Run-time Reconfiguration of Service-Centric Systems

Paris Avgeriou
Department of Mathematics and Computing Science
University of Groningen, the Netherlands
paris@cs.rug.nl

Abstract

Service-centric systems are driven more and more towards self-adaptation in order to satisfy QoS in highly dynamic environments. However, the young age and immaturity of this domain, combined with the increasing size and complexity of these systems, hinder the architects from designing effectively self-adaptive systems. This paper addresses the process of run-time reconfiguration with respect to high level issues such as monitoring, evaluation of QoS, reconfiguring and structuring the service-centric system. All patterns relate their solutions partially to well-established architectural patterns, adapted to the domain of service-centric systems. The aim is to compose a unified framework in the form of a pattern language that will help architects in taking the major design decisions.

1 Introduction

One of the most promising emerging trends in software engineering is that of service-centric systems [5,15,16]: systems consisting of multiple services, possibly from different service providers, working together to perform some functionality. The key notion in this paradigm is that of a *service composition* [2], the process of synthesizing several services into a single composite one, which satisfies a set of functional and quality requirements. The greatest advantage that service-centric computing advocates, is runtime reconfiguration[1]: services are located, bound, and executed at runtime using standard protocols such as UDDI, WSDL, and SOAP [25]. Because services are loosely-coupled and have an explicit interface, this paradigm facilitates the integration of third-party services, and the substitution of one service for another at runtime.

In fact, runtime reconfiguration is better said than done, since highly demanding QoS requirements must be satisfied in constantly changing dynamic contexts. On the one hand the QoS that the service-centric system has to deliver is often formalized in a Service Level Agreement (SLA), i.e. the QoS requirements are legally and financially binding. On the other hand, most service-centric systems operate in highly dynamic environments: they depend on third-party services which are out of their control; user demands may change arbitrarily (e.g. due to change of the physical location of a mobile device); networks may present congestions or other unpredictable behavior; local nodes that execute services have variable workloads

[1] Run-time reconfiguration is not new but has already been supported in component-based systems and earlier. In fact, service-centric systems are considered as an extension of object-oriented and component-based systems. The differences and similarities between these three categories of systems and their respective paradigms are a subject of great debate and out of the scope of this paper, which merely focuses on services.

etc. Therefore the service configuration must be adaptable at all times, so that the systems can meet the constant changes of the dynamic environment and thus deliver the QoS, they have committed themselves to. This problem is exceptionally difficult from the architectural point of view as it involves balancing quality attributes at runtime.

In theory, runtime adaptation of a service-centric system can be initiated by the service users, by the service providers, or by the system itself. In practice however, the first two parties can not realistically perform successful adaptation. First, the majority of the service users are interested only in the result, i.e. the offered functionality and not the underlying service composition. Second, the service providers cannot manually perform the runtime adaptation themselves, since the potential increase of service users would pose serious scalability problems. Therefore the only viable solution for runtime adaptation lies on the service-centric system itself as a form of *self-adaptation:* the system can evaluate its current situation and if necessary adapt itself by reconfiguring the service composition to better meet its QoS requirements.

Unfortunately there are no standard approaches for creating such self-adaptive service-centric systems. Current standardization approaches mainly focus on fundamental implementation-level web services technology, such as message exchange protocols and service description and orchestration languages. Moreover current approaches have mainly dealt with static service composition and not dynamic, runtime composition. Therefore architects of service-centric systems usually provide domain-specific, piecemeal, ad-hoc solutions that focus mostly on the implementation level. This paper attempts to provide some architectural design guidance in this domain in the form of patterns. It contains four patterns for performing the process of run-time reconfiguration in the domain of service-centric systems. The reconfiguration process has the form of a feedback loop: monitoring the system, evaluating the QoS, reconfiguring and structuring the services. The solutions of these four process patterns are partially achieved through the employment of well-established architectural patterns. Also, great emphasis is put on establishing the relations between the patterns as to illustrate as complete a picture as possible of the solution space in this domain. This pattern language does not deal with lower level issues such as service description, discovery, selection and composition. Finally, the paper pinpoints future directions in the form of future patterns, for significant areas that should be explored such as variability and transaction management.

2 The Patterns

Figure 1 depicts an overview of the patterns as well as the main relationships between them and other related architectural patterns. The rest of the relationships as well as names of the shown relationships have been suppressed in order not to clutter the diagram.

The patterns attempt to tackle the complex problem of dynamic reconfiguration of service-centric systems, focusing on the higher application-general layers. The *intended audience* for these patterns is software architects that are designing systems in this domain. The patterns do not imply a waterfall-like, up-front architecture design approach, but rather support iterative and incremental approaches, where the patterns are applied iteratively in cycles, each time refining and fine-tuning.

The process patterns tackle four major issues in the dynamic reconfiguration of service-centric systems:

- MONITOR THE SYSTEM, specifies what needs to be monitored in order to ascertain the status of the service-centric system.

- EVALUATE QOS, focuses on analyzing the monitoring information combined with the service configuration to infer the QoS values of the running system.
- RECONFIGURE THE SERVICES AT RUNTIME, takes care of adapting the system at runtime in order to satisfy its QoS requirements at all times.
- STRUCTURE THE SERVICES, deals with how the services should be actually structured, for the reconfiguration to take place.

The first three patterns form the feedback loop of the reconfiguration process: the monitoring information is used as input to the QoS evaluation, which in turns is used to reconfigure the system if necessary. The fourth pattern is more horizontal as it deals with the general structure of the services, another valuable input to the reconfiguration process.

The following patterns have been referenced in this paper from third-party sources:

- ACID TRANSACTION [7], ensures that a transaction will never have any unexpected or inconsistent outcome, i.e. the transaction will have the ACID properties: atomicity, consistency, isolation and durability.
- OPTIMISTIC TRANSACTION [9], allows for concurrent accesses to happen but detects them and repairs conflicts.
- TWO-WAY LOCKING [9], does not allow for concurrent access by locking the resources, while on the same time trying to avoid deadlocks.

The following patterns complete the big picture in this domain and are not completed yet but are considered as future work:

- VARIABILITY MODEL, defines explicitly all the possible variation points in a configuration, their constraints, as well as all possible variants of those variation points and their corresponding QoS values.
- VARIANT CONFIGURATION, considers a service configuration as a choice of specific variants for all the variation points of the VARIABILITY MODEL
- UPDATE THE VARIANTS, makes sure that when specific variants are chosen as part of the VARIANT CONFIGURATION, their QoS values in the VARIABILITY MODEL will be updated according to the actual values derived from EVALUATING THE QOS.
- SECURE CONFIGURATION, make a configuration of services inaccessible to third parties that are unauthorized or not trusted.

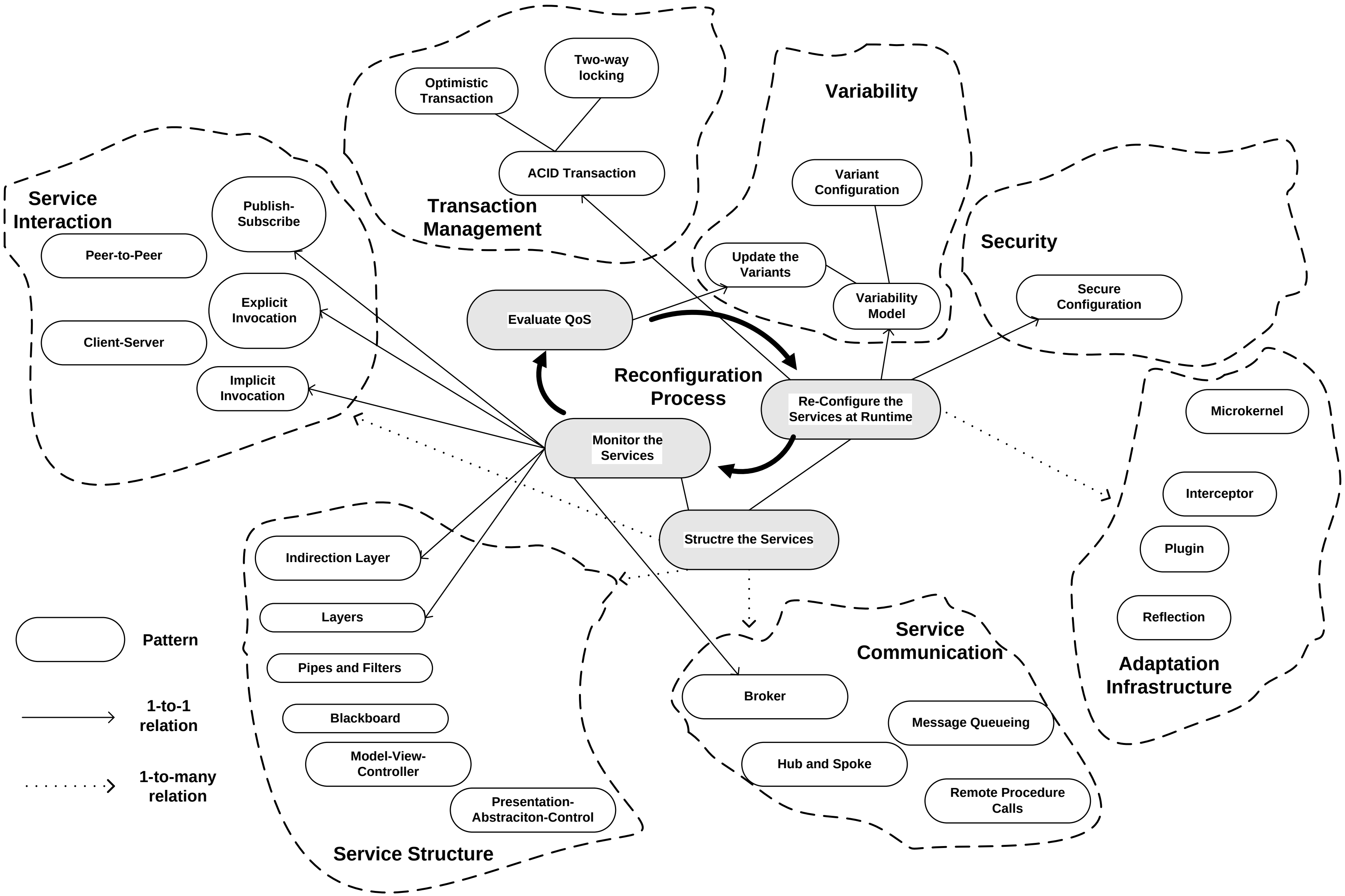

Figure 1 - An overview of patterns and their relations for the domain of Adaptive Service-Centric Systems

2.1 MONITOR THE SYSTEM

Context You have a service-centric system at hand, running on a highly dynamic environment and you need to enforce some sort of a feedback loop into the system.

Problem **How do you examine a highly dynamic Service-Centric System in order to understand its current status with respect to the QoS requirements?**

Forces

- A Service-Centric System needs to be looked upon from different viewpoints and different levels to get the full picture.
- Monitoring a Service-Centric System is not a one-time task. As the system is dynamic, its QoS is bound to change during run-time thus requiring continuous monitoring serving a constant feedback loop.
- In a complex Service-Centric System, comprised of multiple aggregated services, it can be overwhelmingly difficult to pinpoint what exactly needs to be monitored and how exactly this monitoring can be performed.
- There is always an associated cost with monitoring a Service-Centric system, as it implies an interference with its internal workings, probing its inputs and outputs. Such interference may pose additional overhead to the system's non-functional requirements and especially performance.

Solution **Therefore: Monitor the service-centric system continuously from three different aspects: the individual services, the execution environment of the system, and the context of the application user.**

The type of the information that must be monitored depends on the application domain and QoS requirements. However the monitoring information itself can invariably be classified in three categories: a) information about the services per se (e.g. response times, failure rates, and exceptions thrown), b) information about the execution environment (network bandwidth, runtime platform, processor load) and c) information about the application users (user device GPS coordinates).

The collection of the monitoring information with respect to the individual services can be achieved by intercepting and subsequently inspecting system services. It is usually enough to inspect merely the inputs and outputs of the services, because most third-party services are "black boxes" but also for reasons of performance. The LAYERS architectural pattern provides a good solution by intermediating a layer capturing service requests and responses. A relevant, more specialized pattern is also the INDIRECTION LAYER [1] that hides the details of services by acting as an ADAPTER or a FAÇADE, first inspecting and subsequently forwarding invocations of services and responses. Finally, if a BROKER [3] is used for the interaction between the services, then it can also be used as an observing mechanism by logging and storing the service interaction information.

The execution environment of the system is usually defined by the network (servers, routers, topology, bandwidth etc.), the various distributed nodes that execute the services locally and also the central platform of the service-centric system itself. The reason for monitoring the execution environment and not only the services themselves and their compositions, is that the execution environment may also provide critical information for evaluating the QoS. For example if a service that is executed on server X, showed very good performance in the beginning and then it slowed down, violating the SLA, we need to know the cause of the problem. If there was temporary network congestion we can still use the same service from server X by tolerating the passing congestion or requesting a different route. If however the problem lies in the server throughput, then we should request this service or a similar service from other servers.

On the other hand, the context of the application user is typically defined in a *user profile* in the client device (e.g. a mobile phone). The user profile may, for instance, contain user goals and preferences, usage data, user stereotype, demographic data etc. [24]. The user profile needs to be monitored as it changes over time and subsequently affects the QoS requirements, e.g. when the user‘s physical location changes or when the user changes her preferences.

The most efficient and least intrusive way to receive this monitoring information by the execution environment or the application user, is by an IMPLICIT INVOCATION [1] mechanism such as PUBLISH-SUBSCRIBE [3]: events generated by both the environment and the user, are sent to the corresponding subscribers-monitors. Alternatively EXPLICIT INVOCATION [1] can also be used by having the monitors probing the system resources at regular time intervals.

Example Consider an e-learning service-centric system where the service user is a student that needs to follow an e-course on mathematics. The system is comprised of a service that provides multimedia reading material, a second service that offers video conferencing and application sharing with a tutor and a third service that offers on-line assessment. The monitoring information in this case focuses on the second service as it may be a potential performance bottleneck for the system. The monitor can measure the bit rate as well as the resolution of the video stream sent to the user and the response time of the shared application.

The application of the pattern entails the following positive and negative consequences:

Benefits

- All relevant information can be monitored in order to have the complete and correct status of the running system.
- The different types of monitoring information are separated and thus give a clear picture of the different system aspects.
- Monitoring the system can sometimes be weaved into normal program flow, e.g. by using INTERCEPTORS or BROKERS.

Liabilities

- Intercepting the services may prove to be extremely cumbersome since it disrupts the normal application program flow.
- Monitoring arbitrary QoS attributes is not always an easy task, as some attributes are not measurable or quantitative or merely easy to pinpoint.
- Monitoring information is too basic to actually derive QoS attributes. Further processing needs to take place while EVALUATING THE QOS.

2.2 EVALUATE QOS

Context You have MONITORED THE SYSTEM, and thus acquired and stored data about the system status.

Problem **The QoS of the running system can not be evaluated merely by examining the status of the system.**

Forces

- The actual QoS that the service-centric system delivers during runtime is variable. The system must satisfy its QoS requirements at all times.
- Monitoring information cannot usually be directly related to QoS requirements. The former concerns various low-level details of the state of the running system. The latter is more high level and concerns generic non-functional properties of the system.

Solution **Therefore: Analyze the monitoring information in order to deduce QoS information that can be evaluated with respect to the fixed QoS requirements. This analysis can be done by combining the monitoring information (i.e. information from individual services placed in the context of the execution environment and the application user) with the service configuration.**

The analysis of the Monitoring information can take three different paths, ranging from simple to more complex:

1. If the QoS can be directly monitored, then the monitoring information equals the QoS information and can be directly compared to the QoS requirements. This is usually the case for simple service configurations and quantitative QoS requirements.
2. If the QoS cannot be directly monitored and the monitoring information contains contextual information about the execution environment and the application user, then the monitoring information can be transformed into QoS information by combining it with the current runtime service configuration.
3. If the QoS cannot be directly monitored and the monitoring information does not contain contextual information about the execution environment and application user, then the evaluation of this contextual information is also necessary. After the context is established, as in the second path, the monitoring information must be combined with the current runtime service configuration, in order to derive QoS information.

The runtime service configuration at any given time follows a specific structure, according to which services interact with each other. There are several architectural patterns that can be used to STRUCTURE THE SERVICES, according to the requirements and the Service Level Agreement at hand. The runtime service configuration can be modelled according to the architectural patterns, and thus allow for evaluation of complex quality attributes. For example the availability of a PIPES AND FILTERS system depends on the availability on the ‘weakest link’ rather than the sum of the availabilities of the different pipes and filters.

The context of the service-centric system, i.e. the execution environment and the application user are usually specified in terms of a context model, based on a table or ontology, by filling in the values according to the monitoring information. Naturally, an ontology is more powerful than a table in terms of expressiveness and inference abilities. On the other hand, the QoS information can be calculated according to individual metrics. Such metrics vary according to the individual quality attributes that are measured (e.g. performance, availability, reliability), the application domain and the system per se.

Once the QoS information is established, it can be weighed against the QoS requirements, in order to determine “where the application is at”. For each QoS requirement an individual decision may be taken with a quantitative evaluation of how well it is satisfied in the current application. Qualitative evaluation is also possible but it will lead to more uncertainty and risk when the system RECONFIGURES THE SERVICES.

If the evaluated QoS does not satisfy the Service-Level Agreement, it can be used as input in the next step of this process which is RECONFIGURING THE SERVICES AT RUNTIME. A major problem that arises after the evaluation of each quality attribute is that they cannot all be optimally satisfied in the system, and thus the architect needs to perform some kind of trade-off analysis: which quality attributes to support and which ones to compromise. This is a cumbersome task to perform during run-time as the well-known architecture evaluation methods, e.g. Architecture Trade-Off Analysis Method (ATAM) [12], Cost-Benefit Analysis Method (CBAM) [13], are design-time methods. The architect needs to set domain-specific trade-off rules in place that are based on the interactions among quality attributes and the STRUCTURE OF THE SERVICES. The trade-off decisions must always be taken in accordance to the Service-Level Agreement, in order to ensure that compromised quality attributes do not cause a breach in the latter.

Example In the same e-learning application as in the previous pattern, consider that the shared application is a whiteboard that both the student and the tutor can draw on. The QoS requirement is that the synchronization of the shared whiteboard in the two screens should not exceed 0.5 sec. Our system thus periodically checks the monitoring information, in specific the response time, based on the timestamp of the packets and expresses the time delay for each package as a QoS value.

The application of the pattern entails the following positive and negative consequences:

Benefits

- The monitoring information is translated into concrete QoS information that can be evaluated with respect to the requirements of the Service-Level Agreement. A context model of the system and user is also formulated during the process.
- Separating the analysis of the monitoring information and the evaluation with respect to the QoS requirements decouples the corresponding components that will implement these activities, thus increasing flexibility.

Liabilities

- The context model may become overly too complex resulting in miscalculated QoS information.
- Calculating QoS information from monitoring data brings additional overhead to the system responsiveness to QoS failure.
- Designing the ontology for the context model of system and user is usually done by experiments on the domain and is prone to errors.

2.3 RECONFIGURE THE SERVICES AT RUNTIME

Context You have EVALUATED THE QOS of a service-centric system and you need to make the system adaptive in case it fails to satisfy some of the QoS attributes.

Problem **The service-centric system may satisfy its QoS requirements at design-time, but due to its dynamic nature it may eventually fall short in specific QoS requirements during runtime. This will lead to a breach of the Service-Level Agreement (SLA) between service providers and service consumers.**

Forces

- The QoS requirements are specified in a Service-Level Agreement which may be legally binding for the Service Provider.
- The system needs to satisfy the QoS requirements not only at design time but most importantly at run-time.
- The service-centric system is dynamic in its nature due to e.g. failure of third-party services, network problems, adjustable user demands or variable node workloads. Another aspect of this dynamic nature is that some QoS requirements may change during run-time or they may not be known until the run-time.
- Due to the dynamic nature of service-centric systems, their QoS may deteriorate during run-time, probably also violating the SLA.

Solution **Therefore: Provide a special infrastructure or mechanisms in the system architecture that allow for runtime self-reconfiguration of the service-centric system in order to resume a state where QoS requirements are satisfied.**

The reconfiguration should take place, at any given time, by choosing one among alternative configurations and enacting that configuration in the system at run-time. The reconfiguration will not be performed by the service user or the service provider but by the service-centric system itself, since it is the one that MONITORS THE SYSTEM and EVALUATES THE QOS and thus knows when, what and how must be reconfigured [21].

A service-centric system can perform the run-time reconfiguration with the help of a special *adaptation infrastructure*, by separating the main core of the application from the parts that change in specific variation points. The adaptation infrastructure is part of the run-time platform of a service-centric system and it can be designed according to a few architectural patterns from the Adaptation Infrastructure View [1]. A first option is to use a MICROKERNEL [3], which realizes invariant core services, as well as a way to plug in variant internal services. Another option is to use REFLECTION [3], which hides all structural and behavioural aspects into modifiable meta-objects, and thus separates them from application-logic services. Furthermore a third solution can be an INTERCEPTOR [17] infrastructure, where services can be updated or new services can be added, by registering any number of interceptors that implement those services. Finally a PLUGIN [6] infrastructure can solve the problem of runtime reconfiguration by providing a centralized point of plugging in new or updated services without rebuilding or redeployment.

An alternative or complementary to having a special adaptation infrastructure is to change the STRUCTURE OF THE SERVICES itself at runtime. Of course, modifying the system architecture at runtime may impose severe overhead and discontinuation of the normal system functionality. Also, restoring a failed QoS to the values set by the SLA, usually requires smaller-scale modifications rather than a substantial re-engineering of the system. But there are cases where architectural patterns of similar scope can be used as alternatives, for example Client Server and Peer to Peer, or Shared Repository and Active Repository.

In order to change the service configuration, knowledge about what can change is required. This can be achieved by creating and maintaining a Variability Model of the system [22], where the variation points and all possible variants of those variation points are specifically documented. In these terms a service configuration is a Variant Configuration where a set of specific variants are chosen for all the different variation points. Therefore the set of variation points and their variants is sufficient to formulate all possible service configurations and estimated QoS for each configuration. Each variant of a variation point is also associated with specific QoS values in order to be able to choose the appropriate variants that will satisfy the QoS for the whole system. While the variation points are mainly fixed at design time (with few exceptions), the variants can be specified in both design time and the run time. Constant Updating of the Variants after Evaluating The QoS is of paramount importance in order to ensure that the QoS values of each variant in the Variability Model are up to date. Finally, the possible configurations must be able to be validated before they can be enacted in the running system. The correctness of a specific configuration can be defined in terms of presence of deadlocks, or variability constraints.

Another important aspect of service composition reconfiguration, is the high-level strategy that can be applied. In specific, there are four generic strategies that architects can choose from: a) they can continuously try to choose the optimal configuration, reacting to each evaluation of the system state with respect to the QoS requirements; b), they can perform the reconfiguration proactively, when there are signs or threats of potential QoS failure; c) they can perform reconfiguration as a remedy, by instantly reacting when QoS has indeed failed; d) they can act as in c) but delay the reconfiguration for a short while, in case the QoS is restored rapidly.

The transition from the existing configuration to the new one may be implemented by substituting a bound service for an alternative service or by changing the structure and the flow of the service composition. The enactment of the configuration itself is achieved, e.g. by deploying a new orchestration in the BPEL engine of the application system or by reconfiguring a service proxy.

Example In the same e-learning application as in the previous patterns, consider that the shared whiteboard is not synchronized well leading to a confusing interaction between the student and the tutor. A variability point of this system is the communication mechanism between the services. The configurator chooses to override the Broker that was previously used with Remote Procedure Calls in order to improve the performance by coupling the student's component with the tutor service. Before this reconfiguration can take place the system's transaction manager ensures that the state of the whiteboard (i.e. what has been drawn) is consistent between the two remote components.

The application of the pattern entails the following positive and negative consequences:

Benefits

- The Service-Centric System is reconfigurable at runtime, following the changes of the dynamic system context.

- A feedback loop is used to ensure that the system does not fall short on the QoS requirements. The Service-Level Agreement will not be compromised.

Liabilities

- The verification of a chosen configuration is bound to the specific verification technique used. It may prove that it will not meet the QoS requirements, just like the previous configuration.
- A reconfiguration that modifies a number of services may end up having services that contradict each other with respect to the desired QoS requirements.
- Having a service reconfiguration mechanism may bring severe overhead to the whole service-centric system, as it may become intrusive to the normal workflow of the system, both in the client side and within the various services.
- The services in a service composition may have numerous and complex dependencies between each other, that can not easily be modeled in the variability model.
- The variability model may become too complex thus causing problems in specifying effective service compositions.
- There are several alternative service configurations that have a different impact on the QoS. Making an architectural evaluation, possibly including a trade-off analysis, for all the configurations is very resource-consuming.

2.4 STRUCTURE THE SERVICES

Context You are architecting a service-centric system, that needs to run on a highly dynamic environment and at the same time satisfy hard QoS requirements.

Problem **The services need to be structured in such a way that they provide the required functionality and satisfy the QoS.**

Forces

- The QoS requirements are specified in a Service-Level Agreement which is legally binding for the Service Provider.
- Reusing architectural design experience is always essential in software development, saving time and money and preventing from re-inventing the wheel. There are a number of architectural patterns that solve recurrent problems in architectural design and can be reused in different contexts.
- Several issues need to be resolved by the service structure, and especially distribution, communication, message exchange, security and transaction management.

Solution **Therefore: Structure the services by addressing different areas of concern, with the following being a standard minimum: the overall partitioning and control mechanism, the message exchange, the distributed communication, the transaction management and the security**.

A service composition structured in LAYERS [3], is horizontally partitioned into interacting parts, that perform a specific level of functionality and still remain decoupled from each other. In a service-centric system, there should exist a control mechanism that maintains an overall organization scheme by orchestrating the various services. In case of data flowing through different processing nodes, PIPES AND FILTERS [3] are more appropriate (or variants such as BATCH SEQUENTIAL[20]): different services connected in order to successively process streams of data. On the other hand, if the system is heavily data-centric, or does not follow a deterministic processing or interaction model, a BLACKBOARD [3]] can be used (or its variants such as SHARED REPOSITORY [1] or ACTIVE REPOSITORY [1]): one of the services acts as central repository of data, that can be accessed by the rest of the services. The service acting as the data store may be active or passive, and it is possible that there are more than one data stores. Service-centric systems that are highly interactive with the users may benefit from a MODEL-VIEW-CONTROLLER (MVC) [3] or a PRESENTATION-ABSTRACTION-CONTROL (PAC) [3] structure, where the user interface logic is decoupled from application logic and data. In MVC terms, there may be multiple services as Views, presenting data to the user, multiple services as Controllers of the Views, and a service acting as the Model. In PAC terms, the services may be structured into a tree-like hierarchy, where each level of the tree aggregates functionality from the lower level.

The configuration of the services needs to determine the mechanism for message exchange by the independent services in order to satisfy the QoS requirements. Several architectural patterns from the Component Interaction View [1] can be used to define how the services interact by exchanging messages and thus deliver specific QoS. By using EXPLICIT INVOCATION (called “Communicating Processes” in [4,20]) the services are tightly coupled in various aspects (e.g. network location, service name and parameters) in order to improve performance or have an immutable topology or force a client always to initiate the invocation etc. CLIENT-SERVER [4] and PEER-TO-PEER [4] are two patterns that use mostly EXPLICIT INVOCATIONS: in the former a distinction is made between producers and consumer services, while in the latter all services may play both roles. On the other hand, IMPLICIT INVOCATION [19, 20] provides temporal, spatial, functional or other forms of decoupling between the services. In this case performance can be compromised in order to increase adaptivity, flexibility, availability, interoperability etc. PUBLISH-SUBSCRIBE [4] is a pattern heavily based on IMPLICIT INVOCATIONS: it allows service consumers to register for specific events and event producers to publish specific events that reach a specified number of consumers. It thus support flexibility, locality of changes and optimized performance of the subscribers.

Another parameter in service structuring and also that can be modified during service reconfiguration is the communication of the distributed services. There exist several architectural patterns from the Distributed Communication View [1], that determine the QoS and specifically performance, interoperability, scalability, location transparency etc. REMOTE PROCEDURE CALLS [23] extend the well-known procedure call abstraction to distributed systems (remote services are invoked as if they were local) and ensure fast performance. Furthermore, the BROKER pattern [3], or its specialization, the HUB AND SPOKE pattern [15] is a typical way to separate the communication mechanism from the application logic in a distributed environment, by hiding and mediating all communication between the services, thus adding flexibility but compromising performance. Alternatively, MESSAGE QUEUING [23] may be used to decouple the consumer and producer services by implementing intermediate queues that store and forward messages appropriately. The availability and reliability of the system is increased but with a cost on performance.

A precondition to enacting the configuration is to ensure the safe management of **transactions**, i.e. to safeguard the system state. The starting point to tackle this problem is to enforce ACID Transactions [7] that preserve the classical ‘atomicity, consistency, isolation and durability’ properties. There are several strategies to follow when enacting a configuration, e.g. to finishing current transactions without accepting new ones, or to interrupting current transactions and perform rollbacks. Naturally classical transaction management techniques can be used in this case, such as Optimistic Transaction, Two-way Locking or Multiversion Two-way Locking [9].

An important aspect of the service configuration is to ensure Secure Configurations that cannot be accessed or tampered by unauthorized third parties. Numerous security patterns can be used to achieve this purpose, e.g. Single Access Point, Secure Access Layer [8, 18]. Finally the Interceptor architectural pattern is often used to handle cross-cutting issues, such as security and transaction management.

The reconfiguration of the Services may change any of the above mechanisms, in order to satisfy the QoS requirements. Also the Structure of the Services largely determines the service configuration, which is combined with the monitoring information during the Monitoring of the System.

Example In the same e-learning application as in the previous patterns, the student is connected to a number of services through the Hub and Spoke pattern: the hub is a central service that acts as a Broker between the student and the rest of the third party services. This is a specialization of the Broker pattern that extends it from bilateral to multi-lateral interactions. The student and the hub play the roles of a client and server respectively.

The application of the pattern entails the following positive and negative consequences:

Benefits

- The Service-Centric System is structured according to specific architectural patterns that solve problems pertinent to specific areas of concern.
- Architectural design experience in the form of architectural patterns is reused saving precious resources.
- A common language is established among stakeholders for the description of the system architecture using patterns.

Liabilities

- The combination of more than one architectural patterns makes it necessary to perform trade-offs in quality attributes as the latter are supported or opposed by the different patterns.
- The dynamic nature of service-centric systems entails a risk that the patterns chosen for the service configuration may become obsolete. Reconfiguring the services at runtime by changing the service structure may prove to be a cumbersome process.

3 Epilogue

In order for service-centric systems to be self-adaptive, they must be able to self-detect when and what to change and make this change autonomously. This ability includes several activities that take place at runtime: monitoring the system context, evaluating if the current QoS fulfils the QoS requirements, and reconfiguring the service composition when necessary. In essence this is a more general problem of balancing adaptivity and flexibility on one side and keeping the rest of the quality attributes within acceptable boundaries on the other side. The reconfiguration process patterns presented here build upon several existing and well-established architectural patterns, setting them in the domain of service-centric systems and correlating them. Even though, this paper is focused on the domain of Service-centric systems, these patterns could be applied in the more general context of reconfigurable distributed systems. In the future, this pattern language is planned to be completed by customizing architectural patterns in this domain, and also by adding new patterns to tackle security and transaction management.

Acknowledgements

This research has been partially sponsored by the European Union funded project SeCSE (Service-Centric System Engineering) under contract no. IST-511680. I would like to thank Dieter Hammer, Ivor Bosloper, Johanneke Siljee, and Jos Nijhuis for providing valuable input on Dynamic Service-Centric Architectures. Last but certainly not least, many thanks to my shepherd Andy Longshaw, for his constructive feedback.

References

1. P. Avgeriou and U. Zdun. Architectural patterns revisited – a pattern language. In 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), Irsee, Germany, July 2005.
2. G. Alonso, F. Casati, H. Kuno, and V. Machiraju, Web Services - Concepts, Architectures and Applications, Springer Verlag, 2004.
3. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M., Pattern-Oriented Software Architecture, Volume 1: A System of Patterns, John Wiley & Sons, 1996.
4. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J., Documenting Software Architectures: Views and Beyond, Addison-Wesley, 2002.
5. J. Duggan, "Simplify your business with a SOA approach", Gartner, Report AV-18-6077, 2002.
6. M. Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002.
7. M. Grand. Java Enterprise Design Patterns: Patterns in Java Volume 3, John Wiley & Sons, 2001
8. Neil Harrison, Brian Foote, Hans Rohnert. Pattern Languages of Program Design 4, Addison-Wesley, 1999.
9. K. Henrdikx, E. Duval, H. Olivie. Managing Shared Resources. In 5th European Conference on Pattern Languages of Programs (EuroPlop 2000), Irsee, Germany, July 2000.

10. Hull, R., Christophides, V., Su, J., 2003. EServices: A Look Behind the Curtain. Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 9-12, 2003, San Diego, CA.
11. Kazman, R., Bass, L., Abowd, G., & Webb, M. "SAAM: A Method for Analyzing the Properties of Software Architectures", Proceedings of the 16th international conference on Software engineering (ICSE 1994), Sorrento, Italy, pp. 81 - 90
12. R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The Architecture Tradeoff Analysis Method," Proceedings of the Fourth International Conference on Engineering of Complex Computer Systems (ICECCS98), August, 1998.
13. Rick Kazman , Jai Asundi , Mark Klein, Quantifying the costs and benefits of architectural decisions, Proceedings of the 23rd International Conference on Software Engineering, p.297-306, May 12-19, 2001, Toronto, Ontario, Canada
14. Nico Lassing , Perlof Bengtsson , Hans van Vliet , Jan Bosch, Experiences with ALMA: architecture-level modifiability analysis, Journal of Systems and Software, v.61 n.1, p.47-57, March 2002
15. E. G. Nadhan, "Service-Oriented Architecture: Implementation Challenges", Microsoft Architects Journal, MSDN, vol. 2, April 2004.
16. R. Schmelzer, "Service-Oriented Process Foundation Report", ZapThink, Report ZTR-WS108, 2003.
17. D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. Patterns for Concurrent and Distributed Objects. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd., 2000.
18. M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, P. Sommerlad Security Patterns Integrating Security and Systems Engineering, J. Wiley and Sons Ltd., 2005.
19. Shaw, M., Garlan, D.: Software Architecture - Perspectives on an emerging discipline. Prentice Hall, 1996.
20. M. Shaw and P. Clements, A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems, Proceedings of the 21st International Computer Software and Applications Conference (COMPSAC), pp. 6 – 13, 1997
21. Johanneke Siljee, Ivor Bosloper, Jos Nijhuis, Dieter Hammer, DySOA: making Service Systems Self-Adaptive, The Third International Conference on Service Oriented Computing (ICSOC05), Amsterdam, The Netherlands, December 12-15, 2005.
22. M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch, "COVAMOF: A Framework for Modeling Variability in Software Product Families", The Third Software Product Line Conference (SPLC 2004), Boston, USA, 2004.
23. M. Voelter, M. Kircher, and U. Zdun. *Remoting Patterns*. Pattern Series. John Wiley and Sons, 2004.
24. D. Vogiatzis, A. Tzanavari, S. Retalis, P. Avgeriou and A. Papasalouros. The Learner’s Mirror - Designing a User Modelling Component in Adaptive Hypermedia Educational Systems. In 9th European Conference on Pattern Languages of Programs (EuroPlop 2004), Irsee, Germany, July 2004.
25. W3C, "SOAP Version 1.2", Recommendation 2003.