FAS: Introducing a Service for Avoiding Faults in Composite Services

Koray Gülcü[1], Hasan Sözer[2], and Barış Aktemur[2]

[1] Vestel Electronics, Manisa, Turkey
koray.gulcu@vestel.com.tr
[2] Özyeğin University, İstanbul, Turkey,
{hasan.sozer, baris.aktemur}@ozyegin.edu.tr

Abstract. In service-oriented architectures, composite services depend on a set of partner services to perform the required tasks. These partner services may become unavailable due to system and/or network faults, leading to an increased error rate for the composite service. In this paper, we propose an approach to prevent the occurrence of errors that result from the unavailability of partner services. We introduce an external Web service, FAS (Fault Avoidance Service), to which composite services can register at will. After registration, FAS periodically checks the partner links, detects unavailable partner services, and updates the composite service with available alternatives. Thus, in case of a partner service error, the composite service will have been updated before invoking the partner service. We provide mathematical analysis regarding the error rate and the ratio of false positives with respect to the monitoring frequency of FAS for different partner service availabilities. We also provide empirical results regarding these metrics based on several tests we performed using the Amazon Elastic Compute Cloud.

1 Introduction

In service-oriented architectures [14], composite services depend on a set of partner services to perform the required tasks. Some of the partner services can cease to be available due to system and/or network faults, which have been shown in recent experimental studies [28] to be very common. These faults result in an error and possibly a failure of the composite service that relies on the availability of its partner services. Preferably, the composite service should discover and utilize alternative services to tolerate such external faults. As such, there have been several service fault tolerance approaches proposed in the literature [25, 27, 17]. However, fault tolerance increases the response time due to the additional time it takes to detect errors and recover from them[3]. The consequential

[3] If the composite service employs the *Active* fault tolerance strategy (i.e., connects to all of the partner services in parallel and proceeds immediately after receiving a response from a partner), occurrence of a fault in a partner service would not affect the composite service. However, this is not possible in many cases due to constraints imposed by unavailable resources or the problem domain.

delay can be very significant especially for composite services that utilize many other services [4]. Therefore, external faults should be avoided (if possible) to improve the dependability and performance of service-oriented systems. One way to avoid partner service faults is to execute the service selection process per each request [3, 7] or per each flow of requests [2]. However, a partner service might be accessed multiple times during the processing of a request and it can cease to be available at any time. Moreover, executing the service selection process per each request/flow also introduces an overhead, just like the overhead of error detection and recovery.

Research efforts so far have mainly focused on providing service brokers [6, 7], middleware [26, 13, 27] and framework support [10, 12, 5, 17] to compose dependable services. In this paper, we propose *fault avoidance as a service*, whose utilization does not require a particular composite service model. We introduce an external Web service, FAS(Fault Avoidance Service), to which a composite service registers the set of its partner services. FAS periodically checks the availability of the registered partner services and locates the alternatives when they are unavailable. Here, our goal is not to provide health monitoring or fault tolerance, for which many approaches have already been proposed [28, 27, 17]. Instead, FAS aims at proactively updating composite services and as such, *avoiding* faults. Faults are avoided by updating the links for unavailable partner services with available alternatives *before* they are invoked by the composite service. This reduces the error-rate.

We studied the impact of the *monitoring frequency* of FAS on the effectiveness of our approach. In particular, we defined analytical metrics regarding the error rate and the false positive rate for different monitoring frequencies and partner service availabilities. We performed several tests using a prototype implementation deployed on the Amazon Elastic Compute Cloud (EC2) [1]. Our measurements confirmed the accuracy of our analytical metrics, which can be used for determining an optimal monitoring frequency depending on varying partner service availabilities.

Contributions of this paper are twofold. First, we propose the implementation of forecasting, detection and the handling of external faults as external services. In this way, a set of services can provide dependability support for other services, i.e., Dependability as a Service (DaaS). To our knowledge, so far this concept has only been realized in the context of software/service testing (Testing as a Service - TaaS). Second, we provide analytical metrics regarding the impact of monitoring frequency on the error rate and the false positive rate. We also validate these metrics with empirical results for various partner service availabilities. We have not encountered such an analysis in the literature although service monitoring has been employed in many studies.

Organization: Section 2 presents the problem statement. Section 3 introduces our solution approach. In Section 4, we introduce analytical metrics and related mathematical analysis. In Section 5, we present our experimental setup, results and evaluation. In Section 6, related previous studies are summarized. Finally, in Section 7 we discuss some future work issues and provide the conclusions.

2 Background and the Problem Statement

A typical process in service-oriented systems involves a service requester and a service provider, which communicate with each other through service requests [14]. Usually a service provider registers its services at a service broker that maintains a registry of “available” services [14]; a service requester can look up and discover these services through the service broker. For instance, a UDDI [23] service registry is a specialized type of service broker [14]. The service requester can select any service provider among the ones that are discovered from a registry service. In some cases, a service provider can request services of several other service providers to perform a task. Such services are called composite services. Usually, they are defined by service aggregators as a composition of a number of partner services. Composition languages (e.g., WS-BPEL [16]) introduce special structures called partner links, through which partner services can be accessed.

After registering itself to the service registry, or after being discovered by the composite service, or even after being successfully invoked several times, a partner service can become unavailable due to system and/or network faults. In fact, recent experimental studies [28] show that the majority of service invocation failures are caused by these types of faults (connection timeout, service unavailablity, etc.). As a result, the invocation attempt leads to an error. In turn, the composite service can *i)* report a failure to its service requester, or *ii)* discover and utilize alternative services to recover from the error. Figure 1 presents a scenario for the second case. In this scenario, the previously designated partner service fails and becomes unavailable. Hereby, $MTTF$ and $MTTR$ correspond to the *mean time to failure* and the *mean time to recover* for this service, respectively. After the failure and before the recovery of the partner service, the composite service makes an invocation without success. The composite service waits for a timeout duration ($t_{timeout}$) to decide whether the partner service is available or not. Once it is deemed to be unavailable, the composite service discovers an alternative service from the service registry. The duration of this discovery is t_{lookup}. In case there is already a designated alternative service (might be hardcoded in the source code or the WS-BPEL description, or it might be stored in an external cache), t_{lookup} will be negligibly small. In any case, a new invocation has to be made to the designated/discovered alternative service. The total time that is necessary to recover from the error is $t_{overhead}$.

Failure of a partner service is an external fault from the perspective of the composite service that tries to utilize the failed service. A composite service can be exposed to many such external faults and for each of these faults, $t_{overhead}$ will be added to its overall response time as a cost of fault tolerance. The consequential delay can be significant especially for composite services that utilize many other services [4] to perform their tasks. In the following, we introduce an approach, where these external faults are avoided to improve the dependability and performance of composite services.

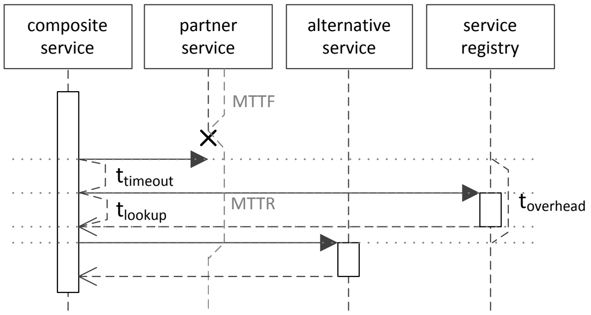


Fig. 1: An error recovery scenario.

3 The Solution Approach

In our approach, we introduce a Web service for avoiding faults. We name this service as *Fault Avoidance Service (FAS)*. A composite service first determines the list of partner services that are going to be utilized, and registers this list of services to FAS. FAS periodically checks the availability of these services. In case a partner service becomes unavailable, FAS locates alternatives and updates the associated partner link of the composite service accordingly. When needed, the composite service uses the updated partner links during invocation. This prevents composite service from trying to reach unavailable partner services, as such reduces the error rate and the overall response time of the process. To be able to incorporate partner link updates, a registered composite service exposes a callback method to receive updates from FAS.

Figure 2 depicts our overall approach. FAS stores a *partner service list* that is provided by the composite service as the list of services to be monitored. This list is used by the *error detection* module to check if the invocation of these services can cause an error due to system/network faults that make the services unavailable. The detected errors are reported to the *fault handling* module. This module is responsible for preventing the detected errors at the composite service by updating its partner links associated with the unavailable partner services. As such, the composite service becomes oblivious to the faults rooted at its partner services. The *fault handling* module may make use of a *service cache* and occasionally the *service registry* to locate alternative and available services. If a faulty service becomes available again, FAS updates the composite service's partner link back to its original setting. FAS checks the availability of the registered partner links periodically. In the following section, we analyze the effect of FAS checking frequency on the error rate and the false positive rate.

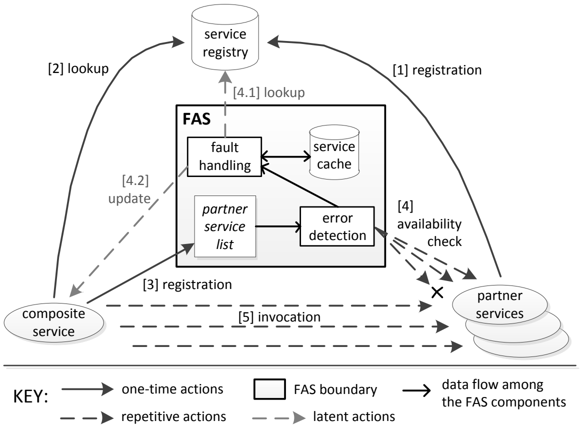


Fig. 2: The overall approach.

4 Mathematical Analysis

In an ideal situation, FAS will immediately detect whenever the partner service becomes unavailable or available. This way, the composite service can be notified right away so that no request from the composite service will fail (i.e., no errors) and no request will be unnecessarily forwarded to the secondary service (i.e., no false positives). However, in real life, there will be cases where the composite service sends its request to the partner service before FAS notices that the service is down, or the cases where the composite service still uses the secondary service because FAS did not notice yet that the partner service is back in life. The error rate and the number of false positives depend on the frequency of requests sent from the composite service, the frequency of FAS checks, and the chance of a FAS check occurring right after a partner service status change. Increasing the frequency of FAS checks would obviously decrease the error rate and false positives, however, an increased frequency means more load and resource usage. Being aware of this trade-off is vital for system administrators in adjusting the checking period for FAS. In this section we provide the mathematical analysis of the expected values of the error rate and the false positive rate.

Figure 3 shows the important events in a system using FAS. In this scenario, we assume that the composite service (CS) periodically sends requests at some frequency C, FAS checks availability of the partner service at a frequency F, and the partner service becomes unavailable for a certain period of its lifetime T_U. For simplicity, we assume that the requests, checks and partner service up/down events are instantaneous. The duration between the moment the partner service becomes unavailable and the time FAS detects this, is the *period of errors*, because any request sent from the CS during this period will fail. Similarly, the

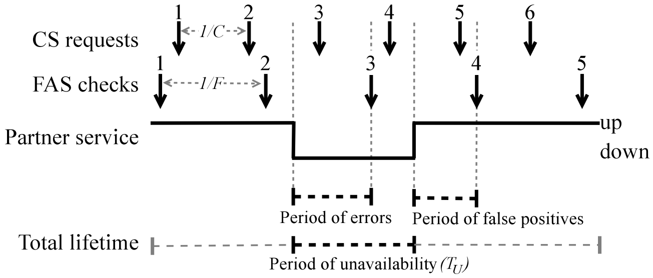


Fig. 3: A scenario showing the important events in a system that uses FAS. This scenario also illustrates the case where $1/F \leq T_U$.

duration between the moment the partner service becomes available again and the time FAS detects this, is the *period of false positives*, because any request sent from the CS during this period will unnecessarily be forwarded to the secondary service. For example, the third CS request in Figure 3 fails because FAS has not notified the CS for the unavailability of the partner service yet. After the third FAS check, FAS notifies the CS, the fourth CS request is succesfully forwarded to the secondary service and the potential error is avoided. However, the fifth request will still be forwarded even though the partner service is back to life, resulting in a false positive. This is because the fourth FAS check occurs after the fifth CS request.

The question we look into at this moment is the expected rate of errors that are not avoided and the false positive rate. The smaller these values are, the more useful FAS is. To calculate these values, we first list the metrics and units we use.

- A (%): Availability of the partner service.
- F $(1/s)$: Frequency of FAS checks.
- C $(1/s)$: Frequency of CS requests.
- T (s): Total lifetime of the system.
- T_U: Period of unavailability of the partner service, i.e., $T_U = (1 - A)T$.
- T_E (s): Period of errors.
- T_F (s): Period of false positives.
- ER: Error rate, calculated as the ratio of the number of errors to the total number of requests.
- FP: False positive rate, calculated as the ratio of the number of false positives to the total number of requests.

Based on these terms, the expected error rate is calculated using the following formulae:

$$E[ER] = \frac{\mathit{Expected\ number\ of\ errors}}{\mathit{Total\ number\ of\ requests}} = \frac{\frac{\mathit{Expected\ length\ of\ } T_E}{\mathit{Duration\ between\ two\ CS\ checks}}}{\frac{\mathit{Total\ lifetime}}{\mathit{Duration\ between\ two\ CS\ checks}}} = E[T_E]/T$$

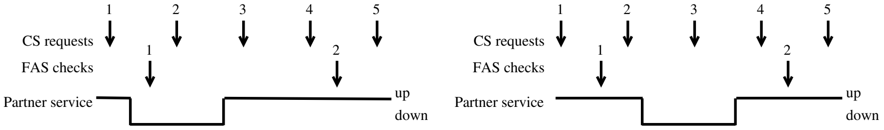


Fig. 4: Two scenarios for when the duration between two FAS checks is larger than the period of unavailability (i.e., $1/F > T_U$). In this case, a FAS check may or may not occur during unavailability.

Similarly,

$$E[FP] = E[T_F]/T$$

$E[T_E]$ and $E[T_F]$ are calculated according to a case analysis as follows.

- *Case 1*: $1/F \leq T_U$. In this case there is at least one FAS check that occurs during the period of unavailability; Figure 3 is a depiction of this case. In this scenario, the minimum value of T_E can be 0 (if a FAS check occurs immediately after the partner service goes down), the maximum value can be $1/F$ (if a FAS check occurs immediately before the partner service goes down). Assuming that the starting time of FAS is uniformly distributed,

  $$E[T_E] = (1/F)/2 = \frac{1}{2F}$$

  Similarly, the minimum value of T_F can be 0, the maximum value can be $1/F$. Assuming uniform distribution,

  $$E[T_F] = (1/F)/2 = \frac{1}{2F}$$

- *Case 2*: $1/F > T_U$. In this case, a FAS check may or may not occur during the period of unavailability. Illustration of both cases is given in Figure 4.
  - *Case 2.1*: A FAS check occurs. In this case, the value of T_E is between 0 and T_U; the value of T_F is between $1/F - T_U$ and $1/F$. Hence, assuming uniform distribution,

    $$E[T_E] = (T_U - 0)/2 = T_U/2$$

    $$E[T_F] = (1/F - (1/F - T_U))/2 = T_U/2$$

  - *Case 2.2*: A FAS check does *not* occur. In this scenario, FAS misses the unavailability of the partner service; CS is never notified by FAS. Hence, there are no false positives and all the requests that occur during T_U result in error:

    $$E[T_E] = T_U \qquad E[T_F] = 0$$

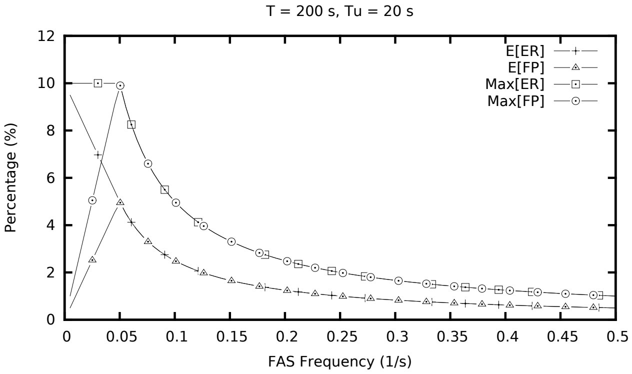


Fig. 5: Change of expected and maximum values of error and false positive rates with respect to FAS frequency.

Again assuming uniform distribution, *Case 2.1* may occur with probability $P_{2.1} = T_U/(1/F) = F\,T_U$; *Case 2.2* may occur with probability $P_{2.2} = 1 - F\,T_U$. Thus, the expected values in *Case 2* are calculated as below:

$$\begin{aligned} E[T_E] &= P_{2.1} \times T_U/2 + P_{2.2} \times T_U \\ &= F\,T_U^2/2 + T_U - F\,T_U^2 \\ &= T_U - F\,T_U^2/2 \end{aligned}$$

$$\begin{aligned} E[T_F] &= P_{2.1} \times T_U/2 + P_{2.2} \times 0 \\ &= F\,T_U^2/2 \end{aligned}$$

Putting the cases together, we have

$$E[ER] = \begin{cases} 1/(2FT), & \text{if } 1/F \le T_U \\ (T_U - F\,T_U^2/2)/T, & \text{otherwise} \end{cases}$$

$$E[FP] = \begin{cases} 1/(2FT), & \text{if } 1/F \le T_U \\ F\,T_U^2/(2T), & \text{otherwise} \end{cases}$$

Note that the expected error and false positive rates are inversely proportional to T. This means, the advantage of using FAS will be higher in longer-running systems.

Following a similar case analysis, below are the upperbounds to ER and FP. The plots of the expected and maximum values are given in Figure 5 for when $T = 200\ s$ and $T_U = 20\ s$ (i.e., $A = 90\%$).

$$Max[ER] = \begin{cases} 1/(FT), & \text{if } 1/F \le T_U \\ T_U/T, & \text{otherwise} \end{cases}$$

$$Max[FP] = \begin{cases} 1/(FT), & \text{if } 1/F \le T_U \\ F\,T_U^2/T, & \text{otherwise} \end{cases}$$

5 Evaluation

We performed several tests to evaluate our approach and analysis. In the following subsections, we discuss the realization of our approach, the experimental setup and the results.

5.1 Realization of the Approach

We developed FAS in Java as a Web service that provides an interface to composite services for registration at start-up. During registration, composite services convey two types of information: *i)* a callback method to be used by FAS to perform partner link updates, and *ii)* a list of partner services and methods to be monitored. FAS uses high-level (service-level) transactions to monitor the partner services. This is to guarantee that the target Web service is functional and reachable. Other, low-level mechanisms (e.g., ping requests) can be used for confirming the availability of a system, however, this does not necessarily imply the functional availability of services. For sending updates, FAS uses nonblocking Web service invocation. Hence, in principle, FAS should be able to handle multiple clients simultaneously without significant delay.

The utilization of FAS does not require the use of a platform/middleware or any composite service model. However, composite services should have *i)* a FAS registration process as part of their initialization, and *ii)* an interface implemented for receiving partner link updates. In accordance with these two requirements, we developed a composite service in Java. We did not use WS-BPEL because it does not directly support stateful (i.e., persistent and global) data. Therefore, partner link updates in a FAS instance cannot be reflected to the other, subsequently created instances. In principle, our approach is agnostic to the composite service implementation and the employed composition language. It is also possible to utilize WS-BPEL, for instance, using the extension proposed by Wu et al. [24].

We also implemented a partner service and replicated it. If FAS updates the partner link before the (unavailable) first replica is invoked, composite service sends the request directly to the second replica. If not, the composite service tries to invoke the first replica. In case of an error, the second replica is invoked and the received response is returned to the client.

5.2 Experimental Setup

We used Axis v2.0 [20] and Tomcat v7.0 [22] to develop and deploy Web services in our experiments. We globally distributed these services using the Amazon EC2 [1]. We utilized *micro instances* [1] and used identical machines, each of which has one CPU core with one *EC2 Compute Unit* [1], 613 MB memory and 8 GB of storage. All instances were running 32-bit Linux operating system. We deployed a composite service and two replicas of our partner service. Partner service replicas were deployed in Ireland and Tokyo, while composite service was in North California and FAS was in Sao Paulo, Brazil. Tests were conducted and controlled with a PC located in Istanbul, Turkey. The PC had Intel(R) Core 2 Duo P8600 at 2.40 Ghz with 4G RAM. As the client to the composite service, JMeter v2.4 [21] was used for executing different test scenarios and collecting measurements automatically.

Throughout our tests, we varied availability (A) only for the first replica of the partner service. The second replica is configured to be 100% available for all tests. Hence, it is assumed that an available replica always exists in the environment.

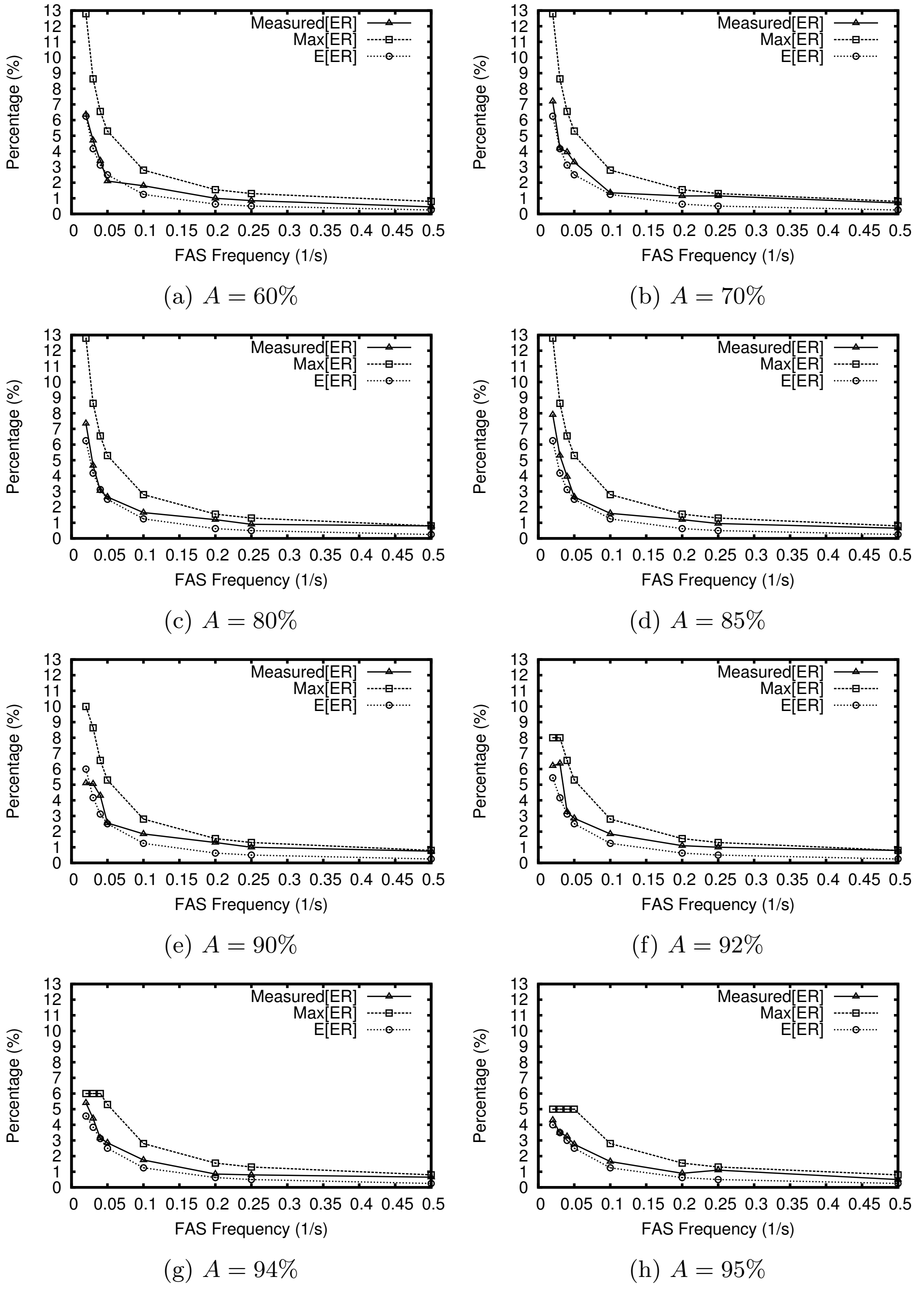
We varied A between 60% and 95%, whereas F was varied between 0.02 ($1/s$) and 0.5 ($1/s$). We performed tests for combinations of these parameters. For each combination, the tests were repeated 20 times; ER and FP were calculated by taking the average of measurements made over these repetitions. During a test, the client sends 100 requests to the composite service at a frequency of 0.25 ($1/s$). Hence, for each parameter combination, 2000 requests were sent in total for calculating the ER and FP. The results are presented in the following subsection.

5.3 Results and Discussion

In this subsection, we present and discuss the results of our tests for different parameter settings. In Figure 6, $E[ER]$ and $Max[ER]$ are plotted together with the measured error rate ($Measured[ER]$) with respect to F. Results are shown when A is 60%, 70%, 80%, 85%, 90%, 92%, 94% and 95%. Figure 7 shows $E[FP]$, $Max[FP]$, and the measured false positive rate ($Measured[FP]$) for the same range and settings of F and A.

It can be seen from the figures that $E[ER]$ and $Max[ER]$ values are consistent with respect to the measured error rates. Likewise, the measured false positive rates confirm the accuracy of our mathematical analysis regarding $E[FP]$ and $Max[FP]$. We could also observe the difference in the change of ER and FP depending on if $1/F \leq T_U$ as shown in Figure 5. See for instance the change of $Measured[FP]$ in Figure 7(h) when F is just less than 0.05. As an interesting observation, we noticed that in many cases $Measured[ER]$ converges to $Max[ER]$ as F increases. We could not observe the same trend consistently for $Measured[FP]$.

The rate of change of ER and FP with respect to F provides us a trade-off curve, which can be utilized for selecting an (pareto-)optimal F for FAS. For our experimental setup and parameter settings for instance, $F = 0.1$ could be a reasonable trade-off point. In general, we can calculate F depending on the value of A and how much we decide to compromise between ER and the load on FAS. The partial derivative of the $E[ER]$ function with respect to F defines the rate of change of $E[ER]$ with respect to F. If we want to balance the objectives of minimizing ER and minimizing the load on FAS for instance, we can find the value of F for which this rate of change (i.e., slope) is -1, e.g., for $1/F \leq T_U$, $E[ER] = 1/2FT \Rightarrow \partial(1/2FT)/\partial F = -1/2TF^2 = -1 \Rightarrow F = \sqrt{1/2T}$.



(a) $A = 60\%$

(b) $A = 70\%$

(c) $A = 80\%$

(d) $A = 85\%$

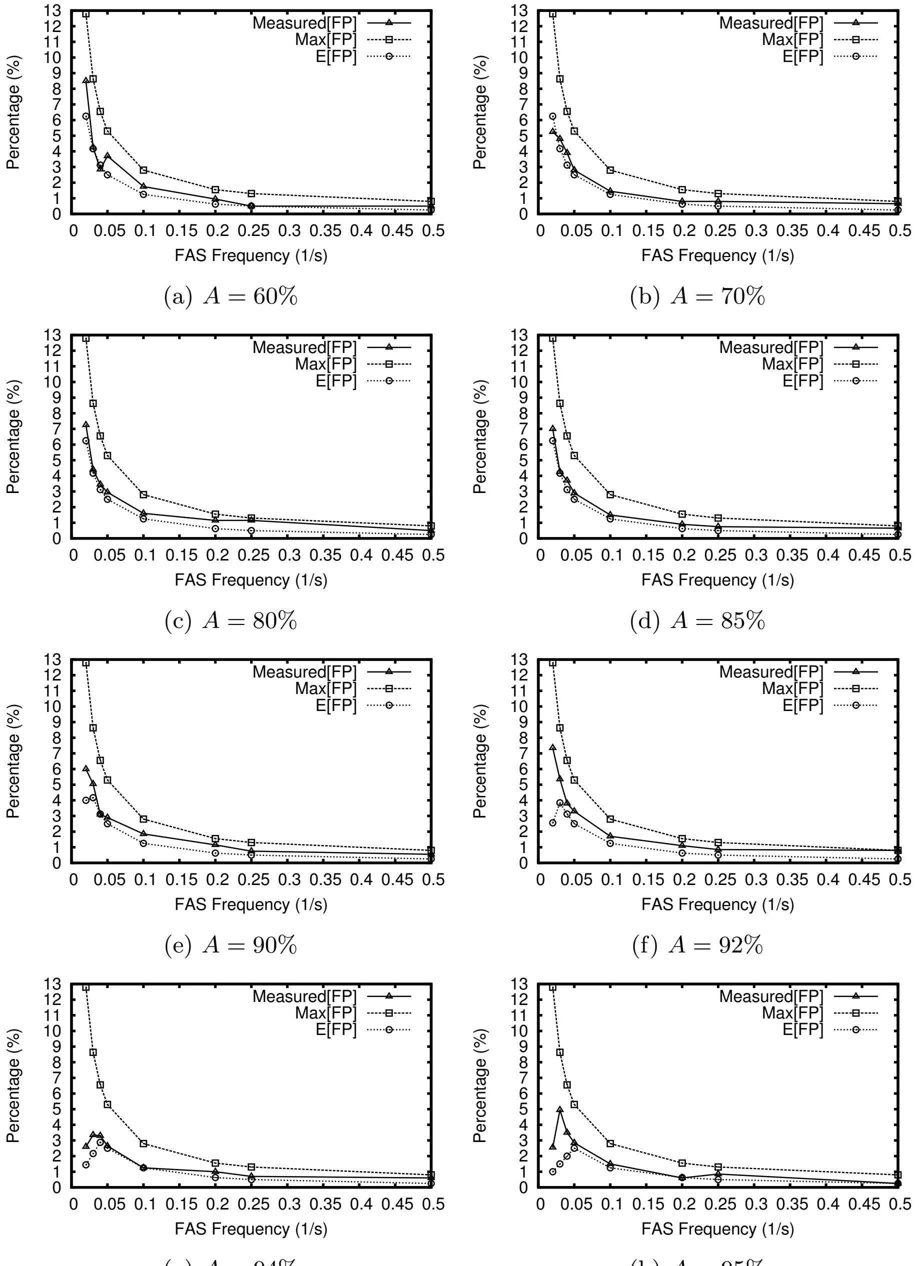
(e) $A = 90\%$

(f) $A = 92\%$

(g) $A = 94\%$

(h) $A = 95\%$

Fig. 6: $E[ER]$, $Max[ER]$, and the measured error rate ($Measured[ER]$) with respect to F, when A is 60%, 70%, 80%, 85%, 90%, 92%, 94% and 95%.



(a) $A = 60\%$

(b) $A = 70\%$

(c) $A = 80\%$

(d) $A = 85\%$

(e) $A = 90\%$

(f) $A = 92\%$

(g) $A = 94\%$

(h) $A = 95\%$

Fig. 7: $E[FP]$, $Max[FP]$, and the measured false positive rate ($Measured[FP]$) with respect to F, when A is 60%, 70%, 80%, 85%, 90%, 92%, 94% and 95%.

5.4 Threats to validity

In our approach, we assume the availability of at least one replica of the partner service. Accordingly, we deployed a partner service replica with 100% availability in our experimental setup. There might be cases where *i)* there is no alternative partner service, *ii)* the alternative service is also unavailable, or *iii)* the alternative service cannot be directly substituted due to stateful properties [13]. We ignored these cases in this work.

The availability of partner services are being monitored from the perspective of FAS, which might possibly mismatch the experience of the composite service. Complementary mediators [9] can be incorporated to monitor the dependability characteristics of partner services from composite services' perspectives.

6 Related Work

So far, research efforts for improving the dependability of service-oriented systems have mainly focused on service fault tolerance [17, 12, 10]. We focus on fault avoidance instead. An analysis of the literature also reveals that dependability improvement has been mainly facilitated by means of frameworks [15, 12], architectural methods [8, 4], reliable service connectors [18], proxies [11] and service dispatchers [19]. We propose implementing a standalone service to which other services can register for improving their dependability.

There exist service brokers and architectural frameworks [6] that are responsible for the creation/composition as well as the adaptation of a composite service. As an advantage of this approach, structural changes (i.e., architecture selection) can also be applied to the composite service [6]. However, such approaches are inherently coupled with the adapted composite service based on a composite service model. FAS does not change the structure and the behavior of the composite service and it does not assume any composite service model.

In this work, we assumed the existence of alternative services that can be directly substituted with unavailable services. However, dynamic service substitution can be problematic in case of stateful services. As a complementary work, SIROCO middleware [13] was introduced to tackle this problem by enabling semantic-based service substitution.

Zheng and Lyu [27] introduce a middleware for composite services to keep track of the QoS information regarding the utilized services. This information is updated at each use of a service and sent occasionally to a common server. The collected QoS information is used for dynamically selecting the most appropriate fault tolerance strategy in case of an error. Empirical results show that their dynamic selection approach performs better than sticking to a statically determined strategy. The differences with their approach to ours are: *i)* They use a middleware; we propose implementing a standalone service to which other services can register. *ii)* Our service actively monitors the replicas. Their monitor is passive; it only stores data. *iii)* We update the user's list of preferred replicas, whereas they update the user's preferred fault tolerance strategy.

7 Conclusions and Future Work

We introduced an approach for avoiding faults during the invocation of partner services and as such, preventing errors in composite services. We developed FAS, an external fault avoidance service that periodically checks the availability of a set of partner services that are registered by a composite service. If one of the partner services ceases to be available, FAS locates alternative services and sends an update to the corresponding composite service, before the faulty partner service is invoked.

We defined analytical metrics for the error rate and the ratio of false positives for different monitoring frequencies of FAS and partner service availabilities. We performed several tests using a prototype implementation deployed on the Amazon EC2. Our measurements confirmed the accuracy of our analytical metrics, which can be used for configuring FAS based on varying partner service availabilities. Our analysis also revealed that FAS is expected to be more effective in reducing the error rate for long-running systems.

In the future, we are planning to enhance FAS so that it can adapt the service checking period at runtime, based on the monitored failure/usage frequencies and response times. We will evaluate the effectiveness of adaptive FAS in the context of an industrial case study for improving the dependability of Smart TVs that utilize many external services.

Acknowledgments

This work is supported by the joint grant of Vestel Electronics and the Turkish Ministry of Science, Industry and Technology (00995.STZ.2011-2). The authors also thank Dr. Ali Özer Ercan for his help in the derivation of analytical metrics.

References

1. Amazon.com: Elastic Compute Cloud (EC2) (2012), http://aws.amazon.com/ec2
2. Ardagna, D., Mirandola, R.: Per-flow optimal service selection for web services based processes. Journal of Systems and Software 83(8), 1512–1523 (2010)
3. Ardagna, D., Pernici, B.: Adaptive service composition in flexible processes. IEEE Transactions on Software Engineering 33, 369–384 (2007)
4. Baresi, L., Ghezzi, C.: Towards self-healing service compositions. Proceedings of the 1st Conference on the Principles of Software Engineering pp. 27–46 (2004)
5. Canfora, G., Penta, M.D., Esposito, R., Villani, M.: A framework for QoS-aware binding and re-binding of composite web services. Journal of Systems and Software 81(10), 1754–1769 (2008)
6. Cardellini, V., Casalicchio, E., Grassi, V., Presti, F.L., Mirandola, R.: Architecting dependable systems vi. chap. Towards Self-adaptation for Dependable Service-Oriented Systems, pp. 24–48. Springer-Verlag, Berlin, Heidelberg (2009)
7. Cardellini, V., Valerio, V.D., Grassi, V., Iannucci, S., Presti, F.L.: A new approach to QoS driven service selection in service oriented architectures. In: Proceedings of the 6th IEEE International Symposium on Service Oriented System Engineering. pp. 102 –113 (2011)

8. Chen, I., Ni, G., Kuo, C., Lin, C.Y.: A BPEL-Based fault-handling architecture for telecom operation support systems. Journal of Advanced Computational Intelligence and Intelligent Informatics 14(5), 523–530 (2010)
9. Chen, Y., Romanovsky, A.: WS-Mediator for improving the dependability of web services integration. Journal of IT Professionals 10(3), 29–35 (2008)
10. Dobson, G.: Using WS-BPEL to implement software fault tolerance for Web services. In: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications. pp. 126–133 (2006)
11. Ezenwoye, O., Sadjadi, S.: A proxy-based approach to enhancing the autonomic behavior in composite services. Journal of Networks 3(5), 42–53 (2008)
12. Fang, C.L., Liang, D., Lin, F., Lin, C.C.: Fault tolerant Web services. Journal of System Architure 53(1), 21–38 (2007)
13. Fredj, M., Georgantas, N., Issarny, V., Zarras, A.: Dynamic service substitution in service-oriented architectures. In: Proceedings of the IEEE Congress on Services. pp. 101–104 (2008)
14. Georgakopoulos, D., Papazoglu, M. (eds.): Service-Oriented Computing. MIT Press (2009)
15. Gorbenko, A., Iraj, E.K., Kharchenko, V.S., Mikhaylichenko, A.: Exception analysis in service-oriented architecture. In: Information Systems Technology and its Applications. pp. 228–233 (2007)
16. Jordan, D., Evdemon, J.: Web services business process execution language version 2.0 (2009), http://docs.oasis-open.org/wsbpel/2.0/serviceref, oASIS Standard
17. Liu, A., Li, Q., Huang, L., Xiao, M.: FACTS: A framework for fault-tolerant composition of transactional web services. IEEE Transactions on Services Computing 3(1), 46 –59 (2010)
18. N. Salatge, N., Fabre, J.C.: Fault tolerance connectors for unreliable Web services. In: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 51–60 (2007)
19. Santos, G., Lung, L., Montez, C.: FTWeb: A fault tolerant infrastructure for Web services. In: Proceedings of the 9th IEEE International Conference on Enterprise Computing. pp. 95–105 (2005)
20. The Apache Software Foundation: Axis (2012), http://axis.apache.org/
21. The Apache Software Foundation: JMeter (2012), http://jmeter.apache.org/
22. The Apache Software Foundation: Tomcat (2012), http://tomcat.apache.org/
23. Tsalgatidou, A., Pilioura, T.: An overview of standards and related technology in Web services. Distributed Parallel Databases 12(2), 135–162 (2002)
24. Wu, G., Wei, J., Huang, T.: Flexible pattern monitoring for WS-BPEL through stateful aspect extension. In: Proceedings of the IEEE International Conference on Web Services. pp. 577–584 (2008)
25. Zarras, A., Fredj, M., Georgantas, N., Issarny, V.: Rigorous development of complex fault-tolerant systems. In: Engineering Reconfigurable Distributed Systems: Issues Arising for Pervasive Computing. pp. 364–386. No. LNCS 4157, Springer-Verlag, Berlin, Heidelberg (2006)
26. Zeng, L., Benatallah, B., Ngu, A., Dumas, M., Kalagnanam, J., Chang, H.: QoS-aware middleware for web services composition. IEEE Transactions on Software Engineering 30(5), 311–327 (2004)
27. Zheng, Z., Lyu, M.: An adaptive QoS aware fault tolerance strategy for web services. Journal of Empirical Software Engineering 15(4), 323–345 (2010)
28. Zheng, Z., Zhang, Y., Lyu, M.: Distributed QoS evaluation for real-world web services. In: Proceedings of the IEEE International Conference on Web Services. pp. 83–90 (2010)