Autonomic Service-Oriented Architecture for Resilient Complex Systems

Pieter den Hamer¹
Information Technology
¹DNV Research & Innovation
Høvik, Norway
+31 6 17850425
pieter.den.hamer@gmail.com

Torbjørn Skramstad¹, ²
Department of Computer and Information Science
²Norwegian University of Science and Technology
Trondheim, Norway
+47 97123246
torbjorn.skramstad@idi.ntnu.no
torbjorn.skramstad@dnv.com

ABSTRACT
Service Oriented Architectures (SOAs) are increasingly applied in industrial systems that need to be very dependable as they are critical for business continuity, safety and/or security. In addition, such systems are getting ever more complex, especially in terms of their number of interacting sub-systems or components and emergent (mis)behavior. In the face of this growing complexity, SOA based systems need to be resilient to meet high dependability requirements, even in increasingly dynamic system environments. This desired resilience is supported, amongst others, by ‘autonomic’ SOA systems that have important run-time capabilities like introspection and self-adaptation. In this paper we explore different concepts and techniques that are needed to achieve this.

Keywords

1. INTRODUCTION
In our work, which amongst others includes the verification and validation of the quality of software-intensive systems, we observe the growing use of SOA for industrial systems [13], including monitoring and control systems that are critical for production continuity, safety and/or security. A good example of this is the growing automation and remote control of offshore energy operations [11] [15] that are increasingly based on SOA.

As a result, SOA systems increasingly have to be able to meet high to very high dependability requirements. Examples include mission critical systems on ships or airplanes, business critical systems for production in process industry and offshore industry [15] and life saving safety systems in all possible industrial and transport domains.

At the same time these (SOA) systems are facing increasing levels of complexity [1]. This complexity typically arises from a number of factors. Amongst these factors is the growing level of interaction and integration of an ever-increasing number of software components and sub-systems, resulting in ‘systems of systems’ [6] and a higher degree of interdependence between software components. Other factors that underlie growing complexity are the increase in the distribution of software components, the growing functional scope and diversity of the systems and increasing technical heterogeneity in terms of variety of the technical run-time environments of the different software components and sub-systems.

Also, systems are ever more ‘open’ in the sense that systems interoperate during run-time with software components of 3rd parties, typically through the internet, for instance provided as ‘software as a service’ or in the context of information exchange between parties in a supply chain. Systems are also more open in the sense that software components that are assembled during design and implementation (or that are added or replaced during the maintenance of an operational system) may come from a variety of suppliers, sometimes as tailor made software, sometimes as ‘commercial off the shelf’ products of known or unknown pedigree [5]. Adding to the complexity of systems is the fact that they have to operate in a dynamic run-time environment, meaning amongst others that systems have to be able to deal with more variation in the contents and quality of (input) data and fluctuations in the magnitude and character of system usage, both by human operators and other interacting systems [16].

The growing complexity of systems is not only true for business information systems, but also for industrial systems, including real-time control systems. We see a tendency of blurring lines between business information systems and industrial automation systems, again a result of higher levels of integration and interaction, which is a response to more demanding business requirements with respect to efficiency improvement, real-time monitoring, decision support, and process optimization [13] [15].

A work initiated by OLF, the Norwegian Oil Industry Association [19] recommends the use of a Service Oriented Approach as the best practice for implementing Integrated Operations, Generation Two in order to ensure reuse of existing solutions and future reuse of new solutions.

The main contributions of this paper is that it explores different concepts and techniques needed for a system to adapt dynamically to changing internal or external circumstances, both foreseen and unforeseen, in order to support the dependability of the system. It also contains a first draft categorization of the techniques with respect to applicability to system criticality or

¹ Interoperation across assets, operation centres and corporate and national boundaries in a timely and secure manner
confident level, maturity, and the expected increase in complexity the introduction of the technique represents.

The paper is structured as follows. Section 2 briefly introduces the concept of Basic Service Oriented Architecture. Section 3 describes the capabilities and challenges related to Autonomic Service Oriented Architectures. Section 4 discusses desired and undesired emergent behavior of such systems. Section 5 gives some criteria to guide the selection of techniques based on criticality of the system, maturity of the technique and the degree of increased complexity by using the technique. Finally, section 6 concludes the paper.

2. ‘BASIC’ SERVICE ORIENTED ARCHITECTURE

Growing system complexity is a strong driver behind the increasing use of SOA in many industries, as SOA in itself is a way of dealing with complexity, while meeting requirements towards dependability and flexibility [7].

The benefits of ‘basic’ SOA (versus ‘autonomic’ SOA as described in the next paragraph) are amongst others due to the ‘loose coupling’ of services, balanced with limitation of service interdependency, and the use of ‘service middleware’ that include process orchestration, fault tolerance, data transformation, dynamic service binding and other mechanisms that are increasingly part of modern ‘enterprise service bus’ software implementations.

Equally important is the fact that SOAs tend to be highly scalable in terms of number of services, number of users and data growth. Modern SOA infrastructure, which may include the use of cloud computing, offers a SOA habitat that is often better able to deal with aspects like peak loads and growth of storage and computing capacity than more traditional monolithic system environments.

In addition, SOA offers potential advantages like service reuse which tends to lead to higher service reliability, late run-time service binding which allows for temporal rerouting and service maintenance, and other characteristics that fit nicely with high availability requirements of any system that needs to be in operation for a long continued time period, with no or very limited maintenance time windows.

3. AUTONOMIC SERVICE ORIENTED ARCHITECTURE

As SOA systems increasingly have to deal with high dependability requirements in very dynamic environments and with increased complexity, there is a growing interest in the area of ‘autonomic computing’ [10] or more specifically ‘autonomic SOA’ [4] [17]. In most ongoing research this terminology mainly refers to improving the capability of dealing with run-time problems in service composition, being the key mechanism of SOA applications that are composed or ‘assembled’ of individual services. Such problems typically include unavailability of services which may occur because of service failures, server breakdowns or because of problems in the network that connect services with each other. As services may be very distributed, and possibly provided by 3rd parties, service composition is quite vulnerable if no measures are taken to deal with the likely event of service unavailability [3]. In addition, service composition is vulnerable to data incompatibility (syntactic and semantic) and other quality issues in the data that is exchanged between services [18].

Although not yet widely applied in practice, ‘autonomic SOA’ promises an increased capability of a SOA to deal with events that were unforeseen at design time, by having self-inspection and (non-deterministic) self-adaptation mechanisms [7] to support service composition. Autonomic SOA may further be supported by ‘life testing’ (during run-time) of new or existing services, graceful service degradation and data provenance, as described in the following paragraphs.

3.1 Self-inspection & self-adaptation

Self-inspection or introspection is typically accomplished [17] by applying run-time testing of service connections that are able to detect deviations or absence of expected responses of the service that is to be invoked. In practice such ‘self-inspection’ could be implemented in service middleware like ‘service mediators’ or an enterprise service bus (ESB) that receives connection requests. Before executing each request, a test is carried out that checks whether a set of pre-defined failure types may occur, using for instance a generic fault taxonomy as described in [17].

‘Self-adaptation’ is obtained by having response mechanisms as part of mediation between service requestors and service providers, to deal with an unresponsive service provider or other failure types. A relatively simple approach in case of unresponsive service providers is to reroute a service request to another identical service provider instance or to initiate a new equivalent service (see Figure 1). Such alternatives may be pre-defined during design time, in which case a service mediator must be in place that monitors service provider availability during run-time, and if necessary can reroute the service request to an alternative provider, using the pre-defined (yet maintainable) list of alternatives, which may or may not be in order of preference. This preference may depend on service level aspects like (data) quality, performance or costs.

An example of this could be the use of a weather-forecast service (derived from [9]): the first choice would be a high quality, low cost weather forecast service of, say, a public sector weather
institute. But if this service would not be available for some reason, perhaps because of network problems or a crash of the server(s) of the weather forecasting organization, an alternative would be to use a service of a commercial meteorological company, with similar quality but with higher costs. Yet a 3rd choice, in case the 2nd choice were also not available, would be the use of a service that provides a weather forecast of lower, yet still useful quality. This might be the case if the forecast is less accurate (on average) or perhaps less actual or less localized.

A more complex approach is to look for service alternatives dynamically, during run-time, rather than using the abovementioned pre-defined and static list of service alternatives. Such a dynamic approach is useful in cases of even higher availability requirements and/or SOA environments with a high level of variety of service providers and/or of their associated service level aspects like costs, performance or data quality. Such a dynamic service identification and invocation mechanism could be implemented by having so called active service repositories or registries that dynamically keep track of the ‘quality of service’ of service providers, combined with algorithms for run-time service search and service selection or matching [4].

A possible criterion to dynamically select services is to keep track of their quality of service in the course of time [3]. A simple example would be to keep track of the running average accuracy of a weather forecast service [9]. The selected service is the one with the highest running average accuracy at the moment of selection. A more advanced method of selection is to use a model to predict the quality of service, based on their historic actual usage behavior and/or testing behavior [9] [2], rather than just using an account of historic quality of service.

3.2 Life service testing

An underlying assumption with self-inspection and self-adaptation is that service providers are able to deal with ‘test’ invocations as part of their functioning in a ‘life system’ [12]. This means at least that testing a service does not lead to unwanted side effects or effects that may be mistaken for or may interfere with real (non-test) events. Such testing capability would be useful in itself (also without self-inspection and self-adaptation), for instance to test service upgrades in the context a life system that can not be easily shut down because of availability requirements. But to limit risks, testing is preferably done beforehand in separate test and acceptance environments that are as similar as possible to the actual production environment. If this is not possible then services and service middleware should also be able to recognize and execute tests in a runtime environment.

3.3 Graceful service degradation

‘Graceful service degradation’ is used here to indicate the approach, identified and being researched in our own work, in which the most critical services of a system are kept ‘alive’ as long as possible, even when less critical services in a system are no longer available. This situation may occur for instance because of a drop-down in available server resources or available connections or bandwidth.

To have graceful degradation, it is necessary to identify the business or industrial processes - and their associated services - that are most critical with respect to one or more organizational goals (continuity, safety, security). This should result in a specific priority that is determined for each service. Then following these priorities, services with higher priority should get a better chance of not being impaired or affected by failures or problems, for which several techniques are available.

One important technique is to monitor available resources for service execution (i.e. memory space, processor time) and if these resources become scarce, allocate resources according to service priority, which may even lead to refusing or postponing resource allocation to low priority services. If the latter is the case, then dependencies (which assumes the existence of well maintained services dependency tables) should be taken into account, preventing that absence or slow responding of low priority services have a negative impact on higher priority services. In addition, there should be a specific response to invocators of low priority services that their wanted service is (temporarily) unavailable, which should lead to an appropriate handling of the situation by the invocator.

3.4 SOA data provenance

In the case of using self-adaptation in service composition, as described earlier, alternative service providers may offer lower quality of data. The requesting service must be able to deal with this, or at least be made aware of the fact that data contained in the service response is less accurate, complete, precise, timely or otherwise less usable or reliable. This could be accomplished for instance by including a data quality indicator in the response, the name of the service provider or any other information that allows the service requestor to deal with possible quality deviations. If relevant, the service requestor should also use this to inform other services that depend on it, reflecting the fact that impaired quality of a service provider may lead to a whole chain of impacts on other services that directly or indirectly depend on the data that came from the selected service provider.

In fact, keeping track of data flows between services is of critical importance in any SOA, not only for reliability but also for security and other system quality aspects. In a wider context, data quality, lineage, invalidation, data definitions, semantics, transformations, ownership, confidentiality, integrity and other aspects are preferably comprehensively tackled in a system of ‘data provenance’ [18]. In contrast with more traditional software architecture and engineering, it is recommendable to implement data provenance in a separate category of ‘information services’ which acts as a layer between higher level business or functional oriented services on the one hand and lower level technical services on the other hand. Business or more high level functional services typically use information services to retrieve, store or change data, to validate, transform or analyze data or to obtain data definitions, lineage, ownership, confidentiality or other metadata [18] [11].

The alternative of not having a separate category of information services is to implement data provenance in a distributed fashion over business/functional services and technical services. This, however, is likely to result in redundancy and inconsistency of the data provenance ‘logic’, because the same data is often used in multiple services. Also, data is typically exchanged between services that are dynamically selected and that may be provided by 3rd parties.
The fact that data is exchanged between and is processed in a variety of services of known and unknown pedigree, calls for data provenance that supports the resilience of a SOA based system. This is why we explore possibilities in our work to design for more adaptive data provenance. More specifically, we are looking into mechanisms that are able to do run-time data quality monitoring and, based on that, execute adaptive data cleaning and data transformations that reduce the likelihood of failing services that can not deal with (unexpected) quality issues in their ‘input’ data.

4. COMPLEXITY AND EMERGENT (MIS)BEHAVIOR

It should be noted that the approaches towards autonomic SOA mentioned above, have a tendency to focus on the interaction at the level of individual components. At the system level, however, this type of dynamic interaction may lead to unexpected emergent behavior of the system (or system of systems) as a whole. A growing number of researchers point out that, based on complexity theory, the behavior of a system can not be well understood by only looking at the local behavior of individual components (or services). Behavior at system level is an emergent property of local interactions, and may be both desirable (i.e. resilience) and undesirable (i.e. unstableness, unexpected feedback loops) [1].

Examples of both desired and undesired emergent behavior are described in [14] and [8]. One of these examples is ‘herd behavior’ [14] or the unwanted cascading unavailability effect that may result from adaptive service selection, as described earlier in this paper. This effect can occur in systems where multiple service requestors try to invoke the same service provider. If that service provider is unavailable, the adaptively selected alternative service provider may also quickly become unavailable, as all service requestors ‘jump’ on the alternative collectively. The collective ‘hunt’ will turn to yet another alternative service, which quickly becomes overloaded as well. A cure for such unwanted ‘emergent’ behavior may be found by introducing some randomness in the selection of alternative services. Or, in other words, to introduce some variation in the adaptation mechanism and prevent ‘collective behavior’. An alternative cure could be to dynamically raise the attractiveness or the likelihood of selection of service alternatives in proportion to the actual selection and invocation of other service alternatives. Or, in reverse, to decrease the attractiveness of a service alternative, for each time it is selected, but not so strong to diminish other selection criteria, reducing the selection to ordinary load balancing. The point is, however, that what ever the ‘cure’ for emergent misbehavior, it will not be considered when our perspective lies only at component level, and not at system level [8].

In the light of complexity it should also be noted that, given the needed design and implementation efforts, an adaptive mechanism to select service provider alternatives and to deal with quality of service variation should only be used in situations where availability is critical, even when quality is impaired. Another perhaps ‘sobering’ observation is that introducing such mechanisms may in itself increase the complexity of the system in such a way that reliability may be impaired. In other words, we should be careful not to introduce a cure that is worse than the disease. Having an autonomic SOA therefore should be carefully balanced with the actual need for such measures. Mechanisms that ‘complexify’ relatively simple SOA systems without very high reliability requirements do not make a lot of sense.

5. TOWARDS A RECOMMENDED APPROACH

To give professional practitioners involved in designing, implementing, testing, assessing or managing systems guidance as to which resilience techniques, as described in the previous paragraphs may be applicable to a specific situation we provide some preliminary ideas. These recommendations are shown in Table 1. DNV RP-D201 [5] introduces the concept of confidence level. The required confidence level for a system or a function is determined as combined safety, business continuity, security and environmental considerations and represents a sort of overall criticality with respect to dependability. The confidence level is between 0 and 3, 0 representing the lowest level and 3 the highest. In the table each technique is related to a minimum confidence level (1-3) (see column 2), the higher the required confidence level the more need for resilience. The third column gives a rough estimate of the maturity of the technique. Some techniques are already used in practice and may be considered as proven technology (high maturity), others are used in practice, but are really not proven (medium maturity) and still others are hardly used in practice or exists only as prototypes or proof of concept (low maturity). The rightmost column (column 4) indicates the increased complexity of the system by introducing the technique. This is low when the system is hardly affected by the introduction of the technique, medium when the complexity is somewhat affected, and high when significantly affected.

<table>
<thead>
<tr>
<th>Resilience technique</th>
<th>Min. conf. level</th>
<th>Maturity</th>
<th>Increased complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self inspection</td>
<td>3</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Self adaption – predefined</td>
<td>3</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Self adaption – dynamic</td>
<td>3</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Life service testing</td>
<td>1</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Graceful service degradation</td>
<td>2</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>SOA data provenance</td>
<td>2</td>
<td>Medium</td>
<td>Low</td>
</tr>
</tbody>
</table>

Table 1: Resilience techniques categorized according to recommended confidence level, maturity, and increased complexity of the system

It should be noted that these categorizations are based on qualitative assessments from a combination of literature studies and own experience.

Further elaboration and validation is still needed and is left for further study.
6. CONCLUSION
In this paper we have explored different approaches for autonomic SOA that are pursued in the growing body of research work in this area, including our own. We have tried to present these approaches in a way that reflects their complementariness and mutual support, as far as present and relevant. However, looking at the current body of work that we have explored, we observe fragmented focus areas and results, and we therefore believe that the field could benefit from a more common framework and research agenda. We also have observed that in order to be effective, several of the current approaches need to take emergent (mis)behavior at system level more into account, rather than only focusing at service level.

At the end we have included a preliminary categorization of the techniques with respect to required system confidence level, maturity of the technique, and estimated increased complexity of the system by applying the technique.

Autonomic SOA is a promising key ingredient to support the much needed resilience of ever more complex systems with high dependability requirements. To fulfill this promise in practice, further work is needed to extend, understand, interrelate and validate the practical effectiveness and feasibility of the different concepts and techniques that underlie autonomic SOA.

In our current and future work we continue to explore possible techniques to support the resilience of complex systems.

Future work also includes developing and testing some of the ideas discussed in this paper and with further elaborations and validation. Ultimately we envision a set of specific guidelines that will help practitioners in selecting and applying relevant techniques, given a certain system and its particular complexity, characteristics and dependability requirements.

7. ACKNOWLEDGMENTS
This paper was written in the context of the ‘resilience of open architectures’ activity, part of the work package ‘architecture & integration’ of the GoICT research project [15] that is headed by DNV Research & Innovation and supported by the Norwegian Research Council by grant #183235/S10.

8. REFERENCES