Designing Survivable Services from Independent Components with Basic Functionality

Andreas Dittrich, Jon Kowal and Miroslaw Malek

Institut für Informatik
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin
{dittrich,malek}@informatik.hu-berlin.de
mail@jonkowal.de

Abstract: Automatic network service configuration techniques focus on the configuration of the attributes describing the different layers involved in service communication and treat service instances monolithically – they either exist in the network which means that they are fully usable or they do not. This approach does not work well in environments where resources are insufficiently dependable and the types of services used are not well known or standardized. This paper proposes a model to compose complex services from independent components with basic functionality that are organized as minimal services in the same service-oriented architecture. The approach promises to better handle run-time diagnostics and on-the-fly (re-)composition of service functionality in networks with highly dynamic capabilities.

Keywords: Survivability, Adaptivity, Composition, Service-oriented Architecture

1 Introduction

Information and communication technology has become ubiquitous in the last two decades. The cheaper production cost and increased networking capabilities allow mobile and embedded devices to pervade areas of computing that used to have fixed environments and help to make them more flexible and dynamic.

Because of the plethora of new devices entering traditional computer networks and the differences in their capabilities and non-functional properties such as availability and performance, there is the need for standards to automatically configure the communication layers that are outside of the semantics of the services provided by those devices. A service client or a server is only interested in the description of the actual service usage. For the sake of simplicity every other detail of network communication should be transparent. Quite a few frameworks have emerged that try to define the different layers involved in service communication and offer protocols to configure the layers in decentralized, dynamic networks with minimum administrative control. These frameworks support and extend what is commonly known as a service-oriented architecture (SOA).
For the purpose of connecting cheap and resource restricted systems the technologies used for auto-configuration need to scale. Only minimal overhead over the requirements for taking part in the network is acceptable. Full-blown SOAs use web services or other elaborate high-level concepts to compose complex business processes. Their use is not feasible in the environments aimed for in this paper. It has been shown in [DK08] that the requirements proposed by the Zeroconf working group in [Wil02] and their implementation described in [CS05] provide a good compromise between functionality and hardware requirements and can be implemented with very little overhead.

Zeroconf essentially takes care of IP interface configuration, the translation between host names and IP addresses and the discovery of services. It also provides basic service description. This is done using the well-standardised concepts of AutoIP, multicast DNS and DNS-Based Service Discovery and thus only depends on the existence of Multicast IP and User Datagram Protocol (UDP). Using the proven and ubiquitous Domain Name System (DNS) for name resolution, service discovery and service description saves resources as well. Following the findings of [DK08] it can generally be said that if a device is able to be a part of an IP network it is also able to be a part of a zeroconf network. This makes zeroconf an ideal framework for realizing service-oriented architectures in which participating nodes might be equipped with limited memory and processing power.

2 Problem statement

Zeroconf focuses mainly on the automatic configuration of the attributes that describe the different layers involved in IP based service communication. Service instances are treated monolithically – they either exist in the network, which means they are fully usable, or they do not. Their functionality in all its possible complexity is seen as a whole. This works well in environments where services are sufficiently dependable and the types of services used are well known and standardized.

But by using this approach it is difficult to quantify non-functional attributes of services that are of key importance in modern dynamic service networks like dependability, fault proneness, fault tolerance and resilience – all of them having impact on the survivability of a certain service. Guaranteeing redundancy is expensive if the failure of one functional component of a service equals the failure of the service as a whole and the replacement of those complex service instances can prove to be non-trivial. Even if the functionality and interfaces of the replacing instance are formally identical, in praxis new properties might emerge that provoke unexpected and possibly unwanted behaviour during service usage. The likelihood of side effects increases with the complexity of the services. It is also difficult to monitor the state of service instances: Zeroconf provides pretty good mechanisms to enumerate service instances, thus monitoring the availability of a service at run-time. However if those mechanisms are only used on complex services seen as single units a lot of their potential remains unused.
We believe that if a complex service can be described by a minimal composition of components with required functionality, those smaller components are easier to monitor, maintain and replace in case of failure. If the functionality is basic enough they might even be replaced by components that are used by different complex services but belong to the same service class (they provide the same functionality with identical interfaces). Given \( m \) complex services realized by \( n \) components with basic functionality, can one determine the maximum number \( k \) of components that may fail while all \( m \) services can still be provided? How many components may fail so that a certain service can still be provided and how can this number be maximized?

Models are needed to extract the different functionalities of complex services and to quantify their impact on the service functionality as a whole. This might lead to more dependable services that are cheaper to maintain and – using the principles of a service-oriented architecture [MP01] – can sustain their functionality longer without manual interaction. The resulting service-oriented architecture can be called a *survivability-oriented architecture*. As in web service composition [MM04], the appropriate methods for component composition should be used, however the orchestration should be largely automatic.

### 3 Proposed solution

In service-oriented architectures, sophisticated business processes are typically composed of services that implement functionalities most humans would recognize as a service, such as filling out an online application for an account or placing a booking order. We claim that most of these complex services can be divided into disjunctive, loosely coupled components with minimized functionality. Those components – being instances of simpler services – can be modelled as regular service instances in a service-oriented architecture, e.g. Zeroconf\(^1\). They are by all means services in their own right and can be used outside the proposed model, automatically configuring their network parameters and propagating their availability to the other nodes and thus benefiting from the same techniques as the complex service instance they provide. The model of such a complex service is as follows:

\[
S = \{s_0, \{s_1, s_2, \ldots, s_n\}, C\}
\]

\(S\) is defined as the functionality of the complex service we want to describe while \(s_i\) being the functionality of the simple component \(i\). \(C\) contains the composition rules for all the \(s_i\) that make up the service. The special component \(s_0\) provides interfaces for the complex service \(S\) to clients and can be seen as a manager service. As long as a service instance of \(s_0\) is able to discover instances for all the \(s_i\) in the network it can compose them via the rules \(C\), so \(S\) is usable to service clients.

\(^1\) Zeroconf fits the requirements for building a resource saving SOA, but other technologies could be used, as long as they scale to systems that are just able to uses IP networks.
The simple components have a small set of functionalities and vary in their dependence on the context of the service they operate in. The context sensitivity is crucial to determine the interchangeability of a certain component. If the functionality of a component provides is basic enough and if it is independent of the service context it operates in, a component can be used to build many different complex services. We call this characteristic service-agnostic because the component itself operates the same way independent of the complex service that leverages its functionality – it does not even need to “know” the service that uses it, stretching the SOA principle of service reusability. In contrast, it can be said that if a component with basic functionality is in any degree dependent on the context of the service it operates in, it cannot be trivially replaced by components of the same functionality. So we extend the description of complex service functionality in (1) as follows:

\[ S_j = \{\{s_{j1}, s_{j2}, \ldots, s_{jn}\}, t_j, \{t_1, t_2, \ldots, t_n\}, C_j\} \] (2)

The \( s_{ji} \) now describe the simple components that are context-sensitive, \( t_i \) are the service-agnostic components. Both of them compose the complex service functionality \( S_j \) by the rules defined in \( C_j \). The special service-agnostic component \( t_j \) is the manager service providing the interfaces to \( S_j \). Service-agnostic components provide several key-advantages:

1. They can easily be replaced in case of failure as their basic functionality is only bound to their existence in the network. This makes redundancy trivial.
2. By their very nature of providing independent abstract functionality they can be used by many services at the same time.

Thus, as long as there is at least one service-agnostic instance of a special type of service left, all the complex services relying on the functionality provided by that instance will remain functional. On the other hand, it is also possible to introduce new services to an already existing service network without deploying additional resources: New services could be composed only from service-agnostic components. This is illustrated in the following example:

\[ S_a = \{\{s_{a1}, s_{a2}\}, t_a, \{t_1, t_2\}, C_a\} \]
\[ S_b = \{\{s_{b1}\}, t_b, \{t_1, t_3, t_4\}, C_b\} \]
\[ S_c = \{\emptyset, t_c, \{t_1, t_2, t_4\}, C_c\} \] (3)

As long as we have at least one instance for the service-agnostic functionality \( t_i \) all three services can remain functional. If \( s_{ji} \) fails however, \( S_j \) cannot maintain its functionality. Service \( S_a \) on the other hand can be seen as a new service. In an already working service network that provides \( S_a \) and \( S_b \), \( S_c \) can be deployed simply by introducing its managing service with an additional need for resources only for the sake of redundancy. \( S_a \) and \( S_b \) already provide all of its basic functionalities.
The simple example in (3) shows the advantages of service-agnostic components: While the minimum number of instances of any context-sensitive component $s_j$ necessary to provide all services in the network equals the number of $s_j$ needing it, the minimum amount of instances of every service-agnostic component $t_i$ is always one. Thinking of redundancy, if $n$ instances of the service-agnostic component $t_i$ exist, $(n-1)$ may fail while leaving all services intact. In case of any context-sensitive component $s_j$, however this just guarantees the availability of $S_j$. So providing redundancy of service-agnostic components helps every service using them while in the case of context-sensitive components this is only true for single services.

Since the point is to make the component functionality as basic as possible – so that its availability is bound to its existence in the network – the manager service $s_0$ gains self-awareness simply by enumerating instances of the component services. Common service discovery mechanisms handle instance enumeration effectively, especially DNS-SD which is used in zeroconf. In a more sophisticated approach, the rules $C_j$ could contain alternative ways to compose the functionality $S_j$ so $s_0$ has the possibility to recompose it when the last instance of a service class drops away.

These thoughts lead to the conclusion that complex services that are composed of a higher degree of service-agnostic components can ultimately be more survivable since their redundancy is distributed across the whole service network and the replacement of failing functionality is trivial. It should be encouraged to maximize the percentage of independent, service-agnostic functionality. However it has to be investigated to what degree complex services can be modelled with service-agnostic components and how relevant it is in practice. This approach is also feasible for other non-functional attributes of services than redundancy but the benefits for different attributes such as availability may vary.

4 Examples

Thinking of an IP-based, service-oriented network this model can be applied to various service classes, all of them being propagated and discovered with zeroconf techniques. Let us look an application that reads the temperature from sensors in a room and – after analyzing the data – regulates the heating in that room. Using the proposed approach, this application can be composed of four services: temperature sensors, heating regulators, calculation of the adjustment and presentation of the front-end. While obviously the heating regulators are context-sensitive regarding their location the temperature sensors do not need to be, in general it is enough to know the temperature somewhere in the room. Also the calculations can be carried out anywhere where there is enough computation power and the front-end reading the sensors and setting the controls can be served by any service instance providing the presentation service. So if the manager service, possibly encapsulated in the front-end, realizes that the last used calculation service is gone it can check the zeroconf network for other instances of that service and, if available, switch to using another one, thus providing run-time error correction.
The sensor service could also be seen as an example of exchanging context-sensitive components with service-agnostic ones. If an application uses an expensive, high precision sensor that was calibrated in a special location it is quite a complex task to replace it once it fails. However one could think of applications where the same precision could be achieved by averaging the measurements of many less precise sensors that are somewhere around that specific location. In addition these sensors are probably a lot cheaper than the high precision one and can be deployed with less effort.

5 Conclusion

The proposed model provides a concept for survivable service architectures but is better classified as a Service Component Architecture [SCA1]. Although this paper focuses on redundancy, the fine granularity of the service description and distribution of their management among simpler, ideally service-agnostic nodes across the whole service network, is believed to improve other non-functional parameters of a complex services as well. While providing a certain overhead in the communication between the components, the approach promises to better handle run-time diagnostics and on-the-fly (re-)composition of service functionality in networks with highly dynamic capabilities. The overhead should not be critical as the communication is done within the already existing service-oriented architecture. However this claim has to be verified in practice. Furthermore, the variability of the service description provides more possibilities for composing service functionalities. If a vital part of the service breaks down, it might – depending on the application – be recomposed with reduced functionality thus facilitating graceful degradation which is in most cases more desirable than no service functionality at all. How the proposed approach handles those topics needs to be examined in further studies.

References