

Reflective Middleware and Security: OOPP meets Obol*

Anders Andersen, Gordon Blair[†], Tage Stabell-Kulø,
Per Harald Myrvang and Tom-Anders Nilsen Røst
Department of Computer Science
University of Tromsø, Norway

{aa,gordon,tage,perm}@cs.uit.no, <http://abean.cs.uit.no/>

Abstract

The manner in which one can apply the security features of current middleware platforms, like Enterprise Java Beans and CORBA, are either too simple and limited or too 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 presents 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 the reflective features of the middleware platform. In particular 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 [1, 2]. 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 [3, 4] 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, it must be possi-

ble to assign different access rights to the two principals Bob *as* Manager and Alice *as* Manager without having to create new users in the system [5]. Another limitation is how to integrate such a solution into an existing security infrastructure (i.e. Kerberos) or take new technology in use within the existing framework (such as smartcards or PKI).

The second approach uses a large and complex set of security programming abstractions. 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 [6] is hard to implement correctly, and subsequently also complex to use. The security subsystem is a challenge to understand and use even by experienced system designers.

2 OOPP

OOPP (Open-ORB Python Prototype) is a prototype of the Open-ORB architecture [7, 8] adding features for quality of service management [9]. The programming model of OOPP is influenced by the ISO Reference Model for Open Distributed Processing (RM-ODP) [10]. RM-ODP provides a rich vocabulary and grammar for describing a distributed system, including implicit and

*This work has been supported by the Norwegian Research Council (IKT 2010, project number 146986/431).

[†]Also at Distributed Multimedia Research Group, Lancaster University, UK.

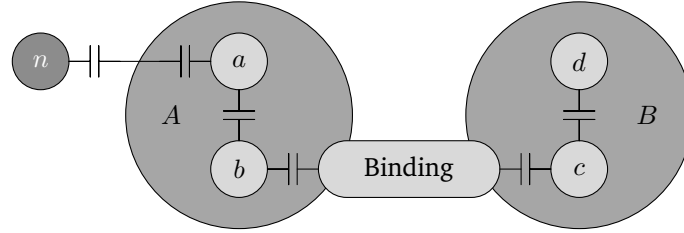


Figure 1: An example with components and two capsules A and B .

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 [11]. Access to the implementation is provided through a distinct set of meta-models [12]. Each meta-model provides a meta-object protocol (MOP) [13] 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 queuing, 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.

3 Programmable security

Experience shows that well designed security regimes entrench the systems they are part of, and few aspects of a system are left unaffected. Experience also shows that the better the security needs of the system are understood, the “better” the design of the security infrastructure will be. The fundamental problem when security mechanisms are designed for middleware systems is that the mere existence of middleware is in contradiction to the lessons learned. In particular, the actual security needs of applications vary widely, not only between applications, but also over the different invocations of an application.

This, of course, is the same line of arguments, albeit in a different setting, that is used to support reflective middleware. We believe that the most viable response is the same as for middleware system: *programmable security*.

3.1 Obol

The programming language Obol is designed solely to program security protocols. The language enables suppliers of software components to augment their applications security requirements in the form of a program rather than requesting it in the deployment descriptor. In particular, the protocol can be supplied or changed after the application itself has been deployed. To see the significance of this, consider the case where a third party supplier changes the protocol she requires her customers to use. To continue to use the service, the software components must either be re-implemented 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.

Obol is a *safe* language in the sense that it can be executed without the risk of compromising the container,

```

1 (believe $P name self)
2 (believe $Q name "...")
3 (believe $K shared-key 0x12345...)
4 (generate $N nonce 128)
5 (send $Q $P $Q $N)
6 (receive $Q $Q $P (decrypt $K $N *1))
7 (send $Q $P $Q *1)
8 (return t)

```

Figure 2: An Obol program.

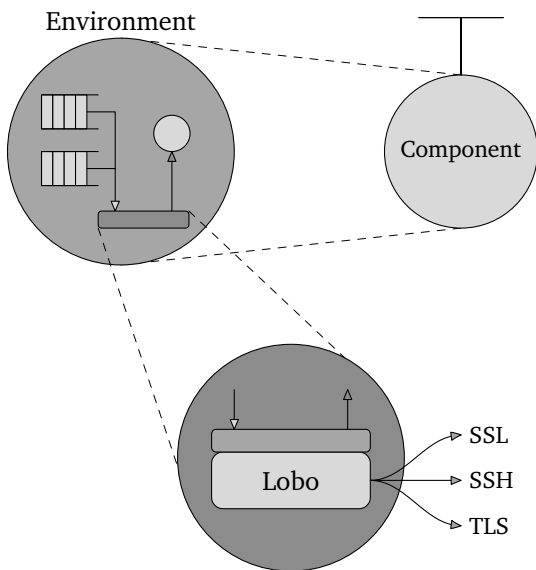


Figure 3: Lobo in OOPP

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. The foremost advantage of using a programming language is that it makes it possible to both impose and benefit from constraints. A language, as opposed to an API, provides both the means to restrict the design choices available to the programmer and at the same time create a setting where the full advantages can be taken in terms of performance and safety.

Figure 2 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 an idea about what an Obol program is.

Variables are identified by having a '\$' prefix. Anonymous (type-less) variables have a '*' prefix. 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 can hold a shared encryption and decryption key (which represents a communication channel). 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.

Obol has been designed for the exploration of replaceable and programmable protocols in the setting of reflective middleware. Sometimes the Obol implementation of the corresponding functionality of a given protocol could be (syntactical) incompatible with the existing protocol implementation, but we believe that Obol can be used to implement any communication semantics.

4 Obol in OOPP

The Obol programs (i.e. protocols) have to be interpreted or executed in a runtime. This runtime is called Lobo. In OOPP, Lobo is included in capsules. An Obol program installed in Lobo provides an implementation of a given security protocol. The MOP of the environment meta-model is used to access and install Obol programs in Lobo. Figure 3 illustrates Lobo in the environment meta-model of OOPP.

As an example of how Obol can be applied, consider a component that needs to access an external service in a secure manner. The client side of the security proto-

```

1 c_environment = environment(c)
2 s = c_environment.obol.add(p)
3 c_environment.obol.run(s)

```

Figure 4: Installing Obol program p .

col for this service is installed in Lobo through the environment meta-model. The application accesses the service in the same way it accesses 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. Figure 4 lists the Python used code to install this client side security protocol p in the component c (c is a binding to the external service). Line 1 is used to get access to the environment meta-model of component c . In line 2 and 3 the Obol program p is installed in the Lobo runtime and then started.

Another example involves two components hosted in two different OOPP capsules. Component C in capsule A acts as a client accessing the services of component S in capsule B . The client side of the 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.

In a setting such as this one, the protocol by which the service is provided need not be part of the client; the client can inquire the server about which protocol to use, and the install it on-the-fly.

5 Conclusion

The flexibility provided by the reflective middleware platform OOPP is a perfect match for the 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 programs in Lobo. This makes it possible to change and replace security protocols used without changing the business logic of the given application, and without changing the implementation of the middleware platform itself.

Obol is a high-level language. Writing protocols in such a high-level language is less error-prone than writing them in a low-level language like Java and C++.

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 the runtime of a high-level language we centralize the code. We believe 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.

6 Acknowledgments

The research described in this paper has been supported by the Norwegian Research Council (IKT 2010, project number 146986/431). We would also like to acknowledge the contributions from the other members of the Arctic Beans project, the Arctic Beans cooperative projects, and the Arctic Beans industrial partners. Particular thanks to Gianluca Dini, Randi Karlsen, Weihai Yu, Andrea Bottoni, Vegard Bønes, Feico Dillema and Ragnhild Varmedal.

References

- [1] Philip A. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2):86–98, February 1996.
- [2] Wayne W. Eckerson. Three-tier client/server architecture. *Open Information Systems*, 10(1):3–22, January 1995.
- [3] Bill Roth. An introduction to enterprise JavaBeans technology. Technical report, Java Software, Sun Microsystems, October 1998.
- [4] Sun Microsystems. Simplified guide to the Java 2 platform, enterprise edition. Technical report, Sun Microsystems, Inc., 1991.
- [5] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. SRC Research Reports 83, DEC’s System Research Center, February 1992.
- [6] Object Management Group. Security service specification. Technical report, Object Management Group, March 2002. (version 1.8).

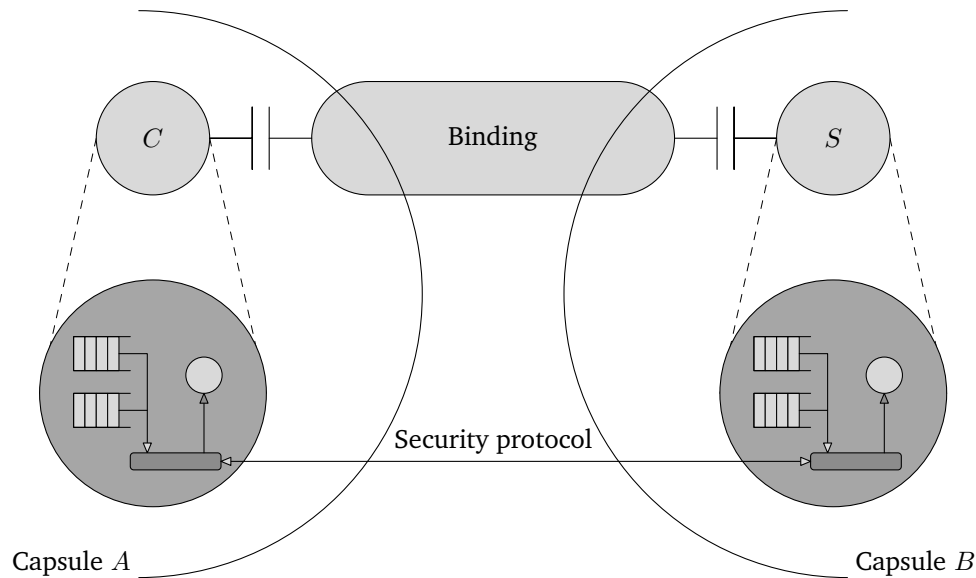


Figure 5: Security protocol between OOPP capsules

- [7] Gordon S. Blair, Geoff Coulson, Philippe Robin, and Michael Papathomas. An architecture for next generation middleware. In *Middleware'98*, September 1998.
- [8] 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 Kattia B. Saikoski. The design and implementation of Open ORB 2. *IEEE Distributed Systems Online*, 2(6), 2001.
- [9] Anders Andersen. *OOPP, A Reflective Middleware Platform including Quality of Service Management*. Dr. sci. thesis, Department of Computer Science, University of Tromsø, Tromsø, Norway, February 2002.
- [10] ISO/IEC. Open distributed processing reference model, part 1: Overview. ITU-T Rec. X.901 — ISO/IEC 10746-1, ISO/IEC, 1995.
- [11] Brian Cantwell Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1982.
- [12] 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*, November 1992.
- [13] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.