WSDM-Enabled Autonomic Augmentation of Classical Multi-version Software Fault-Tolerance Mechanisms

Roeland Dillen, Jonas Buys, Vincenzo De Florio, and Chris Blondia

University of Antwerp
Department of Mathematics and Computer Science
Performance Analysis of Telecommunication Systems
1 Middelheimlaan, B-2020 Antwerp, Belgium
Interdisciplinary Institute for Broadband Technology
8 Gaston Crommenlaan, B-9050 Ghent-Ledeberg, Belgium

Abstract. Web services are increasingly deployed in many enterprise applications. For this type of applications, dependability issues are usually resolved by introducing some form of redundancy in the system. Whereas hardware redundancy schemes have traditionally been defined through static configurations based on worst-case analysis, the enhanced flexibility and interoperability of web services allows for dynamic (self-) management of redundancy at the application layer. Combining this advantage with service-oriented platforms such as OSGi facilitates the replication of software components and their integration within redundancy schemes. The application of such redundancy schemes inevitably comes at a price though — primarily due to the allocation of additional system resources. It is often unknown to the service provider how much redundancy and management complexity is required. Furthermore, the degree of redundancy and the dependability strategy to be employed may be restricted by the budget and requirements of the client, both of which may vary. In this paper, we propose a solution to allow the client to express a trade-off between its dependability requirements and its available budget at request level. A dedicated service provider will then attempt to honour these objectives — failing to do so would obviously result in failure from the client point of view. Furthermore, we show how classical multi-version software fault-tolerance techniques can be augmented with advanced redundancy management leveraging the Web Services Distributed Management standard.

1 Introduction

When constructing complex software systems, dependability issues will eventually unfold. In [1], Laprie defines dependability as the combination of reliability, availability, safety, security and maintainability. Amongst the available techniques to achieve dependability, and improve the reliability and availability in particular, a great deal of attention has been paid in the literature to fault-tolerant redundancy schemes leveraging design diversity. Two prevalent examples of multi-version software fault-tolerance strategies are recovery blocks and n-version programming [2,3]. Recovery blocks subject the response of a call to a replica to an acceptance test, trying each replica in sequence until the output passes the acceptance test or until there are no more replicas left to try. In an n-version programming redundancy scheme, however, replicas are queried in parallel and a decision algorithm is responsible for adjudicating the correct result. Many different types of decision algorithms have been developed, which are usually implemented as generic voters [4]. One example of such voting approaches is plurality voting: for each invocation of the scheme, the replicas will be partitioned based on the equivalence of their results, and the result associated to the largest cluster will be accepted as the correct result.

These classical fault-tolerant strategies have traditionally been applied with a predetermined degree of redundancy on an immutable set of replicas. As such, they are context-agnostic, i.e. they do not take account of changes in the operational status of any of the components contained within the redundancy scheme. It was shown in [5] that this lack in flexibility may jeopardise the effectiveness of the fault-tolerant unit from a dependability, timeliness as well as a resource expenditure perspective. All dependability originating from the use of redundancy inevitably comes at a price, which is primarily due to the additional expenditure ensuing from the allocation of additional system resources. While a fixed amount of redundancy is applicable to hardware systems, applying fault-tolerance strategies at the application layer allows to incorporate advanced redundancy management capable of choosing the amount of redundancy autonomously.

In this paper, we formulate an approach to leverage the flexibility of service-oriented architectures to show how classical multi-version software fault-tolerance techniques can be augmented with advanced redundancy management. Firstly, we propose a solution to allow the client to express a trade-off between its dependability requirements and its available budget at request level. Accordingly, the system will autonomously select the appropriate amount of redundancy, honouring the budgetary constraints stated. Secondly, the system is responsible for maintaining a pool of instances — replicas — of a specific web service and is capable of autonomously deploying additional replicas (if needed), or removing or rejuvinating existing, poorly performing replicas.

The remainder of this paper is structured as follows: Section 2 provides an overview of the technologies used for our service-oriented solution. The design and architecture of the solution is then described in section 3. Next, the performance of the system will be analysed in section 4, after which we conclude this paper by a discussion and future work.

2 Key Technologies and Standards

In order to achieve its design goals, our solution relies on several key technologies. The first employed technology is web services. A web service is defined by the W3C as "a software system designed to support interoperable machine-to-machine interaction over a network" [6, Sect. 1.4]. XML-based web services in particular offer a high degree of interoperability, which mainly stems from the use of the Simple Object Access Protocol (SOAP) protocol to envelop messages to be exchanged, and from the numerous standardisation initiatives.

One such standard is the Web Services Distributed Management [7] (WSDM) family of specifications, which defines how networked resources can be managed by exposing their capabilities and properties by means of a web service interface. The constituent Management of Web Services (MoWS) specification describes how web services themselves can be considered as resources, and require manageability features as well [7, WSDM-MOWS]. More specifically, it defines a number of metrics to expose information regarding the operational status of a web service, which are of particular intrest within the scope of this paper.

Finally, the Java-based Open Services Gateway initiative (OSGi) framework allows bundled applications and services to be remotely and dynamically deployed, without necessitating a reboot of the system. This characteristic will be exploited to automatically deploy new instances of a specific web services (i.e. replicas) should the available amount of system resources prove to be insufficient.

3 Basic Principles and Components

In this section, we provide an overview of the architecture of our service-oriented solution and elaborate on its design aiming to improve the effectiveness of traditional recovery strategies from the following three angles.

Firstly, our solution enables the dynamic and autonomic management of the degree of redundancy of the system. The rationale for this objective is that a predetermined degree of redundancy has traditionally been hardwired within classical software fault-tolerance strategies. Changes in the operational status of the system (i.e. its context) may result in the over- or undershooting of the required degree of redundancy needed to sustain a certain level of dependability [8,5]. It is also likely that the optimal degree of redundancy changes in time.

Secondly, the reliability of the fault-tolerant composite is largely dependent on the quality of the constituent replicas [9]. Our architecture therefore includes a monitor component, which was designed to observe changes in the operational status of the available replicas. As such, replicas that consistently perform poorly may be removed and replaced if necessary, which may further improve the reliability of the composite.

Thirdly, for individual requests issued on the fault-tolerant composite, the system will intelligently determine an appropriate degree and selection of replicas honouring the dependability requirements expressed by the client, *i.c.* the redundancy strategy to be used and timing as well as budgetary constraints. The anticipated cost of invoking a redundancy scheme is primarily determined by the resource allocation expenditure model and the operational status of the available system resources maintained by the service provider though. We have therefore chosen to implement the service provider as a WSDM-enabled resource, exposing the expected cost for the various redundancy strategies it supports by means of resource properties [10, WSRF-RP]. This allows the client to judiciously select a service provider capable of delivering the requested service level.

An overview of the system architecture is shown in Fig. 1. The architecture includes four dedicated components: a replicator, a dispatcher, a monitor and a context data repository (CDR), each of which have been implemented as WSDM-enabled web services. Any web service that will serve as version to be replicated is required to expose a WSDM manageability interface.

In line with our second objective described hereabove, the replicator component is responsible for maintaining a given amount of instances (replicas, that is) of a specific web service (version). Deploying multiple instances of a particular software component in a distributed system has proved successful in lowering the risk of a complete system failure as the result of hardware failures [5]. It is assumed that an increased degree of redundancy results in an increase of the dependability, provided that badly performing replicas are removed. In order to support the dynamic replication of a web service, the replicator will manage of a number of agents. Such utility application web services are deployed on different network hosts and will periodically broadcast a heartbeat to the replicator. When the replicator issues a command to replicate a web service, one or more agents will be instructed to locally deploy a new instance of the service. The replicator exposes a manageability interface for the manipulation of replicas: new instances can be created, the amount of replicas for a given version can be adjusted, and individual replicas can be disabled. Note how the replicator maintains a registry of the system resources, federating different service groups, each exposing the deployed instances of a specific web service [10, WSRF-SG]. This design permits WSDM advertisement messages to be broadcast upon service creation or destruction, such that the monitor and the CDR components can acquire the relevant context information.

The monitor component serves the purpose of monitoring the operational status of the replicas in the system. It was implemented to automatically enrol for participation in a publish-and-subscribe model, so as to receive notifications issued by the WSDM framework on behalf of a replica whenever the value of some MoWS metric changes [11,7]. Notification messages reporting on the change of the value of these metrics need to comply to the format as defined in [10, WSRF-RP]. For instance, a replica that returns faulty responses all too frequently, as can be deduced from the value of the `mows:NumberOfFailedRequests`

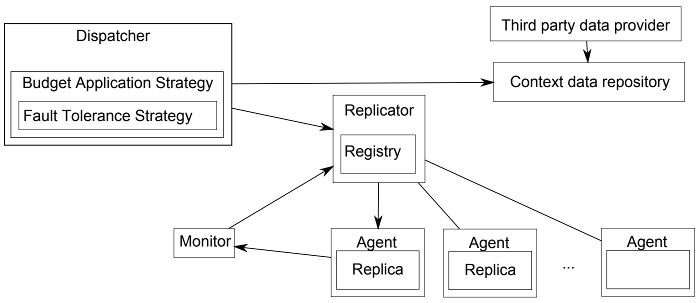

Fig. 1. Overview of the overall system architecture

metric, can be eliminated from the system and replaced by another. In this capacity, the monitor will use the replicator's manageability interface.

Client transparency is attained by means of the dispatcher component that exposes the redundancy scheme as a single web service, shielding the intricacies of the redundancy management. The dispatcher exposes a manageability capability to deploy an OSGi bundle in the system that creates replicas as well as the composite service. Moreover, it can be fitted with support for different fault-tolerance strategies. The dispatcher will select an appropriate degree and selection of resources and integrate them within an appropriate fault-tolerant redundancy scheme, attempting to honour the constraints and preferences expressed by the client. For instance, the client could specify that the dispatcher should use a recovery block strategy, selecting an adequate selection of replicas that does not exceed the budget, or that is guaranteed to return a result within a given time span. We will explain this approach in further detail in Sect. 3.1.

The CDR can be set up acting as a receiver for third party metrics of particular interest, each of which are identified by the QName of the corresponding resource property exposed through the WSDM manageability interface. By default, it will attempt to issue a subscription request so as to enrol in a publish-and-subscribe scheme and receive notifications whenever the value of the relevant metrics change [11,7]. Again, the payload of these messages is formatted as defined in [10, WSRF-RP]. The system will attempt to establish such subscriptions for all replicas registered within the system, driven by the WSDM advertisement messages issued by the replication mechanism.

3.1 Budget Application Strategy (BAS)

We will now elaborate on the redundancy management provided by the dispatcher and how it was designed to determine an adequate selection of replicas matching the requirements stated by the client. An overview of the overall process can be found in Fig. 2.

The cost resulting from the invocation of a redundancy scheme is modelled by two components: the cost to transfer the request message from the dispatcher to the selected replicas and have the response returned accordingly, and the processing cost charged for the use of an invoked replica. We will use an abstract model to quantify the cost resulting from these two components, expressed in currency_unit/byte, respectively currency_unit/ms. The unit price is (dynamically) set for individual replicas and can be retrieved from the CDR. An estimate of the processing cost can be obtained when considering the average processing time, which can be obtained from the `mows:ServiceTime` and `mows:NumberOfRequests` metrics. Similarly, the cost originating from the use of a network datagram service for the invocation of a replica can be calculated from resource properties exposed by the CDR. A new metric was added to store the cumulative message payload size of both the incoming request and the outgoing response. This model, albeit simplistic, enables the dispatcher to rank the available system replicas in terms of their total estimated usage cost, sorting them from cheap to expensive. A subset is then iteratively chosen until the available budget has been exhausted. Note that the budget is not necessarily entirely spent, for it is not always necessary to actually use all the replicas selected — *cf.* recovery blocks *vs.* n-version programming. Having determined an adequate selection of system replicas, the dispatcher will then initialise the selected redundancy strategy, integrating the selected replicas. Our dispatcher implementation currently supports the following strategies:

1. Plain recovery block strategy: the selected replicas are queried in turn until one of them has returned a response that passes the acceptance test.
2. Replicating recovery block strategy: similar to the plain recovery block strategy. If the budget has not been entirely exhausted and all active replicas have been tried, additional replicas will be created and invoked until a valid response is obtained, or until the budget has been spent.
3. Active voting strategy: n-version programming scheme in which all replicas are queried simultaneously; the first response acquired will be returned.
4. Plurality voting strategy: n-version programming scheme in which all replicas are queried simultaneously. If the decision algorithm can establish the existence of a consensus block that constitutes a plurality in the generated partition based on the equivalence of the responses returned, the corresponding response is returned.

```
<soap:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
    <wsa:To>http://localhost:8888/dispatcher/services/service</wsa:To>
    <wsa:Action>http://adss.pats.ua.ac.be/service/version</wsa:Action>
    ...
    <wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
                xmlns:rcfg="http://adss.pats.ua.ac.be/dispatcher"
                xmlns:ba="http://adss.pats.ua.ac.be/service">
        <ba:BudgetAssertion>
            <ba:budget>
                <ba:value>0.05</ba:value>
                <ba:unit>ct</ba:unit>
            </ba:budget>
        </ba:BudgetAssertion>
        <rcfg:RecoveryConfig>
            <rcfg:RecoveryStrategy>
                http://adss.pats.ua.ac.be/strategies/PlainRecoveryBlockStrategy
            </rcfg:RecoveryStrategy>
            <rcfg:parameters>
                <rcfg:parameter>
                    <rcfg:name>timeout</rcfg:name>
                    <rcfg:value>1000</rcfg:value>
                </rcfg:parameter>
            </rcfg:parameters>
        </rcfg:RecoveryConfig>
        <rcfg:AcceptanceTest>
            contains(//versionResponse/text(),'i␣carry␣correct␣result')
        </rcfg:AcceptanceTest>
    </wsp:Policy>
</soap:Header>
```

Listing 1.1. The dependability, timing and budgetary requirements of the client are relayed by means of a WS-Policy element embedded in the SOAP header.

Which of the aforementioned fault-tolerance strategies will be employed is at the behest of the client, a choice which will be conveyed to the dispatcher by means of a special WS-Policy element embedded within a SOAP header block attached to the issued request message, as is shown in List. 1.1 [12]. This WS-Policy element contains a set of assertions, including the mandatory parameterised assertion `rcfg:RecoveryConfig`. Its purpose is to identify the redundancy strategy by means of a predefined Uniform Resource Identifier (URI), and to provide any additional parameters the chosen strategy may require. An optional `ba:BudgetAssertion` can be included to state the available budget. Observe how the acceptance test to be used for the recovery block strategies is relayed through the parameterised `rcfg:AcceptanceTest` assertion, holding an XPath expression that will be used to assess the validity of the response returned by each of the probed replicas. Furthermore, an optional `rcfg:parameter` element can be set by the client to hold a timeout parameter. It is, however, entirely up to the dispatcher's implementation of the redundancy scheme to honour this constraint.

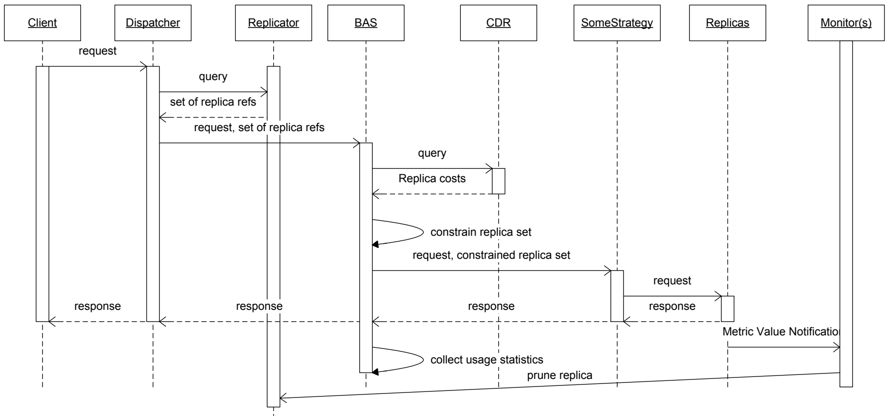

Fig. 2. Request handling overview: sequence diagram

It makes no use to squander the available budget on replicas that do not significantly contribute to the effectiveness of the redundancy scheme. As an additional measure to increase the overall availability of the requested service, the monitor component is configured so as to automatically replace replicas with a high degree of failed requests. In this capacity, the monitor will observe changes in the values of the metrics defined in the MoWS specification so as to assess the health of a replica. It does so by considering the relative number of requests for which a given replica failed to return a valid response, i.e. for which a SOAP fault was returned. When the ratio between the `mows:NumberOfFailedRequests` and the `mows:NumberOfRequests` metrics is found to exceed a certain threshold, the affected replica is considered to be unhealthy and the monitor will instruct the replicator to remove the replica from the system. Note that this process is fully transparent to the client and is driven by the monitor component. The dispatcher allows the plain and replicating recovery block and plurality voting strategies to be enhanced with this replica pruning procedure.

4 Performance Analysis

In this section, we will analyse the performance of each of the proposed strategies in Sect. 3.1, assuming they are operating in an environment subject to transient faults. Failures are injected into the system by means of a special service that will affect the outcome of an acceptance test for strategies built on the recovery block procedure, or that will affect the generated partition for voting-based strategies.

For a given replica, it is assumed the number of requests in between any two successive failures is geometrically distributed with a constant parameter p. This failure probability p is drawn from an exponential distribution with $\lambda = 3.33$. As random variates drawn from an exponential distribution do not necessarily generate values within $[0, 1]$, all the mass above is truncated to 1. The rationale behind the use of the exponential for sampling values for p as described is that a replica is assumed to be affected by transient faults and will therefore only fail periodically. Lower failure rates approaching 0 are more likely, as opposed to permanent faults for which $p = 1$. The choice of λ will result in the generation of probabilities high enough to be visualised easily but not excessively so. By analogy with the findings in [13], variations in the response times of replica invocations are simulated utilising a gamma distribution. In what follows, we will analyse the performance for the following redundancy strategies when subject to the failure model just described:

1. Simplex system: a single replica which obviously is not tolerant of failures.
2. Plain recovery block strategy: classical recovery block, as defined in Sect. 3.1.
3. Replicating recovery block strategy: extends the logic of the plain recovery block in that additional replicas are automatically created if the initial set of replicas to be used is exhausted and a sufficient share of the budget is left.
4. Plurality voting strategy: n-version programming scheme combined with plurality voting, as defined in Sect. 3.1.

Furthermore, some of these strategies have been tested in combination with the replica pruning feature introduced in Sect. 3.1. Replicas are judged unhealthy if more than 20% of the requests previously handled have failed. The monitor will instruct the replicator to remove such replicas from the system.

5. Plain recovery block with replica pruning, extending scenario 2.
6. Replicating recovery block with replica pruning, extending scenario 3.
7. Plurality voting strategy with replica pruning, extending scenario 4.

4.1 Results

We will now provide an overview of the performance of the strategies mentioned. Each of the graphs below compares the basic scenario, without and with replica pruning by the monitor component. Figures 3, 4 and 5 show the same metrics, measuring the (in)effectiveness of a given redundancy strategy:

– Errors (plusses and crosses in the graphs at the right): indicative of the percentage of the total number k of invocations of the composite that resulted in failure ($k = 4000$).

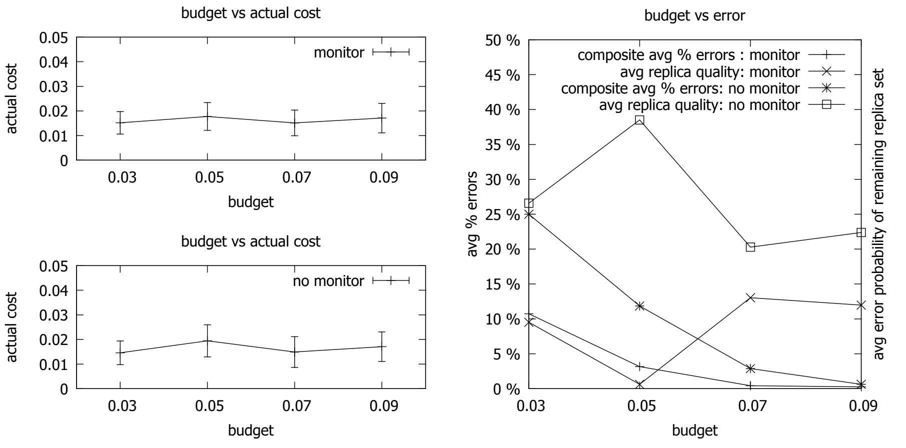

Fig. 3. Scenarios 2 and 5: Plain recovery block strategies

– Costs (shown in the graphs at the left): indicative of the average, actual cost for all requests. Graphs show average and standard deviation.
– Average replica quality (squares and asterisks): at the end of a run, all remaining replicas in the system that were not pruned by the monitor are asked for the probability with which they were struck by a failure. The corresponding average is shown.

There is no need to actually simulate the first scenario, i.e. the simplex system, as one can directly derive the percentage of the k invocations of the system that will result in failure. As the error degree for each replica is chosen from an exponential distribution with $\lambda = 3.33$, the mean failure probability is $\mu = 0.3$. One can then surmise that the Bernouilli process that describes the probability of a failure affecting a request has probability parameter $p = \mu$.

Figure 3 shows the results of scenarios 2 and 5. The percentage of failed invocations is below 30%, for both scenarios with and without the monitor, and thus an improvement compared to the simplex system. A growing degree redundancy because of a bigger budget results in less failures of the scheme. Furthermore, the advantage that the replica pruning appears to deliver diminishes as the budget increases. The effect of the replica pruning is also visible on the average replica quality. The quality of the replicas is distinctly less without replica pruning. The actual cost of the recovery block strategy appears to be, on average, much lower than the actual budget. The cost, however, appears to vary significantly, although not to a degree that it will likely exceed the budget.

Figure 4 shows the results for the replicating recovery block, in which the classical recovery block scheme is augmented with the possibility to create further replicas. The error rate is well below the 30% benchmark for the simplex system. Furthermore, it can be observed that a small advantage is gained from utilising the monitor component. With respect to the cost, the same discrepancy between the actual cost and the budget is noted, including a significant degree

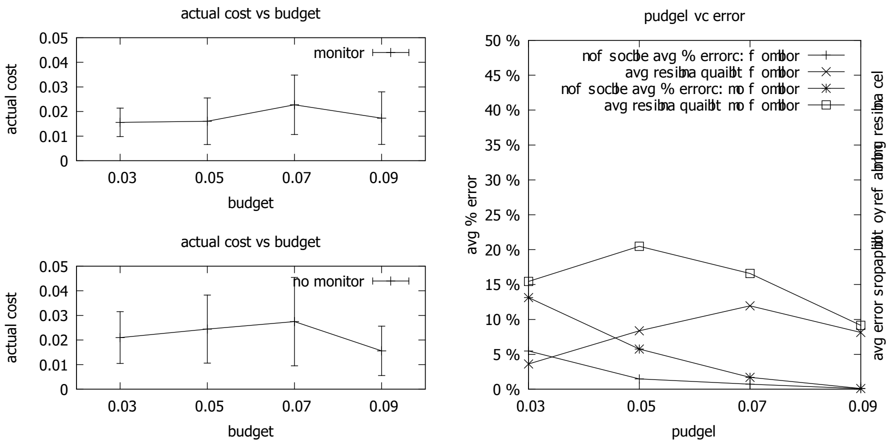

Fig. 4. Scenarios 3 and 6: Replicating recovery block strategies

of variability. The same diminishing returns of the replica pruning mechanism can be observed as in the previous experiment. As the budget and therefore the number of replicas increases, their average quality worsens. This may be an effect of the abundance of replicas causing not all replicas to be queried frequently enough for the monitor to confidently decide to prune a replica.

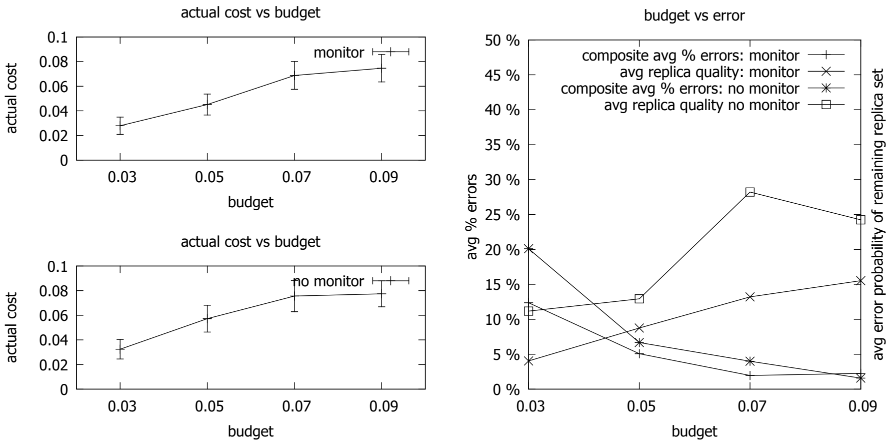

Fig. 5. Scenarios 4 and 7: Voting strategies, with and without replica pruning

Figure 5 shows the classical voting strategy augmented by the measures mentioned in Sect. 3.1. The voting strategy also performs better than the 30% benchmark set by a single service. It also benefits from the replica pruning by about

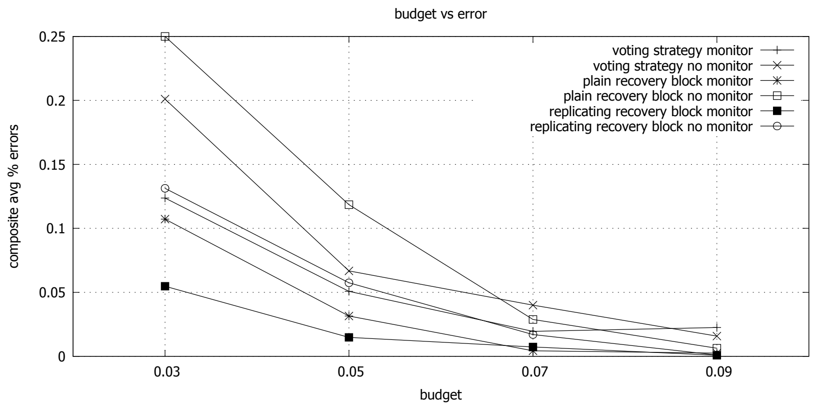

Fig. 6. Percentage of failed composite invocations

10% points in the lower budgets. As more budget becomes available — and therefore more replicas — this advantage diminishes as is also the case for scenarios 5 and 6 based on recovery blocks. This strategy seems to have an effect on the overall replica quality: even in the higher budgets the average replica quality is better in the monitor case. This is most likely caused by the fact that all selected replicas that fit within the budget are actually invoked. This causes the monitor to reach a good confidence about the health of the replica much faster.

Figure 6 shows a comparison of the ratio of invocations resulting in failure of all transient scenarios. It shows that all the strategies provide the best results when fitted with a replica pruning monitor. Of all the pruning strategies the addition of direct replication capabilities in combination with replica pruning as done by the replicating recovery block yields the best result for the smallest budget. We have determined that the use of some simple MoWS features like the `mows:NumberOfFailedRequests` metric already can provide some meaningful improvement on the classical multi-version fault-tolerance strategies.

4.2 Discussion

The replicating recovery block strategy with replica pruning clearly yields the best results. It must be noted though that devising a proper acceptance test is usually very application-specific and may not always be possible due to the limited information exposed in the service interface. Improving the applicability of the recovery block mechanism, our system has been designed so that the acceptance test can be configured at runtime, and is no longer hardwired within the fault-tolerant unit.

The voting strategy can be more widely applied, as it employs a generic plurality voter. The primary disadvantage of the voting system though stems from the fact that all selected replicas will be invoked in parallel, incurring greater actual cost. In this regard, the sequential iterative invocation of individual replicas by the recovery block mechanisms show that, on average, not all replicas are actually used. At the risk of sporadically overstepping the budget, much greater redundancy — and therefore dependability — can be provided.

Because of the monitoring component and its ability to remove poorly performing replicas, an increase in replicas will result in an increase in dependability, provided that the replicas are used sufficiently; it takes a number of requests to achieve sufficient confidence about the health of a replica.

5 Conclusions and Future Work

In this paper, we have shown how software fault-tolerance strategies can be applied to XML-based web services, aiming to increase the dependability, and in particular the availability of the overall service the system seeks to provide. Furthermore, the design of dedicated WSDM-based web services can augment these classical fault-tolerance strategies in that they can accommodate for advanced redundancy management. We have argued that poorly behaving replicas can easily be detected leveraging some simple metrics provided by the MoWS specification. Moreover, it was apparent from our experimentation that it is advantageous to prune such replicas and have them replaced by newly initialised ones. A service-oriented architecture was introduced that builds on top of the OSGi and Apache MUSE frameworks encompassing a monitor component that keeps track of the operational status of the available system resources, and a replicator utility service to ease the dynamic deployment of additional replicas.

The flexibility offered by the service-oriented architecture presented allows to adaptively reconfigure the amount of redundancy and, accordingly, the selection of resources for individual requests. Moreover, our experiments suggest that, on average, combining the devised replica pruning and replacement features with a classical recovery block strategy outperforms the other fault-tolerance mechanisms tested, both in terms of cost and availability.

As part of future work, we envisage investigating more advanced detection mechanisms for the monitor component, encompassing additional metrics extending beyond the set of metrics defined in MoWS. Furthermore, additional experimentation is required to obtain a more general view on the performance of the system using a wide range of failure injection models.

References

1. Laprie, J.C., Avižienis, A., Kopetz, H. (eds.): Dependability: Basic Concepts and Terminology. Springer (1992)
2. Randell, B.: System structure for software fault tolerance. In: Proceedings of the 1st ACM International Conference on Reliable Software, pp. 437–449 (1975)
3. Avižienis, A.: The N-version approach to fault-tolerant software. IEEE Transactions on Software Engineering 11(12), 1491–1501 (1985)
4. Lorczak, P., et al.: A theoretical investigation of generalized voters for redundant systems. In: IEEE Digest of Papers on the 19th International Symposium on Fault-Tolerant Computing (1989)

5. Buys, J., et al.: Towards Context-Aware Adaptive Fault Tolerance in SOA Applications. In: Proceedings of the 5th ACM International Conference on Distributed Event-Based Systems, pp. 63–74 (2011)
6. W3C: Web Services Architecture (2004), `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/`
7. OASIS: Web Services Distributed Management (WSDM) 1.1 (2006), `http://www.oasis-open.org/committees/wsdm`
8. De Florio, V.: Robust-and-evolvable resilient software systems: Open problems and lessons learned. In: Proceedings of the 8th ACM Workshop on Assurances for Self-Adaptive Systems, pp. 10–17 (2011)
9. De Florio, V., et al.: Software tool combining fault masking with user-defined recovery strategies. IEE Proceedings – Software 145(6), 203–211 (1998)
10. OASIS: Web Services Resource Framework (WSRF) 1.2(2006), `http://www.oasis-open.org/committees/wsrf/`
11. OASIS: Web Services Base Notification 1.3 (2006), `http://www.oasis-open.org/committees/wsn/`
12. W3C: Web Services Policy 1.5 - Framework (2007), `http://www.w3.org/TR/ws-policy/`
13. Gorbenko, A., et al.: Real Distribution of Response Time Instability in Service-oriented Architecture. In: IEEE Symposium on Reliable Distributed Systems, pp. 92–99 (2010)