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, flexible 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 evaluate this approach in terms of its security properties, 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.

- *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 environment. Dealing with flexible security policies will be a fundamental challenge in the development of future middleware solutions. Unfortunately, traditional middleware platforms, e.g. EJB [Su91] and CORBA [OM02] typically only provide static, fixed security mechanisms. In this paper, we propose an approach to apply configurable and dynamically reconfigurable security mechanisms in middleware platforms. This involves the integration of two complementary technologies, namely reflective middleware, and programmable security. That is, we develop flexible security policies using Obol [My05], a security protocol programming language to implement security policies. Then we apply them within an existing reflective middleware, Gridkit [Gr05], using a meta-model supporting behavioural reflection (interception). To evaluate the effectiveness of our approach, we present a case study, characterised by diversity, which demonstrates how security policies can be dynamically configured at runtime.

The remainder of the paper is structured as follows. Section 2 and section 3 discuss the two key underlying technologies. In particular, section 2 introduces the reflective middleware platform Gridkit, its component model (OpenCOM) and its interception meta-model. 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. Following this, we evaluate the approach used to build the security architecture in section 6, and present our conclusions and future work in section 7.

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. Figure 1 illustrates the tailorable Gridkit framework; this is essentially a component framework composed of a set of key component frameworks. At the base is the overlays framework (which is typically used by higher-level middleware) into which per-host implementations of overlay networks are plugged, for example, an Application Level Multicast plug-in (ALM), an epidemic routing plug-in, or a Distributed Hash Table (DHT). Above the overlays framework is a set of vertical frameworks providing diverse middleware behaviour. The *interaction* framework supports the plug-in of multiple interaction types (e.g. RPC, Pub-Sub, Group communication, Streaming, etc.) The *resource discovery* framework accepts plug-in strategies to discover resources such as CPUs and storage (e.g. peer-to-peer search), and also discover software services; the *resource management* and *resource monitoring* frameworks are respectively responsible for managing and monitoring resources.

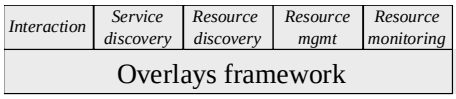

Figure 1: The Gridkit Architecture

[Gr05] 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. 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. Each component implements a set of custom interfaces and receptacles. An interface expresses a unit of service provision, whereas a receptacle describes a unit of service requirement. A connection is the binding between an interface and a receptacle of the same type. OpenCOMJ deploys a standard runtime substrate that manages the creation and deletion of components, acts upon requests to connect/disconnect components and provides service interfaces for reflective operations. The runtime substrate dynamically maintains a system graph of the components currently in use. This explicit maintenance of dynamic dependencies between components provides the support for introspection and reconfiguration of component architectures. OpenCOMJ also supports a component framework model [Gr03]. Here, a framework is a single OpenCOMJ component (seen in figure 2), which then contains its own internal structure (a graph of components). Each framework is extended by the ICFMetaArchitecture interface, which provides reflective operations to inspect and dynamically

reconfigure the framework's local component architecture.

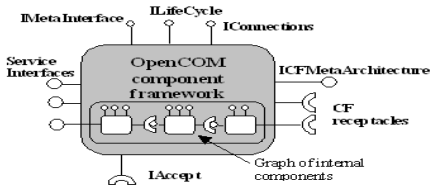

Figure 2: Components and Component Frameworks in OpenCOMJ

Crucially, OpenCOMJ also 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 3. 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.

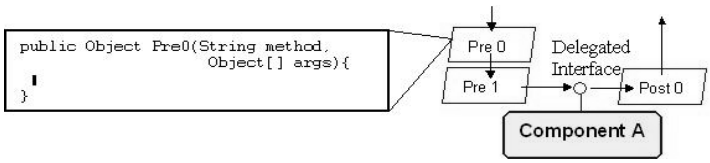

Figure 3: Implementing Pre and Post Interceptors in OpenCOMJ

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 implementation. 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, because the real world is neither static nor globally controlled, change will occur and, gradually, the security solution will become more and more mismatched to the dynamic environment and its applications (the reason for this is often a very tight integration between application and a security solution). 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, and so on.

In order to maximize the flexibility of secure communication solutions, and at the same time separate a solution from the application, we have designed and implemented a language and runtime for expressing and running security protocols in a highly dynamic manner. This allows us to express security protocols at a very high level of abstraction, with clearly defined separation of concerns, boundaries, and interfaces to the application domain.

3.2 Obol: a Security Protocol Language

The security protocol language Obol is greatly influenced by the numerous logics used for analyzing security protocols, e.g. [Bu90] [Sy96]. 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 the security problem at hand. Unlike the logics used for analysis, and other security protocol implementations, Obol is designed for protocol endpoints, and it is not required for Obol to be used by all protocol participants.

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, the Obol language 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 4 shows an overview of the Lobo structure.

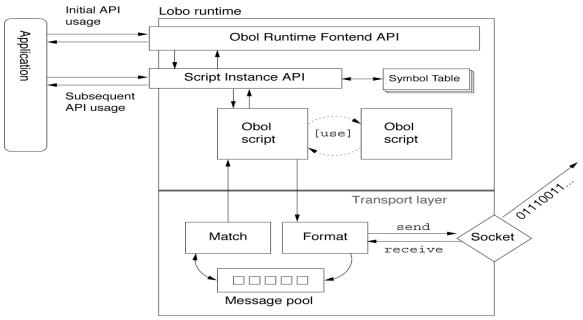

Figure 4: Obol Runtime (Lobo) Overview

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; some protocols yield a result simply by not failing [Go95]. 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, for example, an application can switch the actual protocol used, or just change some parameter of the protocol, such as the encryption algorithm being used, or the manner of communication.

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 [St96], secret sharing technique [Me96], conference protocol [Me96] etc.

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-Shroeder 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 [Me96]

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, and so on. 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 [St88], the Needham-Schroeder shared-key protocol [Ne78] 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 [At00], key graph solution for scalable group security [Wo98], and the Burmester-Desmedt conference protocol [Me96] 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].

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 may be 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; moreover, even if we neglect the cost of encryption technology (e.g. RSA, DES, AES etc.), because encryption or any security-enabling technique will add overhead to communication, this also leads to increased memory and processing costs. In the final analysis, 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 into several parts depending on which security properties it achieves, 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 5 represents a simple Obol program to perform message encryption and transmission (using the *believe*

```
(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 5: An Obol Program

primitive to bind names to values and the *send* primitive to actually send the encrypted message).

As mentioned in section 3.2, Obol defines how to express a given security protocol. These Obol programs must be interpreted and executed in the runtime Lobo. Figure 6 shows how to initiate a Lobo instance and load an Obol program.

```
API _lobo = Runtime.getInstance();
File f = new File("c:/script.text");
LoadedScriptInfo _s = _lobo.loadScript("script", f);
ScriptHandle _script = _lobo.getScriptInstance(_s);
_script.startExecution();
```

Figure 6: Load and Execute an Obol Program

Due to the clean decoupling between protocol implementation and protocol execution, security protocols can be programmed 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.

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. In addition, 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 and so on. 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 API *SecurityConfigurator:InterceptorConfigure()* used to configure security architecture is presented in figure 7.

```
public class SecurityConfigurator{
   public SecurityConfigurator(OpenCOM runtime, IOpenCOM pIOCM, String configureInfoPath){}
   public void InterceptorConfigure(){}
}
```

Figure 7: API for Configuring Security Architecture

The steps involved in the process of configuration are as followed:

1. Read the configuration file (see the example in section 5) according to the "*interaction"* type and the "role"

2. Lookup the required components from the system graph of components; this is supported by 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.

The configuration aims to dynamically set the interception points at runtime. After this, the original call invocation will be intercepted, and the pre- and 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. In summary, the security architecture applied to Gridkit is configurable, orthogonal and crosscuts core middleware functionality to guarantee a series of security objectives including authentication, data integrity, privacy, non-repudiation and others.

5 Case Study

In this section, we present one scenario (shown in figure 8) featuring both RPC and multicast-based group communication. In the scenario, node *A*, node *B* and the server are located in different domains. Client *A* and client *B* invoke services from the server located in $domain_1$. *A* only supports a shared-key system, while *B* supports both shared-key and public-key based systems but no support for the Needham-Schroeder shared-key protocol. Additionally, B joins a chat group and talks to other group members. We configure the security architecture for the two different interaction paradigms on two types of devices (PC and PDA); this demonstrates how programmable security is integrated into our reflective middleware platform, and shows how flexible security policies can be dynamically configured to adapt to the heterogeneous environmental conditions. We adopt the approach mentioned in section 4.3 to build secure distributed applications and illustrate the concrete details behind each step.

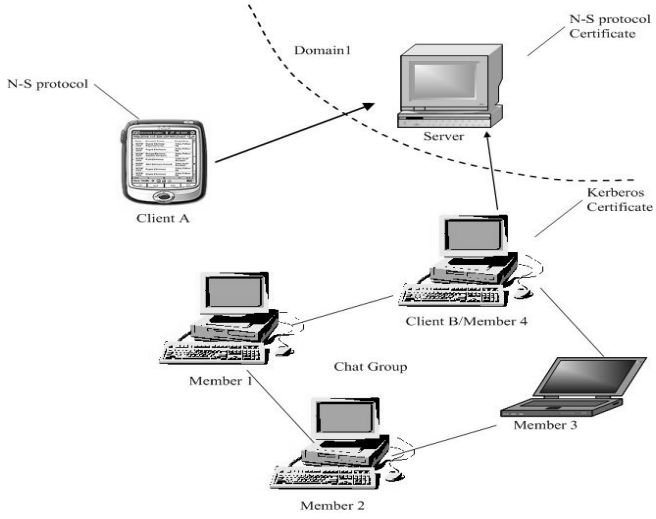

Figure 8: Application Scenario Featuring Client–Server and Group Communication

The configuration of the security architecture is based on the API method: *SecurityConfigerator:ConfigureInteceptor()*. The operations it performs are to discover which points in the middleware need to be intercepted, and then attach the correct interceptors at this point, i.e. adding pre- and post- methods to the call chain. The configuration information based on two different interaction types (RMI and group communication) is tabulated in Table 2.

Interaction Type	Role	Component	Interface	Intercepted Method	Interceptor	Trigger point
RMI	Client	JavaRMI	IClientRemoteProcedureCall	Invoke()	rmi_c_interceptor (pre0,pre1,pre2, post0)	Invoke()
	Server	Endpoint+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, post0)	SendMessage()
		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, post0)	SendMessage()
		GroupManagement	IGroupManagement	LeaveGroup()	leave_interceptor (m_post0)	LeaveGroup()

Table 2: Configuration Information in Heterogeneous Interaction Types

The configuration information is defined in a plain text file (in the future, we will define it using XML). At the application start-up, the *SecurityConfigurator* is initiated and obtains the *contextInfoPath* (see API in figure 7). Following the data from Table 2 for RPC, the *SecurityConfigurator* associated with client *A* and client *B* will realize 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 *A* joins a chat group, the *SecurityConfigurator* associated with it will realize it as "Group/Joiner", look up the component called "GroupManagement", attach the "join_interceptors" to the interface named as "IGroupManagement" and add m_pre0 and m_post0 methods written in the join_interceptor.

The interception meta-model allows programmers to define security behaviours (pre-/post- methods) in advance. The pre-defined actions are triggered at runtime when the invocation happens. 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 9 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.

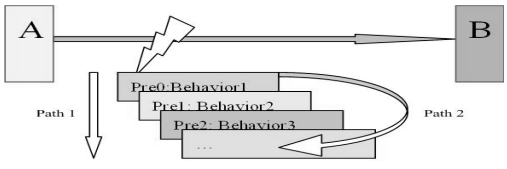

Figure 9: Interception Behaviours in the 'context-based selection' Mechanism

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 10 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 authorized, it will be returned 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*.

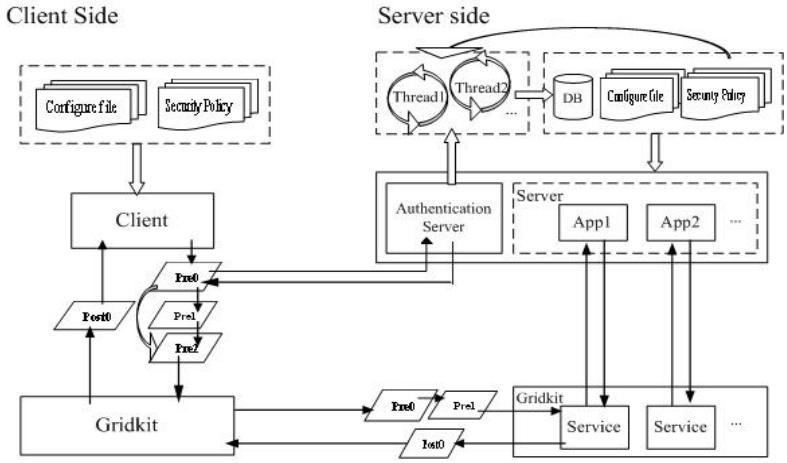

Figure 10: Workflow Overview in RMI Application

As shown in figure 10, 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- and post- method, as well as the *authentication server* will install Lobo and allow the runtime to load and execute an associated Obol program according to the context information.

Needham-Schroeder (N-S)	Public-key Mutual Authentication	Group Communication Protocol
1 A->B: A **2** B->A: $\{R_B\}K_B$ **3** A->KDC: R_A, A, B, $\{R_B\}K_B$ **4** KDC->A: $\{K_{AB}, R_A, B, \{A, R_B, K_{AB}\}K_B\}K_A$ **5** A->B: $\{R_1\}K_{AB}$, $\{A, R_B, K_{AB}\}K_B$ **6** B->A: $\{R_1\text{-}1, R_2\}K_{AB}$ **7** A->B: $\{R_2\text{-}1\}K_{AB}$	**1** A->CA: A, K_A^+* **2** CA->A: A, K_A^+, CA, $\{H(A, K_A^+, CA)\}\ K_{CA}^-$ **3** B->CA: B, K_B^+ **4** CA->B: B, K_B^+, CA, $\{H(B, K_B^+, CA)\}\ K_{CA}^-$ **5** A->B: A, K_A^+, CA, $\{H(A, K_A^+, CA)\}\ K_{CA}^-$ **6** B->A: R_B **7** A->B: $\{H(R_B)\}\ K_A^-$ **8** B->A: B, K_B^+, CA, $\{H(B, K_B^+, CA)\}\ K_{CA}^-$ **9** A->B: R_A **10** B->A: $\{H(R_A)\}\ K_B^-$ [*: A public/ private key pair (K_A^+, K_A^-)]	**Join protocol:** **1** Joiner->SL: A, R_A **2** SL->KDC: SL, A, R_{SL} **3** KDC->SL: $[\pi_{SL,A} = \{A\}K_{SL}$ XOR $\{SL\}K_A]$, $\{R_{SL}\}K_{SL}$ **4** SL->Joiner: SL, $\{R_A, g, \{g, A\}SK_g\}\sigma_{SL,A}$* **5** Joiner->group: A, $\{g, A\}SK_g$ **6** SL->group: g, $(A,\{g+1, SK_{g+1}\}\sigma_{SL,A})$, $(B,\{g+1, SK_{g+1}\}\sigma_{SL,B})$... **Leave protoocl:** SL->group: g, $(A,\{g+1, SK_{g+1}\}\sigma_{SL,A})$, $(B,\{g+1, SK_{g+1}\}\sigma_{SL,B})$... [*: $\sigma_{SL,A} = \{SL\}K_A$]

Table 3: Protocol Description [NB: the number like "1" stands for Message1]

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 Table 3 which lists the protocol description and Table 4 which presents the Obol programs for each participant in the protocol). Once the authentication finishes successfully, A will get a shared key with the server while B will obtain the certificate of the server. At the same time, both A and B get the *connection_ID* for accessing the service. The server also writes messages into the database (DB), including connection_ID as well as security mechanisms related to this connection_ID. After authentication, pre1 or pre2 performs the message encryption operation presented in figure 11, and also attaches the connection_ID to the encrypted messages. The server at the other side authenticates the connection_ID and queries security mechanisms related to this call. It tells Lobo which Obol program is loaded for decrypting the message. The record will be delivered in the same way from server to client.

Protocol	Implementation		
N-S	Client side [A]	Server side [B]	**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 R_A nonce 16) (send KDC R_A "A" "B" *1) **4** (receive KDC (decrypt K_A *K_{AB} R_A "B" *2)) (believe K_{AB} *K_{AB} shared-key ((alg AES)(size 128))) **5** (generate R_1 nonce 16) (send B (encrypt K_{AB} R_1) *2) **6** (believe *R R_1 ((type number))) (generate *R_{1_1} eval lisp **"(- *R 1)"** *R) (believe R_{1_1} *R_{1_1} ((type binary))) (receive B (decrypt K_{AB} R_{1_1} *R_2)) **7** (believe *R *R_2 ((type number))) (generate *R_{2_1} eval lisp **"(- *R 1)"** *R) (send B (encrypt K_{AB} *R_{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 R_B nonce 16) (send *a (encrypt K_B R_B)) **5** (receive *a *1 *2) (decrypt (K_B *2) *A_ID R_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) *R_1) **6** (believe R_1 *R_1 ((type number))) (generate *$R_{1\text{-}1}$ eval lisp "(- *R_1 1)" *R_1) (believe $R_{1\text{-}1}$ *$R_{1\text{-}1}$ ((type binary))) (generate R_2 nonce 16) (send *a (encrypt K_{AB} $R_{1\text{-}1}$ R_2)) **7** (believe *R_2 R_2 ((type number))) (believe *R_{2_1} eval lisp "(- *R_2 1)" *R_2) (receive *a (decrypt K_{AB} R_{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 *R_A *A_ID *B_ID *1) (decrypt (K_B *1) *R_B) **4** (generate K_{AB} shared-key AES 128) (believe *2 (encrypt K_B *A_ID *R_B K_{AB})) (send *a (encrypt K_A K_{AB} *R_A *B_ID *2))
PK Mutual Authen-tication	Client side [A]	Server side [B]	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 *R_B) **7** (send B (sign K_A^- *R_B)) **8** (receive B *B_ID *K_B^+ *CA_ID (verify K_{CA}^+ *B_ID *K_B^+ *CA_ID) **9** (generate R_A nonce 128) (send B R_A) **10** (receive B (verify K_B^+ R_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 R_B nonce 128) (send *a R_B) **7** (receive *a (verify *K_A^+ R_B) **8** (send *a B_ID K_B^+ *CA_ID *s_b) **9** (receive *a *R_A) **10** (send *a (sign K_B^-*R_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 4: Protocol Implementation in Obol [NB: the number like "1" means the implementation for Message1]

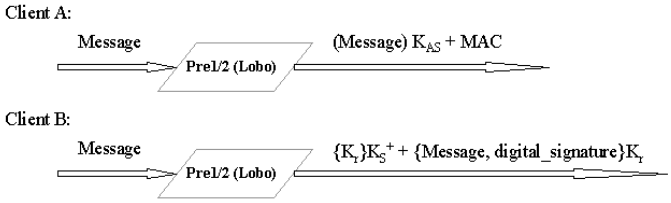

Figure 11: Message Encryption

5.2 Multicast-based Group Communication Security

In this example, to secure group communication, we utilise a lightweight authentication protocol based on Leithton-Micali key distribution algorithm [Mc98] (as described in Table 3). 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 contact the SL before joining. We use the same approach described in section 5.1 to configure the pre- and post- methods according to Table 2. The *JoinGroup()* call will trigger the authentication protocol executed in the m_pre0 method. 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. Member joins and leaves lead to the fluctuation of the group key, so we install a runtime Lobo in the post method *join_interceptor:m_post()* of *JoinGroup()* call to listen to the new group key (as presented in figure 12). Moreover, the *SendMessage()* call will trigger the message encryption before it is transported. The message exchanged among group members is encrypted with the fresh group key, so people outside the group will not understand it. 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 earliest joiner in the current group view will receive a notification and reconfigure itself as a SL, including generating an authentication server (the part distinct from usual membership, seen in figure 12) and loading the authentication protocol. The interceptors that implement this security mechanism are applied in the same manner as for RPC; however, due to space limitations, more details of this implementation are not given here.

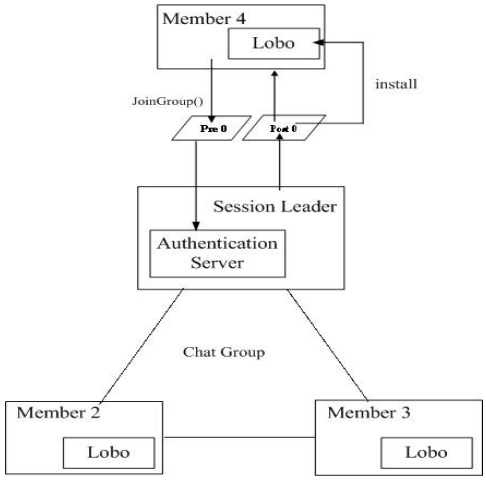

Figure 12: Member 4 Joins a Chat Group

6 Evaluation

In this section, we analyze our approach to building security architectures in Gridkit. We focus on four core aspects as follows:

i) Security. We adopted Obol to define security mechanisms. Currently, we have 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. These security mechanisms protect a given application by enforcing properties such as authentication, integrity, privacy and non-repudiation.

ii) Flexibility. The interception meta-model in our reflective middleware platform provides the possibility to modify the behaviour defined by the business logic of a given application. The combination of this reflective feature within the Gridkit middleware platform and programmable security supported in Obol makes it possible to dynamically configure and re-configure flexible security mechanisms at runtime to meet application requirements in the environmental context.

iii) Ease of use. Firstly, Obol allows the security developer to escape the distractions of low-level implementation efforts. For example, issues such as message representation, cryptographic transformation, etc. are handled in Lobo, so the security developer can focus on designing effective security mechanisms without consideration of low-level cryptographic functions. Additionally, the use of Obol also avoids errors introduced by the implementation of security protocols. Finally, Obol is easy to use because the syntax is similar to the traditional notation of the protocol. In our experience, new protocols can be introduced into the framework with considerable ease; once the overall framework was established, we were able to introduce new security mechanisms by programming security policies in Obol, then updating the interceptors and configuration files (for example, the time to develop a public key mutual protocol was approximately six hours).

iv) Extensibility. We adopted the interception meta-model of the Gridkit reflective middleware platform to construct interceptor based security. This allowed us for example to implement a security architecture to support existing RPC and group communication interaction types; we can now follow this approach to 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.

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 in the Gridkit reflective middleware platform. This combination is capable of supporting configurable, reconfigurable, and flexible security policies.

To date, we have designed and implemented a set of security mechanisms to support the RPC and multicast-based group interaction models. 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.

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

[An03] Andersen, A., Blair, G.S., Myrvang, P.H., Stabell-Kulo, T., “Security and Middleware”, WORDS 2003, Guadalajara, Mexico, January 2003.

[At00] 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] J. Bacon, D. M. Eyers, K. Moody, and L. I. Pesonen, "Securing publish/subscribe for multi-domain systems", In Proc. of the 6th International Middleware Conference (MW'05), Grenoble, France, Nov. 2005.

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

[Bu90] Burrows, M. and Abadi, M., and Needham R., "A logic of Authentication", ACM Transactions on Computer Systems, Vol. 8, No 1, 1990.

[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, Nov 2004

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

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

[Go95] Gong, L., Syverson, P., "Fail-Stop Protocols: An Approach to Designing Secure Protocols", in Proceedings of the 5th IFIP Working Conference on Dependable Computing for Critical Applications, Urbana-Champaign, Illinois, USA, 1995.

[**Gr03**] Grace, P.; Blair, G.; Samuel, S.; ReMMoC: A Reflective Middleware to Solve Mobile Client Interoperability, In Proc. International Symposium of Distributed Objects and Applications (DOA'03), Catania, Sicily, November 2003.

[Gr05] Grace, P., Coulson, G., Blair, G., Porter, B., "Deep Middleware for the Divergent Grid", Proceedings of the 6th IFIP/ACM/USENIX International Middleware Conference 2005, Grenoble, France, November 2005.

[Mc98] McDaniel, P., Honeyman, P., Prakash, A., "Lightweight Security Group Communication", CITI Technical Report 98.

[Me96] Menezes, A., Oorschot, P., Vanstone, S., "Handbook of Applied Cryptography", CRC Press, ISBN: 0-8493-8523-7, October 1996.

[My05] Myrvang, P.H., Skogan, T.S., "The Obol Protocol Language", Department of Computer Science, University of Tromso, 2005

[Ne78] Needham, R., Schroeder, M., "Using encryption for authentication in large networks of computers", Communications of ACM, 21(12): 993-999, December 1978.

[OM02] Object Management Group, "Security service specification", technical report, Object Management Group, Mar. 2002.

[Su91] Sun Microsystems, "Simplified guide to the Java 2 platform, enterprise edition", technical report, Sun Microsystems, Inc., 1991.

[St88] Steiner, J., Neuman, C., and Schiller, J., "Keberos: an authentication service for open network systems", in proceeding Usenix Winter Conference, Berkeley: Calif., 1988.

[St96] 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).

[Sy96] Syverson, P. and van Oorschot, P. C., "A unified cryptographic protocol logic", Naval Research Laboratory, CHACS Report 5540-227, Washington, USA, 1996.

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

[Wo98] C. K. Wong, M. Gouda, and S. S. Lam, "Secure group communications using key graphs," in *Proc. ACM SIGCOMM'98*, Vancouver, B.C., 1998, pp. 68–79.