# Security and Middleware

Anders Andersen, Gordon Blair,[*] Per Harald Myrvang, and Tage Stabell-Kulø
Department of Computer Science
University of Tromsø, Norway
{aa,gordon,perm,tage}@cs.uit.no

## Abstract

*The security features of current middleware platforms, like Enterprise Java Beans and CORBA, are either simple and limited or complex and difficult to use. In both cases are the provided features static and do not support the flexibility needed in a wide range of applications. This paper introduces an approach to flexible security mechanisms in the context of a reflective middleware architecture. The reflective middleware OOPP is a component and capsule (container) based platform providing its reflective features through a set of distinct meta-models. Flexible security mechanisms are provided using a specialized programming language called Obol. In OOPP the flexible security mechanisms based on Obol is a subset of reflective features of the middleware platform. Obol and its machinery is a subset of one distinct aspect or meta-model of the middleware platform.*

## 1. Introduction

Middleware is used to help programmers to create distributed (and complex) applications [2, 6]. It presents a set of useful programming abstractions hiding the details of distributed programming (i.e. providing transparencies). The consequence is that the middleware providers have made a lot of decisions on behalf of the programmer about the behavior of the programming abstractions. This is also true for the security features. The result is that a given middleware platform, either provides a simple programming abstraction for its not-so powerful security features, or it provides a complex set of programming abstractions for its powerful security features.

The first approach with simple security programming abstractions is easy to use, but the provided security features might not be powerful enough for a given application. For example, an Enterprise Java Beans container [14, 16] can have access control lists (ACLs) for provided interfaces. Users listed in an ACL are identified and authenticated by user names and passwords. Such an approach has some limitations. It is difficult to delegate access rights, either to users as such or to composite users acting in specific rôles. In particular, Bob *as* Manager and Alice *as* Manager should be two different *users* and not only Manager. Another limitation is how to integrate such a solution into an existing security infrastructure (i.e. Kerberos).

[*] Distributed Multimedia Research Group, Lancaster University, UK.

The second approach with a complex set of security programming abstractions is more complex to use and it introduces some extra overhead. This complexity and overhead might be present even if the programmer does not use the most complex security features provided. The CORBA security reference model presented in [12] is complex to use and implement correctly. The provided protocols and mechanisms are a challenge to use understand even by experienced programmers.

Still, a lot of the decisions about the security features provided are made by the middleware provider (or the middleware standard committee).

## 2. Reflective middleware

Middleware has a similar role in a distributed system that the operating system has on a computer: to hide the low level details and present a unified programming model. Middleware and operating systems also share common problem related to their usage. The problem was stated clearly by Per Brinch Hansen [5]:

> *One of the difficulties of operating systems is the highly unpredictable nature of the demands made upon them.*

Middleware has to remain responsive to new challenges and demands from existing and new type of applications. Some of these new difficulties that are emerging are (i) support for multimedia, (i) real-time requirements, and (iii) increasingly mobility. These different and unpredictable chal-

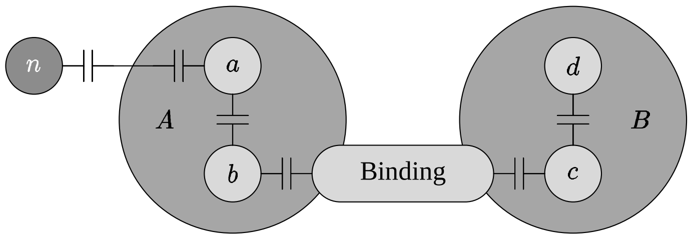

**Figure 1. An example with components and two capsules $A$ and $B$.**

lenges are the motivation for reflective middleware architectures [4]. One key feature to meet these challenges is adaption. Randy H. Katz summaries the requirement for adaption with mobility in the following observation [8]:

> *Mobility requires adaptability. By this we mean that systems must be location- and situation-aware, and must take advantage of this information to dynamically configure themselves in a distributed fashion.*

It should be possible to configure the underlying support offered by the middleware platform to satisfy the requirements from a wide variety of applications. Example of such configurations are scheduling policies, special protocols for multimedia, resource management, and security policies. Another important requirement is the possibility to inspect and adapt the support offered at run-time. This can be done done by adapting an open engineering approach through the concept of reflection.

Current generation middleware only have limited (if any) support for configurability and open engineering. Implementation details are hidden and services are available through a set of interfaces (APIs) to a black box. There are several good reasons for doing this[1] but recent experiences with these platforms suggest that this is to restrictive to a lot of application types. OMG has as a result of this recently added some interfaces to the underlying system i CORBA [11]. But these approaches provide limited openness to a limited set of selected components and they are rather ad hoc.

## 2.1. OOPP

OOPP (Open-ORB Python Prototype) is a prototype of the Open-ORB architecture [4, 3] adding features for quality of service management [1]. The programming model of OOPP are influenced by the ISO Reference Model for Open Distributed Processing (RM-ODP) [7]. RM-ODP provides a rich vocabulary and grammar for describing a distributed system, including implicit and explicit bindings, different types of interfaces, composite components, naming service, and capsules. The capsule is the runtime of OOPP components. It manages and provides services to its local components. Figure 1 is an example of some components in the OOPP programming model. The smaller light grey circles are components and the two large circles are capsules. The small T-shaped attachments to the components are interfaces. Interfaces are connected with local bindings (e.g the local binding between an interface of component $a$ and an interface of component $b$). Component $a$ has an implicit binding to a name server $n$ and component $b$ and $c$ are connected with an explicit binding. This binding is a distributed (and composite) component. The binding and the other components are connected with local bindings (two connected interfaces).

OOPP (and Open-ORB) tries to overcome the limitations in current middleware platforms by opening up the ORB implementation. This is done through the concept of reflection [15]. Access to the implementation is provided through a distinct set of meta-models [13]. Each meta-model provides a meta-object protocol (MOP) [9] used to inspect and manipulated the part of the implementation exposed through this meta-model. The *encapsulation* meta-model provides access to the implementation of OOPP components and interfaces. It can be used to inspect and manipulate their implementation including adding new methods to components and interfaces, installing pre- and post-functions on methods, and changing the implementation (class) of components. The *composition* meta-model provides access to the component graph representing a composite component. It can be used to inspect and manipulate this graph. The *environment* meta-model provides access to the mechanisms and policy for queing, synchronization, scheduling, and dispatching messages (method calls). The *resource* meta-model provides access to the allocation and management of resources associated with a component or an interface. Figure 2 illustrates a component and its four meta-models.

[1]The main reason to provide a middleware platform as a black box with a set of predefined interfaces is to hide the complexity of the underlying system for the application programmer. For a lot of traditional client-server based applications is this hiding of complexity (transparency) a good thing.

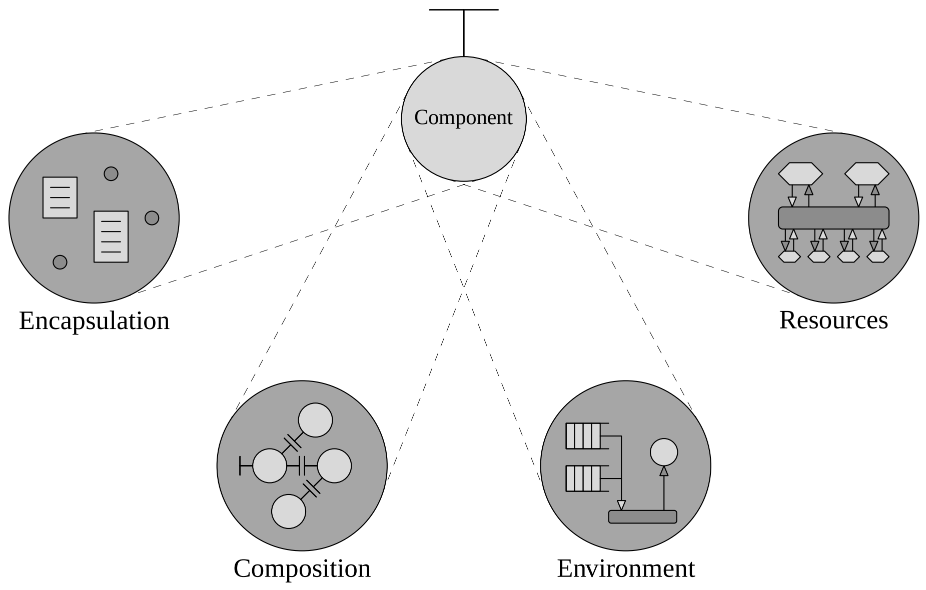

**Figure 2. The four meta-models of a component**

## 3. Programmable security

Three observations about security are important: (i) security is not an add-on feature, (ii) security features are complex to understand and implement, and (iii) the need for security differs a lot depending on the environment and the application. The consequence of observation (i) is that security has to be taken into consideration from the beginning and in all aspects during the design and the implementation of a new middleware platform. The consequence of observation (ii) is that the provided abstractions for security should be expressive and hide details when possible. The consequence of observation (iii) is that the security mechanisms should be flexible and hidden details might have to be exposed and manipulated.

These three observation taken together should warrant for a new approach to security. The very complex but still very rigid security infrastructure offered by CORBA [12] demonstrates that the real problem with security in infrastructure is not the richness, but rather the lack of programmability. To that end we believe that *programmable security* is a viable solution.

### 3.1. Obol

The programming language Obol is designed solely as a security protocol language. The language enables suppliers of software components to describe their applications security requirements in the form of a program rather than in the deployment descriptor. To see the significants of this, consider the case where a third party supplier changes the protocol he requires his customers to use. To continue to use the service, the software components must either be reimplemented with the new protocol, or a container must be augmented with the new protocol. Both are major undertakings, particularly from a logistic point of view. In Obol the provider can specify the protocol as part of the certificates he issues to users (e.g. SSL). Obol is a *safe* language in the sense that it can be executed without risk of compromising the container, the performance is good (it can be compiled to machine code), and since it is geared towards a specific purpose, it is powerful and programs are short.

```
(believe $P name self)
(believe $Q name "...")
(believe $K shared-key 0x12345...)
(generate $N nonce 128)
(send $Q $P $Q $N)
(receive $Q $Q $P (decrypt $K $N *1))
(send $Q $P $Q *1)
(return t)
```

**Figure 3. An Obol program.**

Figure 3 is a short example (less than 200 bytes) of a protocol used to establish the fact that the other party is present (on line). We will not go into details about how this protocol works, the correctness of it, and the server side of the protocol. The purpose of this example is to give the reader an idea about what an Obol program is.

Variables are identified by having a '`$`' prefix. Anonymous (type-less) variables have a '`*`' prefix and are identified by a number. A variable of the type `name` includes an address (enough information to locate whatever the variable is referring to). A variable of the type `shared-key` is a shared encryption and decryption key. The first four lines in the example above creates four variables. `$P` is given the local name (`self` holds system specific information about the local user and her environment), `$Q` is given the name of the other party, `$K` is a key shared between `$P` and `$Q`, and `$N` is a 128 bits nonce generated at runtime. In line 5 a message containing the two names and the generated nonce is sent to `$Q` (the first argument of `send` is the receiver). In the next line a message containing the two names and a block encrypted with the key `$K` is received. The encrypted block should include the nonce `$N` and an anonymous variables given the name `*1`. In line 7 a message containing the two names and the received anonymous variable `*1` is sent back to `$Q`.

Most existing protocol languages focus on verification of the protocols the languages describe. Obol is a language tailored for the implementation of network security protocols. The primitives in Obol are geared specifically towards cryptographic or security protocols. These protocols use cryptographic machinery to establish certain properties of messages. These properties can be integrity, secrecy and origin. Authentication protocols can for example be used to exchange a session key between two parties, to establish mutual authentication, to establish the presence of a participant, or all three at the same time [10].

Three concepts are important in the design of Obol: (i) Security protocols are often described in terms of message passing. In Obol no interesting assumptions are made about how send and receive of messages are actually done. (ii) Security protocols achieve their goals by the beliefs that are assembled by reception of messages with content that is expected. Beliefs stem from two sources: assumptions and new beliefs. Both are made explicit visible in Obol programs. (iii) Security protocols deal to a large extent with random material such as keys an nonces. Obol provides primitives to generate fresh materials and locate keys that are already known.

## 4. Obol in OOPP

Obol provides a language to implement (security) protocols. These protocols (or Obol programs) have to be interpreted or executed in a runtime. This runtime is called Lobo. In OOPP, Lobo is included in the capsules. The MOP of the environment meta-model is used to access and install Obol programs in Lobo. Figure 4 illustrates Lobo in the environment meta-model of OOPP.

A Obol program installed in Lobo provides an implementation of a given security protocol. This can be existing protocols like SSH, SSL or TSL, or it can be special purpose protocols only used in a limited application domain.

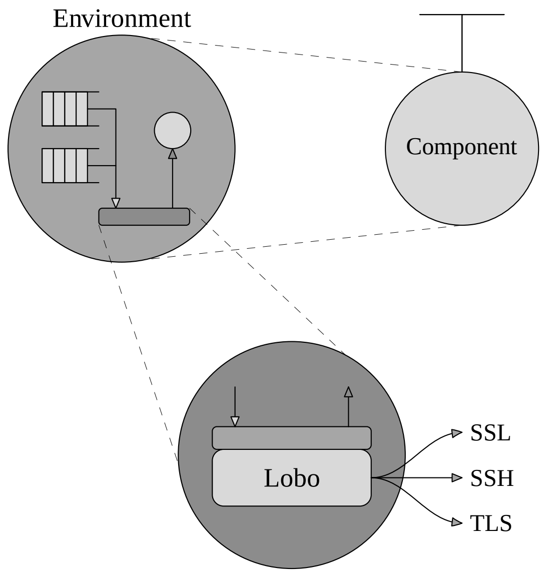

**Figure 4. Lobo in OOPP**

An example could be a component that needs to access an external service that has to be accessed in a secure manner. The client side of the security protocol for this service is installed in Lobo and the application access the service in the same way it access the interface of other (non-secure) services. Lobo performs the actual security protocol, and the security protocol is not exposed at the business logic level.

Another example involves two OOPP capsules. Two OOPP components located in different capsules have to interact in a secure manner. Component $C$ in capsule $A$ is a client accessing the services of component $S$ in capsule $B$. The client side of the security protocol is installed in Lobo in capsule $A$. The server side of the security protocol installed in Lobo in capsule $B$. Figure 5 illustrates this setup. The environment meta-models of component $C$ and $S$ are exposed to show the communication between the client and server side of the security protocol.

## 5. Conclusion

The flexibility provided by the reflective middleware platform OOPP is a perfect match for programmable security of Obol. Reflection provides the mechanisms needed to access and modify the environment of the software components of a given application. In OOPP the runtime of Obol called Lobo is accessed through the environment meta-model. The environment meta-model MOP is used to install and manage Obol program i Lobo. This makes it possible to change and replace security protocols used without changing the business logic of the given application, and without

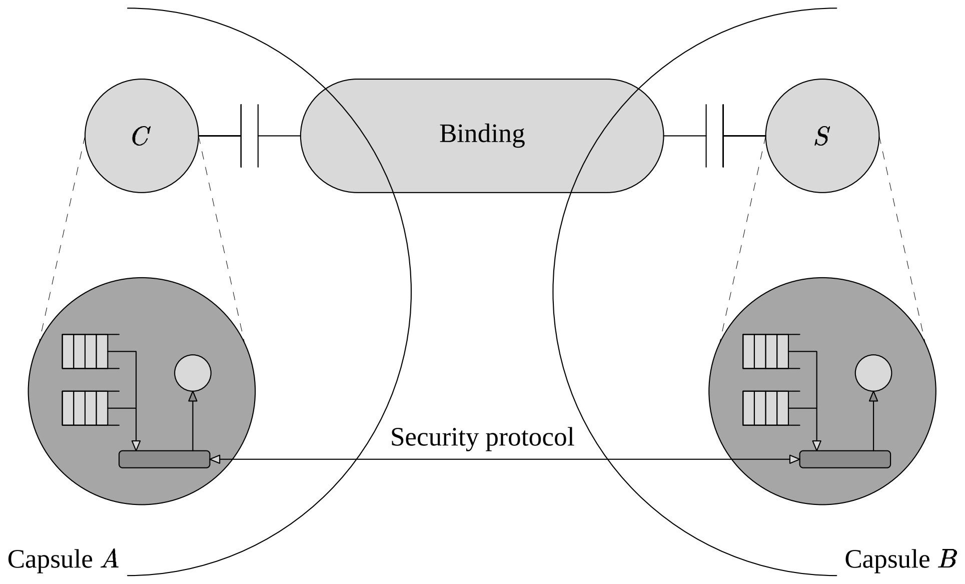

**Figure 5. Security protocol between OOPP capsules**

changing the implementation of the middleware platform itself.

Obol is a high-level language. Writing protocols is such a high-level language is probably less error-prone than writing them in a low-level language like Java and C++. It also makes it possible to upgrade from one version of a protocol to another without starting a major implementation effort.

Obol implements a set of cryptographic primitives. The quality of such code must be very high. By providing such primitives in a high-level language we centralize the code that implements these primitives. We belive that it should be possible for a programmer using Obol to apply potentially complex cryptographic protocols to a system without having to embark on the implementation endeavor necessary to implement every detail of the protocols.

## References

[1] A. Andersen. *OOPP, A Reflective Middleware Platform including Quality of Service Management*. Dr. sci. thesis, Department of Computer Science, University of Tromsø, Tromsø, Norway, Feb. 2002.

[2] P. A. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2):86–98, Feb. 1996.

[3] G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. B. Saikoski. The design and implementation of Open ORB 2. *IEEE Distributed Systems Online*, 2(6), 2001.

[4] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Middleware'98*, Sept. 1998.

[5] P. Brinch Hansen. *Operating System Principles*. Prentice Hall Series in Automatic Computation. Prentice Hall, 1973.

[6] W. W. Eckerson. Three-tier client/server architecture. *Open Information Systems*, 10(1):3–22, Jan. 1995.

[7] ISO/IEC. Open distributed processing reference model, part 1: Overview. ITU-T Rec. X.901 | ISO/IEC 10746-1, ISO/IEC, 1995.

[8] R. H. Katz. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1(1):6–17, First Quarter 1994.

[9] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[10] A. Liebl. Authentication in distributed systems: A bibliography. *ACM Operating Systems Review*, 27(4):31–41, Oct. 1993.

[11] Object Managment Group. The common object request broker: Architecture and specification. Technical report, Object Managment Group, Feb. 2001. (revision 2.4.2).

[12] Object Managment Group. Security service specification. Technical report, Object Managment Group, Mar. 2002. (version 1.8).

[13] H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/D: A distributed programming system with multi-model reflection framework. In *Proceedings of the Workshop on New Models for Software Architecture*, Nov. 1992.

[14] B. Roth. An introduction to enterprise JavaBeans technology. Technical report, Java Software, Sun Microsystems, Oct. 1998.

[15] B. C. Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1982.

[16] Sun Microsystems. Simplified guide to the Java 2 platform, enterprise edition. Technical report, Sun Microsystems, Inc., 1991.