A survey of middleware with focus on application in network based defence

Ketil Lund, Trude Hafsøe and Frank T. Johnsen

Forsvarets forskningsinstitutt/Norwegian Defence Research Establishment (FFI)

12.12.2007

FFI-rapport 2007/02683

Project 1086

ISBN 978-82-464-1306-8

Keywords

Mellomvare

Nettverksbasert forsvar

Tjenesteorientert arkitektur

Approved by

Anders Eggen	Project manager
Vidar S. Andersen	Director

English summary

Interoperability, both inter- and intra-nation, is a main concern when attempting to fully realize Network Based Defence (NBD). The NBD vision implies an information infrastructure that supports prioritized access to information, services, and communications resources from the strategic level, down to the tactical level.

Middleware is an abstraction layer that has the potential to conceal the heterogeneity of the applications, operating systems, commmunication systems and hardware in such a distributed system. In this report, we look into the most important classes of commercial middleware, in the context of military application. This includes the most important aspects of Web services and Enterprise Service Bus. We also present a research prototype, a QoS-aware middleware that is able to understand the requirements of the application and the user, and configure both itself and the application accordingly.

Sammendrag

Interoperabilitet, både nasjonalt og internasjonalt, er en svært viktig faktor i arbeidet med å fullt ut realisere nettverksbasert forsvar (NbF). NbF-visjonen innebærer en informasjonsinfrastruktur som støtter prioritert tilgang til informasjon, tjenester og kommunikasjonsressurser på strategisk nivå, og helt ned til taktisk nivå.

Mellomvare er et abstraksjonslag som gjør det mulig å skjule heterogeniteten i applikasjoner, operativsystem, kommunikasjonsløsninger og maskinvaren i et slikt distribuert system. I denne rapporten ser vi nærmere på de viktigste klassene av kommersiell mellomvare, med tanke på bruk i militær sammenheng. Dette inkluderer en presentasjon av de viktigste aspektene innen Web services og Enterprise Service Bus. Vi ser også på en forskningsprototyp av en mellomvare som er i stand til å forstå krav fra bruker og applikasjon, og som kan konfigurere både seg selv og applikasjon slik at disse kravene møtes.

Contents

1	**Introduction**	**7**
2	**Middleware**	**10**
2.1	Existing middleware	10
2.2	The problems of existing middleware	11
2.2.1	Middleware in a military context	11
2.2.2	Middleware and QoS	12
2.3	Conclusion	13
3	**Enterprise Service Bus**	**14**
3.1	ESB functionality	14
3.2	Pattern vs Product	15
4	**Web services**	**16**
4.1	XML standards	17
4.1.1	Extensible Markup Language (XML)	17
4.2	Messaging-related protocols	18
4.2.1	SOAP	19
4.2.2	WS-Notification	19
4.2.3	WS-Eventing	20
4.3	Security Protocols	20
4.4	Reliable Messaging Standards	20
4.4.1	WS-ReliableMessaging (WS-RM)	21
4.5	Transaction-related Protocols	21
4.5.1	WS-AtomicTransaction	21
4.6	Description-related Protocols	22
4.6.1	Web Services Description Language (WSDL)	22
4.6.2	Web Services Inspection Language (WS-Inspection)	22
4.6.3	Web Services Policy Framework (WS-Policy)	23
4.6.4	Universal Description, Discovery, and Integration (UDDI)	23
4.6.5	Web Services Dynamic Discovery (WS-Discovery)	25
4.7	Composition Standards	26
4.7.1	Business Process Execution Language (WS-BPEL)	26
5	**Configurable and adaptive middleware**	**27**
5.1	The QuA Project	27
5.2	Comparison of dynamic component-based middleware and Web services	28

6 Summary 30

References 32

Appendix A QuA – Quality of Service Aware Component Architecture 35

A.1 Service plans 35

A.2 Applying Service Plans 36

A.3 Deployment 36

A.4 Instantiation 37

A.5 Reconfiguration 39

1 Introduction

Interoperability, both inter- and intra-nation, is a main concern when attempting to fully realize Network Based Defence (NBD), the Norwegian equivalent of Network Enabled Capabilities. The NBD vision implies an information infrastructure that supports prioritized access to information, services, and communications resources from the strategic level, down to the tactical level where communication resources usually are scarce. This encompasses a vast array of different systems, and without interoperability between these heterogeneous systems the NBD vision will be very hard to accomplish.

One well-known way of handling heterogeneity and distribution is through the use of middleware. Middleware is an abstraction layer that can conceal the heterogeneity of applications, operating systems, communication systems and hardware in a distributed system by providing a common interface to applications. In other words, *middleware is a software layer between application and operating system* [1]. In addition to providing a standardized interface, middleware can offer distribution transparency, i.e., hide the consequences of distribution, with respect to things like location, access, concurrency, replication, errors, and mobility. One common programming abstraction is offered, spanning a distributed system. The middleware encapsulates general solutions on tasks that appear repeatedly in distributed systems, offering building blocks on a higher level than the APIs and sockets offered by the operating system. The middleware concept is illustrated in Figure 1.1, where "Middleware services" is the interface provided to the applications, and "Interaction services" is the set of transport mechanisms used to facilitate middleware communication.

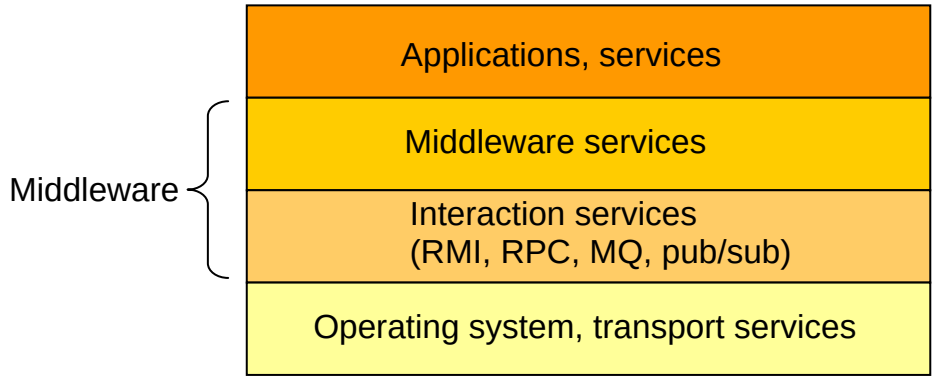


Figure 1.1 Middleware as a layer between applications and operating system

Although middleware is usually viewed as a class of software technologies designed to help managing the complexity and heterogeneity found in distributed systems, there is no general consensus on the precise separation between these areas, nor on the required functions and services. In general, the separation will vary with time, as common application functionality is better understood, such that it can be standardized and moved into the middleware as a service.

The concept of Web services takes the middleware abstraction even further, and can be viewed as a “middleware of middlewares”. The idea is that while traditional middleware is used within a system or platform, for instance a local area network (LAN) within a company, Web services are used to connect such systems. In other words, Web services are intended for use *between* systems/platforms, and, hence, the “middleware of middlewares” analogy. Note however, that this does not preclude the use of Web services also within a system/platform.

Web services are also the most common way of realizing a Service Oriented Architecture (SOA), which in turn is considered a very important component in the realization of NBD [31]. Consequently, there is a strong focus both nationally and within NATO on using Web services on all levels (see for instance [16]). Even if some core standards are mature, other important topics are at the draft level, or are covered by competing non-interoperable standards. Thus Web services are still far from being mature enough for playing the role as a full-blown “middleware of middlewares”.

Enterprise Service Bus (ESB) is a concept that is often mentioned together with SOA. An important purpose of an ESB is to be a layer and access point between service providers and service consumers. The motivation is that by avoiding point-to-point connections between consumers and providers, the number of connections is reduced, and the service consumers do not need knowledge about location or implementation of a service. An ESB supports different communication patterns and protocols, and is therefore not necessarily Web services-based. There is also support for orchestration of services, and different support functions, e.g., for security, maintenance and surveillance. Finally, an important aspect of ESB is that all functionality within the ESB itself is deployed as services, in the same way as the (business) services offered. This means that all functionality uses the same infrastructure.

Within the Web services community, as well as the ESB community, Quality of Service (QoS) has so far received little attention. However, QoS is an important aspect of NBD, and must be taken into consideration to be able to realize the NBD vision. In [2] QoS is defined as *”... the set of those quantitative and qualitative characteristics of a distributed multimedia system necessary to achieve the required functionality of an application”*, and in an NBD context, this could, for instance, include the use of priorities. For a thorough discussion of QoS issues and current middleware solutions, we refer to [3].

In the industry, we see that middleware platforms are being extensively used in both large computational systems, like banking, finance, and on-line content; and small applications for hand-held devices, like games, video players and picture editors. However, existing solutions still have a number of limitations, in the sense that they do not have sufficient adaptation capabilities for such dynamic environments as tactical NBD represents, where both resource and service availability varies continuously. These limitations are currently being addressed by the research community, and the latest generation of middleware, using component technologies, promises to enable application development through logical service specification, i.e., by selecting and interconnecting components. The application developer does not have to consider the physical

locations of the components, since access to remote components is hidden by the underlying middleware. Furthermore, the middleware itself handles the selection of component implementations, in order to configure the application to be able to meet the QoS requirements of the user and the application.

Although such research prototypes by no means are mature enough for operative use, there are techniques and principles used in these prototype that we believe could be successfully transferred to an ESB- or Web services-based SOA used in an NBD context. We have therefore included a presentation of a state of the art research prototype using such technology in this report.

The remainder of this report is organized as follows: In Chapter 2 we provide a brief overview of current, “traditional” middleware technologies, and why these fail in a tactical military context. Chapter 3 presents the ESB concept, while Chapter 4 introduces Web services and presents the most relevant standards. Please note that this report is not intended to be a complete survey of middleware. Rather, we have chosen to present what we consider to be the most important standards and aspects within these areas. In Chapter 5, we present a state of the art research prototype of a configurable and adaptive middleware, and provide some considerations around how elements from this prototype can be used in a SOA-based NBD. Finally, in Chapter 6, we provide a short summary.

2 Middleware

In this chapter we provide an overview of existing “common off-the-shelf” middleware. We give a short overview of the different generations of such middleware, before discussing the main shortcomings of these classes of middleware.

2.1 Existing middleware

The notion of middleware was first used in connection with the introduction of Remote Procedure Call (RPC) and message queuing (MQ). These were technologies aimed at hiding the lower levels of communication, in order to simplify the work of the programmers, and they represent the first generation of middleware.

The second generation of middleware was based on distributed object technology. Instead of a client/server-model, a client/object-model is used, and objects are accessed through interfaces. The abstraction offered, is that a remote object can have its methods invoked as if the object was in the same address space as the caller (this is called location- and access transparency). Examples of second generation middleware are Java Remote Method Invocation (RMI), Microsoft (Distributed) Component Object Model (COM/DCOM), and OMG CORBA.

There is, however, an important problem of this second-generation middleware, namely the presence of implicit dependencies. In other words, it is necessary to have a lot of information about the environment in which the application will run: How is the application deployed, which services will be available on a given server, who will activate my objects, who will manage the life-cycle of my objects, and so on. In addition, CORBA, one of the most well-known middlewares, suffered from a high degree of complexity and lack of interoperability between different vendors’ implementations.

In the third generation of middleware, we see the transition to component-based technology, in order to alleviate this problem. Ideally, the components in a component-based system should only have explicit dependencies, which in turn guarantee *safe deployment*. This is a guarantee that if these explicit dependencies are satisfied, the component will behave as specified. Consequently, the component can be reused by any application needing its functionality.

Middleware platform	Component model
OMG CORBA v3	CORBA Component Model (CCM)
Java 2 Enterprise Edition (J2EE)	Enterprise Java Beans (EJB)
Microsoft .NET	COM+

Table 2.1: Commercial third-generation middleware

Three component technologies are widely known, both in industry and the research community: CORBA component model (CCM), Enterprise Java Beans (EJB), and Common Object Model Plus (COM+). For each component technology there are corresponding groups of middleware platforms; Common Object Request Broker Architecture (CORBA), Java 2 Enterprise Edition (J2EE) and .NET Framework.

2.2 The problems of existing middleware

There are several challenges associated with existing middleware, both general, and specific to military use. One of the most important general challenges is that of QoS management, while from a military point of view, there are additional challenges, in particular associated with security, the use of middleware in international operations, and in tactical networks. In this section, we look closer at some of these challenges.

2.2.1 Middleware in a military context

2.2.1.1 Middleware in international operations

An important limitation of existing middleware solutions is that they are designed to operate within a single company[1], where concerns such as ownership and responsibility are clear. This means that when it comes to what is generally known as business to business (B2B) integration (which in many ways is comparable to an international military operation), several problems arise [32]. When several countries (or military services within a country) need to interconnect their systems, they face the challenge of integrating several heterogeneous and autonomous systems, and possibly also implement automated business processes that span across these.

An important problem faced, is the lack of a natural place to put the middleware. Traditional middleware is (logically) centralized, and controlled by a single company. In the case of an international military operation, this would require that all countries agree on a single middleware platform, which they manage cooperatively. Still, issues like trust, preservation of autonomy, and security would pose significant challenges.

2.2.1.2 Middleware in tactical networks

A tactical environment is characterized by diverse platforms, and a highly dynamic environment, where both resources and services come and go continuously. This means that both the middleware and the application must be adapted to the given environment. In such an environment, it becomes particularly difficult to predict the correct configuration of application and middleware during design time. Information such as what platform is used, and which resources are available and in what quantity, is only available at run-time. Thus, a number of configuration decisions can only be made at run-time. Today's middleware, on the other hand, only supports design-time configuration.

[1] In a military context this would mean within a single country, or even within a single military service of a country.

The dynamicity of tactical environments also necessitates the ability of both application and middleware to be reconfigured as resource availability fluctuates. In general, there are four types of reconfiguration that can be performed, both on an (component-based) application and on the middleware itself:

- *Component-internal*: The application (or component within the application) handles the adaptation or reconfiguration internally. This is the traditional solution, which can be seen, for example, in Windows Media player.
- *"Turning knobs"*: Parameters within the application or component are adjusted externally. One example is the middleware adjusting the buffer size of a streaming application, in order to compensate for increased jitter in the network.
- *Replace components*: In this type of reconfiguration the implementation of a component is replaced. For instance, a compression component could be replaced with one that is less resource demanding (but slower).
- *Change composition*: The last and most comprehensive type of reconfiguration is to change the actual composition of an application. For instance, if the bit error rate of the network increases, a Forward Error Correction (FEC) component could be inserted into the application (on both client and server side), in order to compensate for the increased number of packet errors.

For all four types of reconfiguration, it is a requirement that safe reconfiguration is ensured. In other words, the application must be transferred from one stable configuration to another.

2.2.2 Middleware and QoS

QoS can be many things, encompassing everything from networking issues and prioritization of messages to a consistent service experience delivered at the user level. To achieve QoS in NBD, it must encompass all systems to deliver end-to-end consistency. Thus, the network must support QoS, the applications must be able to negotiate their QoS requirements, and the middleware in between must function as a mediator and broker. For a thorough discussion of QoS issues and current middleware solutions, see [3].

The main problem with respect to QoS is that current component-based middleware offers only *functional* safe deployment. In other words, although a given component is guaranteed to function as advertised, there are no guarantees as for *how well* it will perform its functions. For simple functions such as a file transfer, this may not be that important, but for an application that is sensitive to QoS this may have a considerable impact on the usability of the application. When a programmer designs a component, she will make assumptions about the environment in which the component will run, including assumptions about available resources. However, at design time, it can be hard to make any such assumptions, as the actual resource availability is usually not known until run-time. Consequently, the behaviour of the application becomes hard to predict.

Furthermore, to be able to maintain a given QoS level, we need QoS management. Today, this management is usually hard-coded in the application (i.e., the components). This means once again that one has to make assumptions about the environment, which in turn limits the possibility of reuse. In addition, even if the component designer is able to predict the QoS, how can this be communicated to the user of the component? Finally, how do you predict the end-to-end QoS of an application, based on the QoS characteristics of the individual components? The QoS that can be delivered will be dynamic and change over time as a system is operational. Fluctuations in the available resources demand that the system can adapt dynamically, and thus part of the QoS management should be done in run-time to be effective.

This problem is further amplified by the fact that dynamic middleware solutions for components mainly combine context-awareness with reconfiguration mechanisms, ignoring the QoS characteristics of the different application configurations. As a consequence they fail to support applications where QoS characteristics are critical, such as applications for streaming and conferencing. Furthermore, in current state-of-art solutions, the specifications of the application configurations are defined for a particular middleware. Thus, the specification and its information elements are specialized for a specific set of tasks and platforms, making reuse difficult.

A possible solution to these QoS-related problems is to introduce *separation of concerns*. This means that the application logic and the QoS management are handled separately, which in turn means that one has a much better opportunity of postponing decisions until one has sufficient information available. This is, however, not possible with today's commercial middleware.

2.3 Conclusion

As we have in this chapter discussed several significant problems associated with using traditional middleware in a military context, and in particular for tactical networks. On the other hand, using individual point-to-point connections is not an alternative, since this would prohibit the flexibility required in an NBD.

One possible solution for realizing the NBD vision is to use Web services as middleware, and then possibly extend it with techniques and functionality from research prototypes within component-based middleware. In this report, we therefore give an overview of the most relevant Web services standards, and we present a research prototype of component-based middleware that contain some useful techniques that may be transferred to a Web services middleware.

3 Enterprise Service Bus

Enterprise Service Bus (ESB) is a concept often mentioned along with SOA. The term ESB is used to describe everything from an architectural style down to a number of concrete software products, so it can be difficult to get a good overview of what an ESB is. The most common view is that an ESB is a middleware construct that is based on recognized standards, and provides a number of capabilities or functionalities that can be used as a backbone upon which one can build a SOA. There is however considerable debate if an ESB is simply a catch-all term for this collection of functionality, or if it is an architectural style of its own or even a tangible product. To further add to the confusion, there is also a discrepancy in which functionality the different vendors and standardization organizations consider as part of the ESB.

An ESB is a part of a larger SOA middleware, and is responsible for the interconnectivity between services. Using an ESB is however not the only way to provide this interconnectivity, and it is perfectly possible to build a SOA without the use of an ESB. The ESB can be seen as a virtualization layer for services because it raises the abstraction level from thinking about messages and message processing, such as content-based routing and protocol transformation, to thinking about services. This form of virtualization might not be needed in a small SOA-based infrastructure, but it can be essential for systems consisting of a large number of services. Connecting these services to each other via an ESB rather than directly can greatly reduce the management effort needed when updating or changing service definitions. To realize that level of abstraction, an ESB needs to include functionality such as message routing, protocol transformation, mediation services and more.

3.1 ESB functionality

If developers are supposed to be able to think about implementing functionality in services rather than worry about how to connect them, the ESB used must hide differences in the protocols, communication paradigms and message formats used, in addition to handling service location issues. A client that wants to connect to a service might use a different messaging protocol than the service does, one example could be that the client uses the standard SOAP protocol while the service might be based on REST[2]. The ESB must hide these differences and allow the client to connect to the service without having to change the way it does its messaging. The ESB also needs to hide differences in interaction patterns, so that communication can happen even if one participant uses for instance a synchronous point-to-point protocol and the other uses an asynchronous protocol.

[2] Representational State Transfer (REST) [33] is often used to describe any simple interface that transmits domain specific data over HTTP without an additional messaging layer such SOAP or HTTP cookies.

In addition to message routing and transformation, there is a multitude of other functionality that can be supported by an ESB. These are capabilities that are needed for achieving the virtualization described above, but they are not essential for basic message exchange.

The benefit of using an ESB to provide this functionality rather than using Web services-based point-to-point connections between services is that the developer of the service only needs to focus on the connection to the ESB. This means that the developer can choose one of the many standards to, for instance, secure his Web service. Thus, it is not necessary to implement a number of different interfaces to support all the different security standards a potential service consumer might implement.

However, although an ESB seems to make things simpler, it is important to remember that the complexity is still there. Where the structure of the middleware was visible when using ordinary Web services, it is now hidden within a "box", supplied by some vendor. In fact, the complexity may be even greater than before, because a large amount of functionality is mixed together within the ESB "box". Furthermore, as described above, ESBs are vendor-specific. This means that you cannot easily replace one ESB platform with another. In a military context, this also means that different countries, running different ESBs may have problems interconnecting, unless the different vendors provides compatible adaptors.

3.2 Pattern vs Product

As previously mentioned, there is an ongoing discussion about whether an ESB is an architectural pattern or a software product. The discussion above presents an ESB as a collection of functionality that is needed in order to achieve service virtualization. It is, however, important to note that one does not need an ESB in order to have this functionality. SOA developers can of course choose to implement these capabilities from scratch themselves. The same argument can be used for the use of pre-made library functions in programming; a programmer can of course write every last bit of code herself, but it makes sense to use pre-made libraries when those exist. Similarly, an ESB software product can be used to provide common functionality that all large scale SOA deployments require, without having to implement this from scratch.

To summarize the ongoing pattern vs product discussion, it is useful to think of an ESB as a pattern when defining it, since it is best defined as a collection of functionality. However, the whole idea of having an ESB as a part of a SOA architecture is that the ESB will provide common functionality off the shelf, so that developers can just install, configure, and deploy the ESB and then focus on building their business specific services. The ESB will then provide consistent interfaces and connectivity and make sure that all the Web service standards that are used work together in the best possible way. This is what an ESB software product is all about, so from a business perspective it makes more sense to think of an ESB as a product that can be purchased and used to simplify the SOA deployment and management process. It is, however, important to keep in mind that while an ESB can be an important part of a SOA, it is not sufficient on its own. An ESB will for instance not provide functionality for governance and Web services management.

4 Web services

One of the earliest Web services standards, SOAP, was first introduced in 1999. Since then the number of Web services related standards have been ever increasing and the WS-*standards now cover a large range of topics. The core standards, such as XML, SOAP and WSDL are widely supported, but the sheer number of available standards means that it is difficult for developers to know which standards to adopt. This task is made even more complex when taking into consideration the fact that the maturity of the standards vary. Some standards are fully ratified and have been released in several versions already, while others are early in their development cycle and are currently working drafts or have status as specifications.

In addition to the maturity issue, it is worth noting that there is no one organization that controls all of the ongoing Web services standardization work, and thus there is no set standardization process that ensures that all the standards adhere to a common “Web services architecture”. As a consequence, some topics are covered by multiple, and in some cases competing, standards, while other topics are not covered by a standard at all. Vendors and standardization organizations both use standardization as a political tool, and the rate of vendor adoption of standards vary. It is unlikely that all of these standards will stand the test of time, but it is too early to say which standards will prevail. Vendor support is crucial, so looking into which standards are currently supported by the major vendors such as IBM and Microsoft can function as a guideline when trying to determine if a standard is likely to ever see widespread use. WS-I [35] is an open industry organization chartered to promote Web services interoperability across platforms, operating systems and programming languages. The organization’s diverse community of Web services leaders helps customers to develop interoperable Web services by providing guidance, recommended practices and supporting resources. All companies interested in promoting Web services interoperability are encouraged to join the effort.

This report does not attempt to cover every aspect of the Web services standardization due to the ever changing nature of this work. Instead it focuses on the main categories and topics covered by current Web services standards, and point out the core standards within each of these categories. These standards are the ones that are currently most likely to be a part of a SOA deployment, and form a fundament for using Web services as a SOA middleware. Figure 4.1, gives an overview of the core categories of Web services standards, but does not cover all topics. It does, however, serve as a good starting point for a more detailed categorization and standards description.

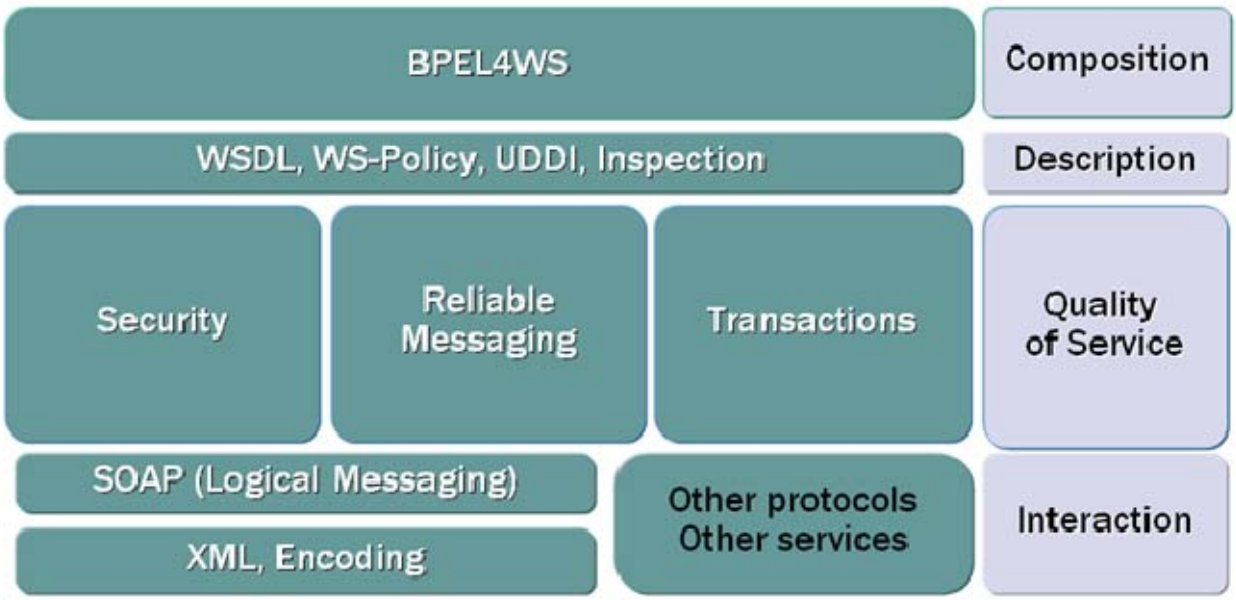


Figure 4.1 Web services middleware components [34]

4.1 XML standards

XML, described in further detail below, is often considered the base standard for Web services, as most of the other standards use the encoding and format rules defined in this standard. There are multiple XML related standards, with the two most important being XML itself, and XML Schema. The latter standard is a description of a type of XML document, typically expressed in terms of constraints on the structure and content of documents of that type, above and beyond the basic syntax constraints imposed by XML itself.

There is also a set of standards for providing security to XML like e.g. XML Digital Signature, XML Encryption, SAML and XACML. These standards are currently under evaluation of a NATO research group (NATO RTO/IST-068) for suitability in military systems.

4.1.1 Extensible Markup Language (XML)

The Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally it was designed to meet the challenges of large-scale electronic publishing, but XML is playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.

An XML-document consists of data that are surrounded by ’tags’. To illustrate the structure of an XML document, consider this simple example from [24]: Tags describe the data they enclose. A tag may have other tags inside it, which allows for a nested structure. An example XML-document is:

```
<?xml version=”1.0” ?>
<greeting>
        <greeting text>Hello, XML</greeting text>
</greeting>
```

One of the benefits of using XML is that an XML document contains metadata, that is, data about the data that are present in the document. In the example above, for instance, we can see that the text string ”Hello, XML” is a ”greeting text”, which in turn is a part of “greeting”. Such tags can be standardized, which allows for the exchange and understanding of data in a standardized, machine-readable way. An XML document can be defined according to an XML Schema, which enables validation of XML documents according to rules defined in the schema.

In its basic form, XML can be seen as a structured, human readable way to organize data. However, in certain cases it is more serviceable to sacrifice human readability for more efficient encoding and transfer. In such cases a binary representation of the XML document should be used, i.e. so-called binary or efficient XML. So far there is no standard for efficient XML, even though there is a proprietary solution available from the company Agile Delta that is called Efficient XML (EFX), but a W3C working group called *Efficient XML Interchange* (EXI) is in the process of standardizing an efficient XML format [25]. The objective of the EXI Working Group is to develop a specification for an encoding format that allows efficient interchange of the XML Information Set, and to illustrate effective processor implementations of that encoding. Earlier this year the group released a working draft [26]. It is worth noting that Agile Delta is actively participating in the EXI work, and are continually adapting their EFX product to conform to the working draft.

4.2 Messaging-related protocols

The base messaging protocol used for Web services is SOAP, but there exist a large number of standards that expand on the SOAP protocol and add further functionality. An example of one such protocol is MTOM, the Message Transmission Optimization Mechanism, that describes how to transfer binary data as a part of a SOAP message.

Additionally, the standards that support event notification are of particular interest in a military context. Publish/subscribe is a well known communication pattern for event-driven, asynchronous communication. In [16] we discuss the benefits of employing the publish/subscribe paradigm in NEC.

At present there are two standardization efforts regarding publish/subscribe for WS: OASIS finished its Web Services Notification (WS-Notification, WSN) standard [17] late in 2006, whereas W3C has produced a public draft version of a similar framework called Web Services Eventing (WS-Eventing) [18]. Both of these protocols are based on SOAP, and uses the functionality provided by SOAP rather than building their own messaging protocol.

Another category of protocols that are closely linked to messaging is transport protocols, but the Web service messaging protocols are largely transport protocol agnostic. Transport protocols are of course needed for message delivery, but the protocols used for this are well-known and already established protocols such as HTTP, SMTP, TCP and UDP, and a discussion of these is beyond the scope of this survey. However, it should be noted that the transport protocol should be message based and have store-and-forward capabilities. This is important since the messages may need to cross heterogeneous networks, and that specially on the tactical level there may be disconnections and distruptions.

4.2.1 SOAP

SOAP is a lightweight, XML-based protocol for information exchange in a decentralized, distributed system. SOAP is an envelope for XML messages, functioning as a transport independent messaging protocol. It was called “simple object access protocol” up to and including its release as a W3C version 1.1 note, but this name did not describe exactly what SOAP was, and so it was later dropped. In its current version, the W3C version 1.2 recommendation [27], the protocol is just called “SOAP”. SOAP messages can be carried by a variety of network (or application) protocols, the most common being HTTP. Other protocols can also be used. For example, we have in our previous work demonstrated that one can run SOAP over STANAG 4406 [10], which could make using Web services technology feasible on the tactical level.

4.2.2 WS-Notification

WS-Notification is a Publish/Subscribe notification framework for Web services. There are three parts in the specification: WS-BaseNotification, WS-BrokeredNotification and WS-Topics. The WS-BaseNotification specification unifies the principles and concepts of SOA with those of event-based programming.

WS-BaseNotification provides the foundation for the WSN family of specifications. It defines the basic roles and message exchanges needed to express the notification pattern. The specification can be used on its own, or it can be used in combination with the WS-Topics and WS-BrokeredNotification specifications in more sophisticated scenarios. The specification defines the message exchanges between notification producer, notification consumer, subscriber, and subscription manager.

The simplest form of a subscribe request message just contains an endpoint reference for a notification consumer. This form of request instructs the notification producer to send each and every notification that it produces to the notification consumer.

The subscribe request message can optionally contain one or more filter expressions. The filter expressions indicate the kind of notification that the consumer requires by restricting the kinds of notification that are to be sent for this subscription.

In summary, WS-Notification encompasses the following standards:

- WS-BaseNotification [19] defines standard message exchanges that allow one service to subscribe and unsubscribe to another, and to receive notification messages from that service.
- WS-BrokeredNotification [20] defines the interface for notification intermediaries. A Notification Broker is an intermediary that decouples the publishers of notification messages from the consumers of those messages; among other things, this allows publication of messages from entities that are not themselves Web service providers.
- WS-Topics [21] defines an XML model to organize and categorize classes of events into "Topics," enabling users of WS-BaseNotification or WS-BrokeredNotification to specify the types of events in which they are interested.

The WSN specifications standardize the syntax and semantics of the message exchanges that establish and manage subscriptions and the message exchanges that distribute information to subscribers. An information provider, known as a notification producer, that conforms to WSN can be subscribed to by any WSN-compliant subscriber.

4.2.3 WS-Eventing

The WS-Eventing specification defines a baseline set of operations that allow WS to provide asynchronous notifications to interested parties. WS-Eventing provides basic publish/subscribe functionality. WS-Eventing provides similar functionality to that of WS-BaseNotification so we will not present further details here. The overall concept is the same, but the two are not compatible with each other at the message level.

Currently WS-I is spearheading an effort to bring WS-Notification and WS-Eventing together in a new, interoperable specification. The publish/subscribe paradigm is well suited to military networks, see [16] for a discussion.

4.3 Security Protocols

Security is an important part of any middleware. Web services technology is increasingly being employed on the Internet, both within organizations and more importantly, between them, providing business to business services. To protect confidential data, security measures are a necessity to guarantee the success of a service and to prevent fraud and financial loss. Even more so, to protect national interests one must consider security issues when contemplating using Web services to realize NBD. However, security is being covered in a separate survey, and further discussion is beyond the scope of this report. For a thorough discussion of security in Web services and XML, see [28].

4.4 Reliable Messaging Standards

Reliable messaging standards focus on improving application reliability by adding functionality such as guaranteed message delivery to SOAP. The purpose of these standards is to add end-to-end reliability in a transport protocol independent way, so that message delivery is reliable over multiple hops even when using unreliable transport mechanisms. The goal is to make sure that message delivery occurs exactly once and in order.

There are two reliable messaging standards, WS-Reliability and WS-ReliableMessaging [30]. WS-Reliability was the first to be approved as a standard, but it lacks transaction support, secure session handling and, due to its early release, does not take other Web services standards into consideration. Because of these limitations vendor support for this standard is limited, whereas the other reliability standard, WS-ReliableMessaging, has more industry backing and is widely supported.

4.4.1 WS-ReliableMessaging (WS-RM)

WS-ReliableMessaging, also known as WS-RM, is a standard that assures reliable delivery of messages between distributed applications. The original specification was released in 2003, and it was approved as a standard by OASIS as late as mid 2007. The standard is accompanied by the WS-RM Policy and WS-MakeConnection specifications.

WS-RM uses two logical handlers or agents to provide reliability. These two, the Reliable Messaging Source (RMS) and Reliable Messaging Destination (RMD), are parts of the SOAP message handling and handle message transfer and retransmission. This process is transparent for the application. Note that for a two-way connection, both communicating parties will have both a RMS and a RMD, and even if these are two logically different entities, they can be implemented as one common handler.

The RMS is responsible for establishment and termination of reliability contracts, known as *sequences*, the adding of reliability headers to messages and message resending. The RMD on the other hand sends acknowledgements of received messages, and handles message reordering.

WS-RM supports four different message delivery assurance types, known as at least once, at most once, exactly once and in order delivery.

4.5 Transaction-related Protocols

Transaction support is often necessary when integrating applications, and protocols for such support is therefore important. WS-Transaction is a set of specifications, built on top of the WS-Coordination framework, that define protocols for transaction support in Web services. Because Web services in many areas differ from traditional middleware (lack of centralized middleware platform, long-running transactions, lack of a fixed resource model), the traditional transactional model (the “ACID”-properties) is relaxed, and instead compensation mechanisms are used [32].

4.5.1 WS-AtomicTransaction

This is a coordination type defined by WS-Transaction, and it is composed of five coordination protocols: “Completion”, “2PC”, “CompletionWithAck”, “PhaseZero”, and “OutcomeNotification”. The basic principle of WS-AtomicTransaction is a two-phase commit (2PC) protocol, where each participant is asked to either commit or abort. A separate service, a coordinator, communicates with all participants, and coordinates the transaction. At any point during the transaction, any participant may query the coordinator about the outcome of the

transaction. 2PC is often not suitable in military systems; it is often better that some units receive an update than none at all. The result can be inconsistencies among units for a period of time, but this can, in many cases, be accepted.

It is important to notice that WS-Transaction does not specify the business logic completely. For instance, the semantics of commit and abort are only informally defined. Admittedly, the general meaning of these operations is commonly agreed upon, but the behavior of the actual Web services when performing these operations may vary considerably [32].

4.6 Description-related Protocols

The standards that are related to Web service description can be divided into two main subcategories, namely service discovery and metadata standards.

The metadata standards focuses on providing a framework that can be used to describe the requirements of a service, including its operations, message format, input and output parameters, location information and which other Web service standards the service supports. The main metadata standard is WSDL, but WS-Policy and WS-Inspection also fall into this category.

Service discovery protocols on the other hand are used to make services available to potential service consumers. UDDI is the most used and most mature standard in this category.

4.6.1 Web Services Description Language (WSDL)

The Web Services Description Language (WSDL) [23] is an XML language for describing Web services. The current version is 2.0, available as a W3C recommendation. Since XML is used, Web service definitions can be mapped to any implementation language, platform, object model, or messaging system. The specification defines a core language which can be used to describe Web services based on an abstract model of what the service offers. It also defines the conformance criteria for documents in this language.

A WSDL service description indicates how clients are supposed to interact with the described service. It represents an assertion that the described service implements and conforms to what the WSDL document describes. A WSDL interface describes potential interactions with a Web service, not required interactions. The declaration of an operation in a WSDL interface is not an assertion that the interaction described by the operation must occur. Rather it is an assertion that if such an interaction is initiated, then the declared operation describes how that interaction is intended to occur. By using WSDL, it is possible to create a formal, machine-readable description of a Web service, making it possible for clients to invoke it.

4.6.2 Web Services Inspection Language (WS-Inspection)

A short overview of WS-Inspection is given by IBM in [22]: The WS-Inspection specification provides an XML format for assisting in the inspection of a site for available services and a set of rules for how inspection related information should be made available for consumption. A WS-

Inspection document provides a means for aggregating references to pre-existing service description documents, which have been authored in any number of formats. These inspection documents are then made available at the point-of-offering for the service as well as through references, which may be placed within a content medium such as HTML.
Specifications have been proposed to describe Web Services at different levels and from various perspectives. It is the goal of the proposed Web Services Description Language (WSDL) to describe services at a functional level. The Universal Description, Discovery, and Integration (UDDI) schema aims at providing a more business-centric perspective. What has not yet been provided by these proposed standards is the ability to tie together, at the point of offering for a service, these various sources of information in a manner which is both simple to create and use. The WS-Inspection specification addresses this need by defining an XML grammar which facilitates the aggregation of references to different types of service description documents, and then provides a well defined pattern of usage for instances of this grammar. By doing this, the WS-Inspection specification provides a means by which to inspect sites for service offerings.

Repositories already exist where descriptive information about Web services has been gathered together. The WS-Inspection specification provides mechanisms with which these existing repositories can be referenced and utilized, so that the information contained in them need not be duplicated if such duplication is not desired.

4.6.3 Web Services Policy Framework (WS-Policy)

WS-Policy is an important framework for introducing policy support to Web services. The framework is supported by several companies, such as IBM, BEA Systems, Microsoft, SAP AG, Sonic Software, and VeriSign [29]: WS-Policy provides a flexible and extensible grammar for expressing the capabilities, requirements, and general characteristics of entities in an XML Web services-based system. WS-Policy defines a framework and a model for the expression of these properties as policies. Policy expressions allow for both simple declarative assertions as well as more sophisticated conditional assertions. WS-Policy defines a policy to be a collection of one or more policy assertions. Some assertions specify traditional requirements and capabilities, for example, authentication scheme, and transport protocol selection. Other assertions specify requirements and capabilities that are critical to proper service selection and usage, for example a privacy policy. WS-Policy provides a single policy grammar to allow both kinds of assertions to be reasoned about in a consistent manner. For further details about WS-Policy and other security related standards, see [28].

4.6.4 Universal Description, Discovery, and Integration (UDDI)

To be able to publish Web services in a service registry, the specification Universal Description, Discovery and Integration (UDDI) [11] can be used. Basically, UDDI allows service providers to register their services and service consumers to discover these services. UDDI defines entities to describe businesses and their services, see Figure 4.2.

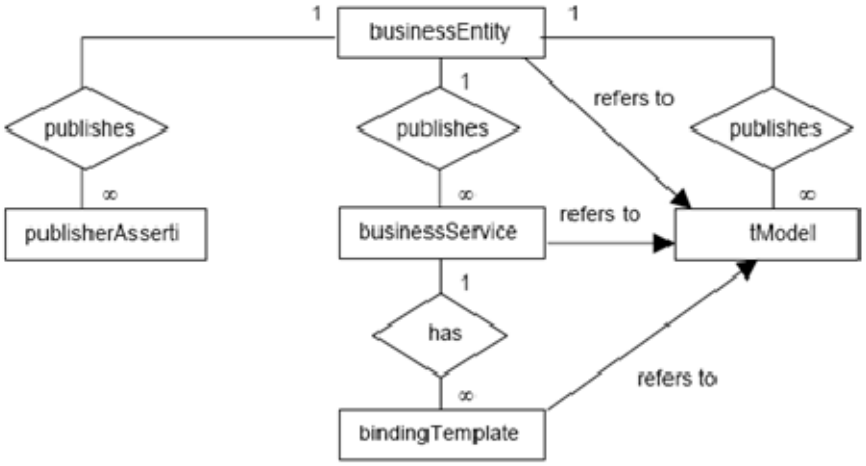


Figure 4.2 UDDI information model

UDDI is defined as a Web service itself, meaning that the SOAP protocol must be used to interact with the registry.

In principle, UDDI is centralized, but mechanisms for federating several registries have also been specified in version 3 of the specification. In this case, a root registry must be chosen, and affiliate registries may be defined as child registries of the root registry. This must be done to avoid duplicate identifiers, or keys. A replication scheme for intra-registry replication between nodes is defined, which allows for fault-tolerance inside a registry. The replication topology must be configured. UDDI has an advanced API that facilitates querying for businesses, their provided services, and technical information about these services, specified in bindingTemplates. It is also possible to subscribe to, and to be notified of changes in the registry.

All core entities in the UDDI information model are assigned unique keys. UDDI provides a flexible model in that specific service types can be registered with the registry and referenced by service instances that implement the service type. This is called a technical model, or tModel, in the UDDI information model. A tModel can be used for different purposes. For instance it can include pointers to further description of a service, for instance its WSDL description and bindings. In the UDDI specification, this is called the technical fingerprint. The tModels give flexibility, but that is also one of the drawbacks with UDDI, as proprietary use of the field can occur. Many solutions use the tModels to store such proprietary information, for example about QoS issues [13]. However, if such proprietary solutions are to work, all publishers and consumers using the registry must understand the information in the tModel and know how to handle it. Another limitation is that tModels are not stored in UDDI registries themselves. A unique identifier referencing a tModel is contained in the registries, and you need a separate repository to store the actual data in.

Furthermore, there is no uniform way of querying about services, service interfaces and classifications: the search for services restricted to Web service name and its classification. Also,

there is no liveness information in UDDI, so it is possible to get out-of-date service documents in the registries. Although UDDI is considered to be one of the core Web service standards, its adoption by enterprises has lagged behind that of the other cornerstones of Web services — SOAP and WSDL. Gartner surveys show that fewer than 10 percent of businesses use UDDI for their Web services registries [14].

As discussed above, the basic, standardized components of UDDI are lacking some important features, such as the mentioned QoS-support and liveness information. However, UDDI can be expanded with proprietary functionality in a way that such features become available. Such expansion could make UDDI better suited to NBD. For further considerations about using UDDI in NBD, see [12].

4.6.5 Web Services Dynamic Discovery (WS-Discovery)

WS-Discovery [15] is a proposal from several vendors, and addresses some of the shortcomings of UDDI. UDDI provides discovery for services that are always connected to the network, but one also needs a discovery system for services that are only connected occasionally. Also, UDDI provides discovery only for registered services, but discovery of services that do not exist in any central registry is also needed. Basically, UDDI is ok to use in static, wired networks. WS-Discovery, on the other hand, is a discovery system that can be used in ad-hoc networks.

WS-Discovery defines a multicast protocol using SOAP over UDP to locate services, a WSDL providing an interface for service discovery, and XML schemas for discovery messages. It allows dynamic discovery of services in ad-hoc and managed networks, and enables discovery of services by type and within scope. WS-Discovery leverages other Web service specifications for secure, reliable message delivery. Inherently scalability is limited due to the use of multicast, but WS-Discovery can scale to a large number of endpoints by defining a multicast suppression behavior if a service registry, i.e. discovery proxy, is available in the network. The discovery proxy is intended to be a registry for Web services (e.g. UDDI). When the discovery proxy is discovered, clients use a discovery-specific protocol to communicate with it. However, this is not a part of the WS-Discovery specification and details are left to the programmers: WS-Discovery neither defines the discovery-specific protocol nor the interaction between the WS-Discovery service and registry.

WS-Discovery is not one of the core Web services standards; in fact, it is not a standard at all. The WS-Discovery specification is provided as-is and is not a standard, so currently it is not in widespread use. However, Microsoft, BEA, Canon and Intel are contributors, and WS-Discovery is implemented in Windows Vista, so we can expect others to implement support for it in the future as well.

There are some limitations in WS-Discovery that limits its usefulness a bit: Just like UDDI, it does not provide liveness information, it does not define a rich data model for service descriptions, and it is not suitable for Internet-scale discovery.

4.7 Composition Standards

The composition standards are the standards that address the business process aspect of Web services and that add support for modeling process flow and service to service integration. There are two main types of service composition, namely orchestration and choreography. Orchestration is the composition of services controlled by one organization, and is mostly used within a business to control how the services owned by that business should cooperate. The WS-BPEL standard is the leading standard within orchestration, and adds workflow type logic to a set of service operations and also allows for maintaining limited business related state within a process. The standard is intended for use within one business, and has limited functionality when it comes to multi party collaboration.

This second type of service to service interaction is choreography, which focuses on composing more advanced services by using basic services provided by several different enterprises. Choreography is focused on interactions that occur between these enterprises rather than on enterprise internal composition. This kind of service composition is covered by the WS-CDL standard, a service choreography language, which can be used to describe collaboration interfaces between parties.

4.7.1 Business Process Execution Language (WS-BPEL)

Business Process Execution Language (WS-BPEL) is a language used for specification of both coordination protocols and composition schemas. Using a single framework is possible because similar techniques are used for the specification of both of these [32]. The BPEL coordination protocols are essentially specifications of abstract processes, that is, the external behavior of a service. The composition schemas of BPEL are specifications of the executable processes, defining the implementation logic for a (composite) service.

In general, BPEL specifications take the form of XML documents, and they define the following aspects of a process [32]:

- The different roles that take part in the message exchanges with the process.
- The port types that must be supported by the different roles and by the process itself.
- The orchestration and the other different aspects that are part of a process definition.
- Correlation information, defining how messages can be routed to the correct composition instances.

The component model of BPEL is fine-grained, and consists of activities. Structured activites are used for defining orchestration, while basic activities represent the components (that is, the operations defined by the WSDL-documents).

5 Configurable and adaptive middleware

In addition to the types of middleware described so far, there also exists a class of dynamic middleware intended for realizing so called self-adaptive systems [36]. Such systems adapt themselves to changing requirements and environments, providing dependability, robustness and availability with minimal human interaction. A self-adaptive system must support both the consistent evolution of applications over time, and adaptation of running applications, that is, a continuous process of detecting changes in the execution context, as well as planning and executing responsive actions to these changes.

While many approaches to adaptation are ad-hoc and specialized, there is consensus in the research community that general solutions, which separate and externalize adaptation mechanisms and control from the application implementation, are necessary to achieve self-adaptation [37]. Several proposed approaches indicate that middleware-based solutions are suitable for enabling such an externalization, see for instance [38] and [39].

However, this is still a relatively new area, and to the best of our knowledge, there are still mostly research prototypes to be found within this class of middleware. One example of such a research prototype is the Quality of Service Aware Component Architecture (QuA) research project. This prototype has some interesting features that may prove useful in a military context.

5.1 The QuA Project

QuA is an open, reflective component architecture, with “hooks” for QoS management mechanisms. Alternative application configurations are deployed, which provide the same functionality (i.e., the same service type), but with different QoS characteristics. When the user wants to start an application, only the application type is specified, together with specification of the desired QoS. The latter is provided as a *utility function*, which corresponds to the user’s priorities with respect to different QoS dimensions. For instance, a user may specify through the utility function that smooth playback (i.e., high frame rate) is more important than sharp pictures (high resolution) when playing back a video.

Then it is up to the platform to select the application configuration that provides the best QoS under the given conditions (i.e., resource availability, user requirements, etc), a solution called *platform-managed QoS*. This means that the platform, i.e., the QuA middleware, understands the QoS requirements of the application, and is responsible for all decisions with respect to service implementations, in order to meet these requirements.

The main motivation for this approach is that the resource situation is usually not known until an application is actually started. This is particularly so in a tactical network, where resource availability (especially bandwidth) may vary considerably. Consequently, it is not possible to configure an application at design time, since the designer has very little knowledge of the

conditions the application will run under. Instead, each implementation of a service type is described by a *service plan*, in such a way that the QuA middleware is able to choose among avaliable implementations of the service type.

One example of the usefulness of this approach in a tactical environment is a person who wants to receive live video from a UAV. When the application is started, the middleware collects information about available bandwidth, the QoS-requirements of the user, the capabilities of the terminal equipment and the characteristics of the video stream. This information is then used by the middleware to select the application (and middleware) configuration that best meets the requirements of the user. For instance, the middleware may choose to add a transcoding component in order to reduce bandwidth consumption. Furthermore, if a better network becomes available after the application has started, the middleware can choose to switch to the new network in order to improve quality.

5.2 Comparison of dynamic component-based middleware and Web services

As we have shown in this chapter, the class of dynamic component-based middleware has qualities that are well suited for SOA in tactical networks. Our main focus within NBD research is on the use of Web services, but the principles of automatic configuration and adaptation are clearly needed also in the NBD context. Therefore, we should investigate if and how the techniques and principles used in component-based middleware can be used in a Web services environment.

If we look at the four types of reconfiguration, described in Section 2.2.1.2, these are also valid for Web Services, and can relatively easily be implemented:

- *Component-internal*: adaptations and/or reconfigurations are made within a Web Service, and are thereby invisible to the clients of the service.
- *"Turning knobs"*: The Web Service offers an interface for adapting or reconfiguring the service. In other words, the client can control the adaptation.
- *Replace components*: This is just a matter of selecting another implementation of a service, i.e., another endpoint address.
- *Change composition*: This is a question of changing the composition of a set of Web Services, and should also be relatively straightforward. Web Service Composition is also a very hot topic, both commercially and in the research community.

Although Web services can be seen as yet another middleware, there are differences. One is that while other types of middleware are typically used within a local domain, Web services are used to connect such local domains over the Internet. In other words, they function as entry points into local information systems, and if we look at the problems described in Section 2.2, Web services seems like a better solution than traditional middleware. Furthermore, Web services can be viewed as an attempt to standardize middleware platforms with respect to language (XML), interfaces (WSDL), business protocols, properties, and semantics [9]. This is also an important advantage, especially in the context of international operations, as described in Section 2.2.

While a component-based middleware must instantiate components in order to activate services, a Web service usually cannot be remotely started or instantiated. If a required service is not running, either a replacement service must be found, or the service provider must be requested to start the service. In other words, the Web Services "middleware" has less control over the services than component-based middleware. This is also reflected in the ability of the middleware to control the resource usage of the services, which is important from a QoS perspective. Consequently, on the client side, monitoring and adaptation becomes more important than resource reservation. On the server side, providing services that allow the clients to specify QoS requirements would increase the possibilities of QoS management.

Another important difference is that Web services are usually more coarse-grained than components. It is reasonable to believe that this implies less control over the composition of services, but the actual consequences need to be further investigated.

Finally, an important difference is the way bindings are realized. Component-based middleware typically uses relatively tight couplings, such as Java RMI. This means that during dynamic reconfiguration of a service, such as replacing a component instance, the old instance must be unbound, and a new binding created to the new instance. In Web services, on the other hand, a binding is only an end-point address, and replacing a service therefore becomes very easy.

In dynamic environments using disadvantaged grids, it is important to reduce the amount of data traversing the network links. The available resources must be used optimally to ensure timely delivery of relevant information. Using Web services over disadvantaged grids requires some adaptation to work. In our previous research, we have experimented with various compression algorithms to reduce the inherent overhead in XML message exchange, and we concluded that data compression is one of several means necessary to make Web Services work over disadvantaged grids. In disadvantaged grids the network is the limiting factor and not the processing capacities of the nodes, so compression is beneficial and should definitely be used.

If we consider the QuA approach, it is clear that resource monitoring becomes important, since the service planner needs to have information about the resource situation in order to select the best service implementations. For the same reason, it must be possible to communicate the user QoS requirements to the service planner.

The service plans in QuA describe how to create a service through instantiation and/or composition. For Web services, instantiations are not necessary, and the service plans would therefore partly correspond to WSDL-files, and partly to a composition specification. Furthermore, the QoS-functions in the service plans may prove useful when choosing between alternative Web services, or when adapting a running service. The actual implications of this, and exactly how to take advantage of the service planning principle in a Web services context need to be investigated further. In addition, the question of to what extent Web services can be utilized in disadvantaged grids must be further investigated.

6 Summary

Interoperability is crucial when attempting to fully realize NBD, but achieving such interoperability is a considerable challenge given the large number and heterogeneity of the different systems currently being used, and the scarcity of bandwidth on the tactical level. Middleware is an abstraction layer that can conceal the heterogeneity of underlying hardware in a distributed system, and thereby represent a solution to this challenge. However, as we have described in this report, traditional middleware is not a suitable solution as infrastructure for NBD.

Web services are the most common way of realizing a SOA, which in turn is considered a very important component in the realization of NBD. Consequently, there is a strong focus both nationally and within NATO on using Web services on all levels. However, Web services are still far from being mature enough for playing the role as a full-blown “middleware of middlewares”; especially on the standardization side, much work remains. Some areas, such as interaction and description (see Figure 4.1), have standards and implementations that are well developed and interoperable. However, topics like QoS and composition are still lacking standardization and non vendor-specific implementation. Thus, it is possible to start employing the mature standards now, and modularly expand the middleware functionality as additional standards gain widespread use.

The core standards, such as XML, SOAP and WSDL are widely supported, but the sheer number of standards means that it is difficult to know which standards to use. This is made even more complex when taking into consideration the fact that the maturity of the standards vary. Some standards are fully ratified and have been released in several versions already, while others are early in their development cycle and are currently working drafts or have status as specifications. Vendor support is crucial, so looking into which standards are currently supported by the major vendors such as IBM and Microsoft can function as a guideline when trying to determine if a standard is likely to ever see widespread use. Furthermore, WS-I develops interoperability profiles and software to test an implementation’s compliance to the standard. Use of these profiles is necessary to ensure that different implementations are interoperable, and helps to alleviate some of the problems mentioned above. The WS-I basic profile helps clear up some ambiguities in some of the core standards, and defines the way a subset of these should be interoperable.

In this report, we have looked into current research within the area of dynamically configurable and adaptive middleware. One important conclusion from this research is that specification of the architecture, dependencies, and QoS characteristics should be separated from the functional code. This, in turn, leads to the problem of whether the implementation of a service and its properties can be specified in a platform independent manner. In other words, in addition to the specification of the service types, we must also be able to describe the implementations of the service types, and their properties.

We have also presented QuA, a research prototype of component-based middleware. QuA is a QoS-aware middleware that is able to understand the requirements of the application and the user, and configure both itself and the application accordingly. Although this prototype is not likely to be used in an NBD implementation, it contains some mechanisms that are very relevant, and that could prove valuable in an NBD context.

References

[1] Bakken, D. E., 2001, Middleware. *Chapter in Encyclopedia of Distributed Computing*, Urban, J. and Dasgupta, P. (eds.), Kluwer Academic Press.

[2] Vogel, A., Kerherve, B., von Bockmann, G., Gecsei, J., Distributed Multimedia and QoS: A Survey, IEEE Multimedia, Vol. 2, No. 2, 1995, pp. 10-19

[3] Johnsen, F.T., Hafsøe T., and Lund K. Quality of Service considerations for Network Based Defence. FFI/RAPPORT-2006/03859

[4] Amundsen, S. L., Lund, K., and F. Eliassen. 2006. Service Plans for Context- and QoS-aware Dynamic Middleware, *In The Second International Workshop on Services and Infrastructure for the Ubiquitous and Mobile Internet (SIUMI'06)*, IEEE, pp. 70–75

[5] Amundsen, S. L., Lund, K., Griwodz, C., and Halvorsen, P., 2005. Scenario Description –Video streaming in the Mobile Domain, *Technical report*, http://www.simula.no:8888 /QuA/uploads/2/techVScenA1.1.pdf

[6] Amundsen, S. L. and Eliassen, F. 2006. Combined Resource and Context Model for QoS-aware Mobile Middleware, In *19th International Conference on Architecture of Computing Systems*, pp. 84–98

[7] Coulson, G., Blair, G., Clarke, M., and Parlavanzas, N., 2002. The design of a configurable and reconfigurable middleware platform, *Distributed Computing Journal*, Vol. 15, pp. 109-126.

[8] Roman, M., Kon, F., and Campbell, R. H., 2001. Reflective middleware, From Your Desk to Your Hand, *IEEE Distributed Systems Online*, Vol. 2, No. 5

[9] Alonso, G., Casati, F., Kuno, H., Machiraju, V. 2004. Web Services Concepts, Architectures and Applications, Springer

[10] Web Services in networks with limited data rate (in Norwegian)
Dinko Hadzic, Trude Hafsøe, Frank T. Johnsen, Ketil Lund, Kjell Rose
FFI/RAPPORT-2006/03886

[11] OASIS, "UDDI Version 3.0.2.", http://uddi.org/pubs/uddi_v3.htm

[12] T. Gagnes et al, “An Architecture for Service Discovery in a Network Based Defence”, FFI-Notat-2006/00115

[13] Min Tian, “QoS integration in Web services with the WS-QoS framework”, Freie Universität Berlin 2005, http://www.diss.fu-berlin.de/2005/326/indexe.html

[14] Gartner research, “Core Web Service Standard UDDI Evolves With Version 3.0.2”, February 2005, ID Number: G00126170,
http://www.gartner.com/resources/126100/126170/core_web_servic.pdf

[15] J. Beatty et al, “Web Services Dynamic Discovery (WS-Discovery)”, April 2005,
http://specs.xmlsoap.org/ws/2005/04/discovery/ws-discovery.pdf

[16] T. Hafsøe et al., “Adapting Web Services for Limited Bandwidth Tactical Networks”, 12th International Command and Control Research and Technology Symposium (ICCRTS), Newport, RI, USA, 2007

[17] OASIS WS-Notification (2006) TC
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn

[18] W3C Web Services Eventing (WS-Eventing) public draft release
http://www.w3.org/Submission/WS-Eventing/
[19] WS-BaseNotification 1.3 OASIS Standard, approved October 1st 2006
http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf
[20] WS-BrokeredNotification 1.3 OASIS Standard, approved October 1st 2006
http://docs.oasis-open.org/wsn/wsn-ws_brokered_notification-1.3-spec-os.pdf
[21] WS-Topics 1.3 OASIS Standard, approved October 1st 2006
http://docs.oasis-open.org/wsn/wsn-ws_topics-1.3-spec-os.pdf
[22] Web Services Inspection Language
http://www.ibm.com/developerworks/library/specification/ws-wsilspec/
[23] Web Services Description Language (WSDL) 2.0, W3C recommendation, June 2007
http://www.w3.org/TR/wsdl20/
[24] T. Gagnes, “A Survey of Service-Oriented Architectures, Event-Driven Architectures and the Current State of Web Services Technology”, FFI/NOTAT-2004/04264
[25] Efficient XML Interchange Working Group
http://www.w3.org/XML/EXI/
[26] Efficient XML Interchange (EXI) Format 1.0, W3C Working Draft July 2007
http://www.w3.org/TR/2007/WD-exi-20070716/
[27] SOAP Version 1.2, W3C Recommendation (Second Edition), April 2007
http://www.w3.org/TR/soap/
[28] N. A. Nordbotten, “Security in Web Services and XML”, (work in progress, to appear as FFI/NOTAT in 2008)
[29] IBM et al., “Web Services Policy Framework”, March 2006
http://www.ibm.com/developerworks/library/specification/ws-polfram/
[30] IBM et al., “Web Services Reliable Messaging”, February 2005
http://www.ibm.com/developerworks/library/specification/ws-rm/
[31] P. Bartolomasi, T. Buckman, A. Campbell, J. Grainger, J. Mahaffey, R. Marchand, O. Kruidhof, C. Shawcross, K. Veum, NATO Network Enabled Capability Feasibility Study, Version 2.0, October 2005
[32] G. Alonso, F. Casati, H. Kuno, V. Machiraju, Web Services – Concepts, Architectures and Applications, Springer-Verlag, 2004, ISBN 3-540-44008-9
[33] R.T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures”, PhD Thesis, University of California, Irvine, 2000
[34] Amit Sheth, "Web Services to Semantic Web processes: Investigating Synergy between Practice and Research" (Keynote Address), The First European Young Researchers Workshop on Service Oriented Computing, April 21-22 - 2005, Leicester, U.K.
[35] Web Services Interoperability Organization (WS-I)
http://www.ws-i.org/
[36] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. IEEE Intelligent Systems, 14(3):54-62, 1999

[37] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer, 37(10):46-54, 2004

[38] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fabio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and Katia Saikoski. The Design and Implementation of Open ORB 2. IEEE Distributed Systems Online, 2001

[39] L. Capra, W. Emmerich, and C. Mascolo. CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. IEEE Transactions on Software Engineering, 2003

Appendix A QuA – Quality of Service Aware Component Architecture

In this appendix, we give a detailed presentation of the Quality of Service Aware Component Architecture (QuA) research project that was introduced in Chapter 5. The description is largely built on [4], which describes a mobile version of QuA, called QuAMobile.

A.1 Service plans

In principle there is no difference between adapting an application and a middleware service, since both provide a service to their clients. The service is accessed through an interface that specifies the interaction with the functionality of the application or the middleware service. Furthermore, a service can be formed by combining functions that are implemented in self-contained software elements with defined interfaces, i.e., components in a service composition. The interface of a service is here referred to as a service type. A service plan represents an association between a service type and one implementation of that type, where a service type is either the interface of the component or the interface(s) provided to the client of a composition of components. As can be seen from Figure A.1, a service plan contains five information elements:

- *Service* is the name of the service type of which the plan specifies the implementation,
- *Implementation* specifies either component or a composition of service types,
- *Dependencies* to context elements in the environment and their properties,
- *ParameterConfiguration* lists values for configuration of the implementation, and
- *QoSCharacteristics* of the specified implementation.

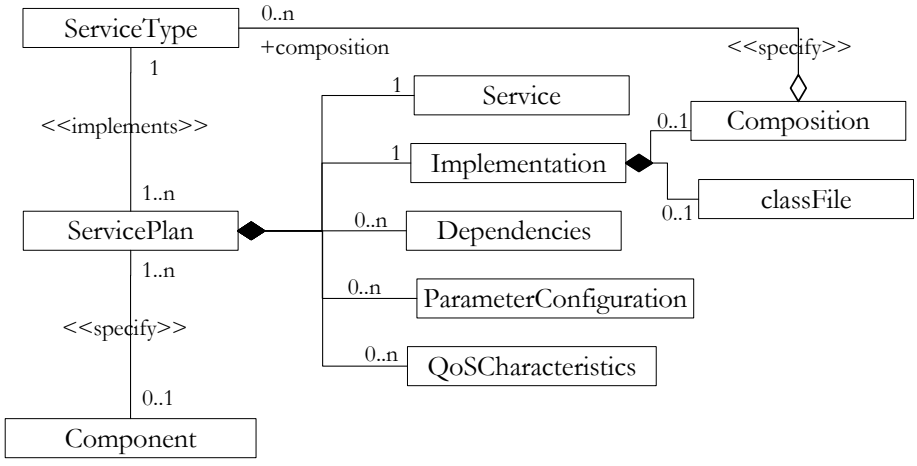


Figure A.1 Conceptual model

The association between service type and plan is one-to-many. Thus, there can be multiple alternative implementations of the same service. This one-to-many association, together with the implementation and parameter configuration information elements, makes it possible to specify

and deploy alternative configurations with different service compositions, component implementations, and parameter configurations. Another important attribute of the concept is the support for compositions of service types. A service composition can in turn be part of another service composition. The result is that the application or middleware service is specified via a recursive structure of service types.

A.2 Applying Service Plans

The service plan is defined as an element in a QoS-architecture, which is implemented in QuAMobile. The core, depicted in Figure A.2, has hooks for a set of domain-specific plug-ins, namely *service planner*, *context manager*, *resource manager*, *configuration manager*, and *reconfiguration manager*. QuAMobile takes advantage of resource and context information to select a suitable service configuration that meets the user's QoS requirements. Service types and plans are deployed alongside the components, and after loading, created components are placed in the *repository*.Service type specifications and plans are published inside the *broker*.

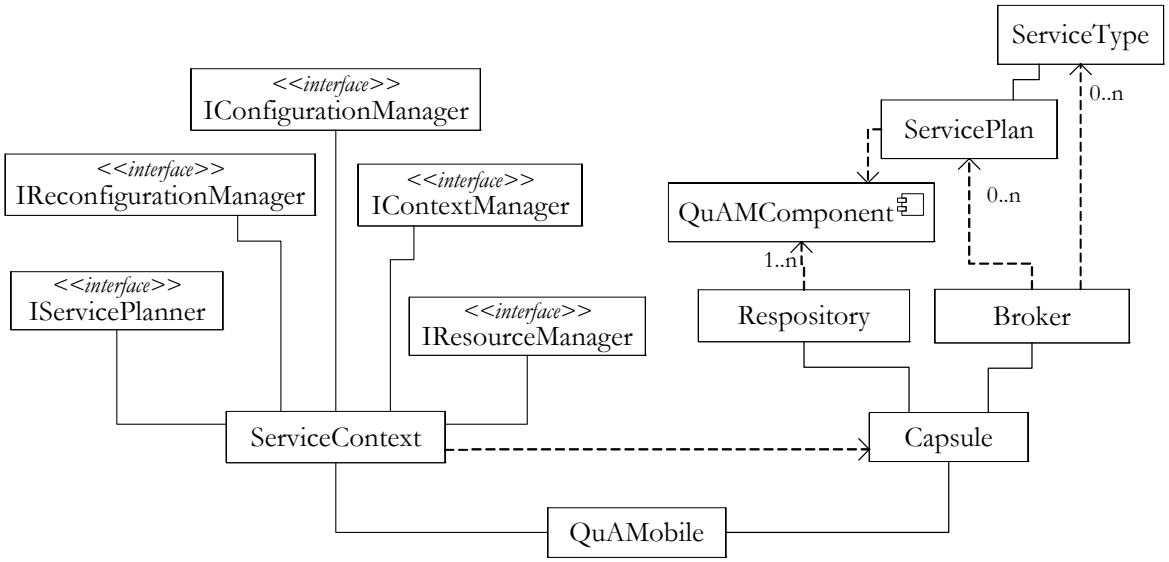


Figure A.2 Core QuAMobile

A.3 Deployment

Service plans are deployed as text files using XML, while service types are described by WSDL documents. A service plan is structured as an ordered tree, as illustrated in Figure A.4. Below the root there are child nodes that branch the tree into five branches, one for each of the information elements specified in the conceptual model, see Figure A.1. Out of these five branches, only *ServiceType* and *Implementation* are mandatory, since these are required to be able to publish the service. The *QoSCharacteristic* node has four child nodes, which reflect the QoS modelling method that is applied to define the QoS characteristics of a service [5]. The implementation node is the only parent with two alternative children, *composition* or *classFile*. Composition is used when the implementation is a service composition, while class file specifies a component, illustrated in Figure A.3.

```
<servicePlan>
   <serviceType>
      <serviceName>RTP_TFRCTransport</serviceName>
      <serviceState>stateless</serviceState>
   </serviceType>
   <implementation>

<classFile>/com/streaming/RTP_TransQuAM.class</classFile>
```

Figure A.3 XML-tags for component

During deployment, service types and plans are loaded and interpreted. This functionality can be implemented in the middleware in different ways. For instance, the loader can be activated from an operations and management console and only upload service types and plans in certain catalogues. Alternatively one may choose to confine loading to a predetermined set of alternative service configurations, by using a configuration file for the loader. The latter is chosen in QuAMobile. During loading, service plan XML-files are interpreted and information from the tree (see Figure A.4) is extracted. In the QuAMobile work, the Java Document Object Model (JDOM) open source software has been used to create the DOM and access the data. The internal representation of the service plans is stored inside the broker, using the service type as key.

A.4 Instantiation

When a user requests access to an application, a service request with the name of the service type and the user's QoS requirements are sent from the presentation layer (Web-pages and Java applets) across to the business layer (i.e., where QuAMobile resides), and finally to the service planner. The planner uses the deployed service types and plans to identify the alternative configurations suitable for the current environment and choose the one that best meets the user's QoS requirements.

For synthesising the alternative configurations from the information inside the service plans, the *service configuration* class (see Figure A.5) is introduced. The planner asks the service configuration to resolve itself, using the type of the requested service, together with a plan for the type, as input. The result is one service configuration object for each alternative configuration of the application.

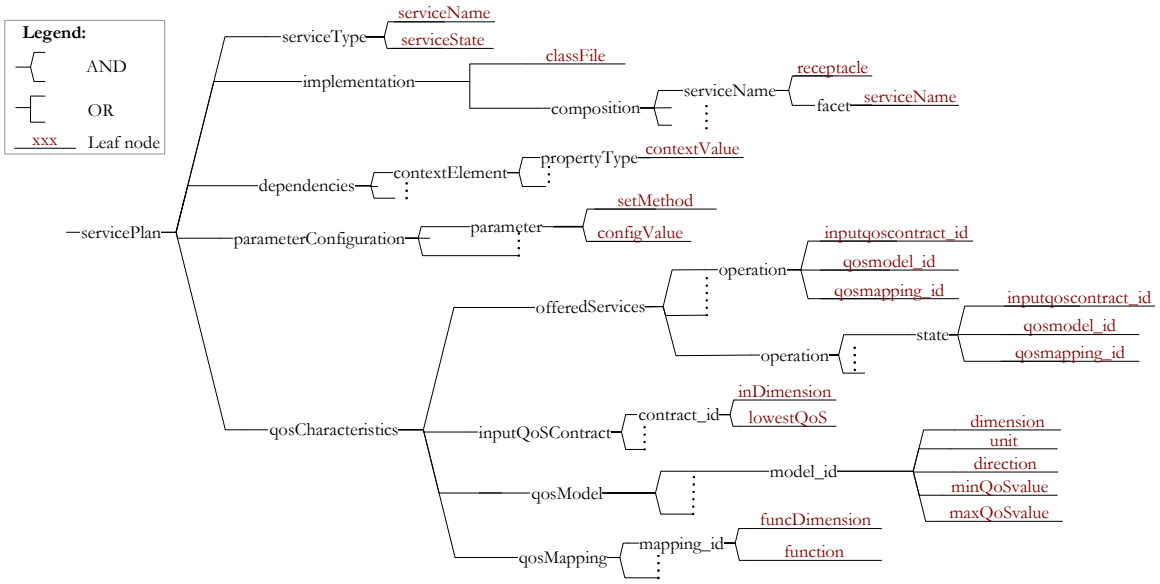


Figure A.4 Ordered tree structure

In case of a service composition, the service configuration analyses the connections between the *receptacle* and *facet* ports of the service types. From this the service configuration creates the *nextLevel* of service types and service plans, as shown in Figure A.5.

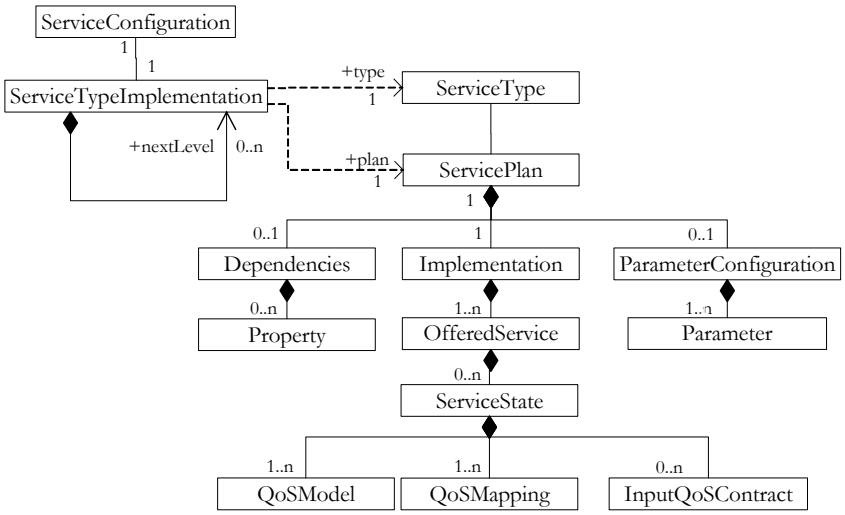
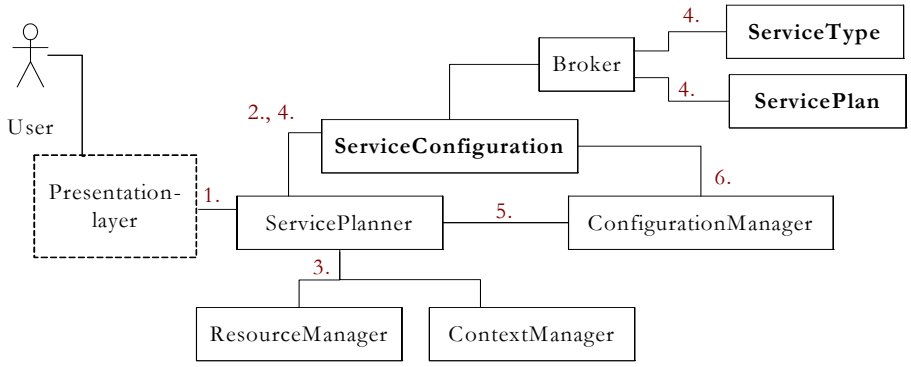


Figure A.5 Service configuration

Synthesized service configurations that can not execute in the current environment are filtered, by checking the specified context dependencies against context information from the shared data model [6] of the resource and context manager plug-ins (in the core of QuAMobile). Next, the service planner compares the service configurations, by predicting end-to-end QoS. This is done by calculating QoS characteristics at each level inside the service configurations, and comparing the results for the different (user) QoS dimensions. The chosen service configuration is then forwarded to the *configuration manager*. The last task of the service configuration object and the service plans is to provide the configuration manager with a list of the components to create and

the receptacle and facet connections between them (also called local bindings). Figure A.6 summarizes the interaction between the middleware, service configuration, and service plans.



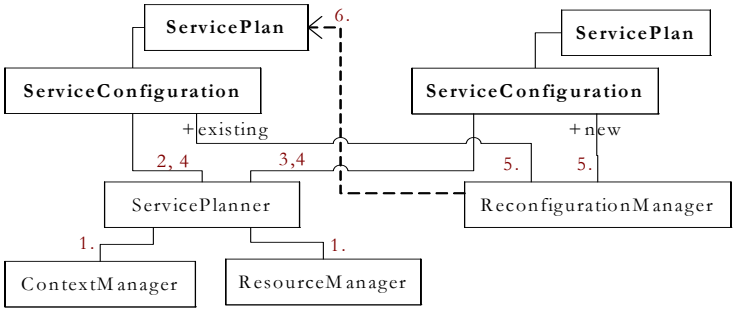
1. Service request for a service
2. Synthesize alternative implementations of the service, i.e., service configurations
3. Fetch resource and context information for the current environment
4. Filter, compare, and choose service configuration
5. Forward chosen service configuration to configuration manager
6. The service configuration provides a list of components to create and connections between ports

Figure A.6 Instantiation

A.5 Reconfiguration

For reconfiguration QuAMobile utilizes the reflection concept [7][8]. This concept introduces a meta-level with application models that describe the running application. The model at the meta-level is casually connected to the application, so any changes made to the meta-level causes corresponding changes in the application. The causal connection is enforced by the middleware.

After instantiation, the selected service configuration, together with the associated service plans, change roles from specifying one possible configuration of the application to specifying the architecture, behaviour, parameter configuration, dependencies, and QoS characteristics of the running application. Thus, during reconfiguration the service plan concept is a model at the meta-level of the application configuration. For reconfiguration this is useful, since it makes the implementation open for inspection without having to involve the components. When QuAMobile detects changes in the environment, the service planner assesses the consequences of the changes by asking the service configuration to re-evaluate context dependencies and re-estimate the end-to-end QoS. If context dependencies or user QoS requirements are violated, the service planner will start to re-plan the service. If one of the other service configurations better meet the users' QoS requirements, both this new and the existing service configurations are forwarded to the reconfiguration manager. It uses the service plans in the existing configuration to get the references (local and remote) to existing components, and then the compare() operation on the new service configuration to identify which components to delete or create, and where to set local bindings. Figure A.7 illustrates the steps in which service plans are involved during reconfiguration.



1. Detect changes in the environment
2. Check for violation of context dependencies and user QoS
3. (If violation) synthesize, filter, compare, and choose an alternative service configuration (as in Figure A.6)
4. Compare existing and new service configurations
5. (If new is better) identify components and connections for reconfiguration
6. Structural reflection on existing plans

Figure A.7 Reconfiguration