<tr>
<td>Header-based Notification HN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>1.7 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.2 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>1.4 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.8 s</td>
</tr>
</tbody>
</table>

4.5.1.8 Experiment DR 8

In the implicit notification; the topic is Disaster Supply Info. USPleasantHillGrain behaves as a producer and consumer (figure 4.33). In the explicit notification, the topic is Disaster Supply Info. BlueSkyHauling behaves as a producer and consumer (figure 4.34). Table 4.16 summarizes the results obtained for experiment 8
Figure 4.33. Implicit Notification of Experiment 8

Figure 4.34. Explicit Notification of Experiment 8
Table 4.17 summarizes the results obtained for experiment 9.

<table>
<thead>
<tr>
<th>Method</th>
<th>Technique</th>
<th>Notification Time</th>
<th>Number of Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Notification CN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>3.9 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.2 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>9.6 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>2.1 s</td>
</tr>
<tr>
<td>Distributed Notification DN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>2.9 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.0 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>7.9 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
</tr>
<tr>
<td>Header-based Notification HN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>2.9 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.7 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>6.3 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
</tr>
</tbody>
</table>

**Table 4.16. The result of the experiment 8**

### 4.5.1.9 Experiment DR 9

In the implicit notification, the topic is Press Release. USPEER behaves as a producer and consumer (figure 4.35). In the explicit notification, the topic is Press Release. AtlantaCDCP behaves as a producer and consumer (figure 4.36). Table 4.17 summarizes the results obtained for experiment 9.

![Diagram](image-url)  
**Figure 4.35. Implicit Notification of Experiment 9**
Figure 4.36. Explicit Notification of Experiment 9

<table>
<thead>
<tr>
<th>Method</th>
<th>Technique</th>
<th>Notification Time</th>
<th>Number of Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Notification CN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>3.9 s</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.6 s</td>
<td>4</td>
</tr>
<tr>
<td>Implicit</td>
<td>Synchronous</td>
<td>4.0 s</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.6 s</td>
<td>5</td>
</tr>
<tr>
<td>Distributed Notification DN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>3.0 s</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.0 s</td>
<td>5</td>
</tr>
<tr>
<td>Implicit</td>
<td>Synchronous</td>
<td>3.8 s</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>0.9 s</td>
<td>6</td>
</tr>
<tr>
<td>Header-based Notification HN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>2.7 s</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.0 s</td>
<td>4</td>
</tr>
<tr>
<td>Implicit</td>
<td>Synchronous</td>
<td>2.7 s</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.0 s</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 4.17. The result of the experiment 9

4.5.1.10 Experiment DR 10

In the implicit notification; the topic is Disaster Info. MSNBCNews behaves as a producer and consumer (figure 4.37). In the explicit notification, the topic is Research Publication. USGS behaves as a producer and consumer (figure 4.38). Table 4.18 summarizes the results obtained for experiment 10.
Figure 4.37. Implicit Notification of Experiment 10.
Figure 4.38. Explicit Notification of Experiment 10

Table 4.18. The result of the experiment 10

<table>
<thead>
<tr>
<th>Method</th>
<th>Technique</th>
<th>Notification Time</th>
<th>Number of Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Notification CN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>5.5 s</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>13.2 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.2 s</td>
</tr>
<tr>
<td>Distributed Notification DN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>4.0 s</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>0.6 s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>10.9 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.1 s</td>
</tr>
<tr>
<td>Header-based Notification HN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>4.4 s</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>0.7 s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>8.3 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
</tr>
</tbody>
</table>

4.5.1.11 Summary

Figure 4.39 depicts the experiments ran implicitly of DR. Figure 4.39.a compares the global notification time (in seconds) of CIN, DIN and HIN synchronous. Notification time is bigger in the case of CIN since all notifiers need to access a remotely located history table. In DIN, each
notifier accesses a local history table. In HIN, there is no table to be accessed; hence HIN takes less time to send notification to other services. Figure 4.39.b compares the global notification time of CIN, DIN and HIN asynchronous. As in the synchronous case, HIN is the fastest to send notification to others. It is interesting to note that notification time is smaller in the case of asynchronous invocations. The reason is that notifications are performed in parallel in asynchronous invocations while they are performed sequentially in synchronous invocations. Figure 4.39.c compares the total number of notifications sent in CIN, DIN and HIN synchronous. Experiments show that DIN generates more notifications than CIN and HIN. As mentioned earlier, the use of CIN and HIN allow avoiding double notifications and notification loops. However, double notifications are still possible in DIN. Figure 4.39.d compares the total number of notifications sent in CIN, DIN and HIN asynchronous. The total number of notifications is similar to the one in the synchronous case. The interaction mode (synchronous vs. asynchronous) affects notification time not the number of notifications.

**Figure 4.39.** Implicit - Notification Time and Number of Notification in CN, DN, and HN
Figure 4.40 depicts the experiments ran explicitly. Figure 4.40.a compares the global notification time (in seconds) of CEN, DEN and HEN synchronous. Notification time is bigger in the case of CEN since all notifiers need to access a remotely located history table. In DEN, each notifier accesses a local history table. In HEN, there is no table to be accessed; hence HEN takes less time to send notification to other services. Figure 4.40.b compares the global notification time of CEN, DEN and HEN asynchronous. As in the synchronous case, HEN is fastest to send notification to others. It is interesting to note that notification time is smaller in the case of asynchronous invocations. The reason is that notifications are performed in parallel in asynchronous invocations while they are performed sequentially in synchronous invocations. Figure 4.40.c compares the total number of notifications sent in CEN, DEN and HEN synchronous. Experiments show that DEN generates more notifications that CEN and HEN. As mentioned earlier, the use of CEN and HEN allow avoiding double notifications and notification loops. However, double notifications are still possible in DIN. Figure 4.40.d compares the total number of notifications sent in CEN, DEN and HEN asynchronous. The total number of notifications is similar to the one in the synchronous case. The interaction mode (synchronous vs. asynchronous) affects notification time not the number of notifications.

**Figure 4.40.** Explicit - Notification Time and Number of Notification in CN, DN, and HN
4.5.2 Case Study 2 – Emerald Ash Borer (EAB)

4.5.2.1 Experiment EAB 1

In the implicit notification; the topic is Surveying an Area-GIP. MADRegulatory Staff behaves as a producer and consumer (Figure 4.41). In the explicit notification; the topic is Potential Damage- Ecological Impact. HousTV behaves as a Producer and consumer (Figure 4.42). Table 4.19 summarizes the results obtained for experiment 1.

Figure 4.41. Implicit Notification of Experiment 1

Figure 4.42. Explicit Notification of Experiment 1
<table>
<thead>
<tr>
<th>Method</th>
<th>Technique</th>
<th>Notification Time</th>
<th>Number of Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Notification CN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Explicit</td>
<td>Synchronous</td>
<td>3.8 s</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
<td>4</td>
</tr>
<tr>
<td>Implicit</td>
<td>Synchronous</td>
<td>3.6 s</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.4 s</td>
<td>4</td>
</tr>
<tr>
<td>Distributed Notification DN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Explicit</td>
<td>Synchronous</td>
<td>2.5 s</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.0 s</td>
<td>5</td>
</tr>
<tr>
<td>Implicit</td>
<td>Synchronous</td>
<td>3.2 s</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
<td>4</td>
</tr>
<tr>
<td>Header-based Notification HN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Explicit</td>
<td>Synchronous</td>
<td>2.6 s</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>0.9 s</td>
<td>4</td>
</tr>
<tr>
<td>Implicit</td>
<td>Synchronous</td>
<td>2.1 s</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>0.9 s</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 4.19. The results of Experiment 1

4.5.2.2 Experiment EAB 2

In the implicit notification; the topic is Original/History-Native Area. CapitalPressNewsPaper behaves as a producer and consumer (figure 4.43). In the explicit notification; the topic is Original/History-Infested Area. BioForestTech behaves as a Producer and consumer (figure 4.44). Table 4.20 summarizes the results obtained for experiment 2.

Figure 4.43. Implicit Notification of Experiment 2
Figure 4.44. Explicit Notification of Experiment 2

Table 4.20. The result of the experiment 2

<table>
<thead>
<tr>
<th></th>
<th>Method</th>
<th>Technique</th>
<th>Notification Time</th>
<th>Number of Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Notification CN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>3.2 s</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>3.2 s</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
<td>2</td>
</tr>
<tr>
<td>Distributed Notification DN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>1.5 s</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.9 s</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>3.1 s</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.0 s</td>
<td>3</td>
</tr>
<tr>
<td>Header-based Notification HN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>1.6 s</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.7 s</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>2.1 s</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.8 s</td>
<td>2</td>
</tr>
</tbody>
</table>

4.5.2.3 Experiment EAB 3

In the implicit notification; the topic is Regular/QuarantineInfo-NeedofRegulation. MichiganForestAss behaves as a producer and consumer (figure 4.45). In the explicit notification; the topic is Original/History-Native Area. EnvironmentPA behaves as a Producer and consumer (figure 4.46). Table 4.21 summarizes the results obtained for experiment 3
Figure 4.45. Implicit Notification of Experiment 3

Figure 4.46. Explicit Notification of Experiment 3
Table 4.21. The result of the experiment 3

<table>
<thead>
<tr>
<th>Method</th>
<th>Technique</th>
<th>Notification Time</th>
<th>Number of Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Notification CN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>3.4 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.2 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>2.8 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
</tr>
<tr>
<td>Distributed Notification DN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>1.9 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.8 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>2.2 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
</tr>
<tr>
<td>Header-based Notification HN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>2.1 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.7 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>1.6 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.9 s</td>
</tr>
</tbody>
</table>

4.5.2.4 Experiment EAB 4

In the implicit notification; the topic is Potential Damage-Ecological Impact. Ann Arbor FPO behaves as a producer and consumer (figure 4.47). In the explicit notification; the topic is Surveying Area-GIP. IllinoiDofA behaves as a Producer and consumer (figure 4.48). Table 4.22 summarizes the results obtained for experiment 4
Figure 4.47. Implicit Notification of Experiment 4

Figure 4.48. Explicit Notification of Experiment 4
Table 4.22. The result of the experiment 4

<table>
<thead>
<tr>
<th></th>
<th>Method</th>
<th>Technique</th>
<th>Notification Time</th>
<th>Number of Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Notification CN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>4.7 s</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.4 s</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>4.6 s</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
<td>6</td>
</tr>
<tr>
<td>Distributed Notification DN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>3.1 s</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.1 s</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>4.1 s</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
<td>7</td>
</tr>
<tr>
<td>Header-based Notification HN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>3.4 s</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.0 s</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>3.1 s</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
<td>6</td>
</tr>
</tbody>
</table>

4.5.2.5 Experiment EAB 5

In the implicit notification; the topic is Original/History-Native Area. CapitalPressNewsPaper behaves as a producer and consumer (figure 4.49). In the explicit notification; the topic is BioInfo-Physiology. Michigan State University behaves as a Producer and consumer (figure 4.50). Table 4.23 summarizes the results obtained for experiment 5
Figure 4.49. Implicit Notification of Experiment 5

Figure 4.50. Explicit Notification of Experiment 5
<table>
<thead>
<tr>
<th>Method</th>
<th>Technique</th>
<th>Notification Time</th>
<th>Number of Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Centralized Notification</strong></td>
<td>Explicit</td>
<td>Synchronous</td>
<td>4.6 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.4 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>3.8 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
</tr>
<tr>
<td><strong>Distributed Notification</strong></td>
<td>Explicit</td>
<td>Synchronous</td>
<td>3.0 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.9 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>3.5 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.2 s</td>
</tr>
<tr>
<td><strong>Header-based Notification</strong></td>
<td>Explicit</td>
<td>Synchronous</td>
<td>2.7 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.0 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>2.5 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.2 s</td>
</tr>
</tbody>
</table>

Table 4.23. the result of the experiment 5

4.5.2.6 Experiment EAB 6

In the implicit notification; the topic is Biological Info-Physiology. AgJournalNewsPaper behaves as a producer and consumer (figure 4.51). In the explicit notification; the topic is Surveying Area-Isolation of Affected Area. AgJournalNewsPaper behaves as a Producer and consumer (figure 4.52). Table 4.24 summarizes the results obtained for experiment 6
Figure 4.51. Implicit Notification of Experiment 6
Figure 4.52. Explicit Notification of Experiment 6

<table>
<thead>
<tr>
<th>Method</th>
<th>Technique</th>
<th>Notification Time</th>
<th>Number of Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Centralized Notification CN</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Explicit</td>
<td>Synchronous</td>
<td>6.8 s</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.1 s</td>
<td>9</td>
</tr>
<tr>
<td>Implicit</td>
<td>Synchronous</td>
<td>6.4 s</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.5 s</td>
<td>9</td>
</tr>
<tr>
<td><strong>Distributed Notification DN</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Explicit</td>
<td>Synchronous</td>
<td>4.6 s</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
<td>10</td>
</tr>
<tr>
<td>Implicit</td>
<td>Synchronous</td>
<td>6.1 s</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.4 s</td>
<td>10</td>
</tr>
<tr>
<td><strong>Header-based Notification HN</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Explicit</td>
<td>Synchronous</td>
<td>4.8 s</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.0 s</td>
<td>9</td>
</tr>
<tr>
<td>Implicit</td>
<td>Synchronous</td>
<td>4.7 s</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.5 s</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 4.24. The result of the experiment 6

4.5.2.7 Experiment EAB 7

In the implicit notification; the topic is Original/History-Native Area. True Green behaves as a producer and consumer (figure 4.53). In the explicit notification; the topic is
Regular/Quarantine-Need of Regulation. True Green behaves as a Producer and consumer (figure 4.54). Table 4.25 summarizes the results obtained for experiment 7.

**Figure 4.53.** Implicit Notification of Experiment 7
**Figure 4.54.** Explicit Notification of Experiment 7

<table>
<thead>
<tr>
<th>Method</th>
<th>Technique</th>
<th>Notification Time</th>
<th>Number of Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Notification CN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>5.7 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.1 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>5.6 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
</tr>
<tr>
<td>Distributed Notification DN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>3.9 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.0 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>5.1 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.2 s</td>
</tr>
<tr>
<td>Header-based Notification HN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>3.8 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.8 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>4.4 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.2 s</td>
</tr>
</tbody>
</table>

**Table 4.25.** The result of the experiment 7

4.5.2.8 Experiment EAB 8

In the implicit notification; the topic is Potential Damage-Economic Impact. Michigan Department of Agriculture behaves as a producer and consumer (figure 4.55). In the explicit notification; the topic is Bio Info-Life Cycle. University of Michigan behaves as a producer and consumer (figure 4.56). Table 4.26 summarizes the results obtained for experiment 8
Figure 4.55. Implicit Notification of Experiment 8

Figure 4.56. Explicit Notification of Experiment 8
Table 4.26. The result of the experiment 8

<table>
<thead>
<tr>
<th>Method</th>
<th>Technique</th>
<th>Notification Time</th>
<th>Number of Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Notification CN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>6.0 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.2 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>6.2 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.5 s</td>
</tr>
<tr>
<td>Distributed Notification DN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>4.0 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.9 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>5.9 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.4 s</td>
</tr>
<tr>
<td>Header-based Notification HN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>4.1 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.8 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>4.6 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
</tr>
</tbody>
</table>

4.5.2.9 Experiment EAB 9

In the implicit notification; the topic is Biological info-Life Cycle. MichiganStateUniversity behaves as a producer and consumer (figure 4.57). In the explicit notification; the topic is Original/History-Native Area. AgJournalNewsPaper behaves as a Producer and consumer (figure 4.58). Table 4.27 summarizes the results obtained for experiment 9

![Figure 4.57. Implicit Notification of Experiment 9](image)
Figure 4.58. Explicit Notification of Experiment 9

<table>
<thead>
<tr>
<th>Method</th>
<th>Technique</th>
<th>Notification Time</th>
<th>Number of Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Notification CN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>2.5 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.0 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>2.6 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>1.1 s</td>
</tr>
<tr>
<td>Distributed Notification DN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>1.5 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.9 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>2.3 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.9 s</td>
</tr>
<tr>
<td>Header-based Notification HN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>1.4 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.6 s</td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>1.5 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asynchronous</td>
<td>0.7 s</td>
</tr>
</tbody>
</table>

Table 4.27. The result of the experiment 9

4.5.2.10 Experiment EAB 10

In the implicit notification; the topic is Biological Info-Life Cycle. Canton Tree Planting Partnership behaves as a producer and consumer (figure 4.59). In the explicit notification; the topic is Regulatory/Quarantine Info-Quarantine Area. MDARregularStaff behaves as a Producer and consumer (figure 4.60). Table 4.28 summarizes the results obtained for experiment 10.
Figure 4.59. Implicit Notification of Experiment 10
**Figure 4.60.** Explicit Notification of Experiment 10

<table>
<thead>
<tr>
<th></th>
<th>Method</th>
<th>Technique</th>
<th>Notification Time</th>
<th>Number of Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Notification CN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>8.0 s</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.5 s</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>6.5 s</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.5 s</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Distributed Notification DN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>5.5 s</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.4 s</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>6.4 s</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.3 s</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Header-based Notification HN</td>
<td>Explicit</td>
<td>Synchronous</td>
<td>5.7 s</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.4 s</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Implicit</td>
<td>Synchronous</td>
<td>6.1 s</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>1.2 s</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.28.** The result of the experiment 10

### 4.5.2.11 Summary

Figure 4.61 depicts the experiments ran implicitly of EAB. Figure 4.61.a compares the global notification time (in seconds) of CIN, DIN and HIN synchronous. Notification time is bigger in the case of CIN since all notifiers need to access a remotely located history table. In DIN, each notifier accesses a local history table. In HIN, there is no table to be accessed; hence HIN takes
less time to send notification to other services. Figure 4.61.b compares the global notification time of CIN, DIN and HIN asynchronous. As in the synchronous case, HIN is fastest to send notification to others. It is interesting to note that notification time is smaller in the case of asynchronous invocations. The reason is that notifications are performed in parallel in asynchronous invocations while they are performed sequentially in synchronous invocations.

Figure 4.61.c compares the total number of notifications sent in CIN, DIN and HIN synchronous. Experiments show that DIN generates more notifications that CIN and HIN. As mentioned earlier, the use of CIN and HIN allow avoiding double notifications and notification loops. However, double notifications are still possible in DIN. Figure 4.61.d compares the total number of notifications sent in CIN, DIN and HIN asynchronous. The total number of notifications is similar to the one in the synchronous case. The interaction mode (synchronous vs. asynchronous) affects notification time not the number of notifications.

Figure 4.61. Implicit - Notification Time and Number of Notification in CN, DN, and HN

Figure 4.62 depicts the experiments ran explicitly. Figure 4.62.a compares the global notification time (in seconds) of CEN, DEN and HEN synchronous. Notification time is bigger in the case of CEN
since all notifiers need to access a remotely located history table. In DEN, each notifier accesses a local history table. In HEN, there is no table to be accessed; hence HEN takes less time to send notification to other services. Figure 4.62.b compares the global notification time of CEN, DEN and HEN asynchronous. As in the synchronous case, HEN is fastest to send notification to others. It is interesting to note that notification time is smaller in the case of asynchronous invocations. The reason is that notifications are performed in parallel in asynchronous invocations while they are performed sequentially in synchronous invocations. Figure 4.62.c compares the total number of notifications sent in CEN, DEN and HEN synchronous. Experiments show that DEN generates more notifications than CEN and HEN. As mentioned earlier, the use of CEN and HEN allow avoiding double notifications and notification loops. However, double notifications are still possible in DIN. Figure 4.62.d compares the total number of notifications sent in CEN, DEN and HEN asynchronous. The total number of notifications is similar to the one in the synchronous case. The interaction mode (synchronous vs. asynchronous) affects notification time not the number of notifications.

![Notification Time - Synchronous Interactions](image)

![Notification Time - Asynchronous Interactions](image)

![Number of Notifications - Synchronous Interactions](image)

![Number of Notifications - Asynchronous Interactions](image)

**Figure 4.62.** Explicit - Notification Time and Number of Notification in CN, DN, and HN
Chapter 5

Conclusion

Service-oriented architecture (SOA) is a recent approach for integrating services in unstable, evolving and highly dynamic environments. SOAs are subject to changes during their lifetime and hence require mechanism for rapidly and consistently detecting those changes. In this Thesis, we proposed two families of protocols for detecting failure-related and content-related changes, respectively. The first family of protocols, called soft-state protocols for SOAs, allows the detection of failure-related changes. We introduced three soft-state protocols for SOAs: pure soft-state, soft-state with explicit removal, and reliable soft-state. The second family of protocols, called notification protocols for SOAs, facilitates the diction of content-related changes in SOAs. We introduced two notification approaches for SOAs: explicit and implicit. For each approach, we proposed three protocols to deal with double notifications and notification loops: centralized notification, distributed notification, and header-based notification. Finally, we developed three (3) prototypes to illustrate the viability of the proposed change detection protocols. The first prototype implements the proposed soft-state protocols. The second and third protocols implement the proposed notification protocols (implicit and explicit) in two application domains: disaster recovery and agriculture (for controlling invasive species). We also conducted experiment studies to compare and analyze the performance of the proposed protocols.

We identify the following directions for future research. First, we propose to explore the issue of predicting changes in SOAs. We will investigate heuristic-based techniques to predict changes in SOAs. These techniques will be based on the hypotheses that (1) a number of computer-related faults have been observed to be preceded by a visible pattern of abnormal behavior and
(2) updates in service policies/data may have causal relationships. Once changes are detected or predicted, there is a need for means to react to those changes. We will investigate techniques for reacting to changes in SOAs. In particular, we will propose a high-level, declarative, and event-driven language for the specification of change reaction policies. The language will rely on four major techniques: replaceability, adaptation, reconfiguration, and recovery. We will explore the use of adopt machine learning mechanisms to assist service providers and composers in specifying change reaction policies.
References


161