FAULT MASKING AS A SERVICE

A Thesis

by

Koray Gülcü

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Department of Computer Science

Özyeğin University
January 2013

FAULT MASKING AS A SERVICE

Approved by:

Professor Hasan Sözer (Advisor)
Department of Computer Science
Özyeğin University

Professor Ali Özer Ercan
Department of Electrical and
Electronics Engineering
Özyeğin University

Professor Barış Aktemur (Advisor)
Department of Computer Science
Özyeğin University

Professor Pınar Yolum
Department of Computer Engineering
Boğaziçi University

Date Approved: 2013

Professor İsmail Arı
Department of Computer Science
Özyeğin University

ABSTRACT

In service-oriented architectures, composite services depend on a set of partner services to perform their 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 dissertation, 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 various partner service availability rates. We obtain empirical results regarding these metrics based on several tests we performed using the Amazon Elastic Compute Cloud. We use these results to evaluate our mathematical analysis.We also introduce an industrial case study for improving the quality of a service-oriented system from the broadcasting and content delivery domain.

ÖZETÇE

Hizmet odaklı mimarilerde birbirinden bağımsız ve dağıtık olarak çalışan hizmetler bir araya gelerek ihtiyaçlarımızı karşılayan bileşik hizmetleri oluşturulur. Bu harici hizmetler sistem ya da ağ kaynaklı hatalardan dolayı kullanılamaz hale gelebilir ve bir bileşik hizmetin hata oranını arttırabilir. Bu çalışmamızda harici hizmetlerin kullanılamaz olmasıyla bileşik hizmetlerde meydana gelebilecek hataları önlemeye yönelik bir yaklaşım sunmaktayız. Bu yaklaşımda bileşik hizmetlerin istedikleri zaman kaydolabilecekleri harici bir Web hizmeti, orijinal adıyla FAS (Fault Avoidance Service) tasarladık. FAS kendisine bildirilen harici servis baglantılarını periyodik olarak gözlemek, ulaşılmaz olanlarını tespit etmek, varsa alternatiflerini bulmak ve kaydolan bileşik hizmete bu tespitleri iletmekle görevlidir. Böylece, harici hizmetlerde bir hata oluştuğunda, bileşik hizmet söz konusu hatalı hizmeti kullanmadan FAS tarafından güncellenebilir. Çalışmamızda harici hizmetin farklı kullanılabilirlik durumlarında FAS'a ait gözleme frekansını degiştirerek hata oranı ve yanlış kabul oranını ölçmeye çalıştık. Bu verileri kullanarak matematiksel analiz gerçekleştirdik. Amazon bulut bilişim servislerini (Amazon Elastic Compute Cloud) kullanarak yaptığımız testlerle bu metrikler ile ilgili deneysel veriler topladık. Bu deneysel verileri matematiksel analizimizi değerlendirmek için kullandık. Ayrıca yaklaşımımızı televizyon ve yayıncılık sektöründe kullanılan bir sistem üzerinde uygulayarak bir vaka analizi gerçekleştirdik.

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).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Service-oriented computing facilitates the development of distributed software systems based on loosely-coupled and self-contained services in heterogeneous environments [1]. These services can be discovered and composed with each other to provide more sophisticated, higher-level, so-called *composite services* [2]. A composite Web service can utilize several other Web services, which are discovered from a *registry service*, such as UDDI [3]. Usually, a *service aggregator* [4] creates and offers a composite service that is defined by means of specialized composition languages such as WS-BPEL [5]. A composite service invokes the aggregated set of services via so-called *partner links*. Hence, the services that are utilized by a composite service are named *partner services*.

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 [6] 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 fault tolerance approaches proposed in the literature [7, 8, 9]. However, error detection and system recovery increase the response time due to the extra time they take.[1] The consequential delay can be significant especially for the composite services that utilize many other services [10]. Therefore, faults should

[1]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 limited resources or the problem domain.

be avoided (if possible) to improve the dependability and performance of service-oriented systems. One way to avoid fault is to execute the service selection process per each request [11, 12] or per each flow of requests [13]. 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 [14, 12], middleware [15, 16, 8] and framework support [17, 18, 19, 9] to compose dependable services. In this thesis[2], we propose *fault masking 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 and independently checks the availability of the registered partner services for the presence of errors and compensates when they are unavailable. Here, our goal is not to provide health monitoring or error handling. Instead, FAS aims at fault avoidance by proactively reconfiguring composite services and as such, *masking* [21] 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 various monitoring frequencies and partner service availabilities. We performed several tests using a prototype implementation deployed in the Amazon Elastic Compute Cloud (EC2) [22]. 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.

[2]This thesis presents an extended version of our previous work [20].

Contributions of this thesis are threefold. First, we propose the implementation of forecasting, detection and 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 [23]). 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 [24, 25, 8, 26]. Third, we present an industrial case study from the broadcasting domain, where the utilization of third party Web services become predominant. We discuss the deployment of FAS in this context and evaluate the effectiveness of fault masking based on real data regarding the availability of third party content providers.

The remainder of this dissertation is organized as follows. Chapter II presents the problem statement and our solution approach. In Chapter III, we introduce a set of analytical metrics and related mathematical analysis with an experimental evaluation. An industrial case study for improving the availability of services employed in Smart TVs is given in Chapter IV. In Chapter V, related previous work is summarized. Finally, in Chapter VI we discuss possible extensions of this work for the future and provide the conclusions.

CHAPTER II

PROBLEM STATEMENT AND THE SOLUTION APPROACH

In this chapter we explain the problem statement and give details of our solution approach.

2.1 Problem Statement

In service-oriented systems, a typical process involves a service requester and a service provider that communicate with each other through service requests [1]. Usually a service provider registers its services at a service broker that maintains a registry of "available" services [1]; a service requester can look up and discover these services through the service broker. For instance, a UDDI [3] service registry is a specialized type of service broker.

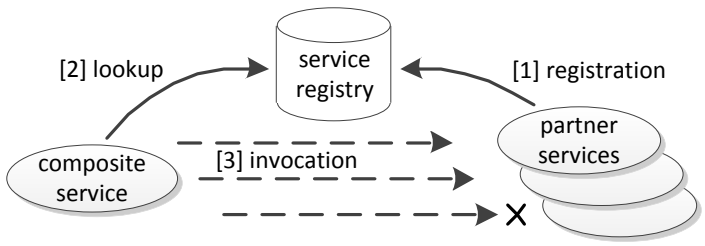


Figure 1: A typical process in service-oriented systems.

Figure 1 depicts a typical process involving a composite service that discovers and utilizes a set of partner services. In this figure, solid arrows represent actions that are supposed to be performed once, during the deployment/activation of services.

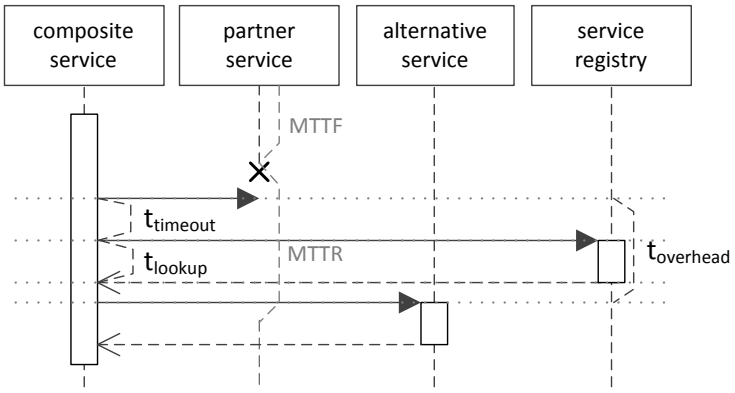


Figure 2: An error recovery scenario.

Dashed arrows, on the other hand, represent actions that are performed many times throughout the life-time of services. The figure also depicts a scenario in which one of the requests fails. 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 [6] show that the majority of service invocation failures are caused by these types of faults (connection timeout, service unavailability, 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 (might be hard-coded in the source code, UDDI and WS-BPEL description, or it might be stored in an external cache) to recover from the error. Figure 2 presents a scenario for the second case where it is assumed that there is an available alternative service in the environment. 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 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 [10] to perform their tasks. In the following, we introduce a fault masking approach, where these external faults are handled to improve the dependability and performance of composite services.

2.2 Overview of the Approach

In our approach, we introduce a Web service for masking faults. We name this service *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 reconfigures the composite service accordingly. When needed, the composite service uses the updated partner links. This prevents composite service from trying to invoke erroneous 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.

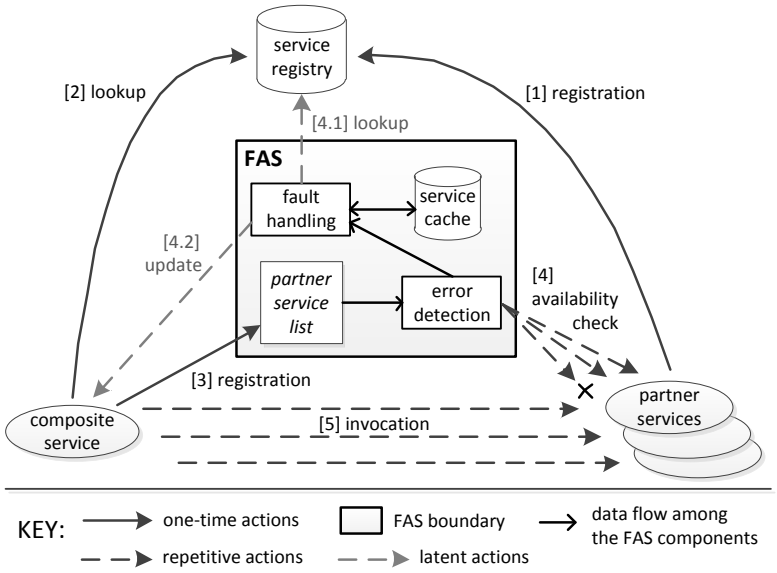


Figure 3: The overall approach.

Figure 3 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 faults are reported to the *fault handling* module. This module is responsible for reconfiguring 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 chapter, we analyze the effect of FAS checking frequency on the error rate and the false positive rate.

CHAPTER III

DEPENDABILITY ANALYSIS

In this chapter, we introduce a set of analytical metrics and related methametical analysis. Afterwards, we explain our experimental evaluation and discuss its results.

3.1 Mathematical Analysis

In an ideal situation, FAS will immediately detect whenever a 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. If the service is down, it threatens the customer satisfaction since the composite service will be trying to invoke an erroneous partner service. If the primary service is back in life, using the secondary service can cause several problems depending on the deployment. The second replica may have limited resources and a higher cost for access. Hence FAS should be utilized as effective as possible to avoid such unwanted consequences.

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

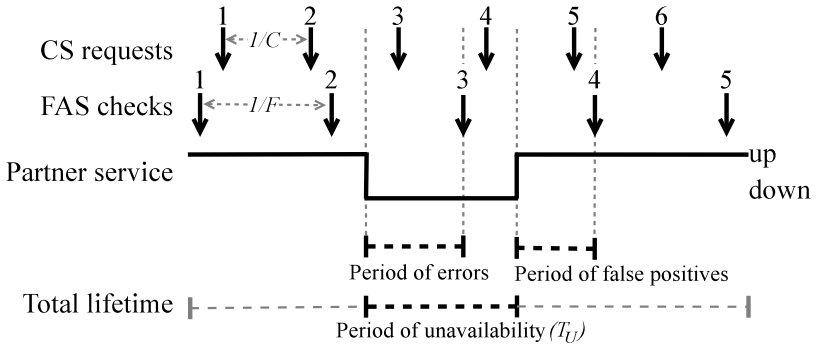


Figure 4: A scenario showing the important events in a system that uses FAS. This scenario also illustrates the case where $1/F \leq T_U$.

FAS. In this section we provide the mathematical analysis focusing on the expected values of the error rate and the false positive rate.

Figure 4 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 T_U is a fixed duration and its starting time is uniformly distributed over the total lifetime. It is also assumed 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 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 4 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 successfully 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 (s): 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.
- N_{check}: Number of FAS checks during T_U.

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

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

Similarly,

$$E[\mathsf{FP}] = E[T_F]/T$$

$E[T_E]$ and $E[T_F]$ are calculated according to a case analysis as follows. We ignored the case when F is 0 because it means that FAS is inactive and $1/F$ is undefined.

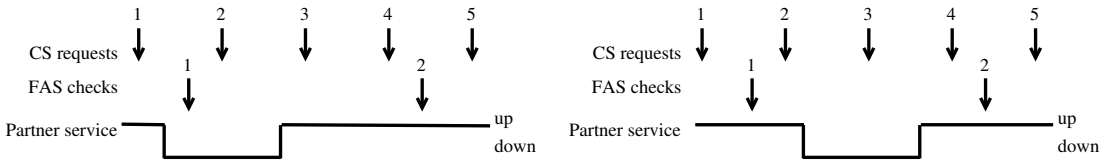


Figure 5: Two scenarios for when the duration between two FAS checks is larger than the period of unavailability (i.e., $1/F > T_U$). In this case, a FAS check may or may not occur during unavailability.

- *Case 1*: $1/F \leq T_U$. Figure 4 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,

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

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

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

- *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 5.

  - *Case 2.1*: A FAS check occurs (i.e. $N_{check} > 0$). 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,

$$E[T_E \mid 1/F > T_U \wedge N_{check} > 0] = (T_U + 0)/2 = T_U/2$$

$$E[T_F \mid 1/F > T_U \wedge N_{check} > 0] = (1/F + (1/F - T_U))/2 = 1/F - T_U/2$$

– *Case 2.2*: A FAS check does *not* occur (i.e. $N_{check} = 0$). 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:

$$E[T_E \mid 1/F > T_U \wedge N_{check} = 0] = T_U \qquad E[T_F \mid 1/F > T_U \wedge N_{check} = 0] = 0$$

Again assuming uniform distribution, $P(N_{check} > 0 \mid 1/F > T_U) = T_U/(1/F) = F\,T_U$; and $P(N_{check} = 0 \mid 1/F > T_U) = 1 - F\,T_U$. Thus, the expected values in *Case 2* are calculated as below:

$$\begin{aligned}
E[T_E \mid 1/F > T_U] &= P(N_{check} > 0 \mid 1/F > T_U) \times E[T_E \mid 1/F > T_U \wedge N_{check} > 0] \\
&+ P(N_{check} = 0 \mid 1/F > T_U) \times E[T_E \mid 1/F > T_U \wedge N_{check} = 0] \\
&= F\,T_U \times T_U/2 + (1 - F\,T_U) \times T_U \\
&= F\,T_U^2/2 + T_U - F\,T_U^2 \\
&= T_U - F\,T_U^2/2
\end{aligned}$$

$$\begin{aligned}
E[T_F \mid 1/F > T_U] &= P(N_{check} > 0 \mid 1/F > T_U) \times E[T_F \mid 1/F > T_U \wedge N_{check} > 0] \\
&+ P(N_{check} = 0 \mid 1/F > T_U) \times E[T_F \mid 1/F > T_U \wedge N_{check} = 0] \\
&= F\,T_U \times (1/F - T_U/2) + (1 - F\,T_U) \times 0 \\
&= T_U - F\,T_U^2/2
\end{aligned}$$

Putting the cases together, we have:

$$E[\mathsf{ER}] = E[\mathsf{FP}] = \begin{cases} 1/(2FT), & \text{if } 0 < 1/F \leq T_U \\ (T_U - F\,T_U^2/2)/T, & \text{if } 1/F > T_U \end{cases}$$

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

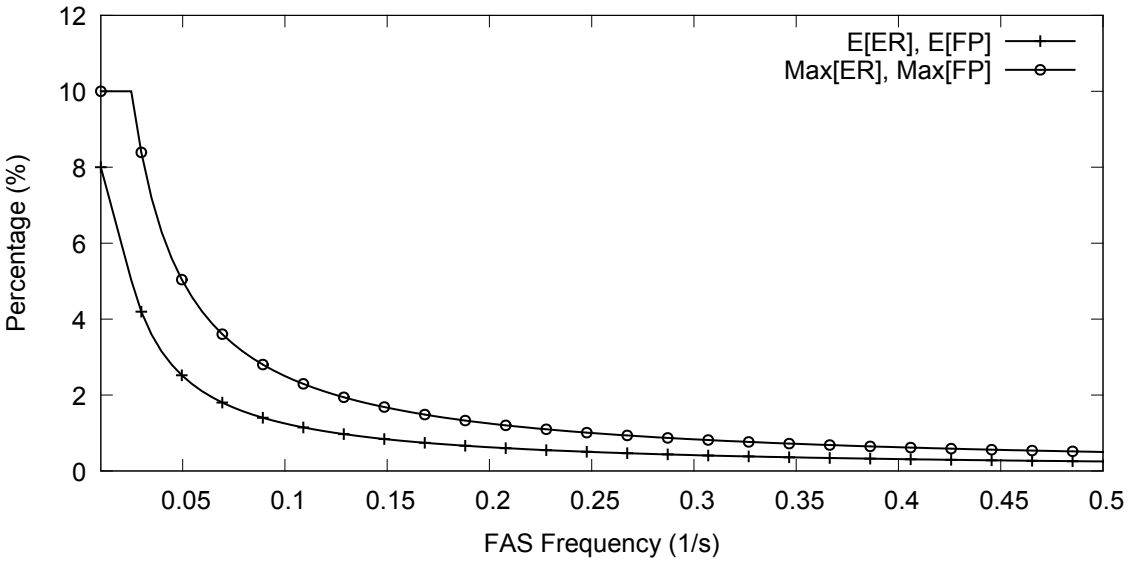


Figure 6: Change of expected and maximum values of error and false positive rates with respect to FAS frequency.

Following a similar case analysis, below are the upper bounds to ER and FP. They are used to obtain the worst case values analytically for any given conditions in the experimental evaluation. The plots of the expected and maximum values are given in Figure 6 for when $T = 400\ s$ and $T_U = 40\ s$ (i.e., $A = 90\%$).

$$Max[\mathsf{ER}] = Max[\mathsf{FP}] = \begin{cases} 1/(FT), & \text{if } 0 < 1/F \leq T_U \\ T_U/T, & \text{if } 1/F > T_U \end{cases}$$

We simulated our mathematical model by using Octave[1] to verify our analysis. We took $T = 400\ s$ and varied A from 60% to 95%. We changed F from 0.01 to 0.5 and repeated each combination of these metrics 10000 times and calculated the average values of T_E and T_F to obtain ER and FP values. Figure 7 shows the results for $A = 90\%$ and the remaining results can be found in the Appendix.

Looking at the simulation results, we see that $E[\mathsf{ER}]$ and $E[\mathsf{FP}]$ values overlap, and the results are coherent with the mathematical analysis. Hence, the derived formulas regarding $E[\mathsf{ER}]$ and $E[\mathsf{FP}]$ are verified by simulation.

[1]http://www.gnu.org/software/octave/

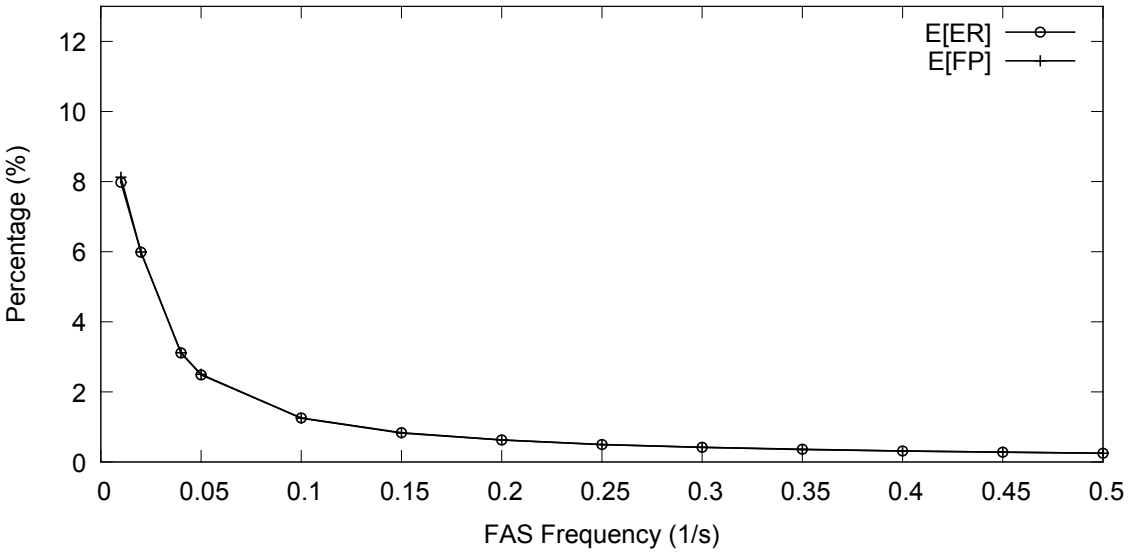


Figure 7: Simulation results of the error and false positive rates for $T = 400$ s and $A = 90\%$, using Octave.

3.2 Experimental Evaluation

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

3.2.1 Realization of the Approach

We developed FAS using Java as a stand alone 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 send 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. [27].

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.

3.2.2 Experimental Setup

We used Axis v2.0 [28] and Tomcat v7.0 [29] to develop and deploy Web services in our experiments. We globally distributed these services using the Amazon EC2 [22]. We utilized *micro instances* [22] and used identical machines, each of which has one CPU core with one *EC2 Compute Unit* [22], 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 4 GB RAM. As the client to the composite service, JMeter v2.4 [30] was used for executing different test scenarios and collecting measurements automatically.

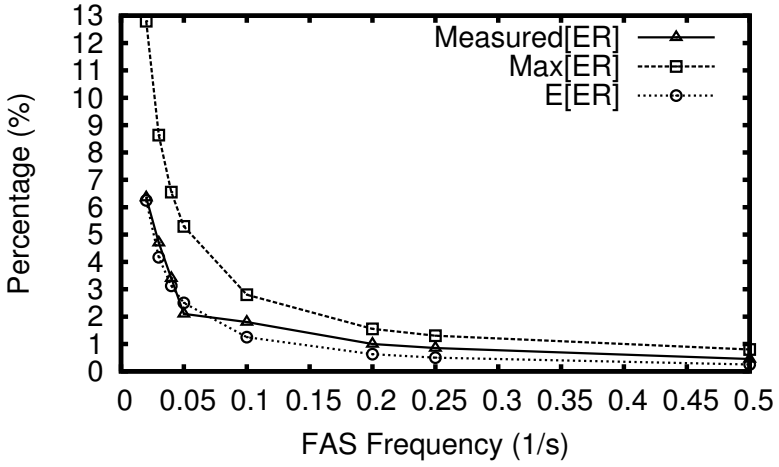
Throughout our tests, we varied availability (A) only for the first replica of the partner service. The second replica is configured to be 100% available for all tests. Hence, it is assumed that an available replica always exists in the environment.

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.

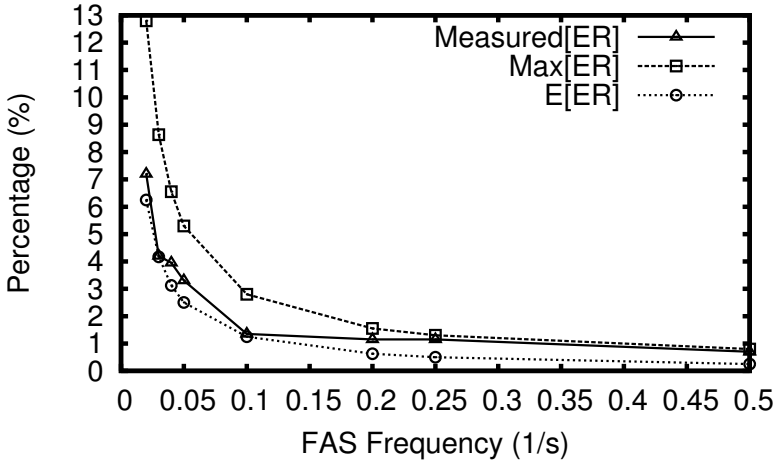
3.2.3 Results and Discussion

In this subsection, we present and discuss the results of our tests for different parameter settings. In Figure 8, 9 and 10, $E[\mathsf{ER}]$ and $Max[\mathsf{ER}]$ are plotted together with the measured error rate ($Measured[\mathsf{ER}]$) with respect to F. Results are shown when A is 60%, 70%, 80%, 85%, 90%, 92%, 94% and 95%. Figures 11, 12 and 13 show $E[\mathsf{FP}]$, $Max[\mathsf{FP}]$, and the measured false positive rate ($Measured[\mathsf{FP}]$) for the same range and settings of F and A.

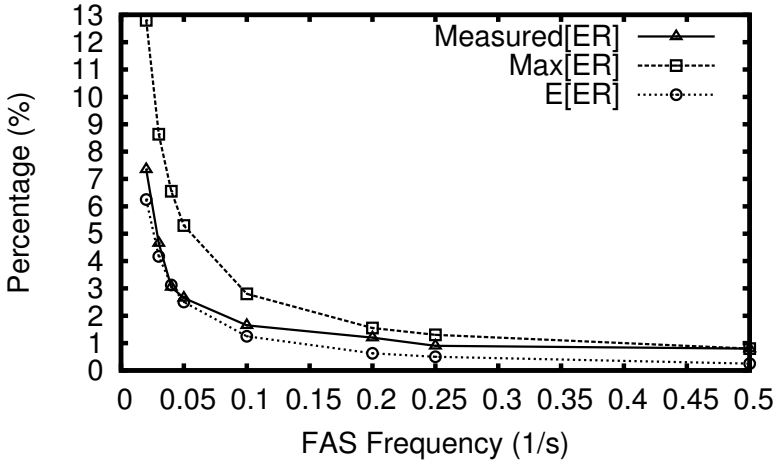
It can be seen from the figures that $E[\mathsf{ER}]$ and $Max[\mathsf{ER}]$ values are consistent with respect to the measured error rates. Likewise, the measured false positive rates confirm the accuracy of our mathematical analysis regarding $E[\mathsf{FP}]$ and $Max[\mathsf{FP}]$. By comparing the figures we can say that in higher availabilities, we obtain low ER even if we use small F values. As an interesting observation, we noticed that in many cases $Measured[\mathsf{ER}]$ converges to $Max[\mathsf{ER}]$ as F increases. We could not observe the



(a) $A = 60\%$

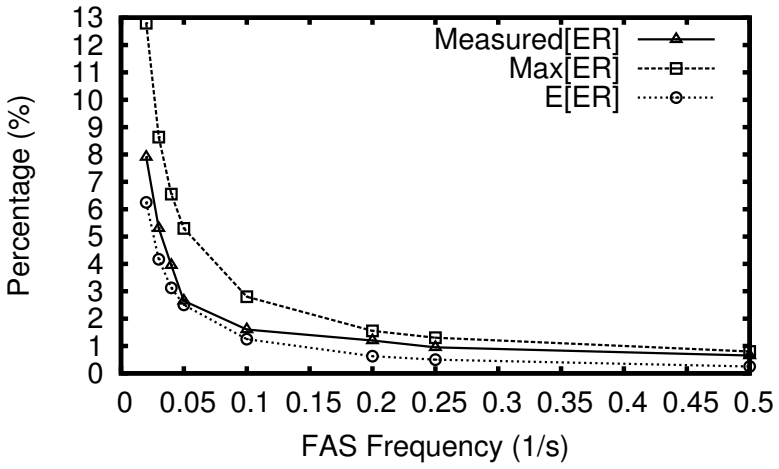


(b) $A = 70\%$

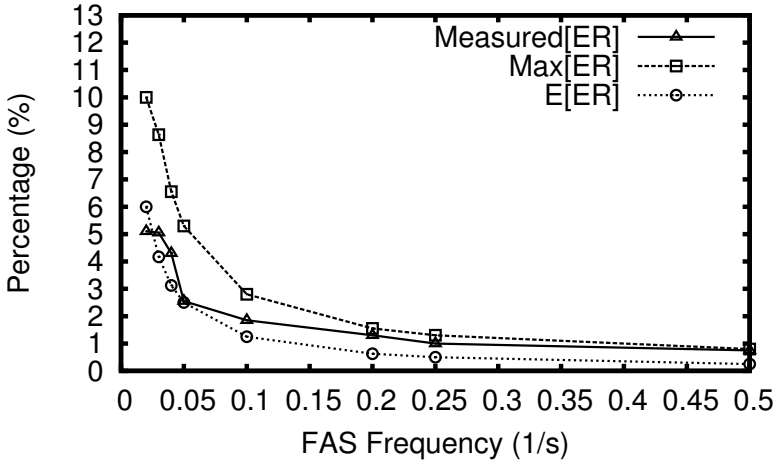


(c) $A = 80\%$

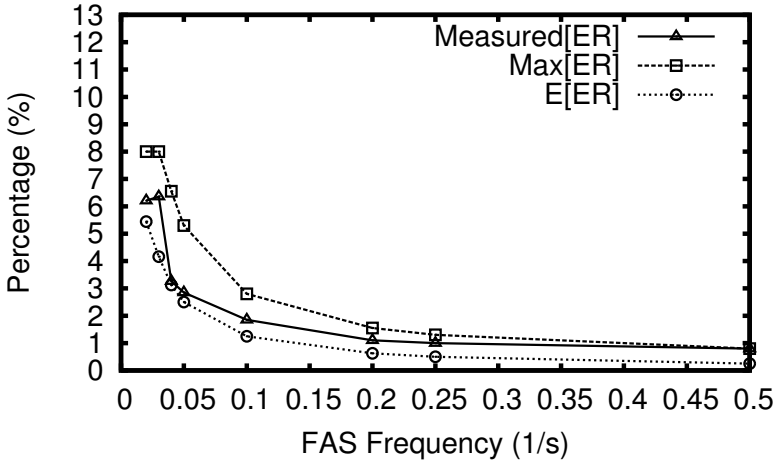
Figure 8: $E[\mathsf{ER}]$, $Max[\mathsf{ER}]$, and the measured error rate ($Measured[\mathsf{ER}]$) with respect to F, when A is 60%, 70%, 80%.



(a) $A = 85\%$

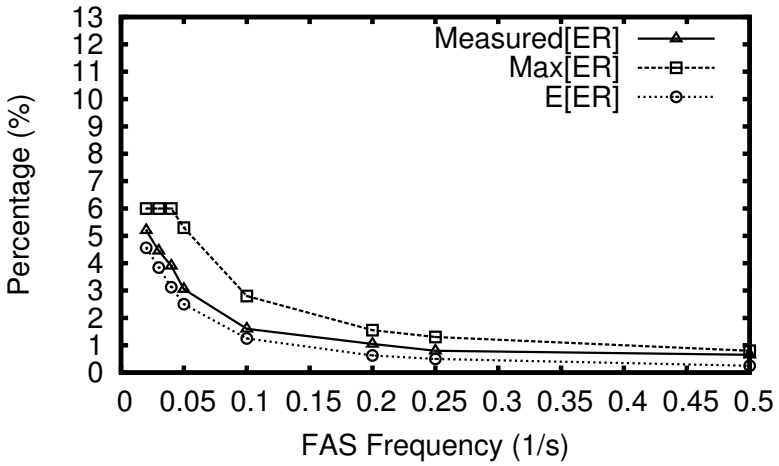


(b) $A = 90\%$

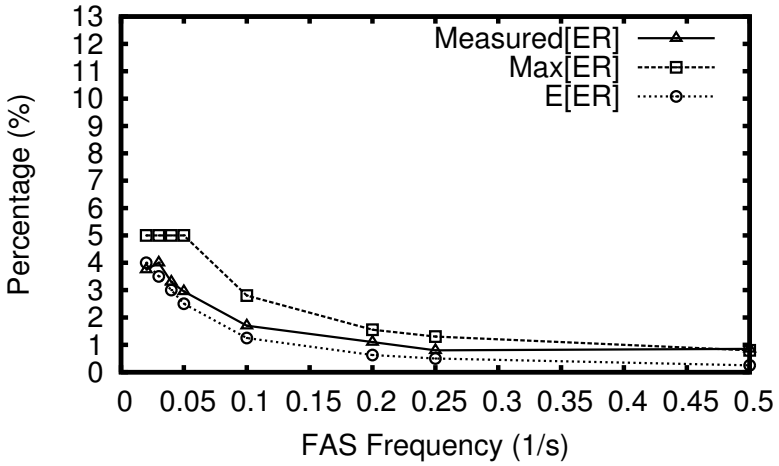


(c) $A = 92\%$

Figure 9: $E[\mathsf{ER}]$, $Max[\mathsf{ER}]$, and the measured error rate ($Measured[\mathsf{ER}]$) with respect to F, when A is 85%, 90%, 92%.

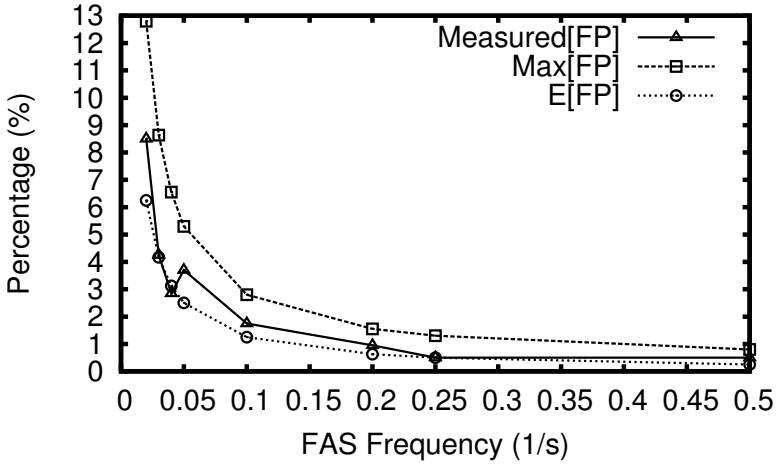


(a) $A = 94\%$

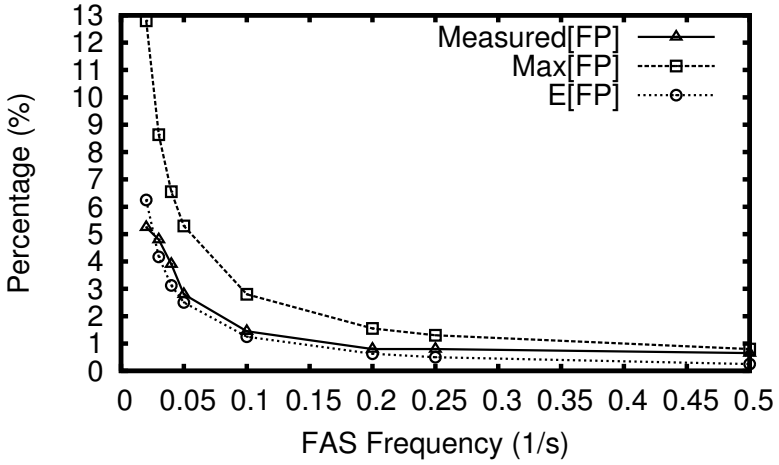


(b) $A = 95\%$

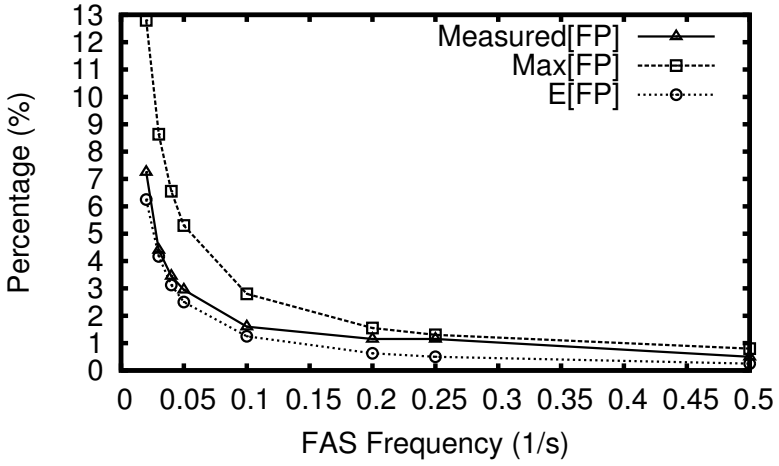
Figure 10: $E[\mathsf{ER}]$, $Max[\mathsf{ER}]$, and the measured error rate ($Measured[\mathsf{ER}]$) with respect to F, when A is 94%, 95%.



(a) $A = 60\%$

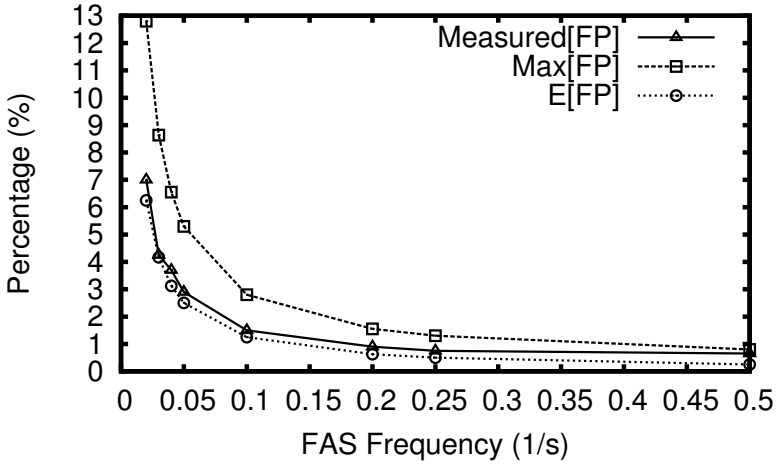


(b) $A = 70\%$

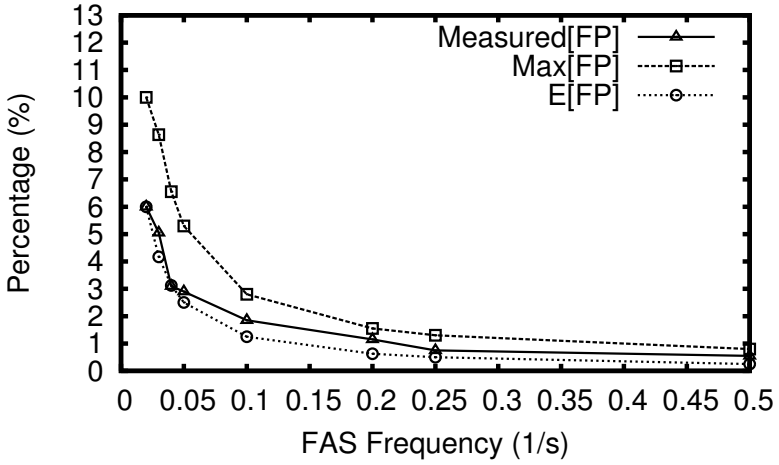


(c) $A = 80\%$

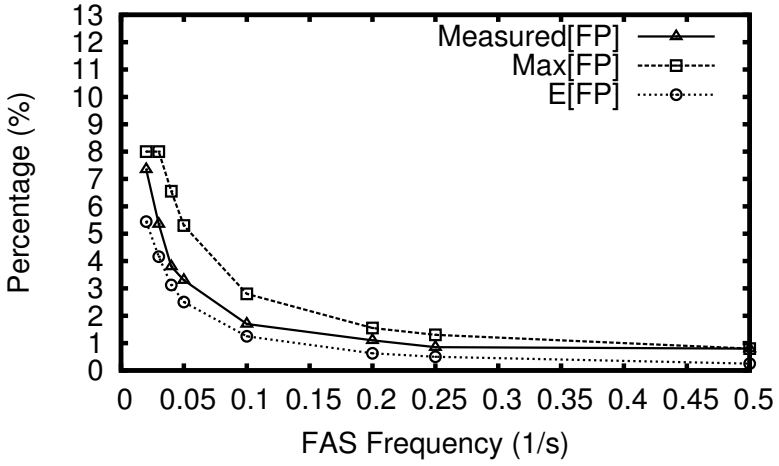
Figure 11: $E[\mathsf{FP}]$, $Max[\mathsf{FP}]$, and the measured false positive rate ($Measured[\mathsf{FP}]$) with respect to F, when A is 60%, 70%, 80%.



(a) $A = 85\%$

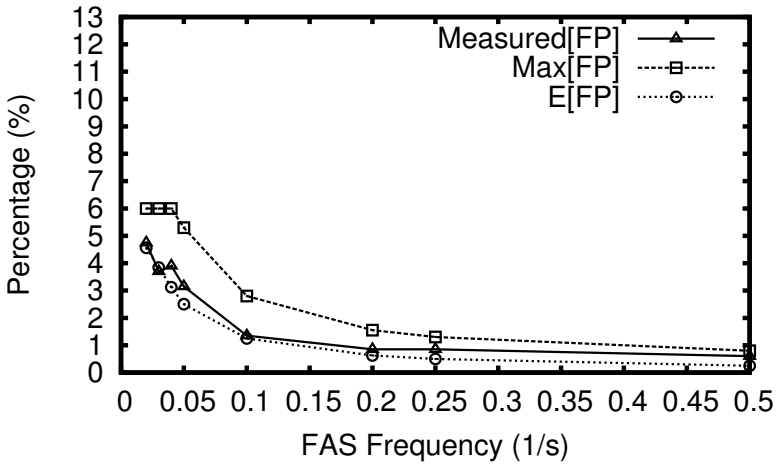


(b) $A = 90\%$

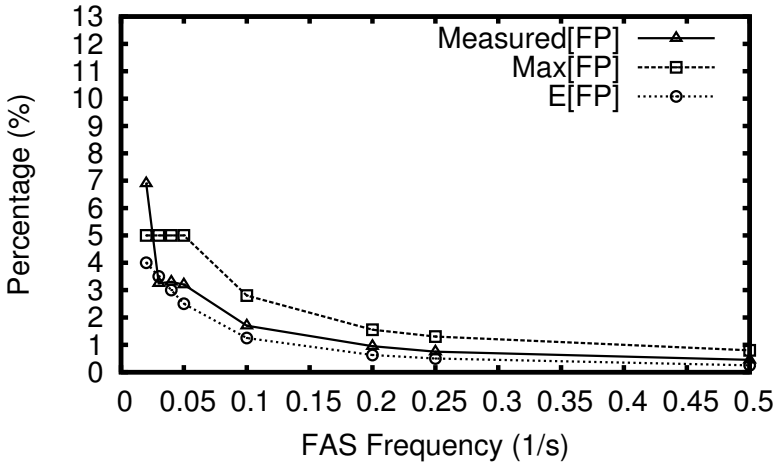


(c) $A = 92\%$

Figure 12: $E[\mathsf{FP}]$, $Max[\mathsf{FP}]$, and the measured false positive rate ($Measured[\mathsf{FP}]$) with respect to F, when A is 85%, 90%, 92%.



(a) $A = 94\%$



(b) $A = 95\%$

Figure 13: $E[\mathsf{FP}]$, $Max[\mathsf{FP}]$, and the measured false positive rate ($Measured[\mathsf{FP}]$) with respect to F, when A is 94% and 95%.

same trend consistently for $Measured[\mathsf{FP}]$. We also observed some outliers especially when F is just less than 0.05. See for instance the change of $Measured[\mathsf{FP}]$ in Figure 13(b) and Figure 13(a). It is also worth noting that the measured values are almost always slightly larger than the expected values. At the practical side, this could be interpreted as a hint to system administrators that when configuring their systems, they can choose to be pessimistic.

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 $E[\mathsf{ER}]$ function with respect to F defines the rate of change of $E[\mathsf{ER}]$ 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 $1/F \leq T_U$, $E[\mathsf{ER}] = 1/2FT \Rightarrow \partial(1/2FT)/\partial F = -1/2TF^2 = -1 \Rightarrow F = \sqrt{1/2T}$.

3.2.4 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 [16]. 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 [31] can be incorporated to monitor the dependability characteristics of partner services from composite services' perspectives.

There might be cases where extra logic is required to decide on partner service substitutions. Even if the primary partner service becomes unavailable, the composite service might have a tolerance margin for reconfiguration. Or it might be costly to substitute a critical partner service. Depending on the process, composite services might need to communicate with FAS to update critical information, service cache, notification interface/protocol as well. We did not take these needs into account.

CHAPTER IV

INDUSTRIAL CASE STUDY

In this chapter, we introduce an industrial case study for improving the quality of a service-oriented system from the broadcasting and content delivery domain. The system is an example of so-called Smart TVs, which emerged after the introduction of broadband connection to the TV systems. These systems utilize various services such as third party video content providers, popular social media platforms and games. Smart TVs combine these services into one portal application to facilitate user experience in various platforms such as set-top boxes and televisions. Nowadays various manufacturers, operators and broadcasters from all around the world provide services for Smart TVs.

In the following section we first motivate the relevance and importance of service quality for these systems. Then, we describe the realization of our approach in this context where different partner services roll into one composite service potentially being consumed by millions of end users. Afterwards, we explain our implementation and experiments. Finally, we discuss our observations and results as an evaluation of the approach.

4.1 Motivation

In this work we investigated a portal application (see Figure 14) developed by Vestek,[1] a group company of Vestel which is one of the largest TV manufacturers in Europe. This application is being utilized by Vestel as an online television service in the local market. The application is a platform comprising dozens of third party services,

[1] www.vestek.com.tr

including the most popular Web applications, audio/video streaming services and games. Among these, there are services with similar content, some of which are the most popular national broadcasters and online video sharing platforms. They serve the same content on different platforms as well, such as PCs, tablets, smart phones and Smart TVs. The application has unique user interfaces particularly designed for various television and set-top box products.

Figure 14: A snapshot from a portal application.

Digital televisions and set-top boxes were renamed as Smart TV after the introduction of broadband connection and Internet. This made televisions more capable than displaying linear content fed by tuners – they could be used for sending emails, surfing on the open Web, using social media services and watching movies, television programs or online videos. Additional features enable users to start, pause, stop and interact with television programs whenever wanted, known as non-linear content. Furthermore, recently introduced technologies mix linear and nonlinear content by launching content-specific Web applications to enrich use cases. These technologies made it possible to watch online videos, movies or popular television shows

from different television channels over broadband connection bypassing broadcasting. Hence, the target platforms are expanded to smart phones, tablets, PCs, MACs, game consoles and media centers which brings the term "Multi-Screen TV". The services providing audio/video content over broadband connection, which are known as OTT (Over the Top) services [32, 33], are considered to be major video services for Smart TVs [34].

Important network and telecommunication companies such as Cisco, Alcatel-Lucent, Ericsson and ATT have already started providing off-the-shelf solutions to television operators or OTT service providers. Besides, there are alternative cloud-based SaaS[2] or CDN[3] approaches. However, in general, the technological needs are almost the same regardless of the adopted workflow: delivery, security and management. In this work, we narrow our focus down to delivery, for assuring high service availability in these composite systems.

The complexity of such environments creates various potential weaknesses which might damage service quality tremendously considering the whole network. Overall system might be affected in various ways depending on the type and location of faults. Each type of problem requires its own way of handling. This makes it difficult to provide a general approach covering all types of problems. For instance, considering only one OTT service, there might be content-related problems which leave front end devices unable to play (e.g. unsupported format) or server side problems which result in faulty feeds, wrong URLs or no response at all. On the other hand, it is also possible to experience intermittent network outages causing video and audio freezes, long buffering periods, end-of-stream or lip-sync errors. In this work, we are particularly interested in content-related faults that propagate through the portal application and affect the user experience negatively.

[2]Software as a Service
[3]Content Delivery Network

Fortunately, most of the challenges are already addressed in existing systems like IPTV[4]. However, it appears that current systems fall short for the next generation Multi-Screen TV platforms. As detailed in [35], the reasons are: *(i)* Systems are getting more complex with the increasing variation in playback devices, platforms, protocols, formats, resolution and bandwidth requirements, causing more faults. *(ii)* Higher demand from new devices will create extra traffic over their existing network. *(iii)*Customer expectations will likely rise as the contents get enriched and extra fees are put in charge by network providers. *(iv)* Viewers are willing to pay more for better quality if content is satisfying. Content providers raise quality constraints to differentiate and compete.

In the following section we focus on OTT services, how we can use FAS to mask and avoid the faults threatening user satisfaction and service availability.

4.2 Realization of the Approach

Figure 15 describes a typical process when a client uses the portal application. When the application is loaded, the client first selects a partner service. If the selected service is a video provider, available categories are retrieved on the server side by invoking the partner service. The response from the partner service is used for the preparation of a page to convey available categories. This page appears on the client device. The client picks a category and triggers another transaction. This time, the portal application invokes the partner service again and retrieves an up-to-date collection of video items for the selected category. Categories are distinguished with a unique identifier that is passed as a parameter while using partner services' APIs. If the request succeeds, video items are displayed with additional meta information such as thumbnail image, title, duration and popularity. Then the client chooses a

[4]Internet Protocol Television

video item and starts streaming which is performed directly between the client and the partner service.

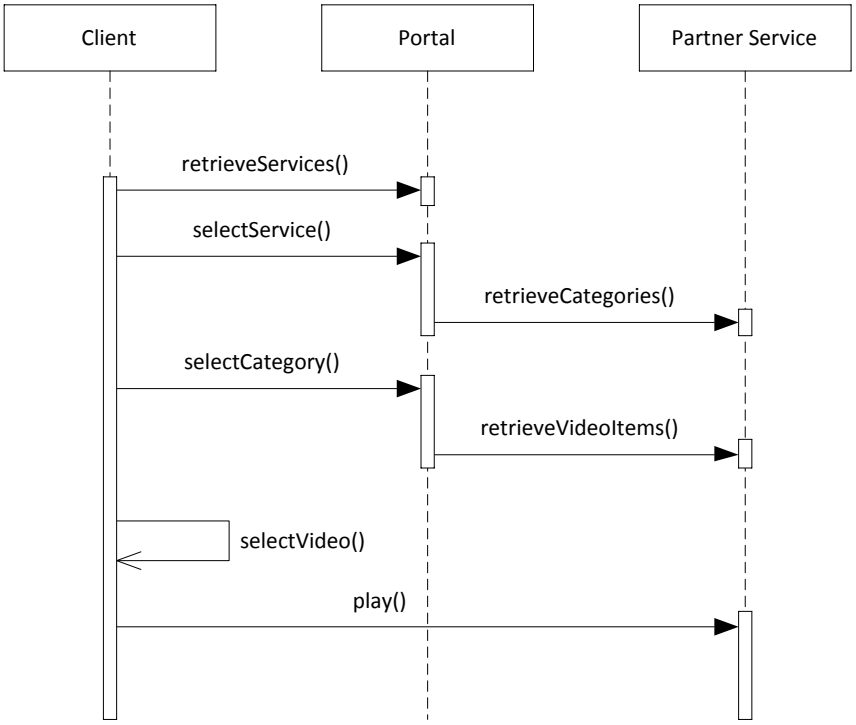


Figure 15: Sequence diagram of video streaming on the portal application.

As we described in Chapter 2, this sequence resembles a typical service-oriented process. The portal application, as a composite service, has dependencies to some third party content providers which are essentially partner services. Multiplicity of these video services fulfills the redundancy requirements to introduce fault masking approach described in Chapter 2.2. Normally, there isn't a mechanism in the portal application to avoid possible external faults originated from the partner services. Hence the application is vulnerable to propagation of these faults causing errors and possibly failures. Here, we aim at detecting possible content problems and masking them with FAS before they are demanded.

To experiment our approach, we exerted several applications. We implemented a prototype composite service taking the role of the portal application. We kept

the utilization of partner services and server side interactions as original. However, we changed the format of the responses so that we can capture, log and analyze them easier. We also implemented the required interfaces so that the portal can communicate with FAS. We deployed FAS as a stand alone web application. We used several client applications to test our prototype portal application. We provide detailed information in Section 4.3 about these implementations.

Before the actual experiments, we first made a preliminary analysis to gather information regarding the type and frequency of faults existing in the video services of the portal application. Afterwards, we picked a particular partner service and performed several tests with different monitoring frequencies. In the following subsection we explain the preliminary analysis and present the results.

4.2.1 Preliminary Analysis

First, we analyzed what type of faults the partner services had. We picked two of them, Izlesene[5] and Ekolay[6], to make observations about their availabilities. We aimed at responding to these questions: *(i)*Are there any content problems in the feeds? *(ii)* How are they changing within a period of time?

To investigate these questions, we used a test application that picked a random video item from a random video category of a specified partner service to demonstrate the real use-case scenario. We sent video URL of the selected item to the FAS service in order to monitor and log the status of the video. The FAS service logged time, URL and test result after each trial. We ran the system in this configuration for 11 days and collected data for approximately 1.6 million requests. For the Ekolay service, 60% of the requests were unsuccessful, i.e. the system detected anomaly. On the other hand, almost 100% of the requests were successful for the Izlesene service.

After observing broken links in the feeds from Ekolay service we decided to unveil

[5]http://www.izlesene.com/
[6]http://video.mynet.com

what type of faults were prevalent in the system. In case of frequent intermittent faults, T would be proportionally small and F would be high as we discussed in Chapter 3.2.3. Higher F causes more network traffic and resource usage that reduces the efficiency. Yet, in case of infrequent changes in the feeds, we can set lower F and leverage FAS more efficiently.

We went further and traced which video links were broken in the feeds. Results pointed out two main observations: *(i)* If a video link was found to be erroneous, its status never changed until removed from the service feed. *(ii)* The root cause was HTTP 404 response code [36] when a video link was found to be broken. Based on the first observation, we concluded that faults associated with the Ekolay service were not intermittent network issues. Based on the second observation, we understood that there was a server-side defect which caused broken links in the feeds. As a result of these inferences, we conclude that it is possible to utilize the FAS service in order to detect the anomalies associated with the Ekolay feeds.

As a next step, we focused on determining a reasonable T. Based on our first observation, we know that a video item remains broken until it is unloaded. Hence, we can estimate an efficient monitoring frequency based on how often Ekolay content gets updated. It is unlikely to see an alteration in any video item until an update occurs. However, if there is an update observed, several outcomes are possible: a broken video item may get fixed, a new broken video item may be introduced, or a video item may be unloaded. In any case, FAS must check Ekolay content again and inform the composite service accordingly.

To reveal how often Ekolay content is updated, we implemented a client application. At each turn, this application requests all video items by iterating through available categories. Then it compares the video items one by one in the same order with the items retrieved within the previous turn. It checks meta data and number of video items for each category in addition to meta data and the order of each video

item. It logs time, service name and whether the service feed is changed or not. We monitored Ekolay, Izlesene and 3 other video services available in the Vestek portal for 14 days. Figure 16 shows the results for the Ekolay service. It can be seen that the changes are observed in some certain time slices. Ekolay's content is changing relatively more sparsely compared to others. The results for other services can be found in Section 4.5.

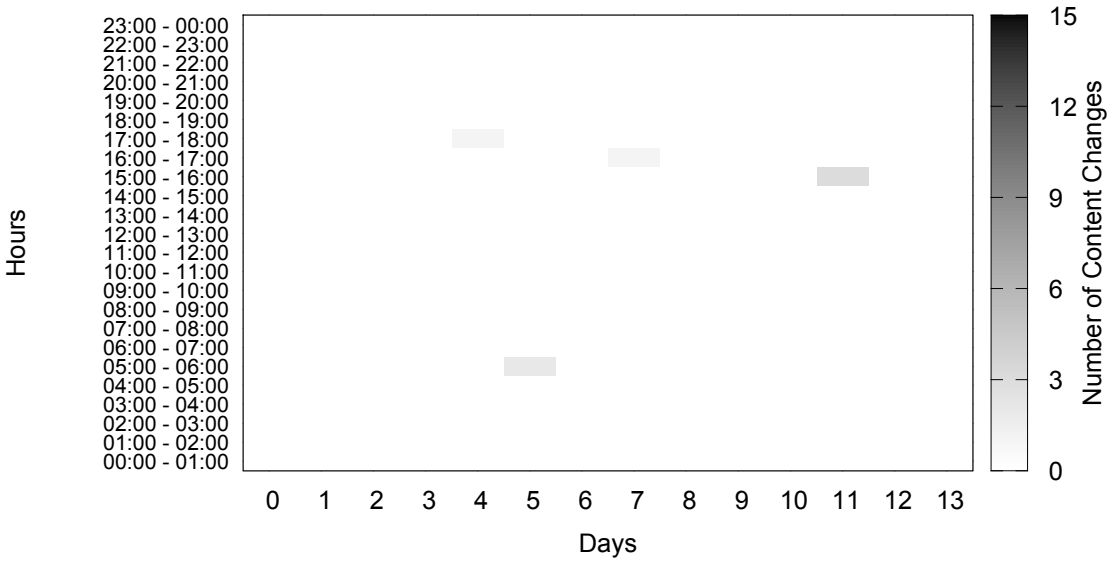


Figure 16: Content changes of the Ekolay service in a 14-day period.

4.3 Implementation

We developed applications using the Python language version 2.6. We did not use any third party libraries. We utilized the HTTP protocol for the communication among the stand alone web applications. Data exchange was performed by using XML to handle serialization/deserialization. We also logged our test results in XML format to facilitate post-processing. In the following subsections we give details about the prototype portal application and the FAS service.

4.3.1 The Portal Application

We implemented a web application akin to the Vestek portal from the point of partner service interactions. This application constitutes a composite service that can be representative for investigating the effectiveness of a fault masking service. The main reasons of utilizing our own composite service are threefold. First, the original implementation uses .NET technology which requires Windows environment to deploy. However, we prefer to stay on the Linux environment with open source tools. Second, the original implementation has authentication and session management causing extra complexity in the experiments. However, we are particularly interested in partner service interactions and anomalies in the feeds. Third, the original HTTP responses are in the form of HTML pages with dynamic contents particularly designed for human interaction on special target platforms. However, we aim at leveraging client side applications that perform automated tests. Hence, our implementation employs XML-based approach for data exchange to interact with automated client applications.

We defined an interface for clients to simulate the use-case scenarios described previously. We ported five content provider services from the original application. We parameterize partner service selection, which can be specified by filling in query parameters in the HTTP requests. Extra business logic is also implemented to communicate with the FAS service and receive callbacks about the faulty video items in the service feeds. To facilitate the testing process, we replicated the Ekolay binding in our portal application. The first replica is left intact to respond original feed without taking into account the FAS interventions. We call this service the "native replica". The second replica is implemented in a way that the original feeds are manipulated according to the FAS interventions. We call this service the "rectified replica".

To manipulate erroneous videos, our application maintains a list of video items according to the FAS service callbacks. Whenever a callback is received, it handles

the notification and updates the list accordingly. A callback might inform a new discovery of an erroneous video and its alternative. Or, it can inform a fix about a video item which was previously reported as broken. When the portal application receives a request for the rectified replica, it retrieves up-to-date content from Ekolay and exchanges the items reported by the FAS service with their alternatives.

4.3.2 The FAS Service

We implemented a stand alone Web application to demonstrate FAS. This implementation is intended to be used in conjunction with the portal application described previously. It maintains a list of video items which the portal application informs by using the HTTP interface. It periodically checks the items in the list and observes their up-to-date status. If an error is detected, it tries to find an alternative video item from another partner service.

To detect errors, we implemented a checker mechanism as depicted with a class diagram in Figure 17. Each checker is a necessary step that will be carried out while testing a partner service. They are fine-grained, reusable, independent components that create a high-level, functional verification utility when aligned together. A detector represents a complete error detection utility that uses a sequence of checkers to perform tests. We create concrete detectors by grouping and configuring desired checkers considering the functional requirements of the services we monitor.

In this case study we particularly deal with video items. Error detection module is responsible for detecting anomalous video items. The portal application sends the video URLs via HTTP POST requests. To ensure that streaming is successful for a particular video item, we make an HTTP request to its URL and follow these steps: *(i)* Check that a valid HTTP response is received. *(ii)* Check that response body of the HTTP response is not empty. *(iii)* Check that the MIME type[7] of the HTTP

[7] An identifier for file types.

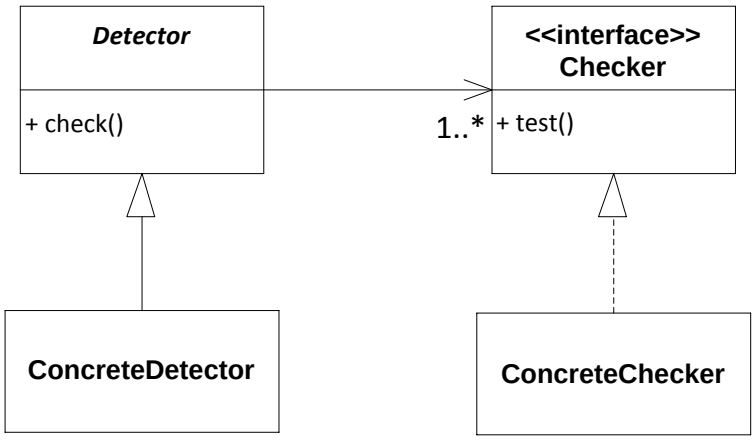


Figure 17: Class diagram of the error detection module.

response is "video/mp4".[8] *(iv)* Check that a chunk of 100 KB can be buffered from the response body. We implemented a checker corresponding to each step in this strategy and created a VideoDetector that aligns these checkers. We parameterized MIME type and buffer size in the related checker implementations so that they can be configured and reused for different types of services.

For fault handling, we deployed only one partner service in the service cache. We chose the Izlesene service to find alternative videos when an anomaly is detected. We did not perform any lookup or interact with a service registry to discover new Web services on the fly. When an anomaly is detected, the implementation uses Izlesene APIs to find an alternative video item. If there is any, it sends a notification to the portal application about the erroneous video and its replacement. If it detects a fixed video which is known to be erroneous, a notification is sent to inform that the original video can be used.

[8]There is a dedicated header in HTTP Response called "Content-Type". It specifies the MIME type of the response body.

We conducted several experiments as described in Section 4.4. We utilized this FAS realization and the portal application for testing our approach.

4.4 Experimental Setup

We performed experiments to observe and measure the effect of FAS to the overall availability of the portal application. For this purpose, we deployed a client application requesting content from the portal by selecting both the native replica and the rectified replica at each iteration. After obtaining the responses, the client picks a video item randomly and tests streaming by using the meta data from two different replicas. At each trial, it logs time stamp, video ID and test results for both partner services. The results give us a chance to compare and count directly how many faulty video items are successfully detected and masked by the FAS service.

We repeated this experiment three times by applying different approaches for the monitoring strategy. These strategies are:

(i) *Average:* Take T as the average period of time between two content changes.

(ii) *Minimum:* Take T as the minimum period of time between two consecutive changes.

(iii) *Exponential back-off:* Take T as the minimum period of time between two consecutive changes and apply exponential back-off when there is no content change.

By using the preliminary analysis results mentioned in Section 4.2.1 and our conclusion $F = \sqrt{1/2T}$ from Subsection 3.2.3, we calculated the values of $1/F$ for each strategy. There were 7 changes in 14 days in the Ekolay feed. Therefore we set $1/F$ to 588 seconds for the *average* strategy. We compared the durations between each consecutive changes and found the minimum as 60 seconds which yields 11 seconds as $1/F$ for the *minimum* strategy. For the *exponential back-off* strategy, we again used the same value with *minimum* but employed the exponential back-off algorithm

to double $1/F$ when there is no change.[9] Until a change is detected, the new value is determined as $1/F_{k+1} = 2^k/F$ where k is the number of iterations starting from the last change.

Tests were conducted using a PC with Intel(R) Core 2 Duo P8600 at 2.40 GHz processor and 4 GB RAM. The applications were run on Ubuntu 10.04 LTS and Python 2.6.5 Runtime Environment.

4.5 Results and Discussion

We monitored five video services from the Vestek portal as we described in Subsection 4.2.1. We observed how often their feeds changed for 14 days. Figure 16 in Section 4.2.1 and Figures 18, 19, 20 and 21 show the results for Ekolay, Izlesene, Bloomberght, Vidivodo and ShowTv services, respectively.

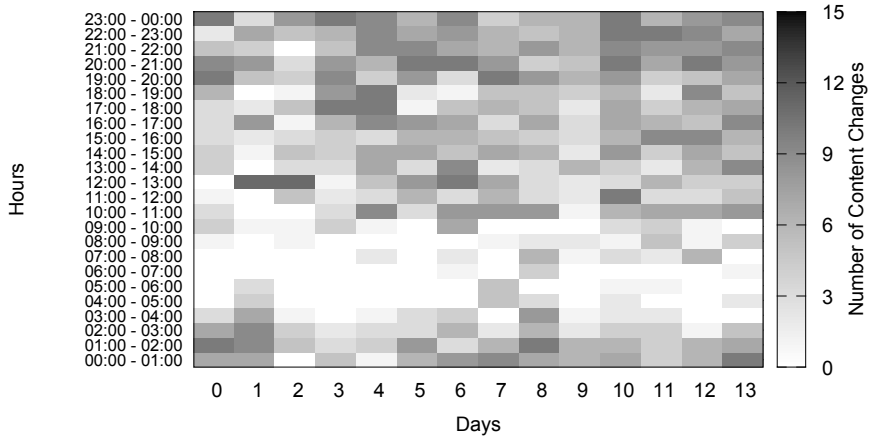


Figure 18: Content changes of the Izlesene service in a 14-day period.

We observe that each service has its own change regime with some similarities as well. Ekolay and ShowTv change very rarely compared to others. Hence, for these two services, we can consider a lightweight monitoring strategy triggered with these changes. On the other hand, Izlesene has a much more frequent change characteristic.

[9] F is bounded by a predetermined limit.

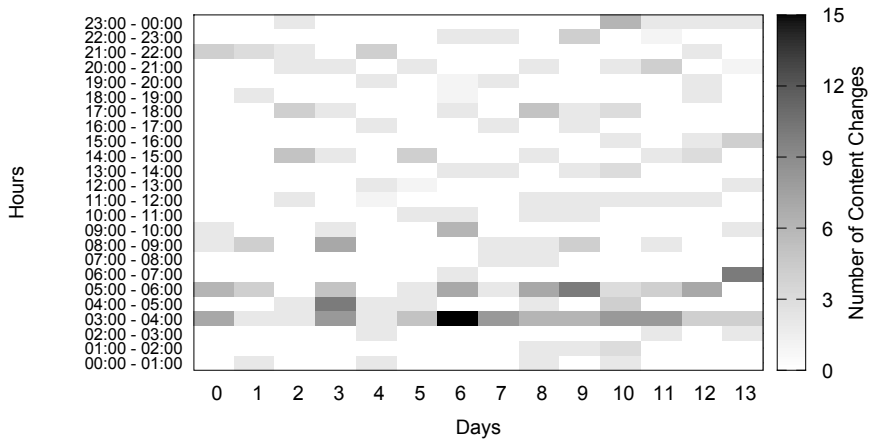


Figure 19: Content changes of the Bloomberght service in a 14-day period.

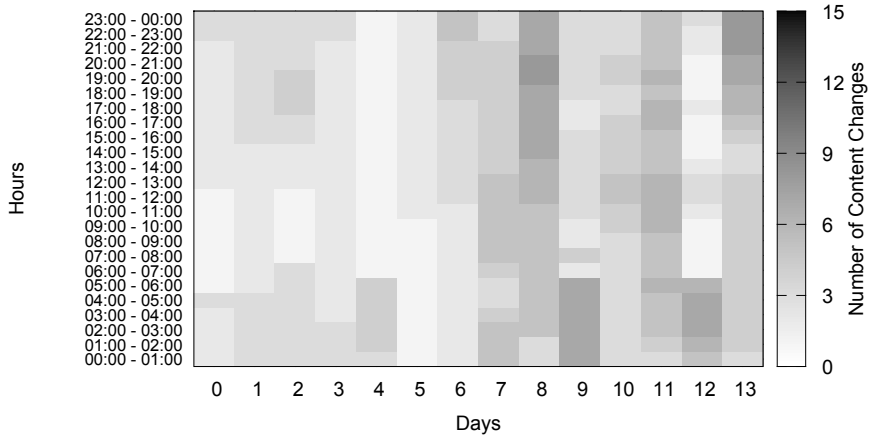


Figure 20: Content changes of the Vidivodo service in a 14-day period.

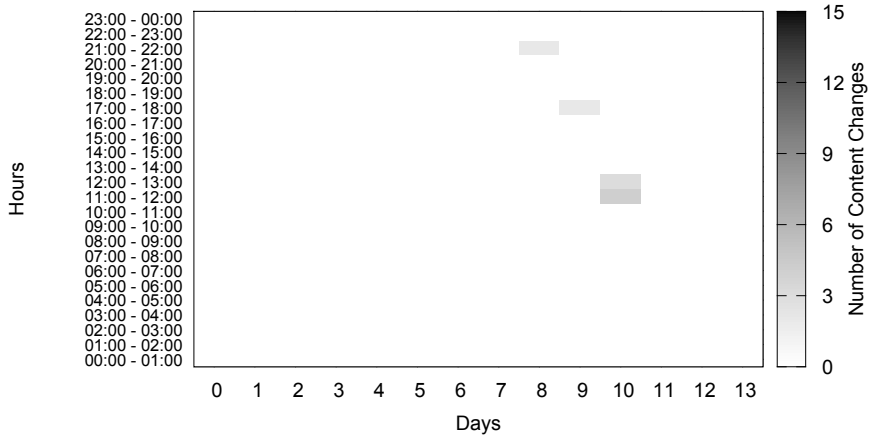


Figure 21: Content changes of the ShowTV service in a 14-day period.

As we can see in Figure 18, it is more stationary between 3:00 – 9:00 and hectic after 18:00. We can utilize a customized strategy based on this daily pattern with different monitoring frequencies per each time slice. Vidivodo and Bloomberght appear to have more uniform distribution of change on a daily basis. A fixed monitoring frequency can be considered for these services. Besides, Bloomberght is slightly erratic between 3:00 – 5:00.

It is important to determine an efficient and suitable monitoring strategy while utilizing FAS. But it can be seen that each service has its own regime which is subject to change. Our observation was 14 days long. It would be more accurate over a longer period of time. Hence an additional subsystem in FAS responsible for determining an accurate and adaptive monitoring strategy would be useful.

Service	Number of Faults	Total Number of Attempts	Availability (%)
Native Replica	9368	55835	83.22
Rectified Replica	10	55835	99.98

Table 1: Experiment results for the *average* strategy.

Table 1 shows the results of the *average* strategy described in Section 4.4. We observed that 9368 streaming attempts out of 55835 failed for the native replica. This corresponds to 16.78%. For the rectified replica, only 10 streaming attempts, which is approximately 0.02%, failed. There are two attempts that failed for the native replica and the rectified replica at the same time, i.e. FAS was unable to detect and fix these. There are 8 attempts where the trial is successful from the native replica but the rectified replica failed.

Service	Number of Faults	Total Number of Attempts	Availability (%)
Native Replica	9269	54918	83.12
Rectified Replica	5	54918	$\succ$99.99

Table 2: Experiment results for the *minimum* strategy.

The results of the *minimum* strategy are given in Table 2. There are 9269 faults out of 54918 attempts for the native replica. This corresponds to 16.88%. However there are only 5 failing requests, which make less than 0.01% for the rectified replica. There is only one failing request both for the native and rectified replica. There are 4 attempts that are successful for the native replica but failed for the rectified replica.

Service	Number of Faults	Total Number of Attempts	Availability (%)
Native Replica	9490	56200	83.12
Rectified Replica	8	56200	$\prec$99.99

Table 3: Experiment results for the *exponential back-off* strategy.

Table 3 shows the results for the *exponential back-off* strategy. For the native replica, 9490 failing attempts are detected out of 56200. This corresponds to 16.88%. Only 8 trials were unsuccessful, which correspond to $\sim$0.01%. 2 attempts failed for both replicas at the same time. There are 2 attempts failing for the rectified replica while succeeding for the native one.

Strategy	$1/F$ (seconds)	Number of Faults	Availability (%)
Average	588	10	99.98
Minimum	11	5	$\succ$99.99
Exponential Back-off	$2^k * 11$ and k=0,1,2...	8	$\prec$99.99

Table 4: Comparison of the monitoring strategies.

Table 4 sums up the results of the three monitoring strategies. The best service availability is obtained with the *minimum* strategy, resulting slightly more than 99.99%. It is followed by the *exponential back-off* with a difference less than 0.01%, and the *average* strategy with 99.98% availability. On the other hand, the *average* strategy introduces the least overhead since network and resource usage is directly proportional to the monitoring frequency. As we can see from Table 1, the *average* strategy increased the availability from 83.22% to 99.98%. Even though the monitoring frequency of the *minimum* strategy is approximately 53 times greater than the

average strategy, service availability is only increased by 0.01%. This makes the *average* strategy more appealing, assuming that the 0.01% difference in the availability is negligible.

CHAPTER V

RELATED WORK

Anatoliy et al.[37] categorized errors and failures specific to service-oriented systems. They introduced three main categories: *i)* network and system failures, *ii)* service errors and failures, and *iii)* client-side binding errors. Our approach focuses on network/system failures and client-side binding errors. Our goal is to forecast these errors/failures, and if possible avoid them to increase the availability of composite services.

So far, research efforts for improving the dependability of service-oriented systems have focused on variety of fault tolerance strategies [9, 18, 17]. We particularly focus on fault masking, a distinctive strategy compared to others. An analysis of the literature also reveals that dependability improvement has been mainly facilitated by means of frameworks [37, 18], architectural methods [38, 10], reliable service connectors [39], proxies [24] and service dispatchers [40]. We propose implementing a stand alone service to which other services can register for improving their dependability.

There exist service brokers and architectural frameworks [14] 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 [14]. 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.

Unlike fault tolerance, the dependability means such as error prevention, fault forecasting and removal at run-time [21] have not received much attention in the

literature for their application to service-oriented systems. This is because, these techniques are mainly considered to be applied at design time to avoid faults during software development and maintenance. Another observation from the analysis of the literature reveals that research efforts mainly focus on providing middleware and framework support to develop dependable services. These results are valuable especially for addressing internal faults; services should be developed/refactored in a rigorous manner with the help of such middleware and/or frameworks. However, forecasting, detection and the handling of many external faults can (partially) be implemented by 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 idea has only been realized in the context of software/service testing (Testing as a Service - TaaS [23]) and not for fault handling at runtime.

Previously, the use of a proxy Web service was proposed to replace failed or slow services with alternative services [24]. Hereby, the quality monitoring is performed by the composite service. The source code of the composite service is automatically instrumented to add this functionality. As a drawback, there is a hard-coded primary service that is always tried first. The proxy service is used for diverting to an alternative service only when/after a failure occurs. So, this approach tolerates faults that are detected when the composite service is demanded. However we aim at detecting and masking faults seamlessly by an external Web service before the composite service is demanded.

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 [16] was introduced to tackle this problem by enabling semantic-based service substitution.

Zheng and Lyu [8] 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 of 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.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

We introduced an approach for masking faults during the invocation of partner services and as such, preventing errors in composite services. We developed FAS, an external fault masking 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.

We examined an industrial use-case from the broadcasting domain. Hereby, we applied our approach to real world video services. We performed a preliminary analysis on commonly used content providers to reveal common error types and possible monitoring strategies while leveraging FAS. We investigated five of such services and analyzed how frequent their feeds change. We applied three different strategies for calculating the monitoring frequency. We conducted several experiments and compared the effectiveness of these strategies. As expected, strategies that employ higher monitoring frequencies resulted in higher availability. However, the improvement in availability turned out to be insignificant with respect to the additional overhead of

increased monitoring frequency. Only 0.01% improvement in availabiliy was observed, when the monitoring frequency was increased 53 times. A strategy based on exponential back-off might provide an acceptable trade-off point among the alternative strategies. The final choice would also depend on the available resources and the importance of the content.

As future work, we plan to enhance FAS so that it can adapt both the service checking strategy and the checking period at runtime, based on the monitored failure/usage frequencies and response times.

APPENDIX A

SIMULATION RESULTS

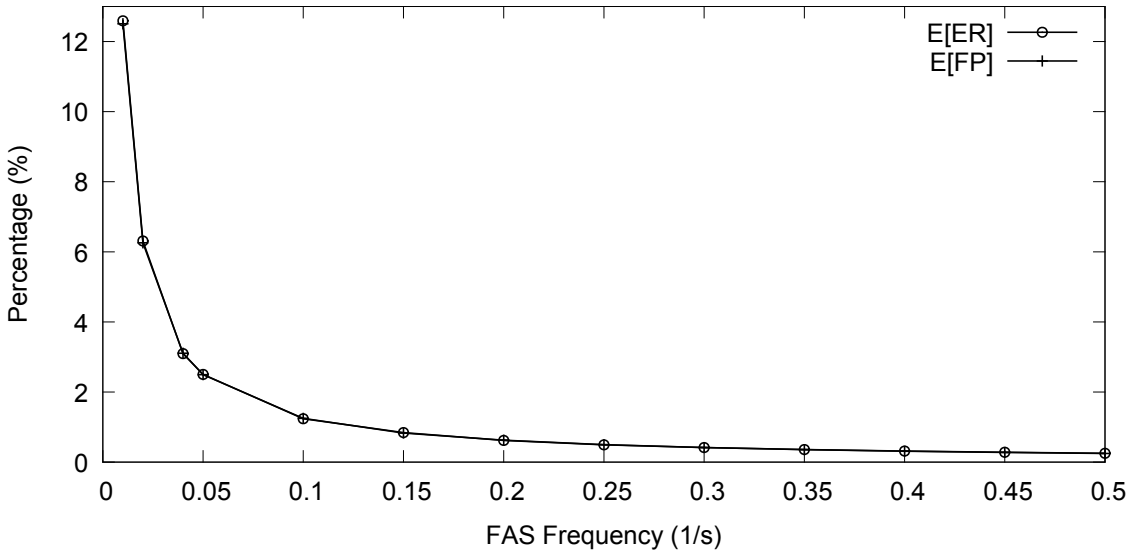


Figure 22: Simulation results for $A = 60\%$.

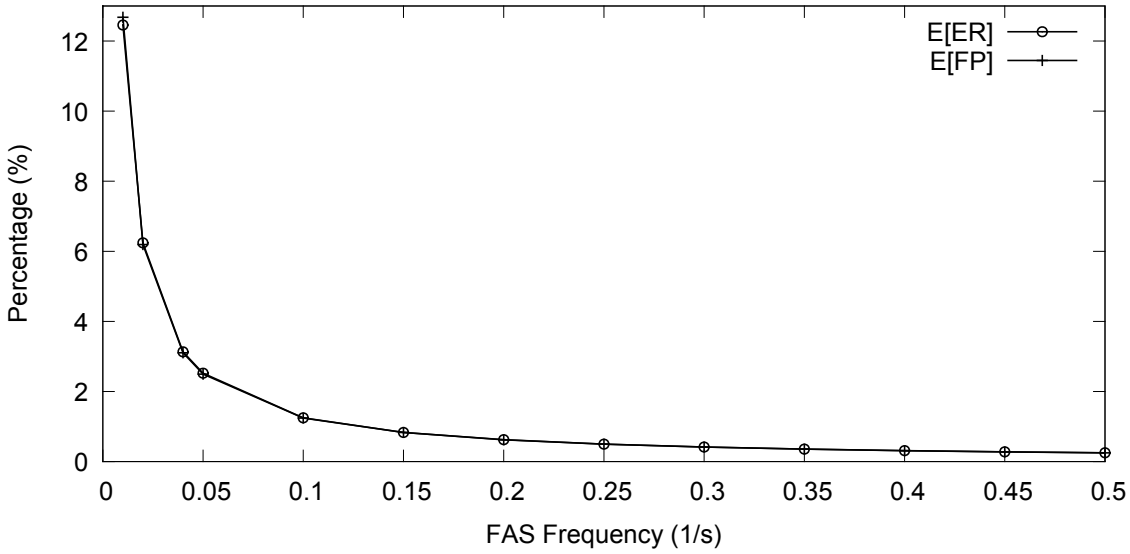


Figure 23: Simulation results for $A = 65\%$.

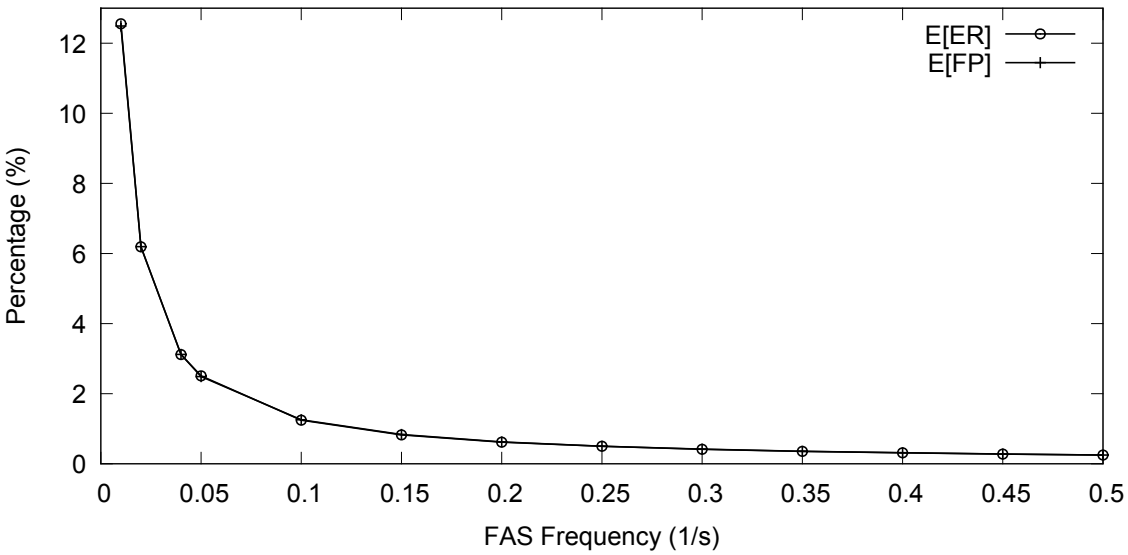


Figure 24: Simulation results for $A = 70\%$.

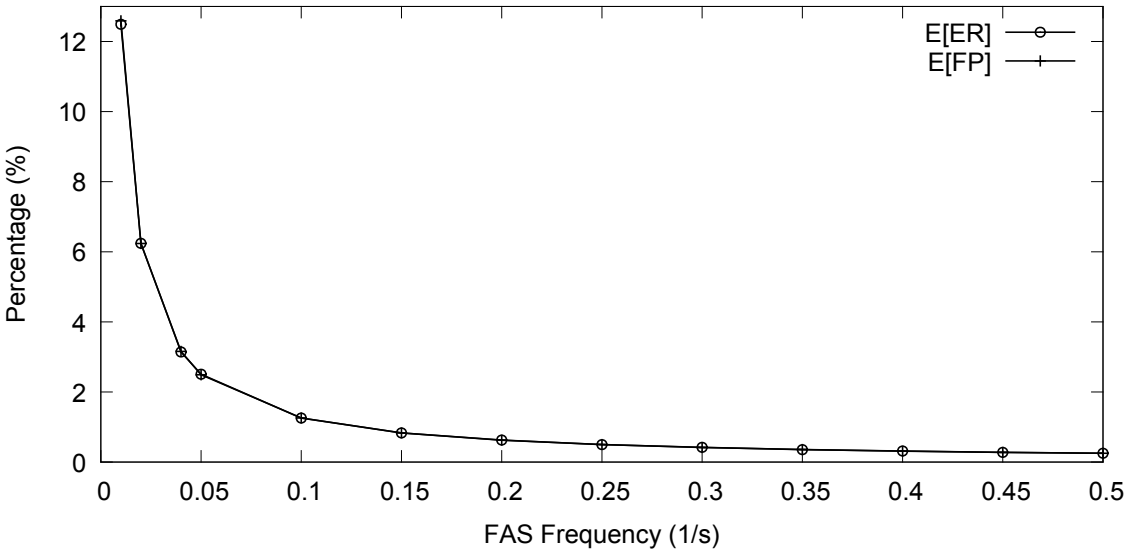


Figure 25: Simulation results for $A = 75\%$.

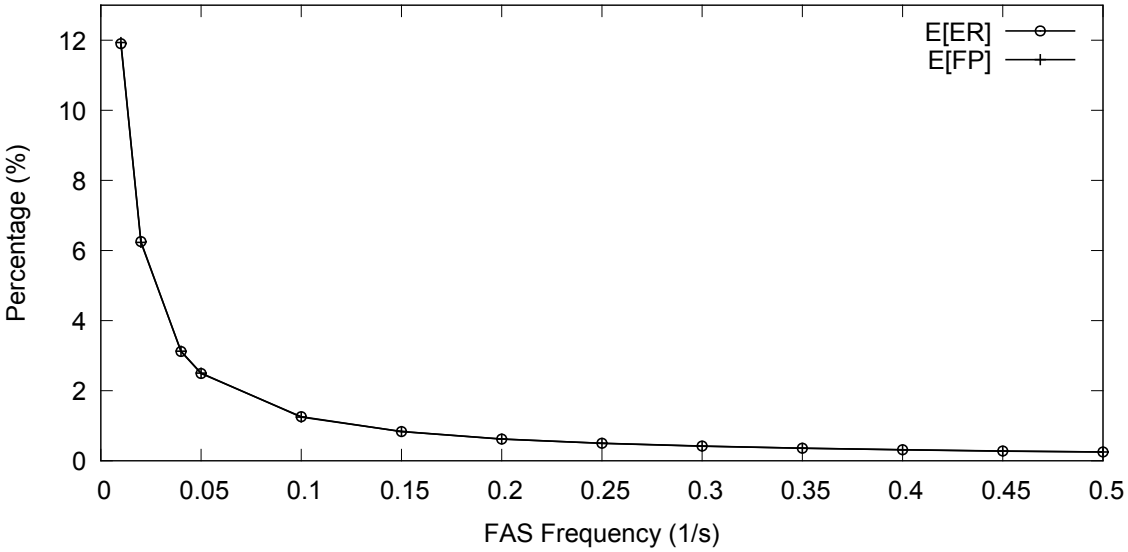


Figure 26: Simulation results for $A = 80\%$.

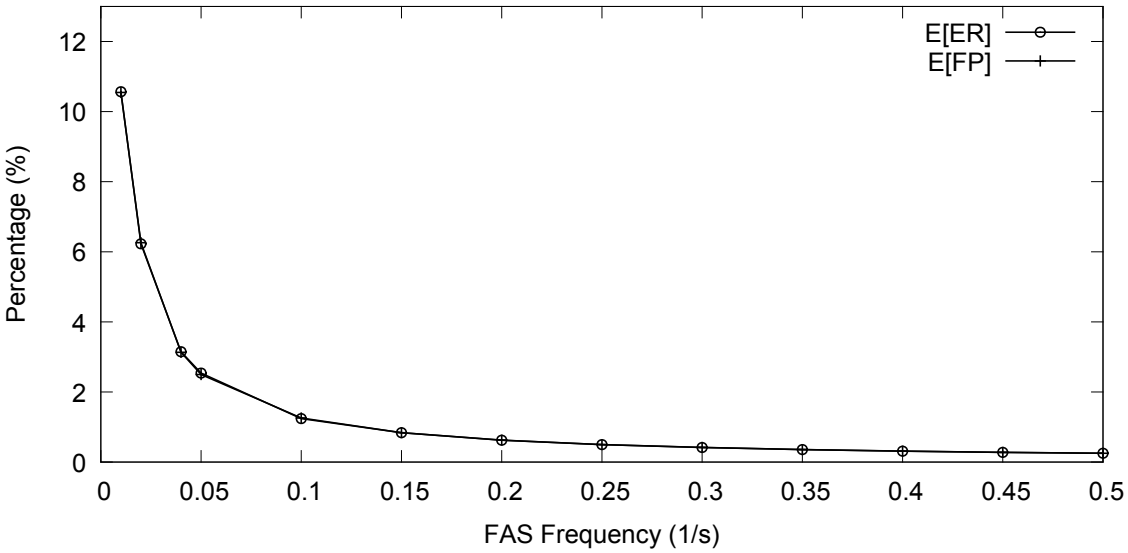


Figure 27: Simulation results for $A = 85\%$.

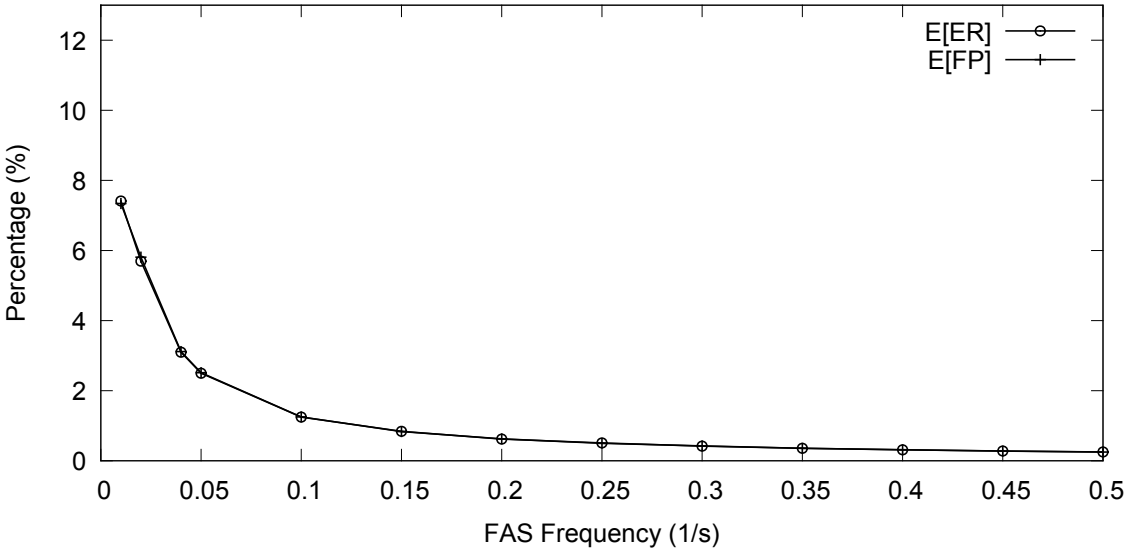


Figure 28: Simulation results for $A = 91\%$.

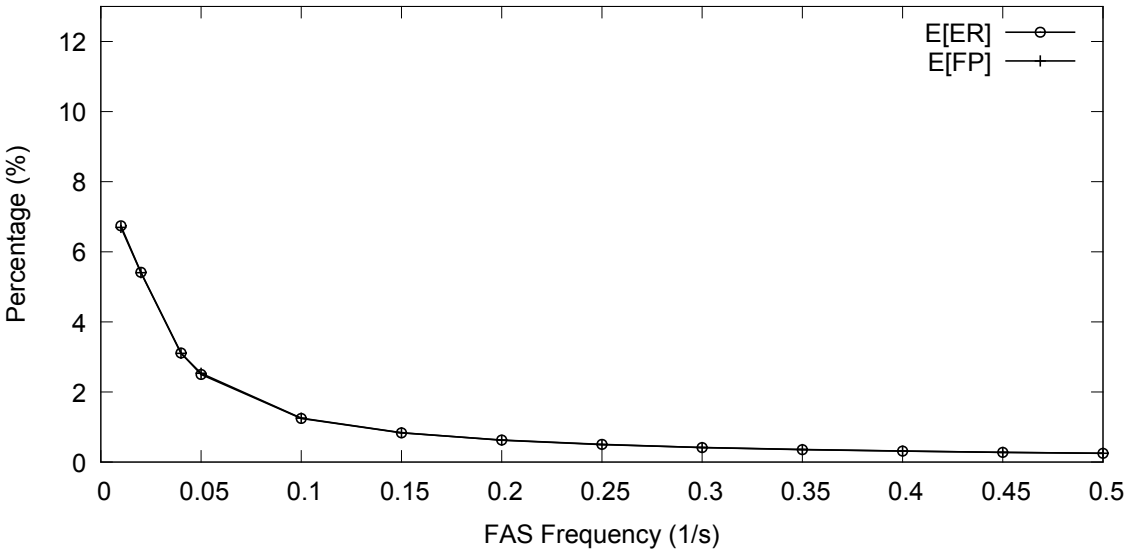


Figure 29: Simulation results for $A = 92\%$.

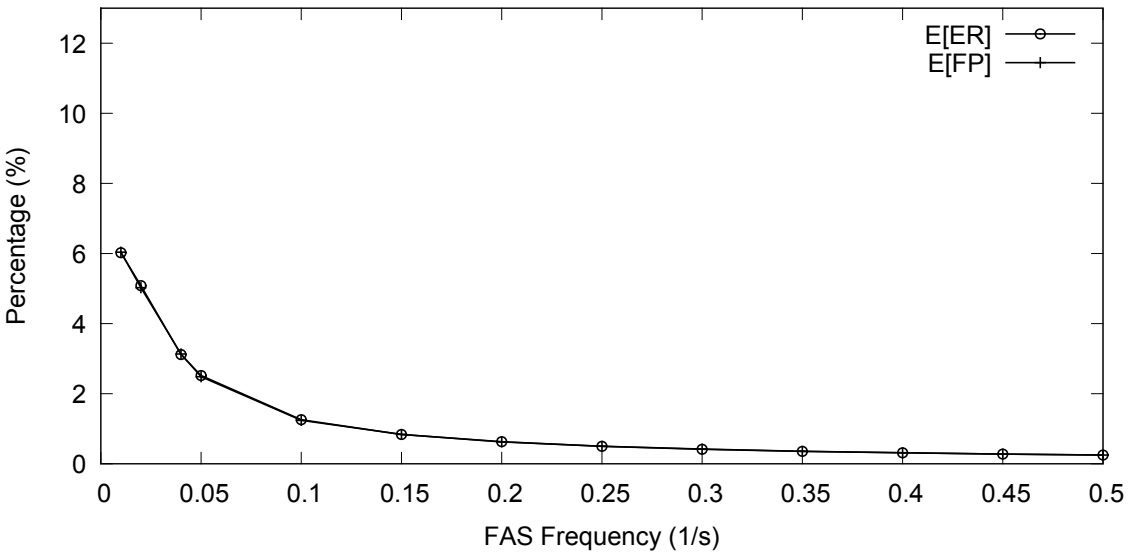


Figure 30: Simulation results for $A = 93\%$.

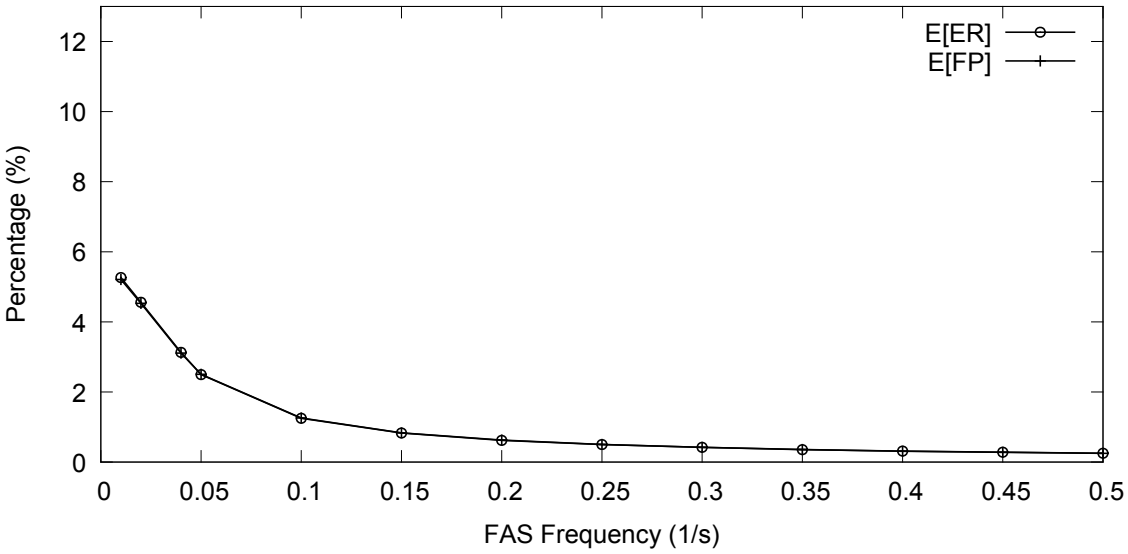


Figure 31: Simulation results for $A = 94\%$.

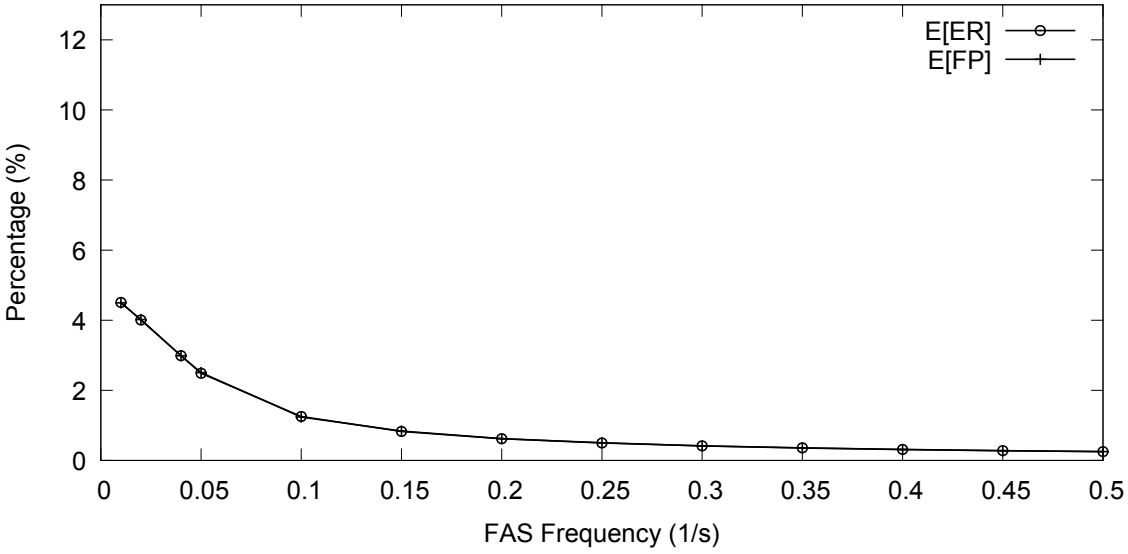


Figure 32: Simulation results for $A = 95\%$.

References

[1] D. Georgakopoulos and M. Papazoglu, eds., *Service-Oriented Computing*. MIT Press, 2009.

[2] B. Medjahed, A. Bouguettaya, and A. Elmagarmid, "Composing Web services on the semantic Web," *VLDB Journal*, vol. 12, no. 4, pp. 333–351, 2003.

[3] A. Tsalgatidou and T. Pilioura, "An overview of standards and related technology in Web services," *Distributed Parallel Databases*, vol. 12, no. 2, pp. 135–162, 2002.

[4] M. P. Papazoglou and D. Georgakopoulos, "Introduction to a special issue on service-oriented computing," *Communications of the ACM*, vol. 46, no. 10, pp. 24–28, 2003.

[5] D. Jordan and J. Evdemon, "Web services business process execution language version 2.0," 2009. OASIS Standard.

[6] Z. Zheng, Y. Zhang, and M. Lyu, "Distributed QoS evaluation for real-world web services," in *Proceedings of the IEEE International Conference on Web Services*, pp. 83–90, 2010.

[7] A. Zarras, M. Fredj, N. Georgantas, and V. Issarny, "Rigorous development of complex fault-tolerant systems," in *Engineering Reconfigurable Distributed Systems: Issues Arising for Pervasive Computing*, no. LNCS 4157, (Berlin, Heidelberg), pp. 364–386, Springer-Verlag, 2006.

[8] Z. Zheng and M. Lyu, "An adaptive QoS aware fault tolerance strategy for web services," *Journal of Empirical Software Engineering*, vol. 15, no. 4, pp. 323–345, 2010.

[9] A. Liu, Q. Li, L. Huang, and M. Xiao, "FACTS: A framework for fault-tolerant composition of transactional web services," *IEEE Transactions on Services Computing*, vol. 3, no. 1, pp. 46 –59, 2010.

[10] L. Baresi and C. Ghezzi, "Towards self-healing service compositions," *Proceedings of the 1st Conference on the Principles of Software Engineering*, pp. 27–46, 2004.

[11] D. Ardagna and B. Pernici, "Adaptive service composition in flexible processes," *IEEE Transactions on Software Engineering*, vol. 33, pp. 369–384, 2007.

[12] V. Cardellini, V. D. Valerio, V. Grassi, S. Iannucci, and F. L. Presti, "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.

[13] D. Ardagna and R. Mirandola, "Per-flow optimal service selection for web services based processes," *Journal of Systems and Software*, vol. 83, no. 8, pp. 1512–1523, 2010.

[14] V. Cardellini, E. Casalicchio, V. Grassi, F. L. Presti, and R. Mirandola, "Architecting dependable systems vi," ch. Towards Self-adaptation for Dependable Service-Oriented Systems, pp. 24–48, Berlin, Heidelberg: Springer-Verlag, 2009.

[15] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "QoS-aware middleware for web services composition," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 311–327, 2004.

[16] M. Fredj, N. Georgantas, V. Issarny, and A. Zarras, "Dynamic service substitution in service-oriented architectures," in *Proceedings of the IEEE Congress on Services*, pp. 101–104, 2008.

[17] G. Dobson, "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.

[18] C.-L. Fang, D. Liang, F. Lin, and C.-C. Lin, "Fault tolerant Web services," *Journal of System Architure*, vol. 53, no. 1, pp. 21–38, 2007.

[19] G. Canfora, M. D. Penta, R. Esposito, and M. Villani, "A framework for QoS-aware binding and re-binding of composite web services," *Journal of Systems and Software*, vol. 81, no. 10, pp. 1754–1769, 2008.

[20] K. Gulcu, H. Sozer, and B. Aktemur, "FAS: Introducing a service for avoiding faults in composite services," in *Proceedings of the 4th International Workshop on Software Engineering for Resilient Systems*, (Pisa, Italy), pp. 106–120, 2012.

[21] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11 – 33, 2004.

[22] Amazon.com, "Elastic Compute Cloud (EC2)," 2012. http://aws.amazon.com/ec2.

[23] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: a software testing service," *SIGOPS Operating Systems Review*, vol. 43, pp. 5–10, 2010.

[24] O. Ezenwoye and S. Sadjadi, "A proxy-based approach to enhancing the autonomic behavior in composite services," *Journal of Networks*, vol. 3, no. 5, pp. 42–53, 2008.

[25] J. Simmonds, G. Yuan, M. Chechik, S. Nejati, B. O'Farrell, E. Litani, and J. Waterhouse *IEEE Transactions on Services Computing*.

[26] W. Robinson and S. Purao, "Monitoring service systems from a language-action perspective," 2011.

[27] G. Wu, J. Wei, and T. Huang, "Flexible pattern monitoring for WS-BPEL through stateful aspect extension," in *Proceedings of the IEEE International Conference on Web Services*, pp. 577–584, 2008.

[28] The Apache Software Foundation, "Axis," 2012. http://axis.apache.org/.

[29] The Apache Software Foundation, "Tomcat," 2012. http://tomcat.apache.org/.

[30] The Apache Software Foundation, "JMeter," 2012. http://jmeter.apache.org/.

[31] Y. Chen and A. Romanovsky, "WS-Mediator for improving the dependability of web services integration," *Journal of IT Professionals*, vol. 10, no. 3, pp. 29–35, 2008.

[32] H. Zeadally, *Media Networks: Architectures, Applications, and Standards*. Taylor & Francis Group, 2012.

[33] T. To and T. Hamidzadeh, *Interactive Video-On-Demand Systems: Resource Management and Scheduling Strategies*. The Kluwer International Series in Engineering and Computer Science, Springer-Verlag GmbH, 1998.

[34] T. Lo, "Trends in the smart tv industry," tech. rep., Digitimes Research, 2012.

[35] A. Breznick, "Assuring multi-screen video quality," tech. rep., Heavy Reading - IneoQuest, 2012.

[36] F. et al., "Hypertext transfer protocol – http/1.1," tech. rep., Network Working Group, 1999.

[37] A. Gorbenko, E. K. Iraj, V. S. Kharchenko, and A. Mikhaylichenko, "Exception analysis in service-oriented architecture," in *Information Systems Technology and its Applications*, pp. 228–233, 2007.

[38] I. Chen, G. Ni, C. Kuo, and C.-Y. Lin, "A BPEL-Based fault-handling architecture for telecom operation support systems," *Journal of Advanced Computational Intelligence and Intelligent Informatics*, vol. 14, no. 5, pp. 523–530, 2010.

[39] N. N. Salatge and J.-C. Fabre, "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.

[40] G. Santos, L. Lung, and C. Montez, "FTWeb: A fault tolerant infrastructure for Web services," in *Proceedings of the 9th IEEE International Conference on Enterprise Computing*, pp. 95–105, 2005.