The Role of Reflective Middleware in Supporting Flexible Security Policies

Na Xu[1], Gordon S. Blair[1], Per Harald Myrvang[2], Tage Stabell-Kulø[3], Paul Grace[1]

[1]Computing Department, Lancaster University, Lancaster, UK
[2]Bodø Graduate School of Business, Bodø University College, Bodø, Norway
[3]Department of Computer Science, University of Tromsø, Tromsø, Norway
n.xu@lancaster.ac.uk, gordon@comp.lancs.ac.uk

Abstract. Next generation middleware must support applications in the face of increasing diversity in interaction paradigms, end system types and network styles. Therefore, to secure applications, security policies must be configured and indeed reconfigured at runtime. In this paper, we propose an approach combining the openness of reflective middleware with the flexibility of programmable security to meet such demands. In particular, we build a security architecture based on the Gridkit reflective middleware platform and the Obol security protocol programming language. The paper then describes a case study that uses flexible policies in order to secure remote procedure calls and secure group communication. We also present the benefits of this approach in terms of flexibility, ease of use and extensibility.

1 Introduction

Developing middleware that can support secure distributed applications is an increasingly difficult task. Computing paradigms such as the Grid, and mobile/ ubiquitous computing all add to the increasing diversity in terms of interaction paradigms, end system types and underlying network styles; therefore, enforcing an appropriate security mechanism in these highly heterogeneous environmental conditions is becoming more challenging. We now analyse how this diversity impacts on security:

– *Varied interaction paradigms*. The development of distributed systems can involve a wide range of interaction styles including: RPC, multicast-based group communication, publish-subscribe, media streaming, and many others. However, security mechanisms developed for the traditional client-server model do not necessarily fit the other interaction styles; experience has shown that there are distinct differences in both the communication models and the security requirements. For example, in the group communication model (e.g. videoconferencing and white-board applications), the group membership can change very often. These fluctuations require updating a group key, making key management more complicated.

– *Different end systems*. Devices can range from: workstations, PCs, laptops to resource-poor and low-speed PDAs and sensors. It is difficult for every device type to support all security policies; for example, the cost of encryption and the processing of some security protocols may exhaust resource-impoverished devices.

Hence, we believe that *flexible security policies* are necessary to dynamically adapt to the divergent application environments. Dealing with flexible security policies will be a fundamental challenge in the development of future middleware solutions. However, traditional middleware platforms typically provide static, fixed security mechanisms (e.g. CORBA, EJB). Alternatively, reflective middleware solutions provide an efficient way to support dynamic system adaptation. In addition, programmable security paves the way for addressing varied security requirements to match the heterogeneous environmental conditions because it facilitates to separate security solutions from the business logic. Given these benefits, we propose an approach to apply configurable and dynamically reconfigurable security mechanisms in middleware platforms; this involves the integration of reflective middleware, and programmable security. In this paper we illustrate this approach by developing flexible security policies using Obol [MS06], a security protocol programming language to implement security policies; and we utilise the interception meta-model supporting behavioural reflection to apply them within an existing reflective middleware, Gridkit [Gr05]. To evaluate the effectiveness of our solution, we present a case study, characterised by diversity, which demonstrates how security policies can be dynamically configured at runtime.

The novel contributions of our approach are: i) it supports the configurability and potential re-configurability of flexible security policies that target particular environments in diverse application domains; ii) it provides a high-level domain specific programmable security language for defining and implementing security policies; iii) our general interception based architecture supports dynamic loading of security policies through reflection.

The remainder of the paper is structured as follows. Section 2 introduces the reflective middleware platform Gridkit. Section 3 then describes the programmable security capability provided by Obol. Following this, section 4 describes the security requirements for diverse environmental conditions, highlights the role of Obol in expressing security policies, and details the approach to integrate flexible security mechanisms within Gridkit. Section 5 describes the development of a case study involving an RPC application and multicast-based group communication using two different device types, i.e. PC and PDA. Subsequently, section 6 discusses related work. Section 7 concludes and presents future work.

2 Reflective Middleware

2.1 Gridkit

Application domains including multimedia, mobile computing, autonomic computing, ubiquitous computing, and many others, are characterised by both diversity and change. Applications can operate on different devices, e.g. sensors, laptops, PDAs, workstations, and clusters; applications can utilise different networks, e.g. fixed infrastructure, wireless and ad-hoc networks; and applications can have very different middleware requirements, e.g. client-server, publish-subscribe, streaming media, resource discovery, etc. Hence, fixed middleware solutions are inappropriate; rather middleware must be adaptable to suit the current application's requirements in the given context, and middleware must be able to dynamically change its behaviour at run-time to manage context changes. In this section, we describe a middleware solution called Gridkit that can be configured, and reconfigured to support a wide variety of application types in highly diverse settings.

Gridkit follows the Lancaster design philosophy [Bl01] that promotes a marriage of *component technologies, component frameworks* and *reflection*. Components are the building blocks of middleware, where a component is "a unit of composition with contractually specified interfaces, which can be deployed independently and is subject to third party creation" [Sz98]. This technique promotes configurability, re-configurability and re-use at the middleware level. Component frameworks manage specific domains of middleware functionality (themselves being composed of other components and frameworks), in particular controlling the configuration and reconfiguration of the elements within. Finally, reflection is then used to provide a principled mechanism to inspect and dynamically adapt the component structure.

In prior work [Gr05], we have described the overall Gridkit approach, focusing on how different elements of middleware functionality can be configured on-demand to meet application requirements in different environmental conditions. It also describes how a declarative policy-based mechanism drives the configuration and reconfiguration of the architecture. Using context information (e.g. current device type, or network style), the correct policy is selected and applied across the framework, plugging the appropriate functionality into each of the core frameworks (i.e. overlay framework, interaction framework, service discovery framework). In this paper, we are investigating the approach further by defining mechanisms to configure and reconfigure non-functional concerns (in this case security) within this framework, and in particular within the interaction framework.

2.2 OpenCOMJ and the Interception Meta-model

OpenCOMJ is a lightweight Java component model that implements the OpenCOM component runtime specification [Co04], and is used to implement every component and framework in the Gridkit architecture. Crucially, OpenCOMJ provides an interception meta-model. This supports the inspection, insertion, and deletion of interceptors to individual interfaces. Interceptors can be either pre, or post, i.e. they are invoked before or after each operation call on that interface. In OpenCOMJ, interceptors are implemented as individual Java methods that follow a particular syntax, as seen in figure 1. The parameters contain the method name and the methods arguments. Hence, the interceptor can monitor and manipulate the behaviour of the interface. Each OpenCOMJ interface is delegated using Java dynamic proxies; essentially the interface call is trapped, the attached pre methods are executed in order, the original method is called dynamically, and finally the post methods are executed. As will be seen below, it is this mechanism that enables the dynamic insertion of security policies into Gridkit.

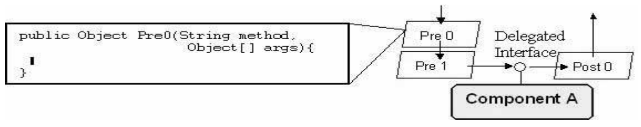

```
public Object Pre0(String method,
                   Object[] args){
  ⁞
}
```

Figure 1: Implementing Pre and Post Interceptors in OpenCOMJ

3 Programmable Security

3.1 The Case for Programmable Security

It is a very challenging problem to address security requirements in dynamic and changeable application domains. Firstly, security is usually treated as an add-on property and is rarely properly considered in the design and implementation of a system. This will inevitably increase the complexity whilst decreasing the effectiveness of security implementations. Secondly, traditional tools used to build security solutions only work as expected in a very specific environment, where all assumptions are clearly defined and supported. However, the real world is not static, change will occur and, gradually, the security solution will become mismatched to the dynamic environment and its applications. In addition, the security features in both low-level cryptography functions and high-level security mechanisms are very complex to understand and implement, especially in the dynamic environment. We propose *programmable security* as a solution to these problems [An03]. We have limited the scope of the security problems we address to the ones related to communication, that is, security protocols. We believe that the most interesting issues (security-wise) revolve around communication (e.g. both message passing and invocation), such as secrecy, integrity, authenticity, non-repudiation. In order to maximize the flexibility of secure communication solutions, and at the same time separate a solution from the application, we have designed Obol, a specific security protocol language to express flexible security policies. Unlike many other policy definition languages (see section 6), security policies in Obol can be dynamically loaded and executed, and are clearly separated from other parts of the system, and hence Obol can be perfectly integrated into a reflective middleware.

3.2 Obol: a Security Protocol Language

The security protocol language Obol [MS06] is greatly influenced by the numerous logics used for analyzing security protocols, e.g. [BAN90] [SO96]. These logics deal with security issues at a very high level of abstraction, leaving other matters to the system of deployment, i.e. the implementation. The Obol language mirrors this by keeping the level of abstraction used to express a security protocol as high-level as possible, while delegating low-level concerns, such as message representation and data transfer, to its runtime. This means that security protocols can be expressed by very short textual descriptions, called *scripts*, which only deal with higher-level security problems it attempts to solve.

For security protocols, the interesting concerns are: manipulating local state, what to encrypt and decrypt, what to digitally sign and verify, what data to send, and what's expected to be received during a correct protocol run. Together with a syntactic notation, Obol provides eight fundamental operands that address these concerns: *believe* and *generate*, for manipulating local state; *encrypt* and *decrypt*, *sign* and *verify*, for the same-named cryptographic operations; and *send* and *receive* for expressing what messages to send and expect to receive. There are other operands for manipulating Obol language objects, and interacting with the Obol runtime.

In its current incarnation, Obol is interpreted in a runtime named "*Lobo"* implemented in Java. The runtime deals with all matters not addressed by the Obol language, such as loading and controlling protocol scripts, message representation, sending and receiving messages, etc. These issues are modularized and can be replaced or updated at need. Figure 2 shows an overview of the Lobo structure.

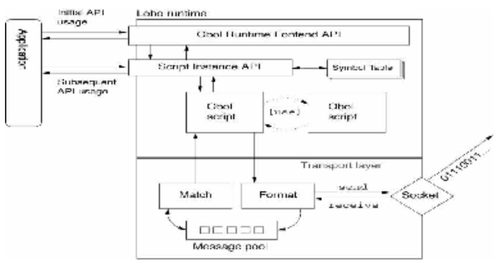


Figure 2: Obol Runtime (Lobo) Overview

Applications interact first with the runtime itself to load, select and start a particular script, and then the application interacts with the script instance through a *script-handle* during the protocol run. The script-handle allows the application to inspect the script instance, to provide or retrieve various parameters and results, as well as interacting with the protocol run, to provide/retrieve intermediate data, error state and so on. Typical parameters provided by the application are long-term identity keys, names, peer-addresses, and payload data. The result retrievable varies greatly depending on the protocol. This reflection is also available to the scripts themselves, allowing one script to use another.

The language does not make any assumptions on how messages communicate; in particular, messages need not be transported over the same medium during a protocol session. The runtime keeps a pool of delivered messages, and a matching algorithm determines if a delivered message is to be received by a script instance. Also, no assumptions are made on how messages are represented nor how they are structured. The exact manner of message transport is handled by the Obol runtime and is modularized so that new ways of communication can be added. The manner of communication can be configured at runtime, either by the Obol scripts themselves, or through parameters passed from an application. This allows an application utilizing Obol as its security protocol machinery to adapt to changing situations.

4 Flexible Security Policies in Gridkit

4.1 Security Requirements of Diverse Environmental Conditions

To prevent attacks in the form of masquerading, tampering, eavesdropping and denial of service, it is necessary to guarantee key security properties such as entity authentication, data integrity, confidentiality, non-repudiation, authorization, validation, access control etc. There are many cryptography techniques provided to support message security. For example, shared-key or public-key based encryption/decryption for confidentiality, MAC and digital signatures for data integrity and non-repudiation, access control technologies (e.g. ACL or ACM) for authorization. Moreover, a series of key establishment protocols are used in authentication, key transport and key agreement. Table 1 shows a selection of basic two-party protocols. Additionally, some protocols provide multi-party support such as n-Party Diffie-Hellman protocol [STW96], secret sharing technique [MOV96], conference protocol [MOV96], etc.

It is well known that the definition of security mechanisms is highly dependent on the requirements of the application you want to protect, i.e., the required security principles, the handling attack types. Therefore, security policies must match the environmental conditions. Heterogeneous interaction paradigms demand flexible and dynamic security policies. Consider for example RPC, group communication and publish-subscribe interaction paradigms. In the client-server model, the system can employ approaches such as Kerberos [SNS88], the Needham-Schroeder shared-key protocol [NS78] or public-key mutual authentication protocol [Gl00] for entity authentication. Moreover, MAC, digital signatures as well as encryption/decryption technology can be used to guarantee privacy and data integrity. Group communication is a significantly more complex interaction type compared to client-server. Its characteristics are: i) potentially large scale groups; ii) dynamic joining and leaving of members resulting in the update of group security parameters (group key and group view) in order to prevent new joiners from eavesdropping previous messages, and leavers from looking at future messages; iii) flexibility: the joiner is allowed to be a member when all other members agree with it. To ensure the validity of a group member as well as the privacy, integrity and freshness of messages delivered between group members, it is necessary to choose appropriate security mechanisms to cope with the generation, distribution and management of group keys. Secure authenticated key agreement protocols for dynamic peer groups [AST00], key graph solution for scalable group security [WGL00], and the Burmester-Desmedt conference protocol [MOV96] are some of the optional techniques to meet different system requirements e.g. in terms of being lightweight, scalable, etc. Finally, in the area of publish-subscribe, security protection focuses more on the cryptographical binding between type name and type definition, as well as the authenticity and integrity of messages [Ba05].

Type	Protocol (properties)
Key transport protocol based on symmetric encryption	Point-to-point key update (no server) Shamir's no key protocol (no server) Kerberos authentication protocol (server based) Needham-Schroeder shared-key protocol (server based) Otway-Rees (server based)
Key transport protocol based on asymmetric encryption	Basic PK encryption (1-pass) (no entity authentication) X.509 (2-pass) -timestamps (mutual entity authentication) X.509 (3-pass) –random (mutual entity authentication) Beller-Yacobi (4-pass) (mutual entity authentication) Beller-Yacobi (2-pass) (unilateral entity authentication)
Key agreement protocol	Diffie-Hellman (entity authentication) ELGamal key agreement (key entity authentication) STS (mutual entity authentication)

Table 1: Selected Protocols [MOV96]

In addition, developers need to consider the trade-offs involved in the security techniques. Public key encryption is slower than symmetric encryption algorithms due to the level of computation involved, so public key cryptography is unusable for resource-poor devices; furthermore, according to [Di03], long-term key based encryption slows performance even using today's high-power processors. Therefore, developers need to weigh the need for strong encryption versus system performance; even if we neglect the cost of encryption technology (e.g. RSA, DES, AES), because encryption or any security-enabling technique will add overhead to communication, it also leads to increased memory and processing costs. To conclude, security mechanisms will vary depending on the end-system types they can execute on.

4.2 Implementing Security Policies in Obol

To support the different security requirements, we adopt Obol to program flexible security policies according to its fundamental characteristics; namely it is "high-level", i.e. easy to implement because the simple syntax is close to the standard description of the security protocol; and it is "programmable", i.e. security policies can be configured and reconfigured at runtime.

Security policies in our architecture are classified depending on security functions, e.g. entity authentication, data integrity, message privacy as well as securing the private key and so on. The implementation of every security policy is an Obol program. Figure 3 represents a simple Obol program to perform message encryption and transmission (using the *believe* primitive to bind names to values and the *send* primitive to actually send the encrypted message).

```
(believe A "localhost:8000" host)
(believe B_id "B")
(believe message "12345")
(believe Key shared-key 0x12345…)
(send A B_id (encrypt Key message))
```

Figure 3: An Obol Program

These Obol programs are interpreted and executed in the runtime Lobo. Due to the clean decoupling between protocol implementation and protocol execution, security protocols can be programmable before or after an application is designed and implemented. Moreover, the loading of Obol programs occurs at runtime so the fluctuation of security policies will not affect other parts of the system. This simplifies the update of security policies and also achieves dynamic configuration and runtime reconfiguration of security policies.

4.3 Integration of Flexible Security Mechanisms into Gridkit

Section 2.2 described the interception meta-model of OpenCOM; this forms the basis of our reflective security architecture. An Obol program (the implementation of the security protocol) must be loaded in the Lobo runtime before it can execute; the reflective mechanism of Gridkit is well-suited to this task, i.e. the interceptor provides an environment to install the runtime Lobo and execute a given security protocol at a particular point in the "middleware path". In this way, the update and replacement of the security protocol used is separated from the logic of the core middleware functionality. Figure 4 presents the security architecture in Gridkit and the runtime loading of Obol scripts.

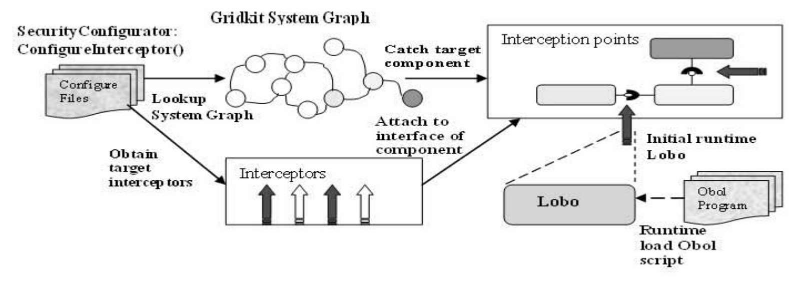


Figure 4: Security Architecture and Runtime Loading of Security Mechanisms in Gridkit

In detail, we designed our security architecture based on the principle of a clear "separation of concerns" between the application logic and the security service. We employ *interceptors* to execute all security related operations so that end-users can focus on the application development rather than security implementation. As a result, in a given application, interceptors are responsible for intercepting the application logic chain and triggering the appropriate security mechanisms.

As mentioned in section 2.2, the interceptor is either pre or post. The pre-/post- methods can be added or deleted from the individual interface. Based on the interception meta-model in Gridkit, security mechanisms can be properly defined and implemented in pre-/post- methods in the design of the system. At runtime, once the pre-/post- methods are attached to an interface, the pre-defined security actions will be the triggered before or after each operation call on this interface.

In order to make the security mechanisms modular, we separate every security policy into different pre- or post- method calls. We then adopt a "*context-based selection*" mechanism to dynamically select which pre-/post- methods will be performed. Figure 5 illustrates that pre methods can be executed in order as shown in path 1, or selectively executed as in path 2 (which uses context information to select a path through the interceptors). The dynamic composition of pre-defined actions not only increases the flexibility of interception behaviours, it also facilitates dynamic configuration and more general extensibility of security mechanisms at runtime.

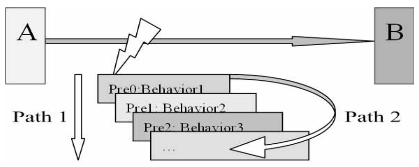
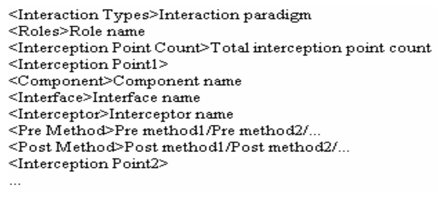


Figure 5: Interception Behaviours in the 'context-based selection' Mechanism

Moreover, we adopt *interaction/role* based configuration in order to adapt the security mechanism to the current requirements and environmental conditions. In other words, "interaction/role" is viewed as a path to the security architecture configuration, e.g. RMI/Client, RMI/Server, Group/SL or Pub-Sub/Publisher. Here, the "interaction/role" decides the interception points while the "role" (potentially together with some other context information at runtime) decides the pre- and post- method-call and the loaded Obol program. The configuration information is defined in a plain text file (see figure 6) including interaction types, roles, components, interfaces as well as pre-/post- methods.



```
<Interaction Types>Interaction paradigm
<Roles>Role name
<Interception Point Count>Total interception point count
<Interception Point1>
<Component>Component name
<Interface>Interface name
<Interceptor>Interceptor name
<Pre Method>Pre method1/Pre method2/...
<Post Method>Post method1/Post method2/...
<Interception Point2>
...
```

Figure 6: Configuration File Template

In our system, the configuration of the security architecture is based on the API method *SecurityConfigurator:ConfigureInterceptor()*. The operations it performs are to discover the points in the middleware that need to be intercepted according to the <Component> and <Interface> tags in the configuration file, using a reflection-based approach, and then attach the interceptors at this point following <Interceptor> and <Pre Method>/<Post Method> tags, i.e. adding pre-/post- methods to the call chain. In summary, the steps involved in the process of configuration are as followed (refer to figure 4):

1. Read the configuration file according to the *interaction* type and the *role*

2. Lookup the required components from the system graph of components; this is supported by the architecture meta-model of OpenCOM [Bl01]

3. Attach the interceptors to the interfaces according to the *role*, or execute other security related operations, such as initiating the authentication server.

This configuration aims to dynamically set the interception points at runtime. After this, the original call invocation will be intercepted, and the pre-/post- methods will be triggered before and after the call. The runtime Lobo will be installed and the appropriate security mechanism (an Obol program) matching the current context information will be loaded dynamically. Therefore, the security architecture in Gridkit is configurable, orthogonal and crosscuts core middleware functionality to guarantee a series of security objectives including authentication, integrity, privacy, non-repudiation.

5 Case Study

In this section, we present a scenario (shown in figure 7) featuring both RPC and multicast-based group communication with different device types. This demonstrates the configurability of flexible security policies for varied interaction paradigms and device types, and shows how programmable security can be applied within Gridkit. Hence, different security mechanisms are automatically configured at runtime to cope with heterogeneous environmental conditions. In the scenario, node *A*, node *B* and the server are located in the different domains. The server provides a variety of services with different permissions. Client *A* and client *B* invoke services from the server. The server provides a variety of security mechanisms, however client *A* is a shared-key system and only supports the Needham-Schroeder shared-key protocol; while client *B* supports both public-key and shared-key systems, but only provides X.509 certificates as well as the Kerberos authentication protocol. At the same time, *B* also joins a chat room and talks to other participants, hence requiring secure multicast. We configure the security architecture for the two interaction paradigms on two types of devices (PC and PDA). We illustrate the concrete details behind each step of our approach described in section 4.3.

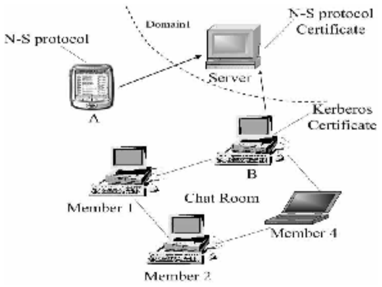


Figure 7: Application Scenario Featuring Client–Server and Group Communication

Consider the security architecture configuration based on two different interaction types (RMI and group communication) that is tabulated in Table 2. It describes one or more different interception points as well as the pre-defined security actions (interceptors) for each interaction. It also shows the pre/post methods inserted at the interfaces. The pre/post methods are executed following the "*context-based selection*" mechanism according to the context information of each device type.

Interaction Type	Role	Component	Interface	Intercepted Method	Interceptor	Trigger point
RMI	Client	JavaRMI	IClientRemoteProcedureCall	Invoke()	rmi_c_interceptor (pre0,pre1 pre2, post0)	Invoke()
	Server	ServiceComponentName +interfaceName	IService	N/A	rmi_s_interceptor (pre0, pre1,post0)	N/A
Group	SL	GroupManagement	IGroupManagement	JoinGroup()	join_interceptor (sl_post0)	JoinGroup()
		GroupCommunication	IGroupCommunication	SendGroup()	send_interceptor (pre0)	SendMessage()
		GroupCommunication	IGroupCommunication	ReceiveFrom Group()	receive_interceptor (post0)	N/A
		GroupManagement	IGroupManagement	LeaveGroup()	leave_interceptor (sl_post0)	LeaveGroup()
	Joiner	GroupManagement	IGroupManagement	JoinGroup()	join_interceptor (m_pre0, m_post0)	JoinGroup()
	Member	GroupCommunication	IGroupCommunication	SendGroup()	send_interceptor (pre0)	SendMessage()
		GroupCommunication	IGroupCommunication	ReceiveFrom Group()	receive_interceptor (post0)	N/A
		GroupManagement	IGroupManagement	LeaveGroup()	leave_interceptor (m_post0)	LeaveGroup()

Table 2: Configuration Information in Heterogeneous Interaction Types

At the application start-up, the *SecurityConfigurator* is initiated and obtains the *contextInfoPath* like RMI/Client or Group/SL from the context. Following the data from Table 2 for RPC, the *SecurityConfigurator* associated with client *A* and client *B* will realise the context information “RMI/Client”, will check the configuration file, look up the component called “JavaRMI”, attach the “rmi_c_interceptor” to the interface called “IClientRemoteProcedureCall” and then add the pre0, pre1, pre2 and post0 methods written in the “rmi_c_interceptor” to the invocation chain. However, if *B* joins a chat room, the *SecurityConfigurator* associated with it will realize it as “Group/Joiner”, look up the component called “GroupManagement”, attach the “join_interceptor” to the interface named as “IGroupManagement” and add m_pre0 and m_post0 methods written in the “join_interceptor”. More details follow in terms of the concrete description of how RPC and group security are deployed.

5.1 RPC Security

The first example focuses on the provision of a security architecture for Gridkit’s RPC interaction type. Figure 8 depicts the workflow of this architecture in detail. In this example, the server provides a simple patient record service allowing doctors to read a patient’s medical record. If the client is authorised, it will return the corresponding record to the passed parameter (*patientID*). In this scenario, there are two clients: *A* (a PDA client being used by a doctor in a hospital) and *B* (a PC client being used by a general practitioner) supporting different security mechanisms; hence, it is essential to negotiate the security mechanism they will use for message exchange. In detail, at the beginning of the application, the server configures the appropriate Gridkit interaction type [Gr05] and hosts the patient record service; it also invokes the *SecurityConfigurator:ConfigureInterceptor()*. After this is done, an *authentication server* is generated and pre- and post- methods are attached. The client configures itself in the same way and invokes the record service with the *contextInfo* (“PDA” or “PC” in this case) and value (*patientID*) as the parameter called *InputParameters*.

The Runtime Lobo is initiated in the pre or post method as shown in figure 8. Pre0 in the client side is triggered first when the client invokes the service. It contacts the *authentication server* and negotiates the authentication protocol to be used. The *authentication server* then creates a new thread for every incoming client for authentication. The pre-/post- method, as well as the *authentication server* will install Lobo, load and execute an associated Obol program according to the negotiation result.

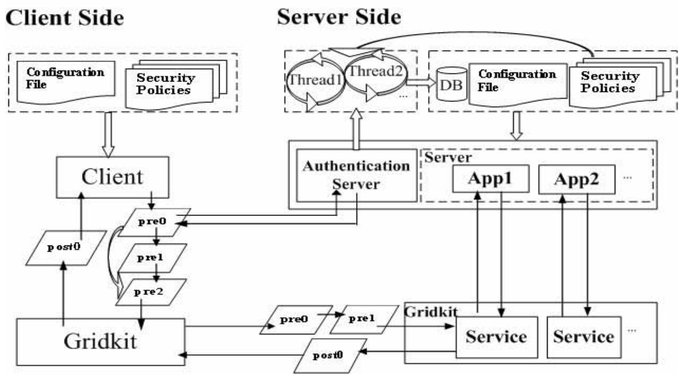


Figure 8: Workflow Overview in RMI Application

In our scenario, the authentication between *A* and the server is based on the Needham-Schroeder protocol, while a public key based mutual authentication protocol is used between *B* and the server (please refer to figure 9 which describes the two protocols and table 3 which shows the protocol implementation in Obol for each participant in the protocol. The number "1" in table 3 is matching to "Message 1" in figure9). At runtime, the Obol program is dynamically loaded into Lobo. For example, when *A* wants to read a patient record, the client side script of the N-S protocol (see table 3) will run in the pre0 method while the server side script of the N-S protocol will run in the authentication server. Once the authentication finishes successfully, *A* will get a shared key with the server. At the same time, it will get the *connection_ID* for accessing the services from the server. The server also writes messages into the database (DB), including *connection_ID* and the security mechanisms related to this *connection_ID*. After authentication at the client side, the pre1 or the pre2 method encrypts messages (please refer to figure 10) and also attaches the *connection_ID* to the encrypted messages. At the other side, the server authenticates the *connection_ID* and queries security mechanisms related to this call. It tells Lobo which Obol program is loaded for decrypting the message.

Message 1 A->B: A
Message 2 B->A: $\{N_B\}K_B$
Message 3 A->KDC: N_A, A, B, $\{N_B\}K_B$
Message 4 KDC->A: $\{K_{AB}, N_A, B, \{A, N_B, K_{AB}\}K_B\}K_A$
Message 5 A->B: $\{N_1\}K_{AB}$, $\{A, N_B, K_{AB}\}K_B$
Message 6 B->A: $\{N_1\text{-}1, N_2\}K_{AB}$
Message 7 A->B: $\{N_2\text{-}1\}K_{AB}$

(i) Needham-Schroeder Shared-key Protocol (N-S)

Message 1 A->CA: A, K_A^{+*}
Message 2 CA->A: A, K_A^+, CA, $\{H(A, K_A^+, CA)\}$ K_{CA}^-
Message 3 B->CA: B, K_B^+
Message 4 CA->B: B, K_B^+, CA, $\{H(B, K_B^+, CA)\}$ K_{CA}^-
Message 5 A->B: A, K_A^+, CA, $\{H(A, K_A^+, CA)\}$ K_{CA}^-
Message 6 B->A: N_B
Message 7 A->B: $\{H(N_B)\}$ K_A^-
Message 8 B->A: B, K_B^+, CA, $\{H(B, K_B^+, CA)\}$ K_{CA}^-
Message 9 A->B: N_A
Message 10 B->A: $\{H(N_A)\}$ K_B^-
[*: A public/ private key pair (K_A^+ , K_A^-)]

(ii) Public-key Mutual Authentication

Figure 9: Protocol Description

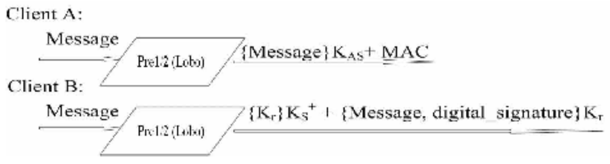


Figure 10: Message Encryption

Protocol	Implementation		
N-S	Client side Obol script	Server side Obol script	**Key Distribution Certer**
	[self “localhost:6000”] (believe B “localhost:7000” host) (believe KDC “localhost:8000” host) (believe K_A (load “c:/K_A.key”) shared-key ((alg AES)(size 128))) **1** (send B “A”) **2** (receive B *1) **3** (generate N_A nonce 16) (send KDC N_A “A” “B” *1) **4** (receive KDC (decrypt K_A *K_{AB} N_A “B” *2)) (believe K_{AB} *K_{AB} shared-key ((alg AES)(size 128))) **5** (generate N_1 nonce 16) (send B (encrypt K_{AB} N_1) *2) **6** (believe *N N_1 ((type number))) (generate *N_{1_1} eval lisp **“(- ***N 1)” *N) (believe N_{1_1} *N_{1_1} ((type binary))) (receive B (decrypt K_{AB} N_{1_1} *N_2)) **7** (believe *N *N_2 ((type number))) (generate *N_{2_1} eval lisp **“(- ***N 1)” *N) (send B (encrypt K_{AB} *N_{2_1}))	[self "localhost:7000"] (believe K_B (load "c:/K_B.key") shared-key ((alg AES)(size 128))) **1** (receive *a *A_ID) **2** (generate N_B nonce 16) (send *a (encrypt K_B N_B)) **5** (receive *a *1 *2) (decrypt (K_B *2) *A_ID N_B *K_{AB}) (believe K_{AB} *K_{AB} shared-key ((alg AES) (size 128))) (believe f "c:/kab_b.key" file.out) (send f K_{AB}) (decrypt (K_{AB} *1) *N_1) **6** (believe N_1 *N_1 ((type number))) (generate *N_{1-1} eval lisp “(- *N_1 1)” *N_1) (believe N_{1-1} *N_{1-1} ((type binary))) (generate N_2 nonce 16) (send *a (encrypt K_{AB} N_{1-1} N_2)) **7** (believe *N_2 N_2 ((type number))) (believe *N_{2_1} eval lisp “(- *N_2 1)” *N_2) (receive *a (decrypt K_{AB} N_{2_1}))	[self "localhost:8000"] (believe K_A (load "c:/K_A.key") shared-key ((alg AES) (size 128))) (believe K_B (load "c:/K_B.key") shared-key ((alg AES)(size 128))) **3** (receive *a *N_A *A_ID *B_ID *1) (decrypt (K_B *1) *N_B) **4** (generate K_{AB} shared-key AES 128) (believe *2 (encrypt K_B *A_ID *N_B K_{AB})) (send *a (encrypt K_A K_{AB} *N_A *B_ID *2))
Public-key Mutual Authen-tication	Client side Obol script	Server side Obol script	Certificate Authority
	[input A_ID string] [self "localhost:6700"] (believe B “localhost:7000” host) (believe CA "localhost:6111" host) (believe K_A^- (load “c:/A_ $K_{private}$.key“) private-key) (believe K_A^+ (load "c:/A_K_{public}.key") public-key) (believe K_{CA}^+ (load "c:/CA_K_{public}.key") public-key) **1** (send CA A_ID K_A^+) **2** (receive CA A_ID K_A^+ *CA_ID *s_a) **5** (send B A_ID K_A^+ *CA_ID *s_a) **6** (receive B *N_B) **7** (send B (sign K_A^- *N_B)) **8** (receive B *B_ID *K_B^+ *CA_ID (verify K_{CA}^+ *B_ID *K_B^+ *CA_ID) **9** (generate N_A nonce 128) (send B N_A) **10** (receive B (verify K_B^+ N_A))	[input B_ID string] [self "localhost:7000"] (believe CA "localhost:6111" host) (believe K_B^- (load "c:/B_$K_{private.}$key") private-key) (believe K_B^+ (load "c:/B_$K_{public.}$key") public-key) (believe K_{CA}^+ (load "c:/CA_$K_{public.}$key") public-key) **3** (send CA B_ID K_B^+) **4** (receive CA B_ID K_B^+ *CA_ID *s_b) **5** (receive *a *A_ID *K_A^+ *CA_ID *s_a) (verify (K_{CA}^+ *s_a) *A_ID *K_A^+ *CA_ID) **6** (generate N_B nonce 128) (send *a N_B) **7** (receive *a (verify *K_A^+ N_B) **8** (send *a B_ID K_B^+ *CA_ID *s_b) **9** (receive *a *N_A) **10** (send *a (sign K_B^- *N_A))	[self "localhost:6111"] (believe K_{CA}^- (load "c:/CA_$K_{private.}$key") private-key) **1** (receive *a *A_ID *K_A^+) **2** (believe *s_a (sign K_{CA}^- *A_ID *K_A^+ "CA")) (send *a *A_ID *K_A^+ "CA" *s_a) **3** (receive *b *B_ID *K_B^+) **4** (believe *s_b (sign K_{CA}^- *B_ID *K_B^+ "CA")) (send *b *B_ID *K_B^+ "CA" *s_b)

Table 3: Protocol Implementation in Obol

5.2 Multicast-based Group Communication Security

In the scenario, to secure group communication, we utilise a lightweight authentication protocol based on Leithton-Micali key distribution algorithm [MHP98]. The founder of the group, or the earliest joiner based on the current group view (if the founder left) is viewed as the session leader (hereafter SL).

Every joiner must be authorized by the group before becoming a group member. We use the same approach described in section 5.1 to configure the pre-/post- methods in interception points according to table 2. As described in figure 11, *B* invokes *JoinGroup()* to join a chat room, and then it will trigger the authentication protocol executed in the m_pre0 method which is inserted into the *IGroupManagement* interface. After the authentication is done, the SL generates a new group key for the new group view and multicasts it to all members in the group. After *JoinGroup()* is called, its m_post0 method will be triggered to install a runtime Lobo (see figure 11) that will wait for the joining of the next authentication process and the receiving of a new group key when a new member comes.

Moreover, the *SendMessage()* call will trigger the message encryption before the message is transported. The message exchanged among group members is encrypted with the fresh group key, so people outside the group will not understand it. At the same time, the group member also listens to the incoming message. After *ReceiveFromGroup()* finishes, its *post0()* method will decrypt the received message and show the original message to the receiver. In addition, when one of the members leaves the group, the post method of *LeaveGroup()* call will trigger the generation and distribution of a new group key. Especially when the SL leaves the group, the *join_interceptor:sl_post()* will notify the earliest joiner in the current group view. This member receives the notification and reconfigures itself as a SL, including generating an authentication server (the part distinct from usual membership, seen in figure 11) and loading the authentication protocol. Due to space limitations, more details of this implementation are not given here.

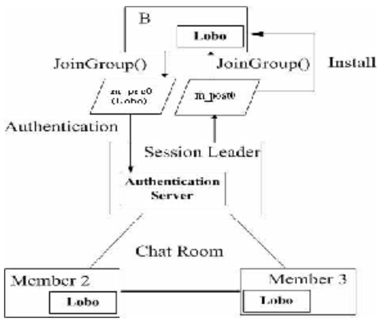


Figure 11: Node B Joins a Chat Room

6 Related Work

There is a large body of research investigating middleware security services, but none as yet includes the range of flexible security policies provided by our security architecture. They either only provide fixed, static security mechanisms or fail to provide configurable and reconfigurable security policies in an effective way. The following are some of the most relevant technologies. CORBA Security Services (CORBASec) [OM02] present a large programming API that encompasses all possible security policies. However, in practice the complexity of security mechanisms makes this standard unusable. Compared to CORBA, the definition of profiles in CCM [OM06] allows developers to avoid re-implementing security service code. However, CCM still reuses the CORBA security services implementation; these are static security mechanisms that cannot deal with the diverse application contexts that our approach tackles. Furthermore, the recently released EJB 3.0 [Su06] adopts the security role mechanism and allows developers to configure user permissions for each method call using annotations (mostly concerned with access control), and also allows the developer to write custom interceptors for EJB methods such as security exceptions. Hence, it aims to support the separation of security behaviour from middleware implementation (as our approach does), but it still remains limited in supporting flexible security policies. The approach lacks a high-level domain specific language, to simplify the inclusion and dynamic selection of a wide range of security protocols. Furthermore, the EJB and CORBA middleware are heavyweight solutions that cannot easily be configured to operate on diverse device types with diverse security requirements.

Apart from these traditional middleware platforms, in the growing Grid computing environment, the major player Globus [Gl00] provides GSI to enforce security for access to Grid resources; however this is again limited by fixed security mechanisms, which does not adequately separate security concerns, and as such is not sufficient to support demanding applications, especially in the next generation Grid that is characterised by high levels of heterogeneity and dynamism. Like CORBA and EJB, Globus is rather heavyweight in terms of implementation and hence is not suitable for many application domains.

Moreover, [ACF99] and [CHV01] present initial exploration into the use of reflection (i.e. Meta-Object Protocols) to implement security mechanisms in distributed object-oriented frameworks. However, these are meta-object approaches that are limited to just providing Java-based security. Hence, the security implementation is not freely extensible or replaceable with new security protocols that have been simply defined in a high-level domain specific language.

Obol is a domain specific programmable security language, and we are not aware of any other work that supports programmable security in the same manner. Of course, there are policy definition languages to define security mechanisms, like ASL [JSS97], Ponder [Da01], PDL [LBN99]. Ponder can be used to specify both management and security policies. It intends to define choices in terms of the conditions under which predefined operations can be switched without recoding or stopping the system, but it is a specification language not implementation language and policies needs to be mapped onto concrete access control mechanisms so it does not freely support the reloading of flexible security mechanisms at runtime (like Obol). Additionally, it also does not support interaction protocols. PDL is event-based language, and the policies are similar to Ponder obligation policies. Hence, they are all insufficient to support flexible security policies in our general interception based security architecture.

7 Conclusions and Future Work

In this paper, we have discussed an approach to integrate our programmable security architecture into the Gridkit middleware platform to support flexible security policies that adapt to heterogeneous environmental conditions. We adopt two complementary technologies: the interception meta-model of the OpenCOM component model and the programmable security capabilities of Obol to build a security architecture. This combination is capable of supporting configurable, reconfigurable, and flexible security policies. To date, we have designed and implemented a rich set of security policies, including authentication protocols, secret key based encryption/decryption, public-key based encryption/decryption, MAC, digital signatures, and private key management, to support authentication, integrity, privacy and non-repudiation. Currently we have a mature implementation of the prototype to support dynamic configuration of flexible security policies to adapt to varied device types in the RPC model and we are now extending the security architecture for the group communication model for robustness, and to include a wider range of selectable security services.

The main benefits of our approach to build security architecture are in terms of: i) *Flexibility*. The combination of the reflective capabilities of Gridkit and the programmable security provided by Obol makes it possible to dynamically configure and re-configure flexible security mechanisms at runtime. ii) *Ease of use*. Firstly, Obol frees the security developer from the distractions of low-level implementation efforts. Additionally, the use of Obol also avoids errors introduced by the implementation of security protocols. Finally, the Obol syntax is similar to the traditional notation of the protocol. iii) *Extensibility*. Following the approach to building a security architecture to support existing RPC and group communication interaction types, we can now freely extend other available interaction styles such as publish-subscribe, media streaming etc, based on the well-defined interface and the programmable features of the security mechanisms. To some extent, we believe there is the potential to extend other traditional middleware platforms (e.g. CORBA and EJB, which support similar interception capabilities) with our flexible security policies.

Although we have made considerable progress in achieving configuration and reconfiguration of security policies in a reflective middleware platform, this is just a start and a lot remains to be investigated. We have focused on security policies to support secure RPC and group communication addressing security properties such as authentication, integrity, privacy and non-repudiation. Future work is planned to complement these security mechanisms to guarantee more security principles such as authorization and access control. We also aim to investigate security in alternative paradigms like publish-subscribe, tuple-space and media streaming. Additionally, we have addressed dynamic configuration of two interaction paradigms upon two types of devices. More ambitious explorations in the future will focus on implementing runtime reconfiguration of flexible security policies (cf. self-organising security policies). Furthermore, Gridkit is characterized by the two-layered component framework [Gr05] featuring an interaction framework layer supported by an overlay framework. It is also interesting to investigate security policies at the overlay level of Gridkit and how these might relate to more end-to-end policies as studied in this paper.

The interception meta-model in the reflective middleware is a cornerstone of our approach to achieve configuration and reconfiguration of flexible security policies. Future work is planned to explore and extend the current interception meta-model to support more flexible interception behaviours. However, the reflection feature also hides some dangers such as arbitrarily loading and deleting interceptors or freely interposing the interceptors without authorization. Therefore, securing interception is also crucial for our approach. We are examining special components called ‘security mediators’ to control access to the component runtime to protect ‘dangerous’ APIs such as interception. In the future, the TCB (Trusted Computing Base) concept will be the base of our security mechanism, authorizing operations on the potentially open interception mechanism.

In addition, a separate project at Lancaster is addressing how to apply aspect-oriented programming (AOP) techniques to the component-oriented approach as used in OpenCOM to enhance how developers deal with crosscutting concerns. There is considerable potential in considering the role of aspect-oriented techniques to identify aspects and join points and investigate how this would be supported through interception (effectively providing a higher level view of statement of cross-cutting concerns such as security). Moreover, future work is also planed to investigate the possibility to apply Model Driven Development (MDD) to our programmable security architecture.

References

[ACF99] Ancona, M.; Cazzola, W., Fernandez, E.B.: Reflective Authorization Systems: Possibilities, Benefits, and Drawbacks. *LNCS*, 1606:35-50, 1999.

[An03] Andersen, A. et al.: Security and Middleware. WORDS 2003, Guadalajara, Mexico, January 2003.

[AST00] Ateniese, G.; Steiner, M.; Tsudik, G.: New Multi-party Authentication Services and Key Agreement Protocols. IEEE Journal of Selected Areas in Communication, Vol. 18, March 2000.

[Ba05] Bacon, J. et al.: Securing publish/subscribe for multi-domain systems. In Proc. of the 6th International Middleware Conference (MW'05), Grenoble, France, November 2005.

[BAN90] Burrows, M.; Abadi, M.; Needham R.: A Logic of Authentication. ACM Transactions on Computer Systems, Vol. 8, No. 1, 1990.

[Bl01] Blair, G. et al.: The design and implementation of Open ORB 2. IEEE Distributed Systems Online, 2(6), September 2001.

[CHV01] Caromel, D.; Huet, F.; Vayssière, J.: A simple security-aware MOP for Java. Proc. of the 3rd International Conference, REFLECTION 2001, Kyoto, Japan, September 2001.

[Co04] Coulson, G. et al.: OpenCOM v2: A Component Model for Building Systems Software. In Proc. of IASTED Software Engineering and Applications (SEA'04), Cambridge, MA, ESA, November 2004.

[Da01] Damianou, N. et al.: The Ponder Policy Specification Language. In Proc. of the International Workshop on Policies for Distributed Systems and Networks, Spring-Verlag, 2001.

[Di03] Diana, A.: Benchmarking Encryption Technology. http://www.macnewsworld.com/story/31311.html

[Gl00] Overview of the Globus Security Infrastructure. http://www.globus.org/security/overview.html

[Gr05] Grace, P. et al.: Deep Middleware for the Divergent Grid. Proceedings of the 6th IFIP/ACM/USENIX International Middleware Conference 2005, Grenoble, France, November 2005.

[JSS97] Jajodia, S.; Samarati, P.; Subrahmanian, V. S.: A Logical Language for Expressing Authorisations. In Proc. of IEEE Symposium on Security and Privacy, 1997.

[LBN99] Lobo, J.; Bhatia, R.; Naqvi, S.: A Policy Description Language. In Proc. of AAAI, July 1999.

[MHP98] McDaniel, P.; Honeyman, P.; Prakash, A.: Lightweight Security Group Communication. CITI Technical Report 98.

[MOV96] Menezes, A.; Oorschot, P.; Vanstone, S.: Handbook of Applied Cryptography. CRC Press, ISBN: 0-8493-8523-7, October 1996.

[MS06] Myrvang, P. H.; Stabell-Kulø, T.: Security Protocol Programming. To appear in Proc. of the 14th Workshop on Security Protocol 2006, LNCS, Springer Verlag.

[NS78] Needham, R.; Schroeder, M.: Using Encryption for Authentication in Large Networks of Computers. Communications of ACM, 21(12): 993-999, December 1978.

[OM02] OMG: Security Service Specification. Technical Report, Object Management Group, March 2002.

[OM06] OMG: CORBA Component Model Specification, v4.0. OMG document - formal/06-04-01.

[SNS88] Steiner, J.; Neuman, C.; Schiller, J.: Keberos: an Authentication Service for Open Network Systems. In Proc. Usenix Winter Conference, Berkeley: Calif., 1988.

[SO96] Syverson, P.; Oorschot, P.: A Unified Cryptographic Protocol Logic. Naval Research Laboratory, CHACS Report 5540-227, Washington, USA, 1996.

[STW96] Steiner, M.; Tsudik, G.; Waidner, M.: Diffie-Hellman Key Distribution Extended to Group Communication. In Proc. 3rd ACM Conference on Computer and Communications System (CCS' 96).

[Su06] Sun Microsystems: EJB Core Contracts and Requirements. JSR 220: Enterprise JavaBeans^{TM}, version 3.0, Final Release, May 2, 2006.

[Sz98] Szyperski, C.: Component Software, Beyond Object-Oriented Programming. ACM Press/Addison-Wesley, 1998.

[WGL00] Wong, C. K.; Gouda, M.; Lam, S. S.: Secure Group Communications Using Key Graphs. IEEE/ACM TRANSACTIONS ON NETWORKING, Vol. 8, No. 1, February 2000.