Assessing Fault Occurrence Likelihood for Service-Oriented Systems

Amal Alhosban[1], Khayyam Hashmi[1], Zaki Malik[1], and Brahim Medjahed[2]

[1] Department of Computer Science
Wayne State University, MI 48202
{ahusban,khayyam,zaki}@wayne.edu
[2] Department of Computer & Information Science
The University of Michigan - Dearborn, MI 48128
brahim@umd.umich.edu

Abstract. Automated identification and recovery of faults are important and challenging issues for service-oriented systems. The process requires monitoring the system's behavior, determining when and why faults occur, and then applying fault prevention/recovery mechanisms to minimize the impact and/or recover from these faults. In this paper, we introduce an approach (defined FOLT) to automate the fault identification process in services-based systems. FOLT calculates the likelihood of fault occurrence at component services' invocation points, using the component's past history, reputation, the time it was invoked, and its relative weight. Experiment results indicate the applicability of our approach.

Keywords: Service-oriented architecture, Fault tolerance, Reliability.

1 Introduction

Over the past decade, we have witnessed a significant growth of software functionality that is packaged using standardized protocols either over Intranets or through the Internet. System architectures adhering to this development approach are commonly referred to as service-oriented architectures (SOA). In essence, SOAs are distributed systems consisting of diverse and discrete software services that work together to perform the required tasks. Reliability of an SOA is thus directly related to the component services' behavior, and sub-optimal performance of any of the components degrades the SOA's overall quality. Services involved in an SOA often do not operate under a single processing environment and need to communicate using different protocols over a network. Under such conditions, designing a fault management system that is both efficient and extensible is a challenging task. The problem is exacerbated due to security, privacy, trust, etc. concerns, since the component services may not share information about their execution. This lack of information translates into traditional fault management tools and techniques not being fully equipped to monitor, analyze, and resolve faults in SOAs.

S. Auer, O. Diaz, and G.A. Papadopoulos (Eds.): ICWE 2011, LNCS 6757, pp. 59–73, 2011.

In this paper, we present a fault management approach (Fault Occurrence Likelihood esTimation: FOLT) for SOAs. We assume that component services do not share their execution details with the invoking service (defined as an orchestrator). The orchestrator only has information regarding the services' invocation times and some other *observable* quality of service (QoS) characteristics. We propose to create fault expectation points in a SOA's invocation sequence of component services to assess the likelihood of fault occurrence. Fault recovery plans are then created for these expectation points and are stored in a data repository to be retrieved and executed when the system encounters a fault at runtime. Due to space restrictions we only focus on the former in this paper, i.e., assessing the likelihood of a fault's occurrence. The latter, i.e., "fault recovery" requires independent discussion.

The paper is organized as follows. Section 2 presents an overview of the service-oriented architecture. We then discuss service invocation models used there in, and overview the relationship between services in each model, and the expected faults for each invocation model. The fault assessment techniques are discussed in Section 3. We present some experiments and analysis of FOLT in Section 4, while Section 5 provides an overview of related work. Section 6 concludes the paper.

2 Service-Oriented Architecture

In this section, we present a brief overview of service-oriented architectures and the different invocation models used by the composition orchestrators. SOA is defined as "a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains" [11]. Boundaries of SOAs are thus *explicit*, i.e., the services need to communicate across boundaries of different geographical zones, ownerships, trust domains, and operating environments. Thus, explicit message passing is applied in SOAs instead of implicit method invocations. The services in SOAs are *autonomous*, i.e., they are independently deployed, the topology is *dynamic*, i.e., new services may be introduced without prior acknowledgment, and the services involved can leave the system or fail without notification. Services in SOAs *share* schemas and contracts. The message passing structures are specified by the schema, while message-exchange behaviors are specified by the contracts. Service compatibility is thus determined based on explicit policy definitions that define service capabilities and requirements [12].

Two major entities are involved in any SOA transaction: Service consumers, and Service providers. As the name implies, service providers provide a service on the network with the corresponding service description [8]. A service consumer needs to discover a matching service to perform a desired task among all the services published by different providers. The consumer binds to the newly discovered service(s) for execution, where input parameters are sent to the service provider and output is returned to the consumer. In situations where a single service does not suffice, multiple services could be *composed* to deliver the required functionality [11].

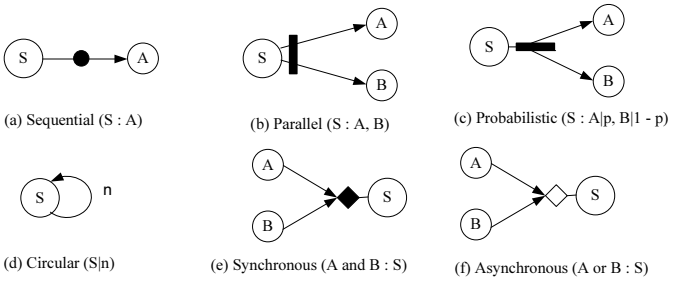

Fig. 1. Major SOA Invocation Models

Each service in an SOA may be invoked using a different invocation model. Here, an invocation refers to *triggering a service (by calling the desired function and providing inputs) and receiving the response (return values if any) from the triggered service*. An SOA may thus be categorized as a 'composite service', which is a conglomeration of services with invocation relations between them. There are six major invocation relations (see Figure 1). (a) Sequential Invocation: the services are invoked in a sequence, (b) Parallel Invocation: multiple services are invoked at the same time, (c) Probabilistic Invocation: one service is invoked from the multiple options, (d) Circular Invocation: a service invokes "itself" x times, (e) Synchronous Activation: all services that were invoked need to complete before the composition can proceed, and (f) Asynchronous Activation: completion of one of the invoked services is enough for the composition to proceed [4]. Due to space restrictions, the detailed discussion about these models is omitted here. The interested reader is referred to [9].

3 Fault Occurrence Likelihood

In this section, we present our approach Fault Occurrence Likelihood esTimation (FOLT) which estimates the likelihood of fault for component services. For the sake of discussion, each service is treated as an independent and autonomous component. This component either performs its desired behavior (i.e., success) or fails to deliver the promised functionality (i.e., fault). FOLT depends on three major factors: the service's past fault history, the time it takes to complete the required task in relation to the composition's total execution time, and the service's weight (i.e., importance) in the composition (in relation to other services invoked). Since, a composed service using one or more of the invocation models described above, may encounter a fault during its execution, the likelihood of encountering a fault is directly proportional to the system's complexity, i.e., the more the invocation models involved, the greater the likelihood of a fault's occurrence. FOLT output values are thus influenced by the invocation model(s) used in the composition. In other words, fault occurrence likelihood is different from one invocation model to the other. In the following, we provide

details of the FOLT approach. We first provide an illustrative scenario where FOLT is applied, and then detail the proposed approach's architecture, and technique.

3.1 Sample Scenario

A student (Sam) intends to attend a conference in London, UK. He needs to purchase an airline `ticket` and reserve a `hotel` for this travel. Moreover, he needs some `transportation` to go from the airport to the hotel and from the hotel to other venues (since this is the first time he's visited the UK, he intends to do some "Site-seeing" also). Sam has a restricted budget, so he is looking for a "deal".

Assume that Sam would be using a SOA-based online service (let's call it *SURETY*) that is a one-stop shop providing all the five options (airline ticket, hotel, attractions, transportation and discounts) through outsourcing. *SURETY* provides many services such as: attraction service which outsources to three services (representing individual services): Art, Museums, and Area tours. This service provides arrangement to visit different areas through sub-contractor companies. For clarity, Figure 2 shows the options at one level. Sam may select Art, Museum, Area tours, or any combination of these services. In terms of transport options, Sam can either use a taxi service, or move around in a rental car, bus, or bike. The different transport companies provide services based on the distance between the places (attractions, etc.) Sam plans to visit. *SURETY* also provides a package optimization service that finds "deals" for the options chosen by Sam.

In Figure 2, the potential services are shown for clarity from "Get request" (when *SURETY* receives Sam's request) to "Send result" states (when *SURETY* sends result(s) to Sam). This is done to show a combination of different invocation models. In reality, service invocations may not follow such a *flat* structure. Since Sam is looking for a travel arrangement that include: booking a ticket, booking a hotel, transportation (rental car, bike or bus) or taxi, and visiting some places, some of these services can be invoked in parallel (here we assume that *SURETY* provides such an option). Booking a ticket and finding attractions is an example of *parallel invocation*. Among the three choices that Sam can select from (Area tours, Museums, and Art), for area attractions, he has to make a choice among these service instances; this is an example of *probabilistic invocation*. Similarly, taxi or rental car, bike and bus services can be classified as probabilistic invocations since *SURETY* has to invoke one service from among multiple services. *SURETY* then provides the results of transport selection to the Package Optimization service, which hunts for available discounts (e.g., if the customer uses the system for more than one year he will get a 20%, etc.). This invocation is an example of *asynchronous invocation*, as one of the transport selections will suffice. *SURETY* then sends the final selection itinerary to Sam. In the following, we show how to use these invocation points to assess the likelihood of a fault's occurrence (similar to [2]).

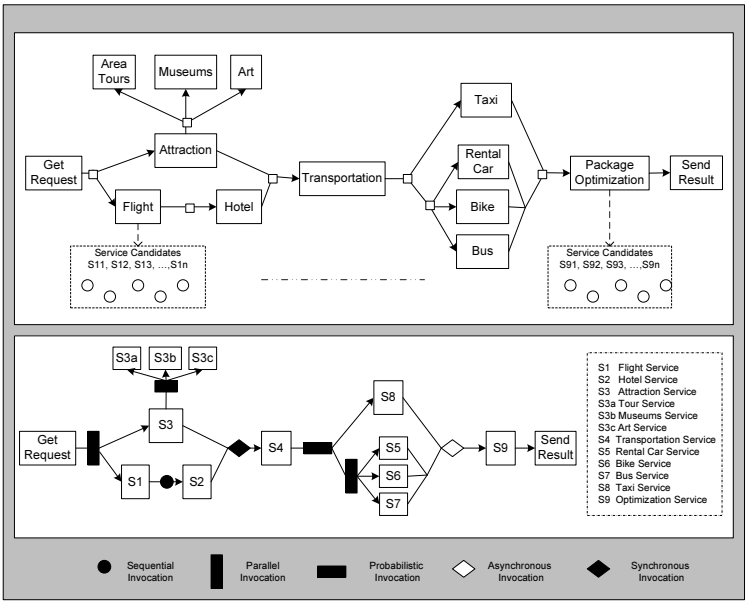

Fig. 2. Scenario with Invocation Models

3.2 Proposed Architecture

In this section, we discuss the architecture of FOLT. FOLT is divided into three phases. In Phase 1, we assess the fault likelihood of the service using different techniques (HMM, Reputation, Clustering). In Phase 2, we build a recovery plan to execute in case of fault(s). Finally, in Phase 3, we calculate the overall system reliability based on the fault occurrence likelihoods assessed for all the services that are part of the current composition. In this paper, we present only the work related to Phase 1. Phase 2 and 3 require independent discussion, which are not presented here due to space restrictions. Details of Phase 1 follow.

3.3 Phase 1: Fault Occurrence Likelihood Assessment

In this Phase, we calculate the fault occurrence likelihood for the service to assess its reliability. The notations used hereafter are listed in Table 1. Most of the terms in the table are self-explanatory. Brief descriptions of other symbols follow: λ_i is the ratio of the time taken by $service_i$ (to complete its execution), to the total composition execution time. On the other hand, λ'_i is the ratio of the time taken by $service_i$ to the total time "remaining" in the composition, from the point when $service_i$ was invoked. Δ_i is the first-hand experience of an invoking service regarding a component $service_i$'s propensity to fault. For cases where the invoker has no historical knowledge of $service_i$ (i.e., the two services had no prior interaction), $\Delta_i = 0$. Similarly, Δ'_i is the second-hand experience regarding a service$_i$'s faulty behavior. This information is retrieved

Table 1. Definition of Symbols

Symbol	Definition
T	The total execution time.
t_0	Start time.
t_n	End time.
t_i	Time at which a new service is invoked.
k	Number of services.
$P(x)^t$	Fault occurrence likelihood for service$_x$ when invoked at time t.
λ_i	Weight of service$_i$ in relation to T.
λ_i'	Weight of service$_i$ in relation to $(T - t_i)$.
Δ_i	First-hand fault history ratio of service$_i$.
Δ_i'	Second-hand fault history ratio of service$_i$.
$f(s_i)$	The priority of service$_i$ in the composition.

from other services that have invoked *service*$_i$ in the past. We assume that trust mechanisms (such as [8]) are in place to retrieve and filter service feedbacks. $f(s_i)$ is the assigned weight of a *service*$_i$ in the whole composition. It provides a measure for the importance of *service*$_i$ in relation to other component services invoked, where $\sum_{i=1}^{n} f(s_i) = 1$.

FOLT architecture (Figure 3) is composed of several modules. These are, *History Module*: This module keeps track of an individual service's propensity to fault. The information is stored in a *History Repository* that includes the service name, invocation time, reported faults (if any), and a numerical score. The *Estimation Module* calculates the fault occurrence likelihood for a service in a given context (execution history). An optional *Priority Module* is used sometimes (details to follow) to indicate the service priority assignment by the invoker in a given execution scenario. Lastly, the *Planning Module* creates plans to recover from encountered faults, and prevent any future ones. As mentioned earlier, details of the module are not the focus of this work.

In summary, the designers store some of the plan details in a plan repository while others are generated at run time. Each plan contains specific fields such as: Plan ID, Plan Name, Plan Duration time, Plan Steps and Plan Counter. When FOLT decides to generate a plan, the system starts the dynamic generation process. The generated plan depends on the chosen invocation model. When the orchestrator invokes a service at any given time (invocation point), it calculates the fault history ratio for the invoked service. Here, we use the maximum value among the external ratio (service's second-hand experience as observed by the community) and internal ratio (first-hand experience of the orchestrator). The system then calculates the fault occurrence likelihood of the invoked service. If the likelihood is greater than a pre-defined threshold (θ_1) the system builds a fault prevention plan. Otherwise, the system re-calculates the likelihood taking into consideration the priority of the current service and compares the value again with θ_1. The purpose of this step is that non-critical services have no plans built for them, and the system can complete the execution even if a fault occurs in any of these services. The newly created plan is tested using a series

of verifications. If the plan fails any of the tests, the system returns back to the planning module, and a new plan is created/checked. The process repeats for x number of times until a valid plan is found. If no plan is still found, the invoker/user is informed. Once a valid plan is created, it is stored in the repository. Then, If the likelihood is greater than another pre-defined threshold (θ_2) the system can execute this fault prevention plan.

Phase 1 is divided into multiple steps: calculating the service's weight (λ), calculating the time weight (λ'), calculating the internal history value (Δ_i) using a Hidden Markov Model, and calculating the external history value (Δ'_i) using clustering and reputation. The likelihood of a fault occurring at time t is defined by studying the relationship between the service's importance, time it takes to execute, and its past performance in the composition. Thus, each invocation model will have a different fault likelihood value. As mentioned earlier, λ is the ratio of the time that is needed to complete the service execution, divided by the total time of completing the execution of the whole system. Similar to the approach used in [10], we use this value of λ as one of the basic constructs

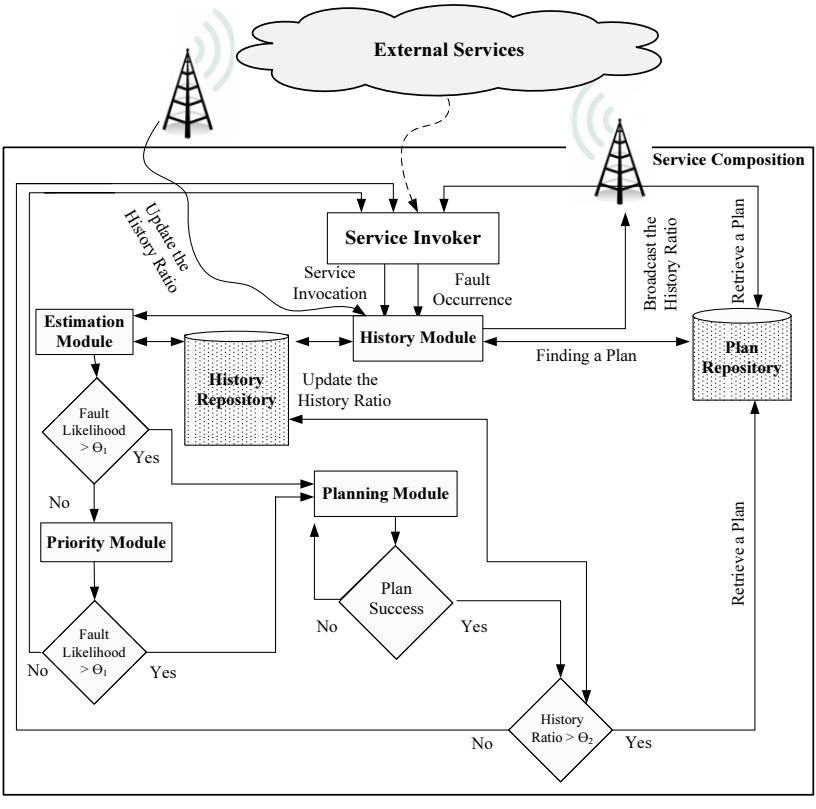

Fig. 3. FOLT Architecture

in FOLT to measure the (relative) weight of the invoked service to the rest of system time. The basic premise is that the likelihood of a fault occurrence for a long running service will be more than a service with very short execution time. Determining the service execution time could be accomplished in two ways. If the system does not know the execution time for a service, then the service's advertised execution time is used. On the other hand, after attaining experience with the service (prior invocations), the service execution time could be recorded and stored in the repository. Then:

$$\lambda_i = \frac{T(s_i)}{T} \tag{1}$$

$$\lambda_i' = \frac{T(s_i)}{T - t_i} \tag{2}$$

where λ_i is as described above, $T(s_i)$ is the total execution time of $service_i$, while t_i is the invocation time of $service_i$ (i.e., when the service was invoked).

A service's past behavior is assessed according to first-hand experience of the invoking service and second-hand experiences of other services obtained in the form of ratings via the community. These *experiences* are evaluated as a ratio of the number of times the service failed, divided by the total number of times the service was invoked. To assess the First-hand Experience, we use a Hidden Markov Model (HMM). The HMM provides the probability that the service will fail in the next invocation, based on the previous behavior of the service within the system. HMMs have proven useful in numerous research areas for modeling dynamic systems [7]. An HMM is a process with a set of states, set of hidden behavior and a transition matrix. In our architecture, all services stay in one of the two states: Healthy or Faulty (Figure 4).

Each time the composition orchestrator invokes service, it records the state of that service (Faulty or Healthy) along with the time of invocation. Let the vector V = the service behavior profile, then to asses the probability that $Service_i$ will be in the Faulty state in the next time instance:

$$P(Faulty|V) = P(Faulty|Healthy) + P(Faulty|Faulty) \tag{3}$$

FOLT also uses other services' experiences with $Service_i$ to assess its reliability. Services are divided into clusters based on their similarity (such as in [1]). These group of services are consulted for the reputation of $Service_i$. We assume

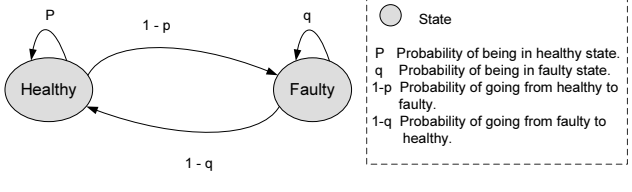

Fig. 4. Finite state machine for a HMM of the service

that other services are willing to share their reputation ratings, which are assimilated using our previous work in [8]. A combination of service time weights and service history ratios (using HMM, and reputation) is used to assess the fault occurrence likelihood:

$$P(s_i)^t = 1 - (\lambda_i')^{\frac{\lambda_i}{1-max(\Delta_i, \Delta_i')}} \tag{4}$$

Note that the fault history is assessed according to $max(\Delta_A, \Delta_A')$. Then, the likelihood of a service executing without any fault is $1 - max(\Delta_A, \Delta_A')$. We use this value in relation to the total execution times (remaining given by λ', and overall given by λ) to assess the likelihood of a service executing *without* a fault. To get the likelihood of the service's fault occurrence we subtract this value from 1 in Equation 4. In cases where we need to incorporate a service's priority weight, Equation 4 becomes:

$$P(s_i)^t = 1 - (\lambda_i')^{\frac{\lambda_i f(s_i)}{1-max(\Delta_i, \Delta_i')}} \tag{5}$$

We observe that with increased service priority, fault likelihood also increases. Based on the fault likelihood, FOLT decides when to build a recovery plan. Services with a high priority are usually critical, and a fault in any of those services may harm the overall QoS. Thus, fault likelihood and service priority are directly proportional in FOLT.

Using Equation 4 as the basis, we define fault likelihood estimation for each invocation model. For instance, the likelihood of fault(s) in a sequential invocation (P_{seq}) is dependent on the successor service(s) [3]. Since FOLT uses invocation points, only a single service can be invoked per time instance/invocation point. Hence the equation stays the same. Let A be the successor service, then

$$P_{seq} = P(s_A)^t = 1 - (\lambda_A')^{\frac{\lambda_A f(s_A)}{1-max(\Delta_A, \Delta_A')}} \tag{6}$$

In a parallel invocation, fault estimation at the invocation point translates to the fault occurring in either of the invoked services. Since all services are independent, we need to add their fault likelihoods. Moreover, due to the likelihood of simultaneous faults occurring in the respective services, we have

$$P_{par} = \bigcup_{i=1}^{h} P_i = \Sigma_{i=1}^{h} P_i - \Pi_{i=1}^{h} P_i \tag{7}$$

$$P_{par} = \Sigma_{i=1}^{h}(1 - (\lambda_i')^{\frac{\lambda_i f(s_i)}{1-max(\Delta_i, \Delta_i')}}) - \Pi_{i=1}^{h}(1 - (\lambda_i')^{\frac{\lambda_i f(s_i)}{1-max(\Delta_i, \Delta_i')}}) \tag{8}$$

where h is the number of services invoked in parallel.

In probabilistic invocation (P_{pro}), fault likelihood depends on the probability of selecting the service (Q). Then, if we have k services:

$$P_{pro} = \bigcap_{i=1}^{k} P_i = \Pi_{i=1}^{k} Q_i \times P_i \tag{9}$$

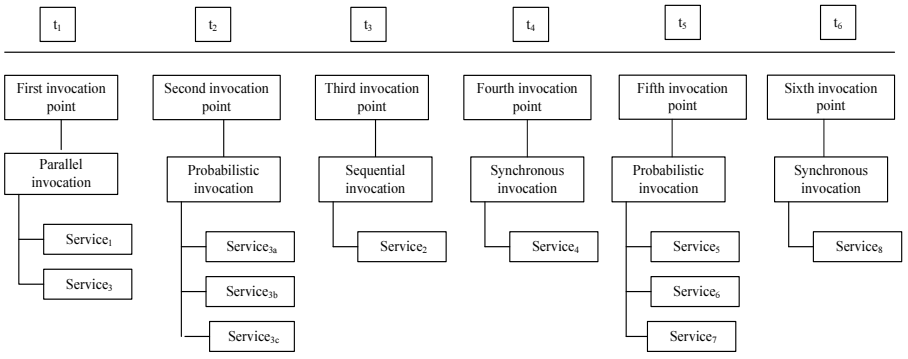

Fig. 5. Simulation Environment of Eleven Services and Six Invocation Points

Similarly, the fault likelihood of a circular invocation is:

$$P_{cir} = \Pi_{i=1}^{n} P_S \tag{10}$$

4 Assessment

We developed a simulator and conducted experiments to analyze the performance of our proposed framework. Our development environment consists of a Windows server 2008 (SP2) based Quad core machine with 8.0 GB of ram. We developed our scenarios using Asp.Net running on Microsoft .Net version 3.5 and SQL as the back-end database. We simulated a services-based system complete with fault prediction, recovery strategies and performance measurement. The input to the system is an XML schema of the system that is used to exhibit the characteristics of a running system.

The experimental results based on the scenario of Figure 2 are discussed below. We are focusing in our experiment in reducing the total execution time, since the service will not be executed if there is a high likelihood that it fails at run time, as this increases the total execution time. In Figure 5, we show a system with 11 services and 6 invocation points. The invocation points are at $t_1 = 30$ ms, $t_2 = 50$ ms, $t_3 = 450$ ms, $t_4 = 560$ ms, $t_5 = 670$ ms, $t_6 = 890$ ms with the total execution time (T) as 1000 ms . At t_1 the system invokes two services ($service_1$, $service_3$) in parallel, at t_2 the system uses probabilistic invocation for three services ($service_{3a}$, $service_{3b}$, $service_{3c}$). At t_3 the system invokes one service ($service_2$) which is sequential invocation, and at t_4 the invocation is synchronous for one service $service_4$. At t_5 the invocation is again probabilistic for three services ($service_5$, $service_6$, $service_7$) and at t_6 the system invokes one service ($service_8$). Table 2 shows a sample (i.e. these are not constant) of the different parameter values for all 6 invocation points.

We assume the different theta values for this experiment i.e., $\theta_1 = 0.50$, $\theta_2 = 0.60$. The table lists the priority of each service involved, the services' time

Table 2. Service Parameters at Invocation Points

Invocation Point	Service	Time	Priority	λ_i	λ_i'	Δ_i	Δ_i'	$P(s_i)$
1	$Service_1$	180	60%	0.18	0.1856	0.30	0.15	0.2289
1	$Service_3$	250	40%	0.25	0.2577	0.40	0.60	0.2875
2	$Service_{3a}$	80	70%	0.08	0.8421	0.90	0.70	0.7498
2	$Service_{3b}$	90	50%	0.09	0.0947	0	0.20	0.1241
2	$Service_{3c}$	80	30%	0.08	0.0842	0.50	0.40	0.1120
3	$Service_2$	150	80%	0.15	0.2727	0.70	0.80	0.5414
4	$Service_4$	1000	80%	0.1	0.2273	0.60	0.80	0.4471
5	$Service_5$	80	90%	0.08	0.2424	0.70	0.65	0.2883
5	$Service_6$	70	80%	0.07	0.2121	0.50	0.80	0.3522
5	$Service_7$	80	90%	0.08	0.2424	0.75	0.90	0.6395
6	$Service_8$	100	80%	0.1	0.9091	0.70	0.80	0.0374

weights and their history ratios (from internal and external experiences). Using Equation 8, FOLT calculated the fault likelihood at the first invocation point (Parallel invocation) to be Ppar =0.4505. Since $service_1$ and $service_3$ had a very low fault likelihood, this in turn implied that the invocation point fault likelihood was lower. In this case $P_{par} < \theta_1$, FOLT did not build any plan and continued with the system execution. For the second invocation point (Probabilistic invocation), as per the given parameters FOLT calculated the fault likelihood using Equation 9 to be P_{pro}=0.0104. Hence, the system did not build a recovery plan and continued its execution. Similarly at third invocation point (Sequential invocation): the fault likelihood was calculated using Equation 6 to be P_{seq}=0.5414. In this case $P_{seq} > \theta_1$, the system did build a recovery plan and continued its execution. However, the execution of the created plan had to wait until the occurrence of fault because $P_{seq} < \theta_2$. Fourth invocation point (Synchronous invocation): The fault likelihood was same as of $service_4 = 0.4471$. Fifth invocation point (Probabilistic invocation): The fault likelihood calculated by FOLT was P_{pro}=0.0649. Since $service_7$ had a high fault likelihood and the other two services had low fault likelihood , this in turn implied that the invocation point fault likelihood was lower. In the case that the selected service was $service_7$, the system would have build a recovery plan and executed it(fault likelihood of $service_7 > \theta_2$). Sixth invocation point (Asynchronous invocation): The fault likelihood of this invocation point was same as that of $service_8 = 0.0374$. For this invocation point the system did not create any plan.

In Figure 6-(a) We can see the eleven services in this system and their fault likelihoods. We notice that $servie_{3a}$ has the highest fault likelihood and $service_8$ has the lowest fault likelihoods. These results are based on the different service's weight, history, behavior, invocation time and priority. Figure 6-(b) shows the fault likelihood for each invocation point where the highest fault likelihood was at t_3 and the lowest fault likelihood was at t_2. Figure 6-(c) shows the relationship between the priority and the fault likelihood. For example, $service_{3a}$ has a priority of 70% and the fault likelihood is 0.7498, however, the priority for $service_7$

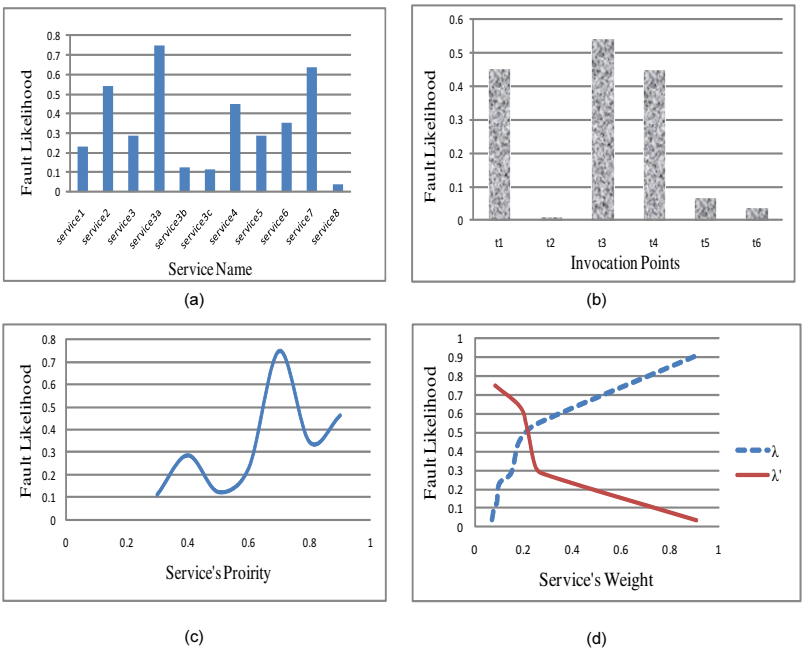

Fig. 6. (a) Services Fault Likelihood (b) Invocations Fault Likelihood (c) Service Priority (d) Service Time Weight

is 90% and the fault likelihood is 0.6395, because it has lower weight. Figure 6-(d) presents the relationship between service weight and the fault occurrence likelihood.

We also performed experiments to assess the FOLT approach's efficiency. Figure 7-(a) shows the comparison between FOLT, no fault and systems that use replace, retry and restart as recovery techniques. Here total execution time is plotted on the y-axis and the number of faults on the x-axis. With increasing number of faults, the execution time also increases. However, FOLT takes less time than compared techniques. This is due to the fact that FOLT preempts a fault and builds a recovery plan for it. Figure 7-(b) shows the total execution time comparisons for the five systems. Here we fix the number of faults to four.

5 Related Work

In this section, we provide a brief overview of related literature on fault management and fault tolerance techniques in service-oriented environments, and the Web in general. Santos et al. [14] proposed a fault tolerance approach (FTWeb) that relies on active replicas. FTWeb uses a sequencer approach to group the different replicas in order. It aims at finding fault free replica(s) for delegating the receiving, execution and request replies to them. FTWeb is based on the

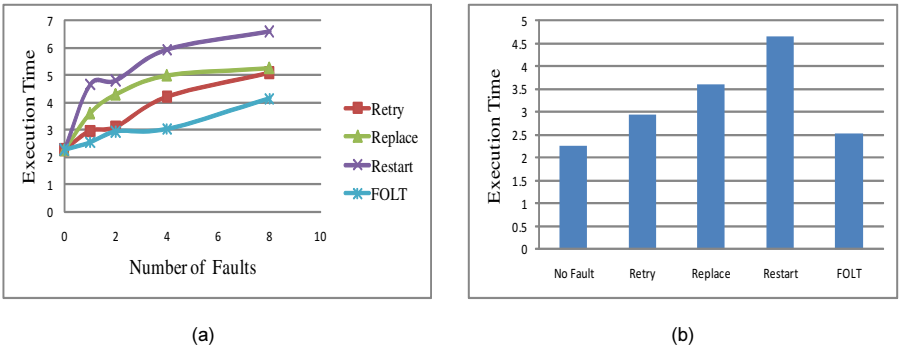

Fig. 7. (a) Total Execution Time Comparisons in Relation to Number of Faults (b) Execution Time Comparisons

WSDispatcher engine, which contains components responsible for: creating fault free service groups, detecting faults, recovering from faults, establish a voting mechanism for replica selection, and invoking the service replicas. Raz et al. [13] present a semantic anomaly detection technique for SOAs. When a fault occurs, it is corrected by comparing the application state to three copies of the service code and data that is injected at a host upon its arrival. Similarly, Hwang et al. [5] analyze the different QoS attributes of web services through a probability based model. The challenge in this approach is composing an alternate work flow in a large search space (withe the least error). Online monitoring (for QoS attributes) also needs some investigation in this approach.

Wang et al's. [16] approach integrates handling of business constraint violations with runtime environment faults for dynamic service composition. The approach is divided into three phases. The first phase is defining the fault taxonomy by dividing the faults into four groups (functional context fault, QoS context fault, domain context fault and platform context fault) and analyzing the fault to determine a remedial strategy. The second phase is defining remedial strategies (remedies are selected and applied dynamically). The remedial strategies are categorized into goal-preserving strategies to recover from faults (ignore, retry, replace and recompose) and non-goal preserving strategies to support the system with actions to assist possible future faults (log, alert and suspend). The third phase is matching each fault category with remedial strategies based on the data levels. The main challenge in this approach is the extra overhead, especially when the selected strategy is a "recomposition" of the whole system.

Simmonds et al. [15] present a framework that guarantees safety and aliveness through the conversation between patterns, and checking their behaviors. The framework is divided in two parts: (1) Websphere runtime monitoring with property manager and monitoring manager. The property manger consists of graphical tools to transfer the sequential diagram to NFAs and check the XML file. The monitoring manager builds the automata and processes the events. (2) Websphere runtime engine. It uses the built-in service component that already exists in BPEL,

to provide service information at runtime. Delivering reliable service compositions over unreliable services is a challenging problem. Liu et al. [6] proposed a hybrid fault-tolerant mechanism (FACTS) that combines exception handling and transaction techniques to improve the reliability of composite services.

6 Conclusion

We presented a new framework for fault management in service-oriented architectures. Our proposed approach Fault Occurrence Likelihood esTimation (FOLT) depends on the past behavior of services, the invocation method of the services, the execution times of services, and the priority of a specific service in the current system. We identified new metrics to measure the fault occurrence likelihood. We evaluated FOLT using simulations and the results indicate the approach's efficiency and ability to recover from faults. FOLT reduces the overall system execution time by replacing the traditional system recovery methods (i.e., restarting the system at check points) by reacting to the faults by expecting the faults ahead of time and pro-actively building the prevention/recovery plans. In the future, we plan to compare FOLT with other similar existing approaches. In this paper, we presented Phase 1 of our approach, and are currently working on Phase 2 (i.e. generating a fault recovery plan) and Phase 3 (i.e. assessing overall system reliability to see when to execute a plan).

References

1. Abramowicz, W., Haniewicz, K., Kaczmarek, M., Zyskowski, D.: Architecture for web services filtering and clustering. In: International Conference on Internet and Web Applications and Services, p.18 (2007)
2. Bai, C.G., Hu, Q.P., Xie, M., Ng, S.H.: Software failure prediction based on a markov bayesian network model. J. Syst. Softw. 74(3), 275–282 (2005)
3. Cardoso, J., Miller, J., Sheth, A., Arnold, J.: Modeling quality of service for workflows and web service processes. Journal of Web Semantics 1, 281–308 (2002)
4. D'Mello, D.A., Ananthanarayana, V.S.: A tree structure for web service compositions. In: COMPUTE 2009: Proceedings of the 2nd Bangalore Annual Computer Conference, pp. 1–4. ACM, New York (2009)
5. Hwang, S.-Y., Wang, H., Tang, J., Srivastava, J.: A probabilistic approach to modeling and estimating the qos of web-services-based workflows. Inf. Sci. 177(23), 5484–5503 (2007)
6. Liu, A., Li, Q., Huang, L., Xiao, M.: Facts: A framework for fault-tolerant composition of transactional web services. IEEE Transactions on Services Computing 99(PrePrints), 46–59 (2009)
7. Malik, Z., Akbar, I., Bouguettaya, A.: Web services reputation assessment using a hidden markov model. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC-ServiceWave 2009. LNCS, vol. 5900, pp. 576–591. Springer, Heidelberg (2009)
8. Malik, Z., Bouguettaya, A.: Rateweb: Reputation assessment for trust establishment among web services. The VLDB Journal 18(4), 885–911 (2009)
9. Menasce, D.A.: Composing web services: A qos view. IEEE Internet Computing 8, 88–90 (2004)

10. Meulenhoff, P.J., Ostendorf, D.R., Živković, M., Meeuwissen, H.B., Gijsen, B.M.M.: Intelligent overload control for composite web services. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC-ServiceWave 2009. LNCS, vol. 5900, pp. 34–49. Springer, Heidelberg (2009)
11. Papazoglou, M.: Web Services: Principles and Technology. Pearson-Prentice Hall, London (2008) ISBN: 978-0-321-15555-9
12. Chen, H.p., Zhang, C.: A fault detection mechanism for service-oriented architecture based on queueing theory. International Conference on Computer and Information Technology, 1071–1076 (2007)
13. Raz, O., Koopman, P., Shaw, M.: Semantic anomaly detection in online data sources. In: ICSE 2002: Proceedings of the 24th International Conference on Software Engineering, pp. 302–312. ACM, New York (2002)
14. Santos, G.T., Lung, L.C., Montez, C.: Ftweb: A fault tolerant infrastructure for web services. In: Proceedings of the IEEE International Enterprise Computing Conference, pp. 95–105. IEEE Computer Society, Los Alamitos (2005)
15. Simmonds, J., Gan, Y., Chechik, M., Nejati, S., O'Farrell, B., Litani, E., Waterhouse, J.: Runtime monitoring of web service conversations. IEEE Transactions on Services Computing 99(PrePrints), 223–244 (2009)
16. Wang, M., Bandara, K.Y., Pahl, C.: Integrated constraint violation handling for dynamic service composition. In: SCC 2009: Proceedings of the 2009 IEEE International Conference on Services Computing, pp. 168–175. IEEE Computer Society, Washington, DC, USA (2009)