Architectural Mismatch in Service-Oriented Architectures

Kevin Bierhoff

Institute for Software Research, Carnegie Mellon University
Pittsburgh, PA 15213
kevin.bierhoff@cs.cmu.edu

Mark Grechanik and Edy S. Liongosari

Systems Integration Group, Accenture Technology Labs
Chicago, IL 60601
{mark.grechanik, edy.s.liongosari}@accenture.com

Abstract

Architectural mismatch results from implicit and conflicting assumptions that designers of components make about the environments in which these components should operate. While architectural mismatch was extensively studied in monolithic and distributed applications, it has not been applied to Service-Oriented Architectures (SOAs).

A major contribution of this paper is the analysis of how architectural mismatch affects SOAs. We study how implicit and conflicting assumptions that designers make about web services and their compositions affect the quality of resulting SOA-based systems. We support our analysis with empirical data that we collected from a large-scale SOA-based project within Accenture and other smaller projects.

1. Introduction

Architectural mismatch results from implicit and conflicting assumptions that designers of components make about the environment in which these components will operate [9]. Architectural mismatch impedes constructing applications from third-party reusable components that, on a superficial level, appear compatible. Even if components are written in the same programming language, run on the same platform, and are intended for reuse, software engineers can encounter significant problems in getting components to work together. Engineers may have to re-implement existing functionality, provide glue code to mediate between components, or even change component implementations in order to overcome mismatch. The resulting systems can be intolerably large and slow [9].

As a concrete example, Garlan et al observed a mismatch between message data models of two components they used in constructing an interactive modeling environment. One component assumed that messages would be passed as heap data structures while another component expected character strings. Even though this mismatch was discovered early, non-trivial message conversions were necessary to make the two components interoperable. The conversion routines represented a significant engineering effort and seriously affected the performance of the resulting system [9]. This paper will show that this and other problems of architectural mismatch are still relevant in developing Service-Oriented Architectures.

Service-Oriented Architectures (SOAs) define how software components called *services* are organized into structures to support business requirements [12]. Web services are software components that exchange information (i.e., interoperate [2]) in heterogenous environments including the Internet. They currently gain widespread acceptance partly because of the business demand for applications to exchange information [8]. SOAs and web services enable organizations to automate business processes by increasing the speed and effectiveness of information exchange.

Architectural mismatch offers a taxonomic framework for understanding challenges in building applications out of re-usable components. This framework was extensively studied for monolithic and distributed applications, however, it has not been applied to SOAs. A major contribution of this paper is our analysis of how architectural mismatch affects SOAs. We study how implicit and conflicting assumptions that designers make about web services and their compositions affect the quality of resulting SOA-based systems. We support our analysis with empirical data that we collected as part of Tarpon, a large-scale SOA-based project within Accenture, and other smaller projects.

We show that architectural mismatch is not only helpful in categorizing and understanding practical challenges in building SOAs. It turns out that all originally described constituents of architectural mismatch are still relevant in the context of SOA. Two primary concerns for architectural mismatch in SOAs are messaging overhead and incompatibilities between SOA platform vendors.

2. Categories of Architectural Mismatch

Architectural mismatch provides a taxonomic framework for understanding how conflicting assumptions arise.

The following categorization of causes for architectural mismatch is based on Garlan et al's original taxonomy [9].

1. Assumptions about the **nature of components** can be divided into four sub-categories.

   - *Functionality supply.* Components provide functionality that may not be needed in the final assembly, leading to excessive code size of resulting applications.
   - *Infrastructure expectations.* Components may assume the presence of certain resources (e.g., libraries or hardware) that may not be available, rendering these components non-usable.
   - *Control model.* Designers assume that their components will own the main thread of control that contains an infinite event processing loop. Coordinating these event loops is non-trivial and may require change to component implementations.
   - *Data manipulation.* Designers make assumptions about how clients manipulate component data structures.

2. Assumptions about **communication between components** can be divided into two sub-categories.

   - *Asynchronous communication.* Asynchronous messaging can force conceptually single-threaded applications to be implemented with multiple threads.
   - *Message data model.* Incompatibilities in the formats of messages that components exchange can lead to massive performance overhead due to costly message conversions.

3. **Global architecture structure.** Designers assume that different clients of a component operate independently. However, clients may delegate tasks to each other, violating the independence assumption. Such dependencies may be subtle, for example, when two components access the same resource.

4. **Construction process.** Designers assume an order in which components should be constructed and how these components are combined into the system. Conflicting order assumptions can complicate the application's construction process.

The following four section discuss how these categories of architectural mismatch affect SOAs. Afterwards we summarize empirical evidence for our findings.

3. The Nature of Services

From an architectural point of view, a service provides a logically coherent piece of functionality to its clients. Mismatch can occur when designers of services define their nature in ways that make it difficult to use these services.

3.1. Functionality Supply

Problems with functionality supply exist in SOA both for individual services and because of the employed SOA infrastructure. Individual services are designed to be reusable and will therefore provide a certain flexibility in the way they can be used. This can lead to oversized services with bloated interfaces. For example, the interest payment calculation service could be used for home mortgages, auto and credit card loans. The rules of calculation are somewhat different for different loan-types. Thus, the service's incoming and outgoing messages now have to include the type of loan as well as different regulatory and location parameters needed for some calculations, regardless of whether clients actually exercise this flexibility. Thus mismatch in functionality supply can lead to oversized messages.

Another source of mismatch in functionality of individual services is the level of service granularity of an SOA. While many SOA experts advocates the use of business processes to define the scope and granularity of the underlying services, the problem of granularity mismatch does not go away because many business processes can be further decomposed into smaller process steps. If processes are too fine-grained making the services too small then messaging overhead becomes overwhelming. If services are too large then messages can become big and cause services to respond slowly.

This messaging overhead directly impacts the performance of the service. Studies have shown that XML messages are typically 10 to 50 times larger than their binary counterparts and that XML-related tasks such as parsing, transformation and serialization consumed over 93% of total processing of typical XML documents [4]. Thus, a single highly used service with oversized messages can impact an entire SOA infrastructure.

Additionally, SOA middleware commonly addresses non-functional concerns such as security. In order to address these concerns, the middleware commonly expect that services expose certain interfaces. However, despite having standards like WS-Security, these interfaces may differ between middleware vendors, leading to mismatches in critical areas such as security or reliability (see section 7.1.1). While mismatches between individual services could potentially be corrected by changing service implementations, mismatches between SOA middleware implementations from different vendors cannot be addressed di-

rectly.

3.2. Infrastructure Expectation

Many traditional component technologies use late binding, i.e. components are not connected until they are executed. Essentially, lately bound components only depend on the interfaces of other components and not on their implementation. This introduces additional flexibility, in particular the ability to change the implementation of one component without the need to re-compile other components. On the other hand, late binding causes brittleness if a component disappears that other components depend on.

In traditional component development, early binding is a prevalent way to couple components at compile time. Because independence of services is highlighted in the context of SOA we suspect that changing service interfaces and service disappearance may be a more common phenomena in SOAs. Loose coupling between services is one of the promises and advantages of SOA but it requires more work in controlling the dependencies between services and across versions of a service. Service directories currently begin to address this issue.

3.3. Control Model

Control model mismatch is a smaller problem in SOAs. Services are assumed to be autonomous and they typically run on independent machines or in different processes. They are orchestrated asynchronously in a workflow-like manner. The orchestration engine owns the "logical" thread of control that drives the overall application. Therefore, the competition for the main thread of control (as found to be a problem with traditional components) is not a significant issue for the services in SOA.

The control mechanism inside an orchestration engine can be fairly complicated. For example, in systems with a federated topology, an orchestrated set of services can be exposed as a "logical" service to a higher-level orchestration engine - thus creating a hierarchy of orchestrations. Furthermore, there are cases where a number of orchestration engines may have to cooperate in a peer-to-peer manner across organizational boundaries to accomplish a set of tasks (see section 7.1.2).

3.4. Data Manipulation

Data manipulation problems are less problematic in SOA than with traditional components. Services typically cannot expose internal data to clients, and therefore clients cannot manipulate this data directly. However, if services are stateless, the entire conversational state may be required to be sent back and forth between client and service thereby exposing internal data to external clients.

As with internal data of traditional components, it is conceivable that clients are only allowed to modify certain parts of the conversational state received from the service. If a service is implemented carelessly unexpected manipulations of conversational state by clients could lead to inconsistent data and ultimately service malfunctioning. Thus architectural mismatch due to restrictions on data manipulation by clients does exist and can lead to various problems as described in Section 7.2.

4. Communication Between Services

Communication between applications and services participating in an SOA is typically handled by messaging middleware. Messages are typically exchanged in XML format and routed through a message bus that connects to all applications and services. Messaging follows an asynchronous model and some services will even publish notification messages without knowing which other services receive these notifications.

4.1. Asynchronous Communication

In theory, services in SOA should be autonomous and the communication among services should be done asynchronously. In reality however, services are rarely completely autonomous and they do not communicate solely through asynchronous means. Asynchronous communication makes it difficult for developers to implement conceptually sequential applications using asynchronous service invocations. Seemingly single-threaded applications are forced into multi-threaded implementations with callback routines in order to process asynchronous replies from services. This can be a significant complication in implementing even simple applications.

On the positive side, asynchronous communication can often be used transparently in a synchronous fashion. Moreover, service orchestration engines are specifically designed to facilitate taking advantage of asynchronous communication without burdening developers with traditional problems of multi-threaded software.

4.2. Message Data Model

Services typically communicate through XML and therefore require support for accessing and manipulating XML. XML messages are difficult to parse and relatively verbose. Thus messaging overhead is a considerable factor in designing an SOA even before mismatch occurs [7].

Mismatch in message data models essentially means that messages from one service do not fit the expectations of an-

other service. Consider the case where output of one service is used as part of the input to another service. If the two services use different message formats then the first service's output needs to be converted so that it matches to the second service's expectations.

It has been pointed out that such conversions quickly become a performance bottleneck in SOAs [7]. This is because XML is not only difficult to parse and therefore difficult to convert but also because the stateless model of services requires messages to carry the entire conversational state, leading to very large messages. Therefore it appears that mismatch in message data models can become a serious problem in SOA-based infrastructures.

5. Global Architecture Assumptions

Architectural mismatch due to global architecture assumptions may be comparable between SOA and traditional components. When services are orchestrated using a control layer in SOAs, their dependencies may be implicitly coded in the orchestration mechanism. Revealing these dependencies is as difficult as in non-SOA-based systems.

6. Construction Process

At first glance it appears that mismatching component assumptions about the construction process are eliminated in SOA. This is because services are developed and built independently. By "building" we mean compiling and linking the service implementation. While the situation is certainly much better than with traditional components it appears that managing the build process can still be difficult because services can depend on the interface of other services.

Consider the case where service A depends on service B. One way of building service A is to obtain B's interface definition and use it for generating glue code for communicating with B. If B (directly or indirectly) happens to depend on A's interface as well then it can be challenging to build A and B fully automatically.

In SOA there is another complication: Services also have to be deployed to on a SOA middleware in order to make them available. This can complicate testing of independently constructed services that invoke each other when an expected service is not (yet) available. Techniques similar to unit testing are needed that essentially simulate the presence of other services with "mock services".

7. Empirical Evidence

In this section, we provide empirical evidence and show models that describe how architectural mismatch surfaces in SOAs. We review the issues of architectural mismatch in an Accenture project and in message exchanges between services.

7.1. Tarpon

Project Tarpon is an SOA R&D initiative within Accenture to explore the promises and pitfalls of SOA. The initiative will test everything from security to performance, delivering a realistic evaluation of the feasibility and benefits of separating cross-enterprise processes from applications that implement the processes.

7.1.1 Basic Messaging

As the first step in this evaluation process, Tarpon performed a series of basic messaging interoperability tests across select SOA platforms from vendors like Microsoft, IBM and Oracle. The test that we performed is a superset of what was published by the WS-Interoperability organization [1]. The idea is to measure the effectiveness of various web services standards such as SOAP, XML and WSDL in alleviating the differences of communication protocols across multiple platforms [14].

We found that even with these established standards, cross-platform interoperability continues to be a challenge. Specifically, we found that:

- different vendors are adopting standards at different pace;
- different vendors are adopting different parts of a standard, and
- complex dependencies among standards further exacerbate confusion in implementation.

Speed of adopting standards. Because standardization is typically a multi-year process, by the time a specification becomes a standard, that standard may be succeeded by a new and better specification. That is exactly what happened to WS-Reliability, a reliable messaging standard that is published by OASIS. By the time it became a standard it was already superseded by a new–and by many measures superior–specification: WS-ReliableMessaging (WS-RM).

The vendors are left with a dilemma of which one to adopt: a published standard or a better specification. To make things worse, the two are completely incompatible. While it is not surprising that some vendors chose to implement the published standard, those that chose the better specification worked hard to ensure that the specification becomes a standard. In the meanwhile, this left users baffled as they had to absorb this incompatibility at their own expense.

Partial adoption of standards. WS-Security is another standard from OASIS that addresses, among other concerns,

encryption and authentication of messages. As a fairly complicated standard it contains multiple parts, some of which are optional. For example, WS-Security specifies a number of authentication protocols such as username-password pair, X.509, Kerberos and SAML. Except for the username-password pair, it is up to the vendor to choose which other protocols to implement.

This implies that while two products can claim that they support one standard, WS-Security, they may implement two different sets of protocols. For example, one vendor supports Kerberos and the other supports SAML. This complicates cross-platform interoperability. Users have to be aware of these variations and are often times left with the least common denominator.

Complex dependencies among standards. Many standards do not stand alone. They depend on other existing standards. Take for example X.509, a decade-old standard for public key infrastructure published by International Telecommunication Union (ITU). One of its essential components is the key certificate. The format of this certificate is defined by other standards such as RSA Data Security's Public Key Cryptography Standard (PKCS). There many versions of PKCS–most notably PKCS #8 and #12. While it is possible to convert one certificate format to another, the process is non-trivial [11].

As each standard evolves independently, keeping track of the dependencies and compatibilities is a serious and time-consuming exercise. The situation is worsened by the fact that the number of standards organizations involved in this area continues to grow due to the popularity of web services and SOA. For example, the World Wide Web Consortium, OASIS, Liberty Alliance, Internet Engineering Task Force, Data Management Task Force are all involved in standardizing security and identity management. As such, it is inevitable that they will create multiple conflicting standards.

7.1.2 Control Model

Business process orchestration is a key component of SOA that distinguishes SOA from JBOWS (just a bunch of web services). The idea is that each service in SOA is autonomous and message exchanges across services are handled through one or more orchestration engines in a workflow-like manner. This way, service invocation is entirely controlled by orchestration engines.

We found that this control model is too constraining for cross-enterprise scenarios. Take for example electronic prescribing: a doctor creates an electronic prescription. Before a prescription can be forwarded to a pharmacy, it has to go through several organizations including the insurance company that checks for drug coverage under an insurance plan. Since there are many independent organizations involved, it is not clear where service orchestration should reside and how the invocation of services should be strung together.

This calls for an event-driven business process orchestration (or event-driven architecture) [6]. The idea is to move from a static control flow structure–similar to what is typically found in a console program–to an event-based flow as found in many GUI environments. We found that many SOA platforms today do not support this type of control model.

7.2. Errors In Exchanging XML Data

It is conservatively estimated that the cost of programming errors in component interoperability just in the capital facilities industry[1] in the U.S. alone is $15.8 billion per year. A primary driver for this high cost is fixing flaws in incorrect data exchanges between interoperating components [3]. An instance of this problem is described in a case study of a large-scale project conducted at KLA-Tencor Corp. [10].

We use a basic model shown in Figure 1 to describe how services interoperate. In this model, `J` and `C` are services that interact using XML data D_2. Service `J` reads in data D_1, modifies it, and passes it as data D_2 to `C`. Service `C` reads in the data D_2 expecting it to be an instance of some schema `S`. Since `J` outputs data D_2 before `C` accesses it, concurrency is not relevant. However, because of design or programming errors, service `J` outputs the data D_2 as an instance of a different schema `S'`, which is not explicitly stated in any design documents. Since `S'` is different from `S`, a runtime error may be issued when `C` reads in D_2.

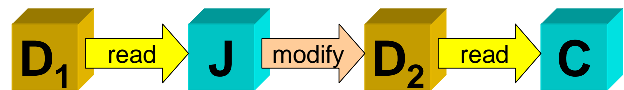

Figure 1. A model of service interoperability.

There are different reasons why programmers make such mistakes when they write the services `J` and `C`. Based on our participation in large-scale projects, we observe that programmers often make wrong assumptions about schemas. Given that many industrial schemas contain thousands of elements and types, it is easy to make mistakes about names of elements and their locations in schemas. The other source of errors lies in the complexity of platform API calls that programmers use to access and manipulate XML data. XML parsers export dozens of different API calls, and mastering them requires a steep learning curve.

Programmers often lack the knowledge of the impact caused by changing the code of some component on other components that interoperate using XML data. This lack of knowledge is an effect of the Curtis' law that states

[1] A capital facility is a structure or equipment which generally costs at least $10,000 and has a useful life of ten years or more.

that application and domain knowledge is thinly spread and only one or two team members may possess the full knowledge of a software system [5]. The effect of this law combined with the difficulty of comprehending large-scale XML schemas and high complexity of platform API calls result in components producing XML data that is incompatible for use by other components.

The other source of errors is the disparity in evolving XML schemas and components. Database administrators usually maintain schemas, and programmers maintain components that interoperate using XML data that should be instances of these schemas. If a database administrator modifies a schema and does not inform all programmers whose services are affected by this change then some services will keep modifying XML data according to the obsolete schema.

The problem of mismatch between XML data and schemas is typically addressed by using schema validators that are parts of many XML parsers. In our model shown in Figure 1, an XML parser can validate that the data D_2 is an instance of the schema S when J produces this data. If the data is not an instance of this schema, then the parser throws a runtime exception. Obviously, it is better to predict possible errors at compile time rather than to deal with them at runtime.

In reality, the situation is even more complicated. Using schemas for validating XML data is often not attempted because it degrades components performance [15, 13], and it even leads to throwing exceptions when there may not be any runtime errors. Suppose that the service J deletes all instances of some data element thus violating the schema S that requires at least one instance of this element be present in D_2. If either of services J and C validates this incorrect data D_2 against the schema S, then a runtime error will be issued. However, when executed, C may never attempt to access the deleted data element, and therefore, no exception will be thrown if the validation step is bypassed. It is important to know what data elements services J and C access and modify, and if no data element accessed by C is modified by J, then J and C may still interact safely even if the data D_2 is not an instance of the given schema S.

8. Conclusion

In this paper we apply a taxonomic framework for understanding how conflicting assumptions arise to Service Oriented Architectures (SOAs). A major contribution of this paper is the analysis of how architectural mismatch affects SOAs. We study how implicit and conflicting assumptions that designers make about web services and their compositions affect the quality of resulting SOA-based systems. We support our analysis with empirical data that we collected as part of Tarpon, a large-scale SOA-based project within Accenture and other smaller projects. We show that architectural mismatch is not only helpful in categorizing and understanding practical challenges in building SOAs. It turns out that all originally described constituents of architectural mismatch are still relevant in the context of SOA. Two primary concerns for architectural mismatch in SOAs are messaging overhead and incompatibilities between SOA middleware vendors.

References

[1] Web services interoperability organization website. *http://www.ws-i.org/*.

[2] *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries.* Institute of Electrical and Electronics Engineers, January 1991.

[3] *Cost Analysis of Inadequate Interoperability in the* U.S. *Capital Facilities Industry, GCR 04-867*. NIST, August 2004.

[4] *Solving the Very Large Messaging Problem in the Enterprise*. www.zapthink.com/report.html?id=WP-0137, February 2005.

[5] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Commun. ACM*, 31(11):1268–1287, 1988.

[6] V. Dheap and P. A. S. Ward. Event-driven response architecture for event-based computing. In *CASCON*, pages 70–82, 2005.

[7] T. Erl. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Prentice Hall, 2004.

[8] C. Ferris and J. A. Farrell. What are web services? *Commun. ACM*, 46(6):31, 2003.

[9] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *ICSE*, pages 179–185, 1995.

[10] M. Grechanik, D. S. Batory, and D. E. Perry. Design of large-scale polylingual systems. In *ICSE*, pages 357–366, 2004.

[11] A. K. Lenstra and B. de Weger. On the possibility of constructing meaningful hash collisions for public keys. In *ACISP*, pages 267–279, 2005.

[12] J. McGovern, O. Sims, A. Jain, and M. Little. *Enterprise Service Oriented Architectures: Concepts, Challenges, Recommendations*, section 1, pages 1–11. The Enterprise Series. Springer, first edition, 2006.

[13] J. Meier, S. Vasireddy, A. Babbar, and A. Mackman. Improving .NET application performance and scalability. *Microsoft Corporation*, 2004.

[14] E. Newcomer. *Understanding Web Services:* XML, WSDL, SOAP, *and* UDDI. Addison-Wesley Professional, May 2002.

[15] R. Schmelzer. Breaking XML to optimize performance. ZapT*hink* LLC - *special to SearchWebServices.com*, Oct. 2002.