Bottom-Up Fault Management in Composite Web Services

Brahim Medjahed[1] and Zaki Malik[2]

[1] Department of Computer and Information Science,
University of Michigan – Dearborn,
brahim@umich.edu
[2] Department of Computer Science,
Wayne State University
zaki@wayne.edu

Abstract. We propose an approach for managing bottom-up faults in composite Web services. We define bottom-up faults as abnormal conditions/defects or changes in component services that may lead to run-time failures in composite services. The proposed approach uses soft-state signaling to propagate faults from components to composite services. Soft-state denotes a class of protocols where state (e.g., whether a service is alive) is constantly refreshed by periodic messages. Its advantages include implicit error recovery and easier fault management, resulting in high availability. We introduce a bottom-up fault model for composite services. Then, we propose a soft-state protocol for bottom-up fault propagation in composite services. Finally, we present experiments to assess the performance of our approach.

Keywords: Service Composition – Fault Management – Bottom-up Fault - Soft-State – Fault Coordinator.

1 Introduction

Service-oriented architecture (SOA) has recently emerged as a promising approach for application integration [1,14]. It utilizes services (commonly Web services) as the building blocks for developing software systems distributed within and across organizations. The primary value of SOA is the ability to (1) reuse pre-developed, autonomous, and independently provided resources (e.g., legacy applications, sensors, databases, storage devices, COTS products) as Web services and (2) combine pre-existing services, called *participants*, into higher level services, called *composite services*, which perform more complex functions [9,15].

Because of the dynamic and volatile nature of SOAs, Web services are subject to unavoidable faults during their lifetime. The relationship between a fault and failure is given in ISO/CD 10303-226 document, where a fault is defined as an abnormal condition or defect at the component, equipment, or sub-system level which may lead to a failure. In their seminal work, Avizienis et al. [2] state that faults cycle between dormant and active states, and a failure occurs when a fault becomes active. In this

H. Mouratidis and C. Rolland (Eds.): CAiSE 2011, LNCS 6741, pp. 597–611, 2011.

paper, we focus on faults that propagate from participants to composite services. We refer to these faults as *bottom-up*. Two examples of bottom-up faults are (1) a shut-down scheduled by a participant's provider for maintenance; and (2) an update to a participant's policy (e.g., new message parameters added to a WSDL specification) that may affect the way that participant is consumed. Hence, composite services must rapidly detect and handle faults in their participants to avoid run-time failures and maintain consistency.

Web services generally use HTTP as the underlying message transport. Hence, they are either guaranteed message delivery or notified if a message was not delivered (e.g., because of a server unavailability). In the latter case, composite services become aware of a fault only at the time they interact with their participants *not* at the time that fault occurred. This may decrease the availability of composite services. Besides, users' requests are pending as long as the composite service did not recover from the fault (e.g., by replacing the faulty participant with an equivalent one). This calls for a framework in which composite services are able to detect and handle bottom-up faults as soon as those faults occur in their participants.

In this paper, we introduce a framework for managing bottom-up faults in composite services. The proposed framework uses *soft-state* signaling to propagate faults from participants to composite services. Soft state denotes a type of protocols where state (e.g., whether a server is alive) is constantly refreshed by periodic messages; state which is not refreshed in time expires [8]. This is in contrast to hard-state where installed state remains installed unless explicitly removed by the receipt of a state-teardown message. Advantages of the soft-state approach include implicit error recovery and easier fault management, resulting in high availability [16]. Soft state was introduced in the late 1980s and has been widely used in various Internet protocols (e.g., RSVP). However, to the best of our knowledge, this work is the first to use soft-state for fault management in composite services. The major contributions of this paper are summarized below:

- We introduce a bottom-up fault model for composite services. The model includes a taxonomy of bottom-up faults, a definition of (composite) service, and peer-peer topology for fault management.
- We propose a soft-state protocol for bottom-up fault propagation.
- We conduct experiments to assess the performance of the proposed framework.

The rest of this paper is organized as follows. In Section 2, we describe the bottom-up fault model. In Section 3, we propose the soft-state protocol for bottom-up fault propagation. In Section 4, we present experiments to assess the performance of the proposed approach. In Section 5, we give a brief survey of related work. We finally provide concluding remarks in Section 6.

2 Fault Model

In this section, we describe our model for bottom-up fault management. We first provide a categorization of bottom-up faults. Then, we define the notion of participant's state. Finally, we introduce a peer-to-peer topology for bottom-up fault management.

2.1 Bottom-up Fault Taxonomy

A fault management approach must refer to a taxonomy that describes the different types of faults that composite services are expected to be able to manage. We identify two types of bottom-up faults: *physical* and *logical* (Fig. 1). *Physical faults* are related to the infrastructure that supports Web services. In this paper, *the underlying communication system is assumed to be failure-free*: there is no creation, alteration, loss, or duplication of messages. However, *node faults* are still possible. A *node fault* occurs if the servers (e.g., application server, Web server) hosting a participant are out of action. *Logical faults* are initiated by service providers; this is in contrast to physical faults which are out of service providers' control. We categorize logical faults as *status change*, *participation refusal*, and *policy change*.

Status change occurs if the service provider explicitly modifies the availability status of its service. The status may be changed through freeze or stop. In the *freeze* fault, providers shut down their services for limited time periods (e.g., for maintenance, unavailability of a product in a supply chain's provider). In the *stop* fault, providers make their services permanently unavailable (e.g., a company going out-of-business).

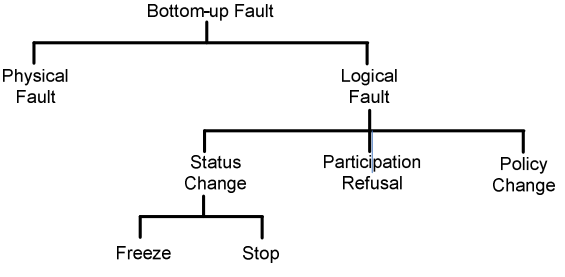

Fig. 1. Bottom-up Fault Taxonomy

Participation refusal occurs if a service is not willing to participate in a given composition. The way participation decision is made varies from a Web service to another. We give below four techniques on how such decisions could be made:

- A service load balancer may check that the server workload will not exceed a given threshold if the service participates in a new composition. The threshold could, for instance, be defined to maintain a minimum quality of service.
- A policy compatibility checker may also verify the compliance of the service policies (e.g. privacy policies) with the composite service policies.
- A service reputation manager may verify that the reputation (e.g., defined by users' ratings) of the composite service is higher than a threshold set by the participant.
- A notification may be sent to the participant's provider who will decide whether the service should participate in the composition or not.

In the rest, we assume the existence of a pre-defined function *Agreed2Join()* used by services to decide whether they are willing to participate in compositions. Each service may provide its own definition and implementation of the *Agreed2Join()* function. However, the way is *Agreed2Join()* defined is out of the scope of this paper.

Policy change 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. We adopt the categorization proposed in [11] considering policies as either vertical or horizontal. The vertical category refers to application domain-dependent policies (e.g., shipping policy in business-to-business e-commerce). The horizontal category refers to policies that are applicable across domains. It is composed of three sub-categories: functional, non-functional, and valued-added. Functional category describes the operational features of a Web service (e.g., in WSDL [1,14], OWL-S [10]). Non-functional category relates to Quality of Service metrics. The value-added category brings "better" environments for Web service interactions. It refers to a set of specifications for supporting optional (but important) requirements for the service (e.g., security, privacy, negotiation, conversation). Changes in the policies of a participant WS_i may impact the way a composite service CS_j interacts with WS_i. Hence, they should be considered as logical faults. For instance, CS_j invocations to WS_i may lead to run-time failures if WS_i provider changed the input message required by WS_i (e.g., new message parameters added, changes made to data types).

2.2 State of a Service

Soft-state signaling enables the propagation of bottom-up faults from participants to composite services. The main idea of this class of signaling is that the *state* of each participant is periodically sent to the composite service. The composite service will then use the received state to determine whether there was any physical or logical fault in the participant. Several questions need to be tackled when designing a soft-state protocol: what is the definition of a state? And how is the state computed? We will give answers to these questions in the rest of this section and paper.

The proposed framework must deal with all types of faults depicted in Fig 2. Physical faults and status changes are detected by composite services in an *implicit* manner; if a node fault occurs or a participant is stopped/frozen, then the composite service will not receive a state from that participant. The participation refusal fault is *explicitly* communicated by participants if they are not willing to be part of a composition.

Policy change faults are transmitted as part of the *participant's state*. To keep track of policy changes, each participant WS_i maintains a data structure called $State_i$ (Fig. 2). $State_i$ is defined by two attributes: *ChangeStatus* and *ChangeDetails*. *ChangeStatus* is equal to True if policy 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 represents a policy change; it is defined by a couple (C,S) where C is the *category* of the policy and S is the *scope* of the change. The initial values of ChangeStatus and ChangeDetails are False and ∅, 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 content of $State_i$ is periodically

communicated to composite services. If a composite service does not receive $State_i$ after a certain period of time, then it assumes a physical fault or status change in WS_i. Otherwise, the composite service reads $State_i$ to find out about the changes made to WS_i.

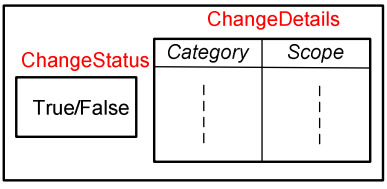

Fig 2. State of a Participant Service

A policy *category* refers to the type of requirements specified by a policy. As mentioned in Section 2.1, it refers to functional, non-functional, value-added, or domain (i.e., vertical) categories. Policies are specified in XML-based Web service languages/standards (e.g., WSDL [1], WS-Security [14]). 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. For instance, let us consider the following WSDL file located in "http://www.ws.com/sq.wsdl":

```
<definitions>
  <types>  <!-- XML Schema -->  </types>
  <message name="getQuote_In"> ….
  <message name="getQuote_Out"> …
  <portType name="StockQuoteServiceInterface">
    <operation name="getQuote">
      <input message="getQuote_In" />
      <output message="getQuote_Out" />
    </operation>
  </portType>
...
```

Let us assume that the name of the operation "getQuote" has been modified in the WSDL document. The category and scope of the change are defined as follows:

- Category = (Functional,WSDL).
- Scope = (URL,Q) where:
  - URL = "http://www.ws.com/stockquote.wsdl"
  - Q = "definitions/portType/operation/@name"

The following definition summarizes the properties of $State_i$ maintained by a participant WS_i.

Definition: The state, denoted $State_i$, of a participant WS_i is defined by (ChangeStatus,ChangeDetails) where:

- ChangeStatus = True ⇔ 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 = ∅.
- At the occurrence of a change (C,S) in WS_i do: $State_i$.ChangeStatus = True, $State_i$.ChangeDetails = $State_i$.ChangeDetails ∪ {(C,S)}. ❑

2.3 Fault Coordinators

In the proposed framework, fault management is a collaborative process between architectural modules called *fault coordinators*. Each Web service (participant or composite) has a coordinator associated to it. This peer-to-peer topology distributes control and externalizes fault management, hence creating a clear separation between the business logic of the services and fault management tasks.

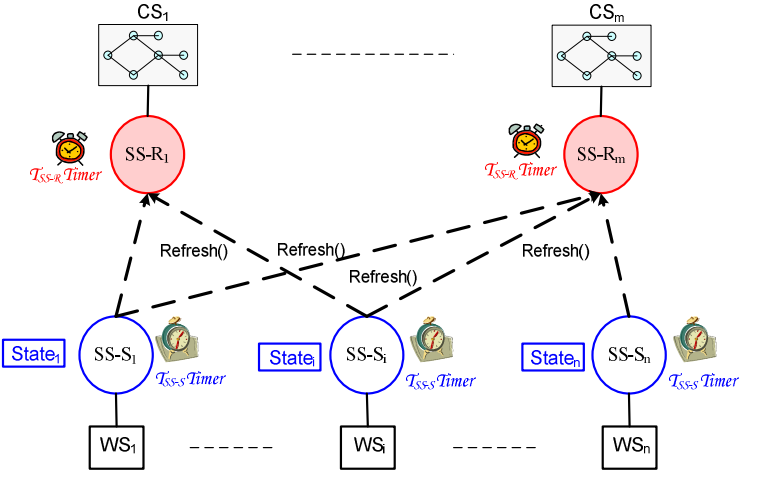

Fig. 3. Fault Coordinators

We define two types of coordinators (Fig. 3): *soft-state senders* (SS-S) and *soft-state receivers* (SS-R). Each participant (resp., composite service) has a sender (resp., receiver) attached to it. A sender $SS\text{-}S_i$ maintains the $State_i$ data structure. To keep track of its receivers, $SS\text{-}S_i$ maintains a Receivers($SS\text{-}S_i$) data structure. If WS_i (attached to $SS\text{-}S_i$) participates in CS_j (attached to $SS\text{-}R_j$) then $SS\text{-}R_j \in$ Receivers($SS\text{-}S_i$). $SS\text{-}S_i$ periodically sends $State_i$ to its receivers via Refresh() messages. The refresh period is determined by the τ_{SSS} timer maintained by $SS\text{-}S_i$. A receiver $SS\text{-}R_j$ maintains two data structures: Senders($SS\text{-}R_j$) and τ_{SSR}. Senders($SS\text{-}R_j$) is the set of senders from which $SS\text{-}R_j$ expects to receive Refresh(). If WS_i participates in CS_j then

$SS\text{-}S_i \in Senders(SS\text{-}R_j)$. τ_{SSR} is a timer used by $SS\text{-}R_j$ to process Refresh() messages received from its senders.

Bottom-up fault management involves three major tasks: *fault detection*, *fault propagation*, and *fault reaction*. The sequence diagram in Fig. 4 depicts the relationship between these tasks:

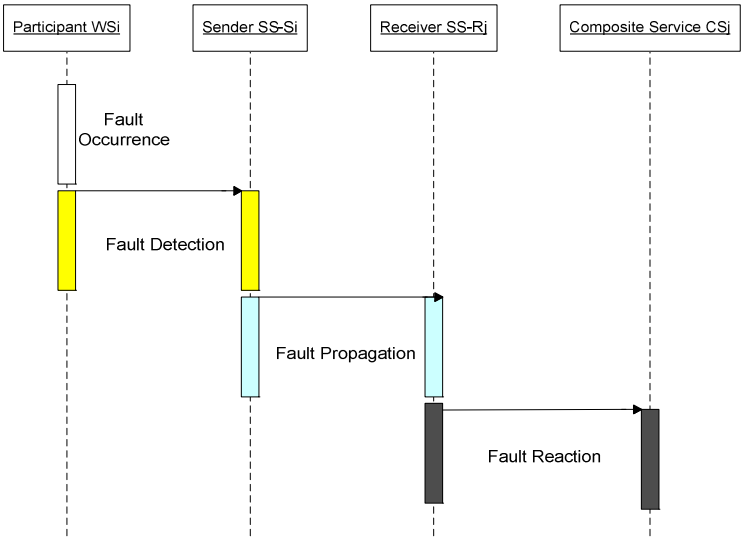

Fig. 4. Sequence Diagram for Bottom-up Fault Management

- *Fault detection*: $SS\text{-}S_i$ first detects faults that occurred in the attached WS_i. $SS\text{-}S_i$ does not need to detect physical and status change faults in WS_i as these are implicitly propagated to receivers if the latter do not receive Refresh() messages from faulty senders. Participation refusal faults are communicated to $SS\text{-}S_i$ using one of the techniques mentioned in Section 2.1. Policy changes in WS_i may be detected by $SS\text{-}S_i$ using various techniques. For instance, $SS\text{-}S_i$ may use XML version control algorithms to detect changes in WS_i policy specifications; another solution is to provide an interface in $SS\text{-}S_i$ through which WS_i provider submits all policy changes; a third solution is to use the publish/subscribe model [3] where $SS\text{-}S_i$ subscribes with WS_i on policy changes. Due to space limitations, details about the detection of participation refusals and policy changes are out of the scope of this paper.
- *Fault propagation*: $SS\text{-}S_i$ then propagates the fault detected in the previous phase to $SS\text{-}R_j$. We propose a soft-state protocol for fault propagation. Details about this protocol are given in Section 3.
- *Fault Reaction*: Finally, SS-Rj and/or CSj execute appropriate measures to react to the fault. The techniques proposed in [12,7] can be extended for that purpose. In [12], we use ECA (Event Condition Actions) to react to changes: the event part of an ECA rule refers to change notifications; the action part allows for the specification of change control policies; the use of conditions allows the specialization of

these policies depending on pre-defined parameters. In [7], we define a Petri Net model for the specification of high-level recovery policies in composite services. These recovery policies are generic directives that model exceptions at design time together with a set of primitive operations used at run time to handle the occurrence of exceptions. The proposed model identifies a set of recovery policies that are useful and commonly needed in many practical situations. Details about the techniques for fault reaction are out of the scope of this paper. We assume the existence of a procedure React (FT, SS) implemented by each composite service to react to faults. The FT parameter is the fault type and takes one of the following values: "Refusal" to refer to a participation refusal fault, "No Refresh" to refer to the non-reception of a Refresh() message (i.e., physical fault or status change), and "policy" to refer to a policy change fault. The SS parameter refers to the sender of faulty participant; the reaction mechanism may in fact vary from a participant to another.

3 The Fault Propagation Protocol

In this section, we describe the algorithms executed by senders and receivers for propagating bottom-up faults. We assume that WS_i (with SS-S_i as attached sender) participates in CS_j (with SS-R_j as attached receiver). As mentioned in Section 2.1, we assume that there is no creation, alteration, loss, or duplication of messages. The propagation protocol adapts the well-know soft-state signaling described in [8,16] to service-oriented environments. It enables the propagation of participation refusal and policy changes faults to receivers. Physical faults and status changes are implicitly propagated to receivers if the latter do not get Refresh() messages from faulty senders.

3.1 Soft-State Sender Algorithm

Table 1 gives the algorithm executed by SS-S_i. SS-S_i receives two types of messages from SS-R_j: Join(SS-R_j) and Leave(). Join(SS-R_j) is the first message that SS-S_i receives from SS-R_j; it invites WS_i to participate in CS_j (lines 1-10). SS-S_i calls the *Agreed2Join()* function to figure out whether WS_i is willing to participate in CS_j (see Section 2.1). If *Agree2Join(SS-R_j)* returns False, 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 receiver of SS-S_i, SS-S_i initializes $State_i$ and starts its τ_{SSS} timer. Finally, SS-S_i sends its decision to SS-R_j through the Decision2Join(SS-S_i,True) message. At any time, SS-S_i may receive a Leave() message from SS-R_j (lines 11-13). This message indicates that CS_j is no longer using WS_i as a participant. In this case, SS-S_i removes SS-R_j from its receivers.

At the detection of a policy change (with a category C and scope S) in WS_i (lines 14-17), SS-S_i sets $State_i$.ChangeStatus to True. SS-S_i keeps track of that change by inserting (C,S) in $State_i$.ChangeDetails. In this way, the state of SS-S_i to be sent to receivers at the end of τ_{SSS} cycle includes all changes that have occurred during that cycle. At the end each period (denoted by τ_{SSS} timer), SS-S_i sends a Refresh()

message to each one of its receivers (lines 18-23). This message includes $State_i$ as a parameter, hence notifying SS-R_j about all policy changes that occurred in WS_i during the last τ_{SSS} period. SS-S_i then reinitializes $State_i$ and restarts its τ_{SSS} timer.

Table 1. Sender's Propagation Algorithm

```
(01) At Reception of Join(SS-Rj) Do
(02)   If Agreed2Join(SS-Rj) = True  Then  Receiversi = Receiversi ∪ {SS-Rj};
(03)         If |Receiversi| = 1 Then  Statei.ChangeStatus = False;
(04)                                   Statei.ChangeDetails = ∅;
(05)                                   Start τSSS timer of SS-Si;
(06)         EndIf
(07)         Send Decision2Join(SS-Si,True) to SS-Rj
(08)   Else  Send Decision2Join(SS-Si,False) to SS-Rj
(09)   EndIf
(10) End
(11) At Reception of Leave(SS-Rj) Do
(12)    Receiversi = Receiversi − {SS-Rj};
(13) End
(14) At the detection of Change(C,S) in WSi Do
(15)   Statei.ChangeStatus = True;
(16)   Statei.ChangeDetails = Statei.ChangeDetails ∪ {(C,S)};
(17) End
(18) At the end of τSSS timer of SS-Si Do
(19)   For each SS-Rj / SS-Rj ∈ Receiversi Do Send Refresh(Statei) to SS-Rj;  EndFor
(20)   Statei.ChangeStatus = False;
(21)   Statei.ChangeDetails = ∅;
(22)   Re-start τSSS timer of SS-Si;
(23) End
```

3.2 Soft-State Receiver Algorithm

The aim of SS-R_j protocol is to detect faults in senders. For that purpose, SS-R_j maintains a local table called *SR-Table*$_j$. SR-Table$_j$ allows SS-R_j to keep track of Refresh() messages transmitted by senders. It contains an entry for each SS-S_i that belongs to Senders(SS-R_j). Each entry contains two columns:

- *Refreshed: SR-Table*$_j$*[SS-S*$_i$*,Refreshed]* equals True iff SS-R_j received a Refresh() from SS-S_i in the current τ_{SSR} cycle.
- *Retry: SR-Table*$_j$*[SS-S*$_i$*,Retry]* contains the number of consecutive cycles during which SS-R_j did not receive Refresh() from SS-S_i.

A temporary node failure in SS-S_i may prevent SS-S_i from sending Refresh() to SS-R_j during a τ_{SSR} cycle. In this case, SS-R_j may want to give SS-S_i a second chance for sending Refresh() during the next τ_{SSR} cycle. For that purpose, SS-R_j maintains a variable (positive integer) *Max-Retry*$_j$. If SS-R_j does not receive Refresh() from SS-S_i during Max-Retry$_j$ consecutive τ_{SSR} cycles, it considers WS_i as faulty. The value of Max-Retry$_j$ is set by CS_j composer and may vary from a composite service to another. The smaller is Max-Retry$_j$, the more pessimistic is CS_j composer about the occurrence of faults in participants.

Table 2. Receiver's Propagation Algorithm

```
(01) At addition of WSi to CSj Do
(02)    Send Join(SS-Rj) to SS-Si;
(03) End
(04) At deletion of WSi from CSj Do
(05)    Send Leave(SS-Rj) to SS-Si;
(06)    Delete SS-Si entry from SR-Tablej;
(07) End
(08) At Reception of Decision2Join(SS-Si,decision) Do
(09)   If decision = True Then  Sendersj = Sendersj ∪ {SS-Si};
(10)                            Create an entry for SS-Si in SR-Tablej;
(11)                            SR-Tablei[SS-Si,Refreshed] = False;
(12)                            SR-Tablei[SS-Si,Retry] = 0;
(13)                            If |Sendersj| = 1 Then Start τSSR timer of SS-Rj EndIf
(14)   Else React("Refusal",SS-Si);
(15)   EndIf
(16) End
(17) At Reception of Refresh(Statei) From SS-Si Do
(18)   SR-Tablei[SS-Si, Refreshed] = True;
(19)   If Statei.ChangeStatus = True  Then React("policy", SS-Si, Statei.ChangeDetails) EndIf
(20) End
(21) At the end of τSSR timer of SS-Rj Do
(22)  For each SS-Si / SS-Si ∈ Sendersj Do
(23)   If SR-Tablej[SS-Si, Refreshed] = True Then  SR-Tablej[SS-Si, Refreshed] = False;
(24)                                               SR-Tablej[SS-Si, Retry] = 0;
(25)   Else  SR-Tablej[SS-Si, Retry]++
(26)         If SR-Tablej[SS-Si,Retry] = Max-Retryj Then
(27)             React("No Refresh", SS-Si);
(28)         EndIf
(29)   EndIf
(30)  EndFor
(31)  Re-start τSSR timer of SS-Rj;
(32) End
```

SS-R_j submits two types of messages to SS-S_i: Join() and Leave(). It also receives two types of messages from SS-S_i: Decision2Join() and Refresh(). Table 2 gives the algorithm executed by SS-R_j. Whenever a new participant WS_i is added to CS_j, SS-R_j sends a Join(SS-R_j) message to SS-S_i (lines 1-3). At the deletion of WS_i from CS_j, SS-R_j sends a Leave(SS-R_j) message to SS-S_i and removes SS-S_i entry from SR-Table$_j$ (lines 4-7). At the reception of Decision2Join(SS-S_i,True), SS-R_j adds SS-S_i to the list of senders (lines 8-16). It also creates a new entry for SS-S_i in SR-Table$_j$ and initializes the Refreshed and Retry columns of that entry to False and 0, respectively. If SS-S_i is the first sender of SS-R_j, SS-R_j starts its τ_{SSR} timer. At the reception of Decision2Join(SS-S_i,False), SS-R_j calls the *React()* procedure to process the participation refusal fault issued by SS-S_i (see Section 2.3).

At the reception of Refresh(State$_i$), SS-R_j sets SR-Table$_j$[SS-S_i,Refreshed] to True (lines 17-20). If State$_i$.ChangeStatus is True, SS-R_j calls the *React()* procedure to process all changes that occurred in SS-S_i during the last τ_{SSR} cycle. At the end of τ_{SSR} timer (lines 21-32), SS-R_j checks if it received Refresh() from each of its senders. If SS-R_j received Refresh() from SS-S_i, it re-initializes the Refreshed and Retry

columns of SS-S_i entry in SR-Table$_j$ to False and 0, respectively. Otherwise, SS-R_j increments SR-Table$_j$[SS-R_j,Retry] If SR-Table$_j$[SS-S_i,Retry] equals Max-Retry$_j$ (i.e., SS-R_j did not receive Refresh() from SS-S_i during Max-Retry$_j$ consecutive τ_{SSR} cycles), SS-R_j assumes a physical (node) fault in SS-S_i and hence, calls the *React()* procedure to process that fault. SS-R_j finally restarts its τ_{SSR} timer.

3.3 Example

Let us consider a Web service WS_3 (with a soft-state sender SS-S_3) that participates in two composite services CS_1 and CS_2 (with soft-state receivers SS-R_1 and SS-R_2, respectively). We assume that $\tau_{SSR1} = \tau_{SSR2} = 2 \times \tau_{SSS3}$. Fig 5 depicts the interactions between SS-S_3 and SS-R_1/SS-R_2.

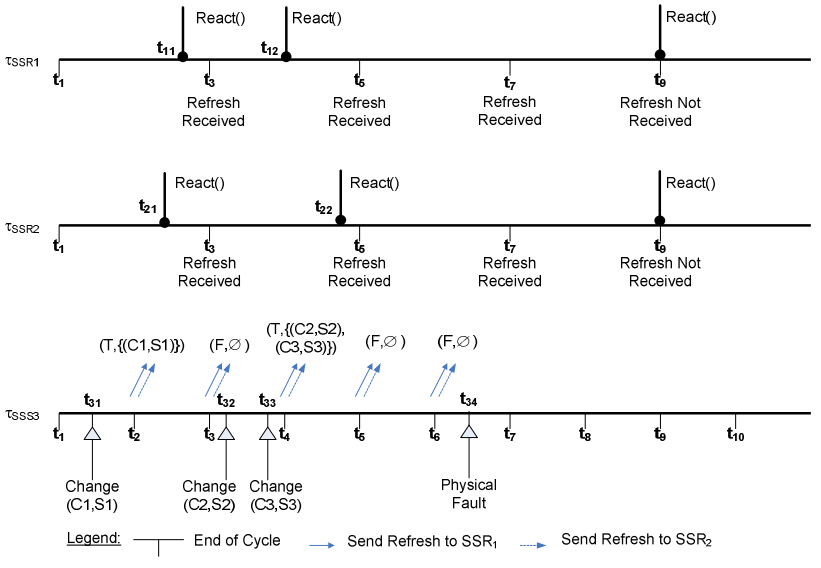

Fig. 5. Example of Fault Propagation

At time t_{31}, SS-S_3 detects a change (with category C_1 and scope S_1) in WS_3. SS-S_3 assigns True to State$_3$.ChangeStatus and inserts (C_1,S_1) in State$_3$.ChangeDetails. At time t_2, SS-S_3 sends Refresh(True,{(C_1,S_1)}) to SS-R_1 and SS-R_2, and reinitializes State$_3$. SS-R_1 and SS-R_2 process those changes by calling their *React()* procedure at times t_{11} and t_{21}, respectively. At t_3, SS-R_1 and SS-R_2 note the reception of the Refresh() sent by SS-S_3. At this same time, SS-S_3 sends Refresh() to both receivers with the parameters (False,∅) since no changes have been detected in the second SS-S_3 cycle. SS-S_3 detects two changes (C_2,S_2) and (C_3,S_3) in WS_3 at t_{32} and t_{33}, respectively. At t_{33}, State$_3$.ChangeStatus equals True and State$_3$.ChangeDetails contains {(C_2,S_2),(C_3,S_3)}. At t_4, SS-S_3 sends Refresh(True,{(C_2,S_2,),(C_3,S_3)}) to SS-R_1 and SS-R_2. SS-R_1 and SS-R_2 process those changes at t_{12} and t_{22}, respectively. At t_5, SS-R_1 and SS-R_2 note the reception of the Refresh() sent by SS-S_3. At times t_5 and t_6,

SS-S_3 sends Refresh() to SS-R_1 and SS-R_2 with the parameters (False,∅) since no changes have been detected in the corresponding SS-S_3 cycle. At t_7, SS-R_1 and SS-R_2 note the reception of the Refresh() sent by SS-S_3. Let us now assume a server failure in WS_3 (and hence SS-S_3) at t_{34}. At t_9, SS-R_1 and SS-R_2 find out that they did not receive Refresh() from SS-S_3 during the last SS-R_2 cycle. If Max-Retry$_2$ is equal to 1, SS-R_1 and SS-R_2 conclude that SS-S_3 failed and hence call the *React()* function.

4 Performance Evaluation

We conducted experiments to assess the different parameters that may impact the performance of the proposed protocol. We used Microsoft Windows Server 2003 (operating system), Microsoft Visual Studio 8 (development kit), UDDI Server, IIS Server, and SQL Server. We ran our experiments on Intel(R) processor (1500MHz) and 512MB of RAM. Soft-state senders and receivers have been developed in C#. We created twenty (20) receivers and fifty (50) senders, and registered them in UDDI. Each receiver has ten (10) senders randomly selected among the existing senders. In the rest of this section, we analyze the relationship between τ_{SSR}/τ_{SSS} values and the following two parameters: *fault propagation time* and *false faults*.

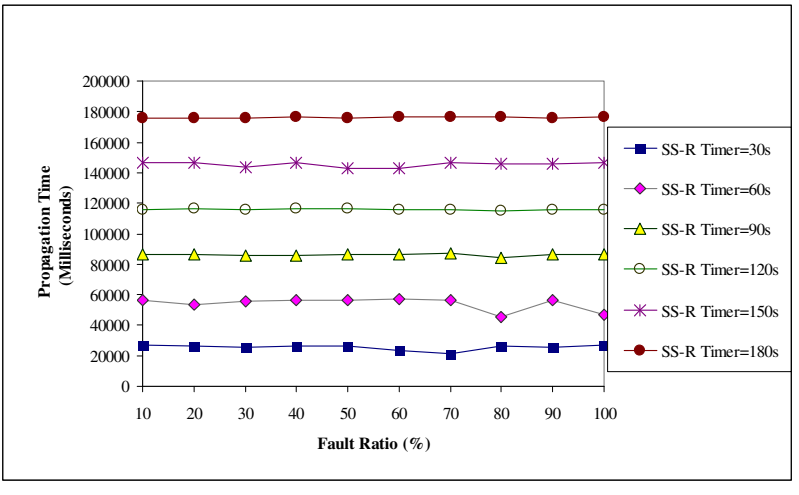

Fig. 6. Impact of τ_{SSR} on Fault Propagation Time

Fault propagation time is the first performance parameter we analyze in our study (Fig. 6). 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 fault propagation time is equal to t_2-t_1 (i.e., the time it took to the receiver to detect a fault in its senders). Fig. 6 compares the average fault propagation time for various τ_{SSR} timer values. We consider different fault ratios for each τ_{SSR} timer value. For instance, a fault ratio of 10 means that 10% (1 out of 10) of participants within a composite service failed. We focused on physical node faults; these are created by physically stopping the services corresponding to

faulty senders (selected randomly). Fig. 6 shows that the τ_{SSR} timer value has an impact on the fault propagation time. The smaller is τ_{SSR}, the shorter is the fault propagation time.

False faults refer to the situation where receivers assume faults that did not occur in their senders. Fig. 7 depicts the relationship between false faults and timer difference (i.e., τ_{SSS}-τ_{SSR}). We set τ_{SSR} to 20s and vary τ_{SSS} from 20s to 25s, 30s, 35s, 40s, etc. Fig. 7 shows that false faults occur if τ_{SSS}-$\tau_{SSR}\geq 0$ (i.e., $\tau_{SSS}\geq\tau_{SSR}$). In addition, the bigger is τ_{SSS} (compared to τ_{SSR}), the larger is the number of false faults. These faults correspond to cases where Refresh() messages are sent after the end of the corresponding τ_{SSR} cycles.

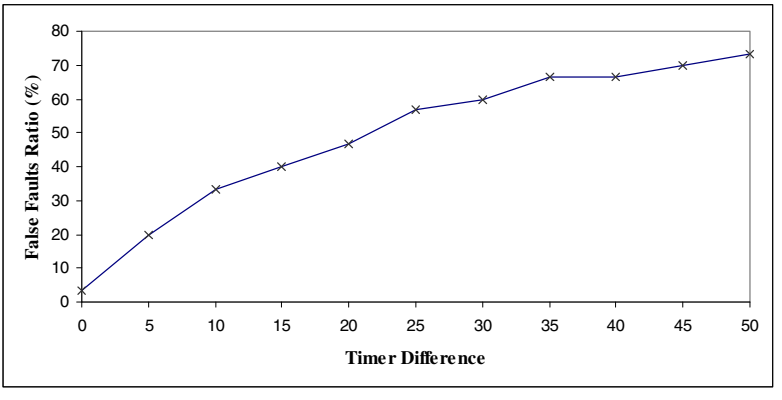

Fig. 7. Relationship between τ_{SSS} and τ_{SSR} and impact on False Faults ratio

5 Related Work

Mechanisms that support fault management in software systems have been around for a long time. Workflow has traditionally been used to deal with faults in business processes. Faults in workflows have usually been modeled as exceptions. SOAs consider faults as the rule, and any solution to fault management would need to treat them as such. Traditional software engineering solutions for fault management (e.g., exceptions and run-time assertion checking) have hard-coded, internal, and application-specific capabilities that limit their generalization and reuse. They disperse the adaptation logic throughout the application, making it costly to modify and maintain [4]. These factors have actuated research dealing with the concept of self-healing systems. A comprehensive survey of major self-healing software engineering approaches is presented in [5]. However, such approaches focus on "traditional" applications *not* service-oriented. The peculiarity of faults, interaction models, and architectural style in SOAs as well as the autonomy, distribution, and heterogeneity of services make these approaches difficult to apply in SOAs.

Current techniques for coping with faults in SOAs allow developers to include constructs in their service specifications (e.g., fault elements in SOAP and WSDL, exception handling in BPEL). Such techniques are static, ad hoc, and make service

design complex [13]. Efforts have recently been made to add self-healing capabilities to SOAs (e.g., [6]). However, these efforts mostly focus on monitoring service level agreements and quality of service.

6 Conclusion

In this paper, we proposed a soft-state approach for managing bottom-up faults in composite services. The approach includes techniques for fault detection, propagation, and reaction. We then focused on describing a fault propagation protocol. The protocol inherits the advantages of the soft-state concept: implicit error recovery and easier fault management resulting in high availability. A future extension of our protocol is to combine soft-state with hard-state signaling. For instance, if WS_i provider is scheduling a shut-down, SS-S_i sends an explicit Shutdown() message to SS-R_j; this allows SS-R_j to differentiate between physical faults and status changes and hence, detect status changes as soon as they occur. Another possible extension is to transmit Refresh() reliably (e.g., via the use of ACK timers) and integrate a notification mechanism through which receivers inform senders about their view (in terms of failure) on those senders.

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architecture, and Applications. Springer, Heidelberg (2003) ISBN: 3540440089
2. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. ACM Trans. on Dependable and Secure Computing 1(1), 11–33 (2004)
3. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.-M.: The Many Faces of Publish/Subscribe. ACM Computing Surveys 35(2), 114–131 (2003)
4. Garlan, D., Schmerl, B.R.: Model-based Adaptation for Self-healing Systems. In: WOSS Workshop (November 2002)
5. Ghosh, D., Sharman, R., Rao, H.R., Upadhyaya, S.: Self-Healing Systems - Survey and Synthesis. Decision Support Systems 42 (2007)
6. Guinea, S.: Self-healing Web Service Compositions. In: ICSE Conference (May 2005)
7. Hamadi, R., Medjahed, B., Benatallah, B.: Self-Adapting Recovery Nets for Policy-Driven Exception Handling in Business Processes. Distributed and Parallel Databases (DAPD), An International Journal 22(1) (February 2008)
8. Ji, P., Ge, Z., Kurose, J., Towsley, D.: A Comparison of Hard-state and Soft-state Signaling Protocols. In: SIGCOMM Conference (August 2003)
9. Khalaf, R., Keller, A., Leymann, F.: Business Processes for Web Services: Principles and Applications. IBM Systems Journal 45(2) (2006)
10. Martin, D., Paolucci, M., McIlraith, S., Burstein, M., McDermott, D., McGuinness, D., Parsia, B., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., Sycara, K.: Bringing Semantics to Web Services: The OWL-S Approach. In: First International Workshop on Semantic Web Services and Web Process Composition (July 2004)
11. Medjahed, B., Atif, Y.: Context-Based Matching for Web Service Composition. Distributed And Parallel Databases 21(1) (February 2007)

12. Medjahed, B., Benatallah, B., Bouguettaya, A., Elmagarmid, A.: WebBIS: An Infrastructure for Agile Integration of Web Services. International Journal on Cooperative Information Systems (IJCIS) 13(2) (June 2004)
13. Verma, K., Sheth, A.P.: Autonomic Web Processes. In: ICSOC Conference (December 2005)
14. Papazoglou, M.P.: Web Services: Principles and Technology. Prentice Hall, Englewood Cliffs (2007) ISBN: 9780321155559
15. Qi, Y., Liu, X., Bouguettaya, A., Medjahed, B.: Deploying Web Services on the Semantic Web. VLDB Journal 17(3) (March 2008)
16. Raman, S., McCanne, S.: A Model, Analysis, and Protocol Framework for Soft State-Based Communication. In: ACM SIGCOMM 1999 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, pp. 15–25 (September 1999)