SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2014; **44**:835–854
Published online 3 February 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2255

Fault masking as a service

Koray Gülcü, Hasan Sözer*,†, Barış Aktemur and Ali Özer Ercan

Özyeğin University, İstanbul, Turkey

SUMMARY

In SOA, composite services depend on a set of partner services to perform their tasks. These partner services may become unavailable because of 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, dubbed Fault Avoidance Service (FAS), 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 attempting an ill-destined request. We provide mathematical analysis regarding the error rate and the false positive rate with respect to the monitoring frequency of FAS for two models. We obtained empirical results by conducting several tests on the Amazon Elastic Compute Cloud to evaluate our mathematical analyses. We also introduce an industrial case study for improving the quality of a service-oriented system from the broadcasting and content delivery domain. Copyright © 2014 John Wiley & Sons, Ltd.

Received 1 March 2013; Revised 1 August 2013; Accepted 1 January 2014

KEY WORDS: service-oriented systems; availability; fault avoidance; fault masking; service monitoring

1. INTRODUCTION

Service-oriented architectures and cloud computing facilitate the development of distributed software systems based on loosely coupled and self-contained services in heterogeneous environments [1, 2]. These services can be discovered and composed with each other to provide more sophisticated, higher-level, so-called *composite services* (CSs) [3]. CSs are usually defined by means of specialized composition languages such as Web Services Business Process Execution Language (WS-BPEL) [4], and they invoke other services via the so-called *partner links*. Hence, the services that are utilized by a CS are named *partner services*. Some of the partner services can cease to be available because of system and/or network faults, which have been shown in recent experimental studies to be very common [5]. These faults result in an error and possibly a failure of the CS that relies on the availability of its partner services. Preferably, the CS should discover and utilize alternative services to tolerate such external faults. As such, there have been several fault tolerance approaches proposed in the literature [6–8]. However, error detection and system recovery increase the response time because of the extra overhead they incur. The consequential delay can be significant especially for CSs that utilize many other services [9]. Therefore, faults should be avoided (if possible) to improve the dependability and performance of service-oriented systems. One can employ an *active* fault tolerance strategy (i.e., connect to all of the partner services simultaneously and proceed with the fastest response) to avoid faults. However, this is not possible in many cases

*Correspondence to: Hasan Sözer, School of Engineering, Özyeğin University, Nişantepe Mah. Orman Sk. No: 13, Alemdağ – Çekmeköy 34794, İstanbul, Turkey.

†E-mail: hasan.sozer@ozyegin.edu.tr

Copyright © 2014 John Wiley & Sons, Ltd.

because of constraints imposed by limited resources or the problem domain. Another way is to perform service selection process per each request [10, 11] or per each flow of requests [12]. 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 [11, 13], middleware, [7, 14, 15] and framework support [8, 16–18] to compose dependable services. In our previous work [19], we proposed 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, that is, dependability as a service. To our knowledge, this concept has only been realized in the context of software/service testing (testing as a service [20]) so far. Our experiments showed that fault masking could be realized as a service. We have introduced a Web service, Fault Avoidance Service (FAS) [19], to which a CS registers the set of its partner services. FAS periodically and independently checks the availability of the registered partner services and compensates for their unavailability. FAS does not just provide health monitoring or error handling but aims at fault avoidance by proactively reconfiguring CSs 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 CS. 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 [19].

Contributions. This paper presents two novel contributions with respect to our previous work [19]. First, we refine our analytical metrics regarding the impact of monitoring frequency on the error rate and the false positive rate. We eliminate major assumptions to propose a more realistic model. 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. We have not encountered such an analysis in the literature although service monitoring has been employed in many studies [7, 23–25]. Second, 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.

Organization. The remainder of this paper is organized as follows. Section 2 presents the problem statement and our solution approach. In Section 3, we introduce a set of analytical metrics and related mathematical analysis. We present an experimental evaluation of our analysis results in Section 4. Section 5 presents an industrial case study for improving the availability of services employed in Smart TVs. In Section 6, related previous work is summarized. Finally, in Section 7, we provide the conclusions.

2. PROBLEM STATEMENT AND THE SOLUTION APPROACH

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 Universal Description, Discovery and Integration [26] service registry is a specialized type of service broker.

After registering itself to the service registry or after being discovered by the CS or even after being successfully invoked several times, a partner service can become unavailable because of system and/or network faults. In fact, recent experimental studies [5] show that the majority of service invocation failures are caused by these types of faults (connection time-out, service unavailability, etc.). As a result, the invocation attempt leads to an error. In turn, the CS can (i) report a failure to its service requester or (ii) discover and utilize alternative services (might be hard coded in the source

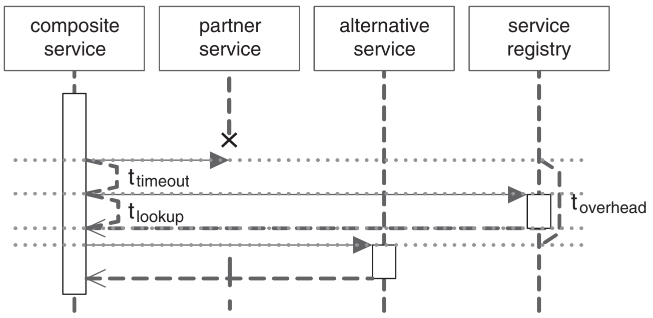


Figure 1. An error recovery scenario.

code, Universal Description, Discovery, and Integration and WS-BPEL descriptions, or it might be stored in an external cache) to recover from the error. Figure 1 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.

After the failure and before the recovery of the partner service, the CS makes an invocation without success. The CS waits for a time-out duration ($t_{timeout}$) to decide whether the partner service is available or not. Once it is deemed to be unavailable, the CS 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 $t_{overhead} \approx t_{timeout} + t_{lookup}$.

Failure of a partner service is an external fault from the CS's perspective, which invokes the failed service. A CS can be exposed to many external faults, each of which increases the overall response time by an additional $t_{overhead}$. The consequential delay can be significant especially for CSs that utilize many partner services [9]. In the following, we introduce a fault masking approach, where these external faults are handled to improve the dependability and performance of CSs.

Overview of the approach. We introduce a Web service for masking faults. We name this service as FAS. A CS first determines the list of partner services that are going to be utilized and registers this list to FAS. FAS periodically checks the availability of these services. Once a partner service becomes unavailable, FAS locates alternatives and reconfigures the CS accordingly. When needed, the CS uses the updated partner links. This prevents CS 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 CS 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 CS 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 CS by updating its partner service links associated with the unavailable partner services. As such, the CS 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 partner services. If a faulty service becomes available again, FAS updates the CS's partner link back to its original configuration. FAS checks the availability of the registered partner links periodically.

In fact, FAS does not require a service composition to be applied. Systems comprising only one partner service interaction can also exploit our approach. In this case, however, FAS would be responsible for only one partner service. As the number of partner services increases, the impact of external faults also increases. Hence, the benefits of fault masking would be more significant in the context of service composition. Otherwise, our approach and the mathematical models, simulations, and case studies are agnostic to the number of services that are monitored. In the following

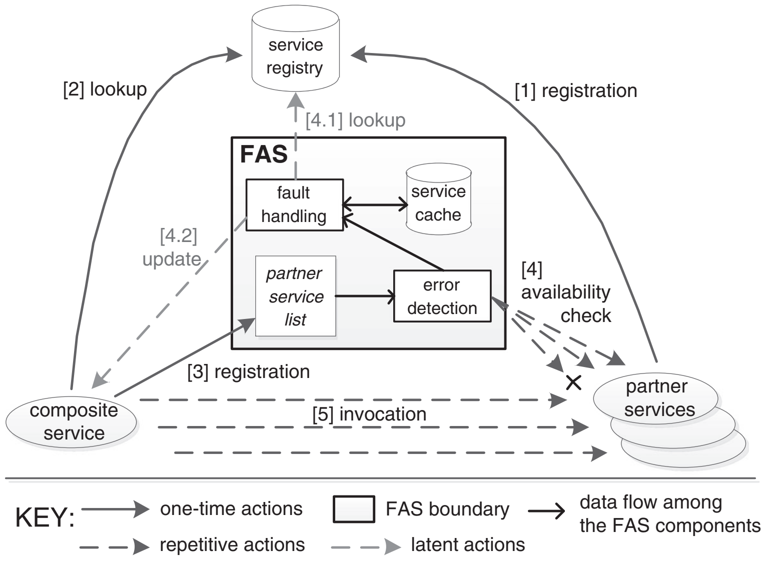


Figure 2. The overall approach.

section, we present mathematical analysis regarding the effect of FAS monitoring frequency on the error rate and the false positive rate.

3. MATHEMATICAL ANALYSIS

In this section, we first introduce a set of analytical metrics and related mathematical analysis. Then, we present simulation results and define an objective function for optimal monitoring frequency.

3.1. Derivation of analytical metrics

In an ideal situation, FAS will immediately detect whenever a partner service becomes unavailable or available. This way, the CS can be notified right away so that no request from the CS 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 CS sends its request to the partner service before FAS notices that the service is down or the cases where the CS still uses the secondary service because FAS did not notice yet that the partner service is back to life. If the service is down, it threatens the customer satisfaction because the CS will be trying to invoke an erroneous partner service. If the primary service is back to life, using the secondary service can cause several problems depending on the deployment. For instance, 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. Increasing the frequency of FAS checks would 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 focusing on 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 CS periodically sends requests at some frequency C, FAS checks the availability of the partner service at a frequency F, and the partner service becomes unavailable for a certain period T_U of its lifetime T. We assume that the requests, checks, and partner service up/down events are instantaneous. We also assume that an available replica always exists in the environment in case the partner service is unavailable (further discussed in Section 4.3). 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

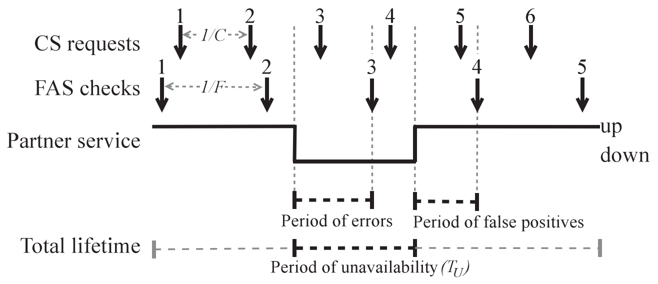


Figure 3. A scenario showing the important events in a system that uses FAS when $1/F \leqslant T_U$.

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 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 (ER) and the false positive rate (FP). ER (repectively FP) is calculated as the ratio of the number of errors (respectively false positives) to the total number of CS requests. The smaller these values are, the more useful FAS is. In our previous work [19, 27], we performed an analysis by assuming that for a given availability, T_U is a fixed duration and its starting time is uniformly distributed over the total lifetime. On the basis of this assumption, the expected error rate ($E[\mathsf{ER}]$) and the expected false positive rate ($E[\mathsf{FP}]$) are calculated as follows.

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

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. We have also derived the upper bounds for ER and FP, which reflect the worst case (maximum) values.

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

The plots of the expected and maximum values are given in Figure 4 for when $T = 400$ s and $T_U = 40$ s (i.e., availability is 90%). We have validated the derived formulas by means of simulations as well as controlled experiments on the Amazon EC2 [22]. Details regarding our derivation and experimental results can be found in [27]. In the following, we provide a complementary analysis, in which T_U is not assumed to be a fixed duration.

Markov chain-based partner service behavior. We also analyzed a Markov chain-based partner service behavior model. According to this model, the partner service state $X(t)$ is assumed to be a continuous-time random process, where $X(t) = 1$ denotes that the partner service is up at time t, and $X(t) = 0$ denotes that the service is down. For a first-order analysis, we assumed that the *future* states are conditionally independent of the *past* states, given the *present* state. This implies that $X(t)$ is a continuous-time Markov chain [28], the state transition diagram of which is depicted in Figure 5. There, the transition rates λ and μ refer to the reciprocals of mean time to recovery (MTTR) and mean time to failure (MTTF), respectively. Thus, the availability $= \frac{MTTF}{MTTF+MTTR} = \frac{\lambda}{\mu+\lambda}$

According to this model, the partner service stays in state 1 (resp. state 0) for an exponentially distributed random time with mean MTTF (resp. MTTR) and then switches to state 0 (resp.

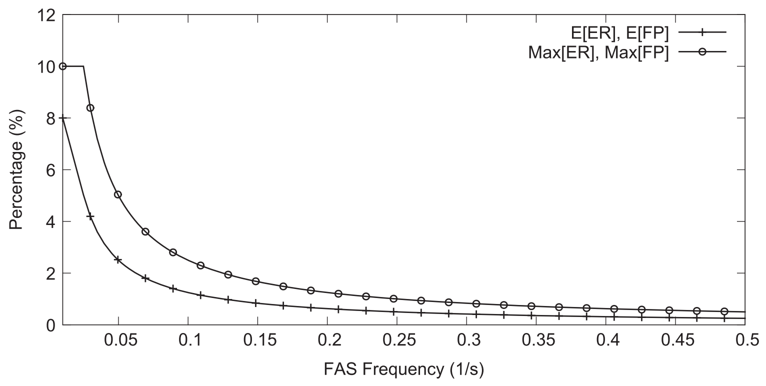


Figure 4. Change of expected and maximum values of error and false positive rates with respect to Fault Avoidance Service (FAS) frequency according to the mathematical model (1) and (2).

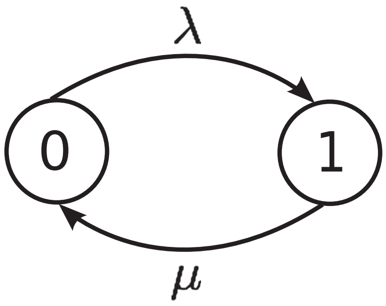


Figure 5. The continuous-time Markov chain-based partner service behavior model.

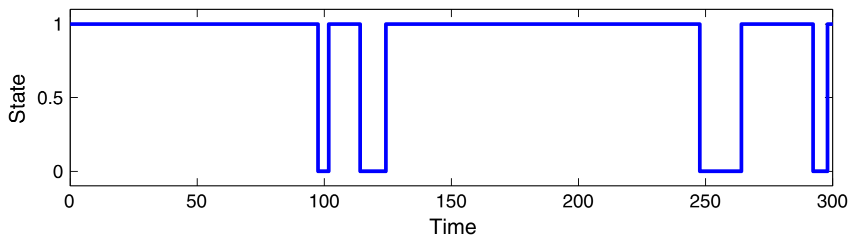


Figure 6. A sample partner service behavior with mean time to failure = 90 and mean time to recovery = 10 units.

state 1), and this is repeated forever. A pseudo-algorithm that generates sample transition times and corresponding states for this behavior is given in Algorithm 1. On Line 3, $\frac{\lambda}{\mu+\lambda}$ is the steady-state probability of $state = 1$. On line 10, the algorithm is drawing an exponential random variable with parameter μ. A sample behavior generated with this algorithm for 300 units of time for MTTF = 90 units and MTTR = 10 units (i.e., availability = 90%) is given in Figure 6.

We calculate $E[\mathsf{ER}]$ and $E[\mathsf{FP}]$ according to this Markov chain model as follows. Let $Y_i = 1$ if the i^{th} CS check results in an error, and $Y_i = 0$ otherwise.

Then, $\mathsf{ER} = \frac{1}{N}\sum_{i=1}^{N} Y_i$, where N is the total number of CS checks. Thus, $E[\mathsf{ER}] = \frac{1}{N}\sum_{i=1}^{N} E[Y_i] = \frac{1}{N}\sum_{i=1}^{N} P(Y_i = 1) = \frac{1}{N}\sum_{i=1}^{N} \mathrm{P(Error)} = \mathrm{P(Error)} = \mathrm{P}$ (CS = down, previous FAS = up). Hereby, $CS = down$ means that the CS check results in partner service being unavailable. $FAS = up$ means that the last FAS check results in partner service being available.

Algorithm 1 Partner service behavior generator

```
Inputs: $MTTR$, $MTTF$
Outputs: $transition\_times$, $states$
1: $\lambda \leftarrow \frac{1}{MTTR}$
2: $\mu \leftarrow \frac{1}{MTTF}$
3: $state \leftarrow (\texttt{random}() \leqslant \frac{\lambda}{\mu+\lambda})$
4: $states \leftarrow [\,]$
5: $transition\_times \leftarrow [\,]$
6: $time \leftarrow 0$
7: while TRUE do
8:     $states \leftarrow [states, state]$
9:     if $state$ then
10:        $time\_in\_state \leftarrow \frac{1}{\mu} \ln(1 - \texttt{random}())$
11:    else
12:        $time\_in\_state \leftarrow \frac{1}{\lambda} \ln(1 - \texttt{random}())$
13:    end if
14:    $time \leftarrow time + time\_in\_state$
15:    $transition\_times \leftarrow [transition\_times, time]$
16:    $state \leftarrow \neg state$
17: end while
```

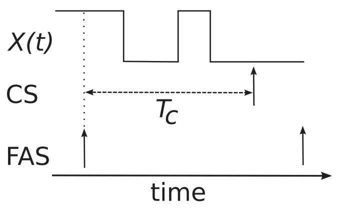


Figure 7. An example case where an error occurs.

Let the time between a CS check and the previous FAS check be the random variable T_C (Figure 7) with probability distribution function $f_{T_C}(t)$. Then,

$$\begin{aligned}
E[\mathsf{ER}] &= \int \mathrm{P}\left(\text{CS} = \text{down},\ \text{previous } \mathsf{FAS} = \text{up}, T_C = t\right) dt \\
&= \int \mathrm{P}\left(\text{CS} = \text{down} | \text{previous } \mathsf{FAS} = up, T_C = t\right) \mathrm{P}\left(\text{previous } \mathsf{FAS} = \text{up} | T_C = t\right) f_{T_C}(t) dt \\
&\overset{(a)}{=} \int \mathrm{P}\left(\text{CS} = \text{down} | \text{previous } \mathsf{FAS} = up, T_C = t\right) \mathrm{P}\left(\text{previous } \mathsf{FAS} = \text{up}\right) f_{T_C}(t) dt \qquad (3) \\
&= \int \mathrm{P}(X(s+t) = 0 | X(s) = 1) \mathrm{P}(X(s) = 1) f_{T_C}(t) dt \\
&\overset{(b)}{=} \int \mathrm{P}(X(t) = 0 | X(0) = 1) \frac{\lambda}{\mu + \lambda} f_{T_C}(t) dt,
\end{aligned}$$

where step (a) uses the fact that the result of the FAS check is independent of T_C, and step (b) assumes that the Markov chain is time-homogeneous and in steady state. To find the first term in the integral, consider the state transition probability matrix $P(t)$ for this Markov chain, which is given by $P(t) = e^{tQ}$, where Q is the transition rate matrix [29]:

$$Q = \begin{bmatrix} -\lambda & \mu \\ \lambda & -\mu \end{bmatrix}.$$

Diagonalization of Q is given by

$$Q = V \Sigma V^{-1} = \begin{bmatrix} \frac{\mu}{\lambda+\mu} & \frac{1}{2} \\ \frac{\lambda}{\lambda+\mu} & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & -\lambda-\mu \end{bmatrix} \begin{bmatrix} 1 & 1 \\ \frac{2\lambda}{\lambda+\mu} & -\frac{2\mu}{\lambda+\mu} \end{bmatrix}.$$

Thus,

$$\begin{aligned} P(t) = e^{tQ} &= V \begin{bmatrix} 1 & 0 \\ 0 & e^{-(\lambda+\mu)t} \end{bmatrix} V^{-1} \\ &= \begin{bmatrix} \frac{\mu}{\lambda+\mu} + \frac{\lambda}{\lambda+\mu} e^{-(\lambda+\mu)t} & \frac{\mu}{\lambda+\mu} - \frac{\mu}{\lambda+\mu} e^{-(\lambda+\mu)t} \\ \frac{\lambda}{\lambda+\mu} - \frac{\lambda}{\lambda+\mu} e^{-(\lambda+\mu)t} & \frac{\lambda}{\lambda+\mu} + \frac{\mu}{\lambda+\mu} e^{-(\lambda+\mu)t} \end{bmatrix}. \end{aligned}$$

By definition, $\mathrm{P}X(t) = 0|X(0) = 1 = P(t)_{0,1} = \frac{\mu}{\lambda+\mu} - \frac{\mu}{\lambda+\mu} e^{-(\lambda+\mu)t}$. Thus, Equation (3) becomes

$$\begin{aligned} E[\mathsf{ER}] &= \int \frac{\mu}{\lambda+\mu} \left(1 - e^{-(\lambda+\mu)t}\right) \frac{\lambda}{\mu+\lambda} f_{T_C}(t) dt \\ &= \int \frac{\lambda\mu}{(\lambda+\mu)^2} \left(1 - e^{-(\lambda+\mu)t}\right) f_{T_C}(t) dt. \end{aligned}$$

Note that the minimum value that T_C can take is 0, while the maximum value is $1/F$. We assume T_C is distributed uniformly in this interval, yielding

$$\begin{aligned} E[\mathsf{ER}] &= \int_0^{1/F} \frac{\lambda\mu}{(\lambda+\mu)^2} \left(1 - e^{-(\lambda+\mu)t}\right) F dt \\ &= \frac{\lambda\mu}{(\lambda+\mu)^2} \left[1 - \frac{F}{\lambda+\mu} \left(1 - e^{-(\lambda+\mu)/F}\right)\right]. \end{aligned} \tag{4}$$

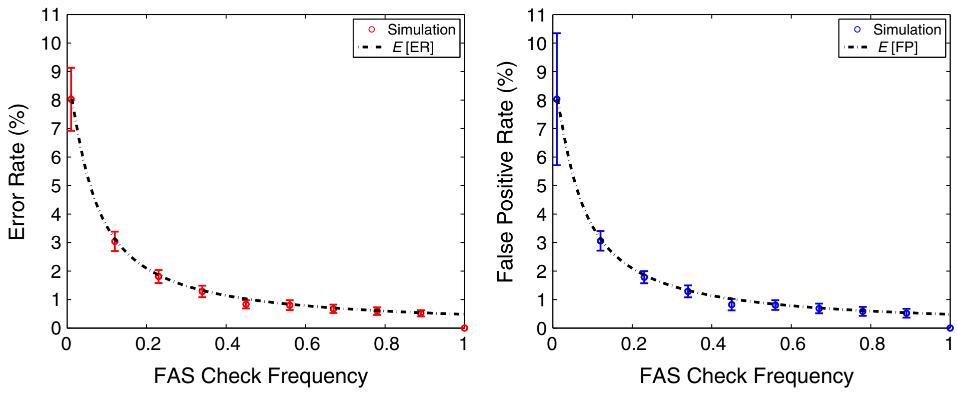
A similar derivation yields the same formula for $E[\mathsf{FP}]$:

$$\begin{aligned} E[\mathsf{FP}] &= \mathrm{PCS} = \mathrm{up}, \mathrm{previous}\mathsf{FAS} = \mathrm{down} \\ &= \frac{\lambda\mu}{(\lambda+\mu)^2} \left[1 - \frac{F}{\lambda+\mu} \left(1 - e^{-(\lambda+\mu)/F}\right)\right]. \end{aligned} \tag{5}$$

3.2. Simulations

We tested the validity of Equations (4) and (5) with simulations executed in Octave. We generated sample partner service behaviors for a total duration of 100,000 units with MTTF = 90 and MTTR = 10 units according to Algorithm 1. Then, for a constant CS check frequency of 0.25 and for varying FAS check rates between 0.01 and 1, we counted the number of times that error events and false positive events happen. The ratio of these numbers to the total number of CS checks are the error rates and the false positive rates found in the simulations. We repeated this experiment 50 times and found the averages and standard deviations of these rates. The results are reported in Figure 8 by the circles and error bars. On top of these, the theoretical graphs conforming to (4) and (5) are plotted in dashed lines.

As seen from the figure, the theoretical values are generally within three standard deviations of the averages. When Equations (4) and (5) are not in $\pm 3\sigma$ of the simulations, the reason is the uniform distribution assumption on T not being valid. That is, for example, consider the case when FAS check rate is 1 where the disagreement is most noticeable. Because CS check rate is 0.25, the CS check period is an integer multiple of FAS check period. Because the checks in the simulations are periodic, the time between each CS check and the previous FAS check is always constant during each run, rendering the uniform distribution assumption invalid. However, we can say simulations



(a) Error Rate (b) False Positive Rate

Figure 8. The simulation results. The circles denote the average rates found, and the error bars denote ± 3 standard deviations of the total 50 runs. Here, mean time to failure = 90, mean time to recovery = 10, and each simulation is done for 100,000 time units. The dashed lines denote the theoretical curves calculated by Equations (4) and (5).

generally agree well with the theoretical values. And also, because in reality, the CS checks are not going to be exactly periodic, we believe the formulae in Equations (4) and (5) are realistic.

The simulations are performed on a local machine for probabilistic validation of our mathematical analysis. There is no service creation or invocation during simulations (we discuss these experiments in the next section). By simulations, we were able to conduct large amount of tests to measure meaningful average rates. Standard deviations turn out to be too high for real-world implementations, which cannot be tested long enough. Moreover, we isolated runtime effects, some of which might be significant on the average values, to provide a more controlled approach for a first-degree validation of our mathematical analysis.

3.3. *Optimal Fault Avoidance Service check frequency*

Equations (4) and (5) are monotonously decreasing functions of FAS check rate F. Thus, according to the Markov chain-based partner service behavior, the more frequent the FAS checks are, the less the errors and false positive rates are. There is no optimal FAS check rate value considering only the error and false positive rates. On the other hand, there is an 'energy' cost of more frequent FAS checks. One might think of a multi-objective cost function such as

$$COST = E[\mathsf{ER}] + \alpha F, \tag{6}$$

where F is the FAS check frequency and α is the conversion rate between the cost of FAS checks and error rate. In other words, α denotes how much reduction in E[ER] one would be willing to increase F by one unit. The smaller the α is, the more important reduction in E[ER] becomes; thus, one is willing to increase F by one unit for smaller reductions in E[ER]. Clearly, it is possible to use a linear combination of E[ER] and E[FP] in the cost function (6); however, because ER and FP are equal to each other in (4) and (5), this approach would be equivalent.

According to this cost function, the dashed curves in Figure 8 at the same time denote the trade-off (i.e., the Pareto curve) between the E[ER] (or E[FP]) and the cost of FAS checks. Depending on the conversion rate α, one can pick an optimal point on this curve. In Figure 9, we plotted the optimal FAS check frequency as a function of α. As seen in the figure, the optimal frequency decreases as α increases (when the cost of FAS checks becomes 'more expensive'). Depending on the practical situation, once α is determined, one can choose the optimal FAS check frequency using this graph.

We also analyzed the optimal F value based on our previous analysis, when T_U is a fixed duration, and its starting time is uniformly distributed over the total lifetime [27]. We used the same cost function (6) and set the parameter $\alpha = 1$. $E[\mathsf{ER}] = \frac{1}{2FT}$ for $1/F \leqslant T_U$ (1). Hence, $COST = \frac{1}{2FT} + F$. To find the optimal value of F, we solve

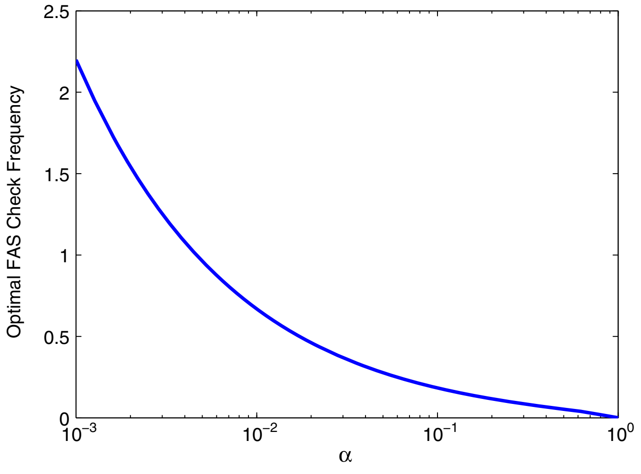


Figure 9. Optimal Fault Avoidance Service (FAS) check frequency for the Markov chain partner service behavior as a function of the conversion rate α. The smaller the α is, the more one would be willing to incur the cost of higher F for reductions in ER. Here, mean time to recovery = 10, and mean time to failure = 90 units.

$$\frac{\partial COST}{\partial F} = 0 \Rightarrow \frac{\partial}{\partial F}\left(\frac{1}{2FT} + F\right) = 0 \Rightarrow \frac{\partial}{\partial F}\left(\frac{1}{2FT}\right) + 1 = 0 \Rightarrow \frac{-1}{2TF^2} = -1$$

As a result, we obtain the following equation, which can be used for calculating an optimal monitoring frequency that is dependent on the parameter, *T*, that is, *total lifetime*.

$$F = \sqrt{1/2T} \tag{7}$$

To determine the value of the parameter *T*, we investigated real services from the broadcasting domain. In Section 5, we discuss our observations and strategies for adjusting this parameter. In the following section, we explain our experimental evaluation and discuss the results.

4. EXPERIMENTAL EVALUATION

In Section 3, we mathematically tested the validity of Equations (4) and (5) with the aid of Algorithm 1. The simulations are based on a probabilistic execution with no real service request or invocation. Hence, they reflect ideal conditions where network delays, service deployment, distribution, and execution are absent. For an experimental evaluation of our mathematical analysis and simulation results, we implemented a prototype [27] and performed several tests, which we discuss in this section.

In our experimental setup, we developed FAS as a stand-alone Web service that provides an interface to CSs for registration at start-up. During registration, CSs 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 CS model. However, CSs 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 CS 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 CS implementation and the employed composition language. It is also possible to utilize WS-BPEL, for instance, using the extension proposed by Wu *et al*. [30].

We also implemented a partner service and replicated it. If FAS updates the partner link before the (unavailable) first replica is invoked, CS sends the request directly to the second replica. If not, the CS 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.

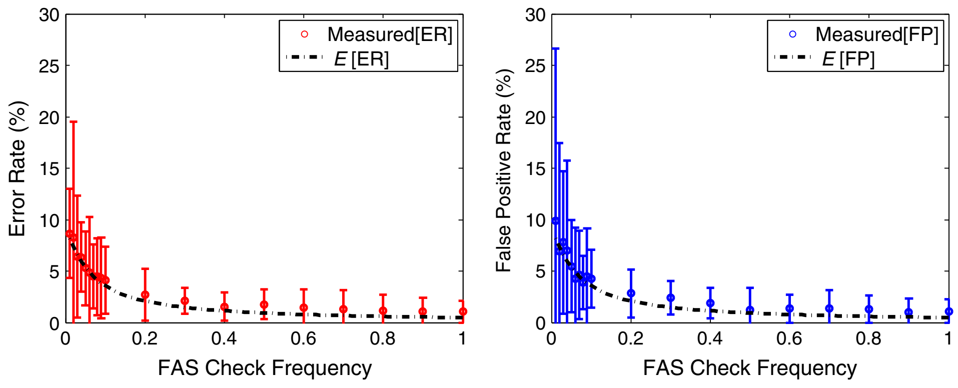
4.1. *Experimental setup*

We used Node.js [31] 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 64-bit Linux operating system. We deployed a CS and two replicas of our partner service. Partner service replicas were deployed in North Virginia and Tokyo, while CS was in Ireland, and FAS was in Sao Paulo, Brazil. Tests were conducted and controlled from another instance located in Sydney, Australia.

Throughout our tests, we generated a partner service lifetime by using Algorithm 1 and assigned it to the first replica. The second replica is configured to be 100% available for all tests. We generated sample partner service behaviors for a total duration of 2000 s with MTTF = 90 and MTTR = 10 units. Then, for a constant CS check frequency of 0.25 and for varying FAS check rates between 0.01 and 1 (between 0.01 and 0.1 with a 0.01 step size and between 0.1 and 1 with a 0.1 step size), we performed several tests and calculated error rates and false positive rates during the experiments. Repeating each configuration 10 times, we obtained averages and standard deviations of these metrics. The results are presented in the following subsection.

4.2. *Results*

Figure 10 depicts the results. In Figure 10(a), $E[\mathsf{ER}]$ is plotted together with the experimental results ($Measured[\mathsf{ER}]$) with respect to F. Figure 10(b) shows $E[\mathsf{FP}]$ and the measured false positive rate ($Measured[\mathsf{FP}]$) for the same range and settings of F and service availability. It can be seen from the figures that $E[\mathsf{ER}]$ is consistent with respect to the experimental results. Likewise, the measured false positive rates confirm the accuracy of our mathematical analysis regarding $E[\mathsf{FP}]$. By comparing the figures, we can say that in higher availabilities, we obtain low ER even if we use small F values. It is also worth noting that the measured values are almost always slightly larger than the



(a) Error Rate (b) False Positive Rate

Figure 10. Experimental results for ER and FP. The circles denote the measured average rates and the error bars denote ±3 standard deviations of the total 10 runs. Here, mean time to failure = 90 s, mean time to recovery = 10 s, and each experiment is done for 2000 s. The dashed lines denote the theoretical curves calculated by Equations (4) and (5).

expected values. This can be due to communication delays that were ignored in our analysis. 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. Of course, this is only one dimension to consider during system configuration; several other criteria [32] must also be taken into account for a full deployment.

4.3. *Threats to validity*

The service behavior model (Figure 5) is a statistical one that aims to model the partner service behavior. That is, each realization of a partner service behavior will be different, and Figure 6 is an example realization among infinitely many different possible ones. The continuous-time Markov chain model in Figure 5 models the *statistical* properties of the ensemble of different possible realizations. We use these statistical properties to calculate expected error and false positive rates. In reality, there can be second-order effects that are not reflected in this model. For example, the Markov assumption implies that the conditional distribution of the future states given the current state and the past states is independent of the past states and only depends on the current state [28]. A real partner service behavior might not exactly satisfy this assumption; however, Markov chains are widely used to model and analyze system reliability [33]. Therefore, we chose to use a Markov chain-based model to be able to reflect the up–down behavior of the partner service statistically to a first degree, while being able to calculate analytically the desired quantities such as expected error and false positive rates. In the model, the parameters λ and μ are expected to be different for different partner services and must be determined for each service by observing MTTF and MTTTR. Thus, these parameters give the DOFs to differentiate between services behaving differently.

Our analysis considers the existence of one service replica that is assumed to be always available. In case multiple and possibly unavailable replicas exist, our analysis is still valid for the following reasons: assume each replica can be up or down according to the Markov chain model in Figure 5, independently of the other replicas. FAS server will check these replicas and choose an available one as the replica to be used by the CS. An error will occur if the chosen service replica is down at the time of CS request, *independent* of what happens at the other unchosen service replicas. Thus, our error rate analysis still applies to this case. Similarly, if the original service is down during the FAS check but happens to be up at the time of CS request a false positive happens independent of the other service replicas. Hence, the false positive rate analysis also is unaffected by the behavior of the other replicas. However, there is one scenario that is not considered here, which is the case when all replicas are down at the time of FAS check. This case leaves the FAS server unsuccessful in designating a service replica for the CS. If this case happens, the FAS server might go to a different regime, for example, searching for an alternative available service replica with much more frequent FAS checks and so on. This regime is out of the scope of our analysis, and for the sake of illustration, we left this case out.

The availability of partner services is being monitored from the perspective of FAS, which might possibly mismatch the experience of the CS. Complementary mediators [34] can be incorporated to monitor the dependability characteristics of partner services from CSs' 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 CS might have a tolerance margin for reconfiguration. Or it might be costly to substitute a critical partner service. Depending on the process, CSs might need to communicate with FAS to update critical information, service cache, and notification interface/protocol. Finally, it is also possible that the alternative service cannot be directly substituted for the original service because of stateful properties [15]. We did not take these needs into account.

4.4. *Experimental evaluation versus simulation*

Experimental evaluation is complementary to the simulations. By means of simulations, we were able to eliminate unpredictable runtime effects such as network delays.

In our experiments, we used Algorithm 1 to generate partner service behaviors the same way we did in the simulations. By implementing prototype services and deploying them in the cloud, we

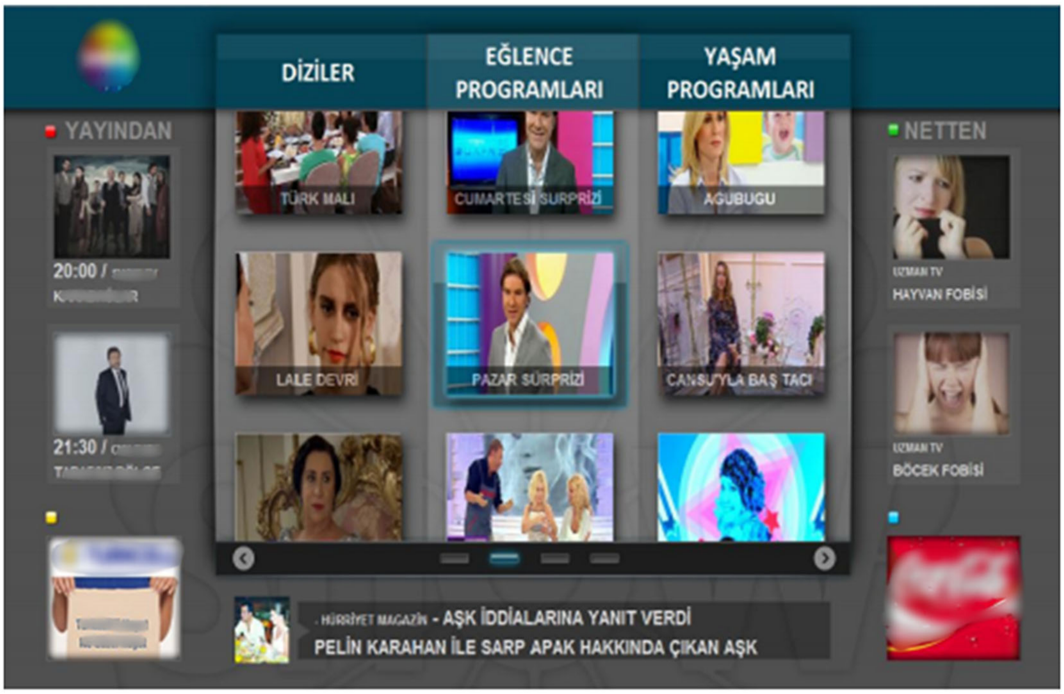


Figure 11. A snapshot from a portal application.

were able to take the runtime effects into account, as illustrated by Figure 8 versus Figure 10. By experimentation, we observed the effect of service invocations and network delays on the average values and deviations. Using the EC2 infrastructure enabled us to distribute our services globally. This provided a more realistic evaluation of our approach; we were able to create a fully distributed test environment and automate most of the data-gathering and post-processing jobs in the cloud.

5. INDUSTRIAL CASE STUDY

In this section, 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 TV systems. These systems utilize various services such as third-party video content providers, popular social media platforms, and games. In particular, we investigated a portal application (Figure 11) developed by Vestek,[‡] a group company of Vestel that is one of the largest TV manufacturers in Europe. This application is being utilized by Vestel as an online television service in Turkey. The application is a platform comprising dozens of third-party services, including the most popular Web applications, audio/video streaming services, and games. Among these, there are services that provide similar content as well.

Services that provide audio/video content over broadband connection are considered to be among the most important services for Smart TVs [35]. These services might be affected in various ways depending on the type and location of faults. For instance, there might be content-related problems that leave front-end devices unable to play (e.g., unsupported format) or server-side problems that result in faulty feeds, wrong URLs, or no response at all. Moreover, it is also possible to experience intermittent network outages causing video and audio freezes, long buffering periods, end-of-stream errors, or lip-sync errors. In this case study, we are particularly interested in content-related faults

[‡]http://vestek.com.tr, accessed in July 22013.

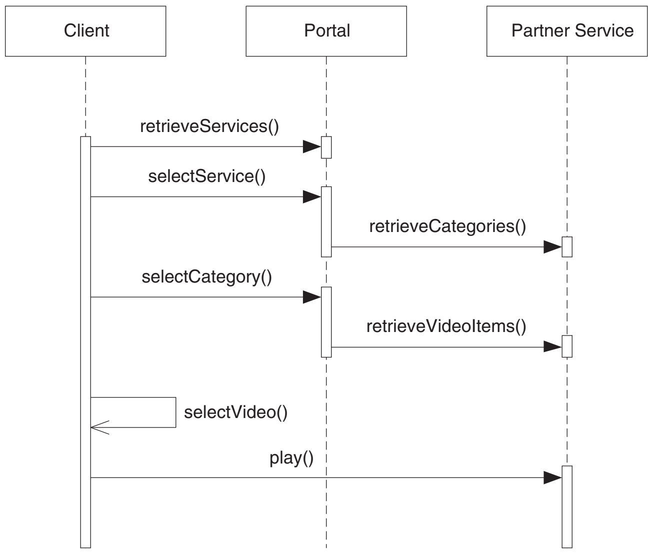


Figure 12. Sequence diagram of video streaming on the portal application.

that propagate through the portal application and affect the user experience negatively. We investigate how we can use FAS to mask and avoid the faults threatening user satisfaction and service availability. In the following, we first describe the realization of our approach in this context, where different partner services roll into one CS potentially being consumed by millions of end users. Then, we discuss our experimental setup, observations, and results.

5.1. Realization of the approach

Figure 12 depicts a typical process when a client uses the portal application. Hereby, 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 to retrieve 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 the partner service's APIs. If the request succeeds, video items are displayed with additional meta information such as the thumbnail image, title, duration, and popularity. Then, the client chooses a video item and starts streaming by directly communicating with the partner service.

To experiment with our approach, we implemented a prototype CS 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. Implementation details can be found in [27].

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 used by the portal application. We implemented a test application that picked a random video item from a random video category of a specified partner service to simulate a usage scenario. The video URL of the selected item was sent to FAS in order to monitor and log the status of the video. We ran the system for 11 days and collected data for approximately 1.6 million requests. For one of the services, ServiceE[§], 60% of the requests were unsuccessful; that is, the system detected anomaly.

[§]Because of confidentiality, we do not disclose the names of the utilized services.

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 symptom was the HTTP 404 response code when a video link was found to be broken. On the basis of the first observation, we concluded that faults associated with ServiceE were not intermittent network issues. On the basis of the second observation, we understood that there was a server-side defect that caused broken links in the feeds. As a result of these inferences, we concluded that it is possible to utilize the FAS service in order to detect anomalies associated with the ServiceE feeds. We have designed our experimental setup as described in the following subsection.

Our first observation also denotes that it is unlikely to see an alteration in the availability of 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 the content after every update and inform the CS accordingly. We implemented a test application that monitors the change behavior on content providers [27]. We monitored five different video services available on the Vestek portal for 14 days. Figure 13 shows the results for ServiceE, Figure 14 for the others. It can be seen that every

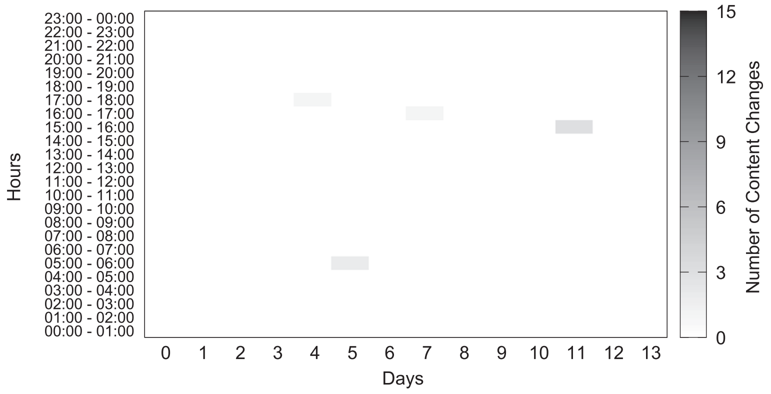


Figure 13. Content changes of ServiceE in a 14-day period.

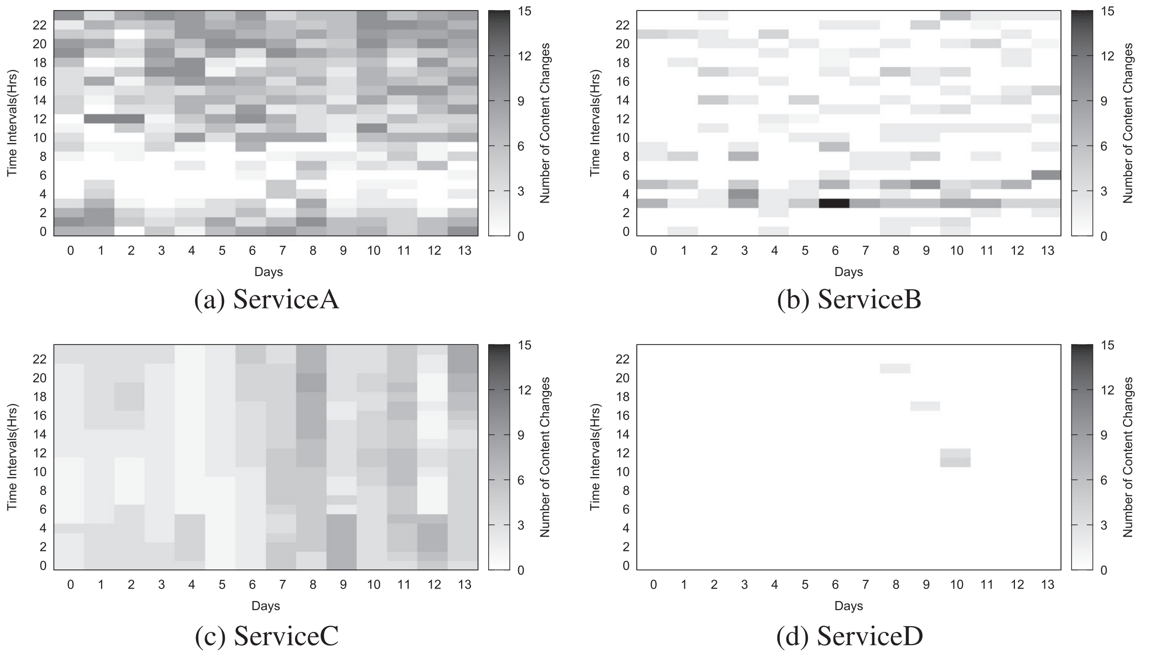


Figure 14. Content changes in a 14-day period.

service has a different content change behavior. For instance, the content provided by ServiceE is changing less often compared with the others. In accordance with this observation, we designed our experimental setup as described in the following subsection.

5.2. *Experimental setup*

We configured FAS monitoring frequency (F) according to Equation (7), which is dependent on the parameter, T, that is, *total lifetime*. We applied three different strategies for setting the value of this parameter differently for each service according to its content change behavior:

(i) *Average:* take T as the average period of time between two consecutive content changes.
(ii) *Minimum:* take T as the minimum period of time between two consecutive changes.
(iii) *Exponential back-off:* take the minimum period of time between two consecutive changes to calculate the initial value of F and apply exponential back-off at the end of each period if there is no content change.

For instance, there were seven changes in 14 days in the ServiceE feed. Therefore, we set $1/F$ to 588 s for the *average* strategy. We compared the durations between each consecutive changes and found the minimum to be 60 s, which yields 11 s as the value of $1/F$ for the *minimum* strategy. For the *exponential back-off* strategy, we used the minimum time difference (i.e., 60 s) to calculate the initial value of F (i.e., F_0) and employed the exponential back-off algorithm to double $1/F$ each time when there is no change. That is, we used the formula $F_{k+1} = F_k/2$ to calculate the frequency where k is the number of iterations starting from the last change. In this approach, we also bounded F by a predetermined limit.

We performed experiments for each strategy to observe and measure the effect of FAS on 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 metadata from two different replicas. At each trial, it logs the 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.

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. Results are discussed in the following subsection.

5.3. *Results and discussion*

Table I shows the results of using the *average* strategy. We observed that 9368 streaming attempts out of 55,835 failed for the native replica. This corresponds to 16.78%. For the rectified replica, only 10 streaming attempts (~0.02%) failed. There are two attempts that failed for the native replica and the rectified replica at the same time; that is, FAS was unable to detect and fix these. There are eight attempts where the trial is successful from the native replica, but the rectified replica failed.

The results of the *minimum* strategy are given in Table II. There are 9269 faults out of 54,918 attempts for the native replica. This corresponds to 16.88%. However, there are only five 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 four attempts that are successful for the native replica but failed for the rectified replica.

Table III shows the results for the *exponential back-off* strategy. For the native replica, 9490 failing attempts are detected out of 56,200. This corresponds to 16.88%. Only eight trials were

Table I. Experiment results for the *average* strategy.

Service	Number of faults	Total number of attempts	Availability (%)
Native replica	9368	55835	83.22
Rectified replica	10	55835	99.98

Table II. Experiment results for the *minimum* strategy.

Service	Number of faults	Total number of attempts	Availability (%)
Native replica	9269	54918	83.12
Rectified replica	5	54918	$\succ$99.99

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

Service	Number of faults	Total number of attempts	Availability (%)
Native replica	9490	56200	83.12
Rectified replica	8	56200	$\prec$99.99

Table IV. Comparison of the monitoring strategies.

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

unsuccessful, which correspond to ~0.01%. Two attempts failed for both replicas at the same time. There are two attempts failing for the rectified replica while succeeding for the native one.

Table IV 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 because network and resource usage is directly proportional to the monitoring frequency. As we can see from Table I, 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.

We monitored five video services in total from the Vestek portal. We observed how often their feeds changed for 14 days. We previously presented the results for ServiceE in Figure 13. Figure 14 shows the results for the other four services: ServiceA, ServiceB, ServiceC, and ServiceD. We observed that each service has its own change regime with some similarities as well. ServiceD and ServiceE change very rarely compared with the others. Hence, for these two services, we can consider a lightweight monitoring strategy triggered with these changes. On the other hand, ServiceA has a much more frequent change characteristic; as seen in Figure 14(a), it is rather stationary between 3:00–9:00 and hectic after 18:00. A customized strategy can be utilized on the basis of this daily pattern with different monitoring frequencies per each time slice. ServiceB and ServiceC appear to have more uniform distribution of change on a daily basis. A fixed monitoring frequency can be considered for these services. Besides, ServiceB is slightly erratic between 3:00 and 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 that is subject to change. Our observation was for 14 days; more accurate results can be obtained over a longer period of time. An additional subsystem in FAS responsible for logging the history of services and determining an accurate and adaptive monitoring strategy would hence be useful.

In this case study, we integrated FAS as a singular service whose responsibility is to monitor and take necessary actions for all partner services (one-to-many). But there might be cases where a single FAS system can fall short to handle enormous number of services. By abstracting out FAS from the system and realizing it as a service facilitates scalability for such scenarios. It is possible to replicate FAS on the cloud and divide up the monitoring responsibility. This can even be done across clusters [36]. Thus, each replica can be deployed independently and distributed geographically, monitoring a small portion of all partner services. At the extreme, a single FAS service can be utilized per each partner service (one-to-one) or even multiple FAS services for a highly critical partner service (many-to-one). This is the main motivation of externalizing fault masking subsystem and considering it as a service. From our experience so far, our approach fits very well to cloud-based implementation scenarios, which makes it scalable.

6. RELATED WORK

Anatoliy *et al*. [37] categorize errors and failures specific to service-oriented systems. They introduce 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 detect these errors/failures and warn the services prior to invocation to increase availability.

So far, research efforts for improving the dependability of service-oriented systems have focused on variety of fault tolerance strategies [8, 16, 17]. We particularly focus on fault masking, a distinctive strategy compared with others. An analysis of the literature also reveals that dependability improvement has been mainly facilitated by means of frameworks [17, 37], architectural methods [9, 38], reliable service connectors [39], proxies [23], 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 [13] that are responsible for the creation/composition as well as the adaptation of a CS. As an advantage of this approach, structural changes (i.e., architecture selection) can also be applied to the CS [13]. However, such approaches are inherently coupled with the adapted CS based on a CS model. FAS does not change the structure and the behavior of the CS, and it does not assume any CS model.

Previously, the use of a proxy Web service was proposed to replace failed or slow services with alternative services [23]. Hereby, the quality monitoring is performed by the CS. The source code of the CS 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 CS is demanded. However, we aim at detecting and masking faults seamlessly by an external Web service before the CS 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 [15] tackles this problem by enabling semantic-based service substitution.

Zheng and Lyu [7] introduce a middleware for CSs 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 the following: (i) they use a middleware, whereas we propose implementing a stand-alone service to which other services can register; (ii) our service actively monitors the replicas, whereas their monitor is passive: it only stores data; and (iii) we update the user's list of preferred replicas, whereas they update the user's preferred fault tolerance strategy.

7. CONCLUSIONS

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

We defined analytical metrics for the error rate and the false positive rate for different monitoring frequencies of FAS and partner service availabilities. We performed several tests using a prototype implementation deployed on the Amazon EC2. Our experimental results confirmed the accuracy of our analytical metrics, which can be used for configuring FAS on the basis of varying partner service availabilities.

We examined an industrial use case from the broadcasting domain. We investigated five video services to reveal common error types and possible monitoring strategies while leveraging FAS. 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 availability 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.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their feedback to improve this paper. This work is supported by Vestel Electronics and the Turkish Ministry of Science, Industry and Technology (00995.STZ.2011-2). The contents of this article reflect the ideas and positions of the authors and do not necessarily reflect the ideas or positions of Vestel Electronics and the Turkish Ministry of Science, Industry and Technology.

REFERENCES

1. Georgakopoulos D, Papazoglu M (eds). *Service-oriented Computing*. MIT Press: Cambridge, Massachusetts, USA, 2009.
2. Wang L, Ranjan R, Chen J, Benatallah B. *Cloud Computing: Methodology, Systems, and Applications*. CRC Press Taylor & Francis Group: London, UK, 2013.
3. Medjahed B, Bouguettaya A, Elmagarmid A. Composing Web services on the semantic Web. *VLDB Journal* 2003; **12**(4):333–351.
4. Jordan D, Evdemon J. Web services business process execution language version 2.0, 2009. Available at: http://docs.oasis-open.org/wsbpel/2.0/serviceref [last accessed July 2013], OASIS Standard.
5. Zheng Z, Zhang Y, Lyu M. Distributed QoS evaluation for real-world web services. In *Proceedings of the IEEE International Conference on Web Services*, Miami, Florida, USA, 2010; 83–90.
6. 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*, LNCS 4157. Springer-Verlag: Berlin, Heidelberg, 2006; 364–386.
7. Zheng Z, Lyu M. An adaptive QoS aware fault tolerance strategy for web services. *Journal of Empirical Software Engineering* 2010; **15**(4):323–345.
8. Liu A, Li Q, Huang L, Xiao M. FACTS: a framework for fault-tolerant composition of transactional web services. *IEEE Transactions on Services Computing* 2010; **3**(1):46 –59.
9. Baresi L, Ghezzi C. Towards self-healing service compositions. In *Proceedings of the 1st Conference on the Principles of Software Engineering*, Buenos Aires, Argentina, 2004; 27–46.
10. Ardagna D, Pernici B. Adaptive service composition in flexible processes. *IEEE Transactions on Software Engineering* 2007; **33**:369–384.
11. Cardellini V, Valerio VD, Grassi V, Iannucci S, Presti FL. 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*, Irvine, California, USA, 2011; 102 –113.
12. Ardagna D, Mirandola R. Per-flow optimal service selection for web services based processes. *Journal of Systems and Software* 2010; **83**(8):1512–1523.
13. Cardellini V, Casalicchio E, Grassi V, Presti FL, Mirandola R. Towards self-adaptation for dependable service-oriented systems. In *Architecting Dependable Systems VI*, de Lemos R, Fabre JC, Gacek C, Gadducci F, Beek M (eds). Springer-Verlag: Berlin, Heidelberg, 2009; 24–48.
14. Zeng L, Benatallah B, Ngu A, Dumas M, Kalagnanam J, Chang H. QoS-aware middleware for web services composition. *IEEE Transactions on Software Engineering* 2004; **30**(5):311–327.
15. Fredj M, Georgantas N, Issarny V, Zarras A. Dynamic service substitution in service-oriented architectures. In *Proceedings of the IEEE Congress on Services*, Honolulu, Hawaii, USA, 2008; 101–104.

16. 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*, Cavtat/Dubrovnik, Croatia, 2006; 126–133.
17. Fang C, Liang D, Lin F, Lin C. Fault tolerant Web services. *Journal of System Architure* 2007; **53**(1):21–38.
18. Canfora G, Penta MD, Esposito R, Villani M. A framework for QoS-aware binding and re-binding of composite web services. *Journal of Systems and Software* 2008; **81**(10):1754–1769.
19. Gulcu K, Sozer H, Aktemur B. 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, 2012; 106–120.
20. Ciortea L, Zamfir C, Bucur S, Chipounov V, Candea G. Cloud9: a software testing service. *SIGOPS Operating Systems Review* 2010; **43**:5–10.
21. Avizienis A, Laprie JC, Randell B, Landwehr C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 2004; **1**(1):11 –33.
22. Amazon. Elastic compute cloud (EC2). Available at: http://aws.amazon.com/ec2 [last accessed July 2013].
23. Ezenwoye O, Sadjadi S. A proxy-based approach to enhancing the autonomic behavior in composite services. *Journal of Networks* 2008; **3**(5):42–53.
24. Simmonds J, Yuan G, Chechik M, Nejati S, O'Farrell B, Litani E, Waterhouse J. Runtime monitoring of web service conversations. *IEEE Transactions on Services Computing* 2009; **2**(3):223 –244.
25. Robinson W, Purao S. Monitoring service systems from a language-action perspective. *IEEE Transactions on Services Computing* 2011; **4**(1):17 –30.
26. Tsalgatidou A, Pilioura T. An overview of standards and related technology in Web services. *Distributed Parallel Databases* 2002; **12**(2):135–162.
27. Gulcu K. Fault masking as a service. *Master's Thesis*, Ozyegin University, Turkey, 2013.
28. Leon-Garcia A. *Probability, Statistics, and Random Processes for Electrical Engineering.* 3rd Edition, Prentice Hall: Upper Saddle River, New Jersey, USA, 2007.
29. Johari R. Lecture notes of MS&E 221: Stochastic Modeling, Stanford University, 2011. Available at: http://eeclass.stanford.edu/cgi-bin/handouts.cgi?cc=msande221&action=handout_download&handout_id=ID117268377726079 [last accessed July 2013].
30. 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*, Beijing, China, 2008; 577–584.
31. Node.js. Available at: http://nodejs.org/ [last accessed July 2013].
32. Menzel M, Ranjan R. CloudGenius: decision support for web service cloud migration, In *Proceedings of the International ACM Conference on World Wide Web*, Lyon, France, 2012; 16–20.
33. Baier C, Haverkort B, Hermanns H, Katoen J. Model checking continuous-time Markov chains by transient analysis. In *Proceedings of the 12th International Conference on Computer Aided Verification*, Chicago, Illinois, USA, 2000; 358–372.
34. Chen Y, Romanovsky A. WS-mediator for improving the dependability of web services integration. *Journal of IT Professionals* 2008; **10**(3):29–35.
35. Lo T. Trends in the Smart TV industry. *Technical Report*, Digitimes Research, 2012.
36. Wang L, Tao J, Ranjan R, Marten H, Streit A, Chen J, Chen D. G-Hadoop: MapReduce across distributed data centers for data-intensive computing. *Future Generation Computer Systems Journal* 2013; **29**(3):739–750.
37. Gorbenko A, Iraj EK, Kharchenko VS, Mikhaylichenko A. Exception analysis in service-oriented architecture. In *Proceedings of the 6th International Conference on Information Systems Technology and its Applications*, Kharkiv, Ukraine, 2007; 228–233.
38. Chen I, Ni G, Kuo C, Lin CY. A BPEL-Based fault-handling architecture for telecom operation support systems. *Journal of Advanced Computational Intelligence and Intelligent Informatics* 2010; **14**(5):523–530.
39. Salatge N, Fabre JC. Fault tolerance connectors for unreliable Web services. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Edinburgh, UK, 2007; 51–60.
40. 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*, Enschede, The Netherlands, 2005; 95–105.