A Java-Based Distributed Platform for Multilateral Security*

A. Pfitzmann**, A. Schill*, A. Westfeld**, G. Wicke**, G. Wolf*, J. Zöllner*

Dresden University of Technology, 01062 Dresden, Germany
*Institute for Operating Systems, Databases and Computer Networks
**Institute for Theoretical Computer Science
{pfitza, schill, westfeld, wicke, g.wolf, zoellner}@inf.tu-dresden.de

Abstract. We describe a new approach and system platform for enabling multilateral security in distributed applications. The major goal is to support users configuring their end systems and to negotiate among security requirements of different users with heterogeneous roles. Typical security features such as confidentiality or integrity of transmitted data are presented to the user at different levels of abstraction based on an inheritance hierarchy, according to his background knowledge and experiences. The system platform is implemented in Java, with distributed interaction based on Java RMI (Remote Method Invocation). It enables flexible integration of existing security libraries and facilities. As a validation example, we present a teleshopping scenario that has been realized using the support platform.
Keywords: Multilateral security, Java RMI, distributed applications, distributed platforms, teleshopping

1 Introduction

In a typical distributed application, there are several parties interconnected by a network interacting with each other. Each of the parties may have different security requirements; as an example, a teleshopping scenario with clients, merchants and a bank may be considered and will be detailed later. Allowing each of them to express these requirements and actually use their chosen level of security is what multilateral security is about. In real life, multilateral security is generally well accepted, e.g. when negotiating and signing contracts.

There are technical as well as organizational means to achieve this goal. The following paper concentrates on the technical aspects. In particular, we present a new approach and system platform to make multilateral security possible and usable for the end-user. It enables (1) pre-configuration of security issues of local systems and distributed applications (2) negotiation of the configurations between communicating

* This research is sponsored by the German Ministry of Education, Science, Research and Technology (BMBF).

users, and (3) different levels of abstraction concerning the presentation of security features to the end user and application programmer. This way non-experts in the field of cryptography and security are explicitly supported to use our multilateral security platform.

The system implementation is based on Java. Remote interactions as well as coarse-grained local interfaces are mapped onto Java RMI (Remote Method Invocation). The RMI allows easy and transparent distribution of applications (and method calls, for that matter).

Inheritance features of Java are intensively used for organizing security mechanisms at different abstraction levels that are accessible via an API (Application Programming Interface). Existing security facilities can be integrated flexibly; for example, Java-to-C mappings were exploited for incorporating C-based security libraries.

The paper is organized as follows. Section 2 discusses related approaches in the areas of security facilities and platforms. Section 3 presents the design and implementation of our security platform. Major conceptual aspects are configuration, negotiation and abstraction. Section 4 presents a teleshopping scenario that was built with our support platform as a validation example. Finally, Section 5 discusses our first experiences and gives an outlook to future work.

2 Related approaches / foundations

SSL [SSL] operates on the network socket layer, focusing on end-to-end connection security. It allows for different configurations on the client's side and a pseudo-negotiation between client and server. Configuration is limited to selecting one or more of some predefined "ciphersuites" which describe fixed combinations of algorithms for key exchange, encryption and hashing (e.g. DES, RSA and SHA) referenced by an index number. The client sends a list with its preferred ciphersuites to the server. The server then accepts the highest-ranked one it can support or denies the connection. Further negotiations are not intended.

PLASMA [Gehr_94] does not have the perspective of multilateral security when speaking of negotiation. This platform deals with divergences of security policies of autonomous domains, end-systems and applications but not the end-user. Local rules as formal security policies are the negotiation base. Although they do not directly support the security requirements of users and do not offer a user-oriented abstraction of security features like our platform they try to design a possibly automatic negotiation.

The Java Security API [Java_97] is Suns approach to provide security for Java programs. It supports - to a certain degree - abstraction of security mechanisms, but the actually used algorithms have to be determined at development time, thus giving up possible flexibility. The API is still under development and therefore not exceedingly extensive. The Cryptix libraries by Systemics, Inc. [Cryp_97] are another product for Java Security. They provide a lot of implemented algorithms and are freeware, but are not standardized like the Java API. Both lack the flexibility and

abstraction we need for our platform, but are integrated as providers for cryptographic algorithms.

Most other security related platforms we know, e.g. DCE [DCE_92], CORBA [CORB_97], TINA [StWi_97], Microsoft CAPI [Wiew_96] or PLASMA [Kran_96, PLASMA], offer security features such as user authentication (DCE Kerberos) or encryption (Microsoft CAPI).

None of these realize multilateral security in a convincing way, cf. [WSWZ_97].

3 Design and Implementation of the support platform

What we need for multilateral security is:

1. user-defined external **configuration** of the security features of applications to express security preferences,
2. **negotiation** between the systems of users to overcome the problems of differing configurations,
3. an **abstract view** on security mechanisms as well as an abstract **API** to provide the flexibility needed for external configuration.

Our technical approaches to these three problems are discussed in the following chapters. Figure 1 shows the architecture of the whole system.

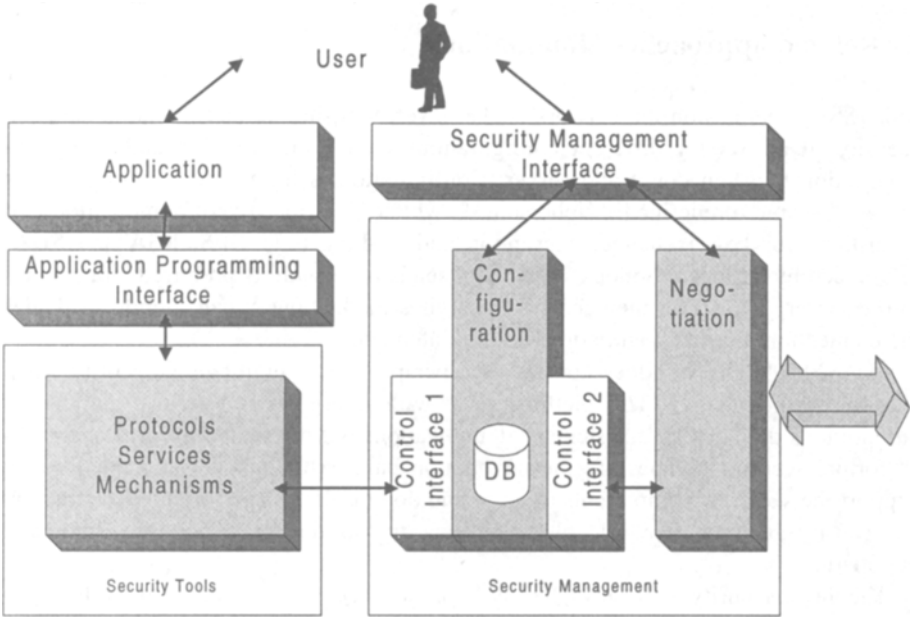

Fig. 1. Architecture of the support platform

The major interfaces shown in Figure 1 are:
- the Security Management Interface (SMI) which consists mainly of the presentation of the configuration (see 3.1) and negotiation (see 3.2) to the end-user,
- the Application Programming Interface (API) (see 3.3),
- the Control Interface 1 which provides access for the Security tools (and the API, respectively) to the current configuration, and
- the Control Interface 2 which provides similar functionality for the negotiation component.

The Control Interfaces offer a simple method to access the configuration data. They are implemented using RMI, which would allow the distribution of the platform, although this is not planned yet.

3.1 Configuration

The configuration is the main part of the platform visible to the end-user. Therefore, special attention had to be paid to the ease of use for the inexperienced user. Our solution to this problem is the following:

User interface: The user interface enables end-users to configure security mechanisms and services for their system and distributed applications easily. The system security configuration interface (see Figure 2) is part of the SMI and provides the possibility to enter central configuration data which are the base for the security configuration of all distributed applications. For each security requirement such as confidentiality, integrity, anonymity or accountability, the user can choose his preferred security mechanisms. The system provides a list of available mechanisms. Further security mechanisms can be downloaded from platform-supporting Internet servers. The end-user has the possibility to accomplish detailed settings for each security mechanism, e.g. choosing of different DES modes (ECB, CBC, OFB, ...) or the key length for an RSA key. To support the end-user, the platform includes an initial system configuration file at delivery.

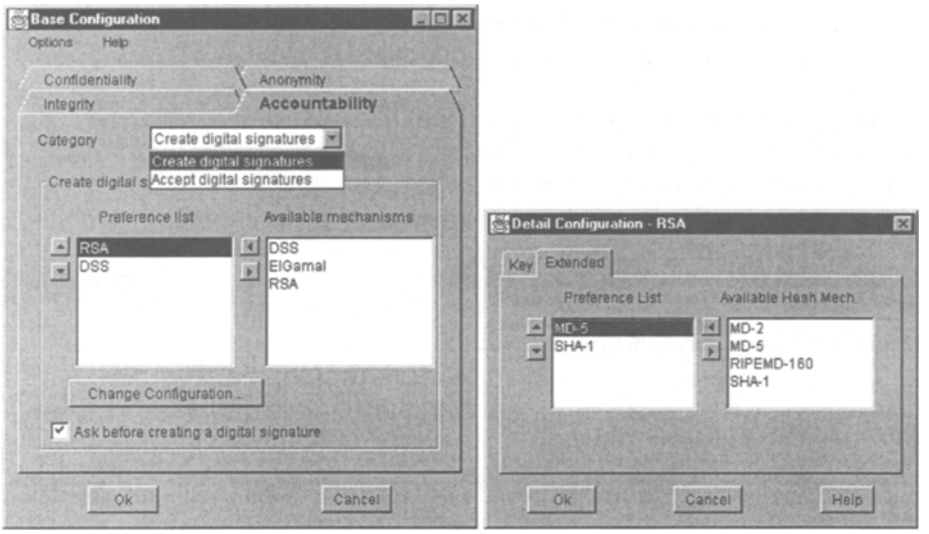

Fig. 2. System security configuration

Building on the central system security configuration, the end-user can express his security requirements for each application within the application security configuration (see Figure 3).

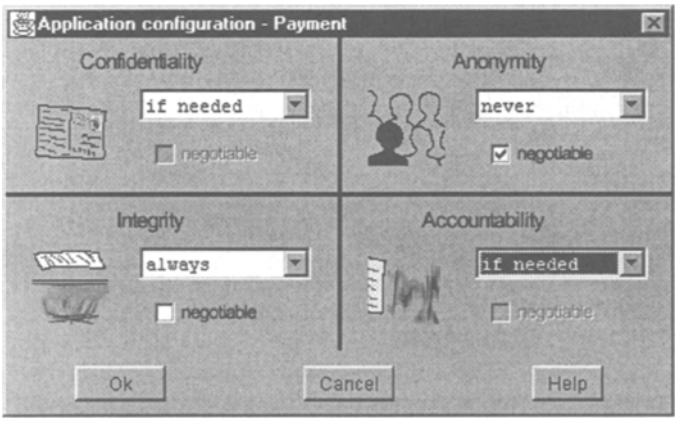

Fig. 3. Application security configuration

The user has to make a decision about which security requirements he would like to be met for this specific application. A mouse click on the icons leads to the more detailed configuration similar to the system security configuration (see Figure 2).

User support: The modules rating and plausibility check provide further user support for the configuration of system and applications. The rating helps non-experts to verify the achieved security level of security mechanisms by means of selected evaluation criteria. The platform offers two kinds of rating: absolute rating giving detailed information about the security each mechanism provides, such as cryptographic quality, performance and maintenance, and relative rating listing evaluations for several mechanisms and supporting especially non-experts in comparing of unknown security mechanisms. The example in Figure 4 shows a comparison between DES and IDEA. These ratings have to be certified.

One ore more criteria can be defined as essential. The overall placing results from the lowest value among the essential criteria and the mathematical average of all criteria.

algorithm	DES	IDEA
security of the algorithm		
cryptographic quality	4	4
resources needed for cryptoanalysis	2	4
algorithm implementation	DES (SSLeay)	IDEA (cryptix-java 2.2)
security of the implementation	2	2
operational costs	3	3
costs if not available at local system		
purchase price	1	1
installation, logistics, maintenance	3	2
overall placing	15/6 = 2.5	16/6 = 2.7

Fig. 4. Relative rating for DES and IDEA

Before saving the system and application security configuration data in a secured database, the platform checks the consistency of the settings – the so-called plausibility check. For this purpose, the platform tests the configuration data on the basis of rules (see below) and notifies the user if it found a missetting or unrecommended combination. Such plausibility rules are for instance:

- `if` anonymity of the sender `then` encryption recommended,
- `if` anonymity of the sender `then` no digital signatures,
- `if` anonymity of the recipient `then` encryption recommended.

3.2 Negotiation

All participants in the distributed applications can configure their systems according to their security requirements, as described in Chapter 3.1. Obviously, differences in the configurations will emerge. The natural way to overcome these differences in real life is negotiation between the parties involved. In order to free the user of needless interactions, the negotiation should work as automatically as possible.

The negotiation itself is split into two parts. First, the parties agree on general security requirements, such as confidentiality or accountability. After that, the mechanisms actually used to accomplish these goals are negotiated.

In Figure 5, the designed protocol for phase 1 is illustrated. It basically shows that both partners create a proposal for a possible common security configuration. These proposals are exchanged and evaluated. The initiator of the communication creates a further proposal which incorporates the proposals of all partners. If the system can not generate this proposal the negotiation component informs the user that no common ground for secure communication can be found. As a result, at least one of the users has to change his security configuration or the communication can not take place - just like in real life.

In phase 2, building on an achieved agreement on general security requirements, the systems negotiate actual crypto-algorithms. The principle is the same as described above, but the solution to hard conflicts extends to the possibility to load new mechanisms from a server or the incorporation of a security gateway which can transform between different formats and mechanisms.

As shown in Figure 1, the negotiation component is the central access point to a computer which runs applications developed using our platform. The Java RMI is used for the communication between the negotiation components of the different systems. Because the negotiation components act as the gateway of a system to the outside world, RMI is actually used for all the communication between the distributed applications or the parts thereof.

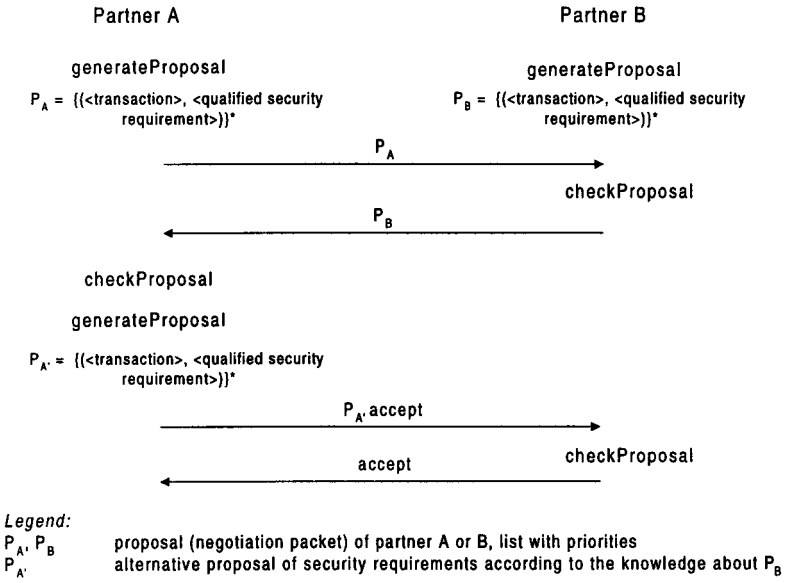

Fig. 5. The negotiation protocol for phase 1

Basically, the interface consists of the remote method `access`. This method is used exclusively by the negotiation components. The applications use the method `send`, which is implemented by the negotiation component of the local system. In Figure 6, the interface for the method `access` and the (current) implementation of `send` are shown.

```
// interface for remote access
public interface Access extends java.rmi.Remote {
    Message access(Message m) throws java.rmi.RemoteException;
}

// implementation of send
public Message send(String serverHost, String serverName,
Message m, int destination) {
  if (securMan == 0) {
      // create and install the security manager
      System.setSecurityManager(new RMISecurityManager());
      securMan = 1;
  }
  String serverStr = "rmi://" + serverHost + "/" + serverName;
  Message answer;
  try {
    /* find server "serverName" */
    Access server = (Access)Naming.lookup(serverStr);
    answer = server.access(m);
  }
  catch (Exception e) {
    System.out.println("Client: exception occured: " + e.getMessage());
    answer = new Message(m.id);
    answer.act = 0;
    answer.text = "RMI Exception" ;
  }
  return answer;
}
```

Fig. 6. The usage of RMI for communication

The negotiation component redirects the (via `access`) incoming messages to the appropriate applications.

3.3 The API

The API (Application Programming Interface) has to be flexible and abstract in order to allow external configuration. Figure 7 shows the hierarchy for the cryptographic core functions.

Figure 8 shows an excerpt of the API functions. Actually, the subtree for confidentiality is shown. Note the absence of any functions dealing with actual algorithms. Providing these would significantly limit the possibility of external configuration. The lower the abstraction level used by the developer, the less flexible the application is to configure. For example, if the developer chooses the function `encrypt()`, it is vital to him that the data are encrypted, not which actual algorithm is used (in this case, it could be any symmetric or asymmetric encryption algorithm supported by the platform). If the programmer explicitly chose

`cryptAsymHybrid()`, the user still could configure which asymmetric and symmetric algorithms would be used for the hybrid system, but he could not decide to use pure symmetric or pure asymmetric encryption instead.

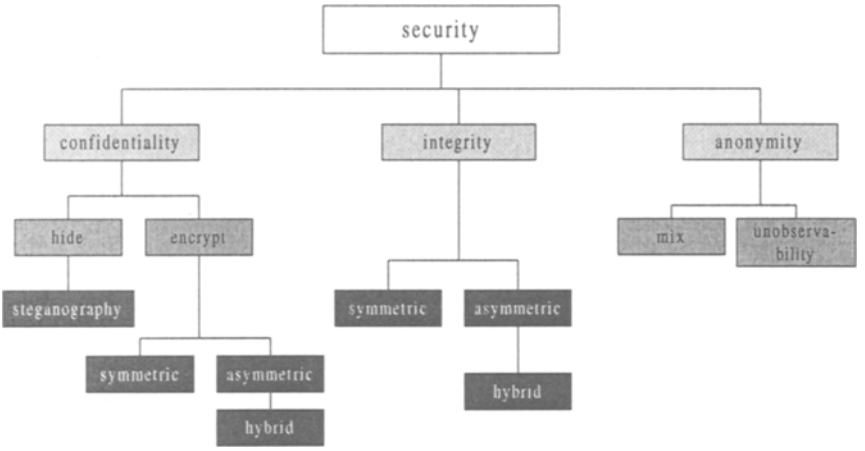

Fig. 7. Cryptographic core functions of the API

```
conceal(String plainText)
deconceal(String cipherText)
```
→ Provides confidentiality. Different forms of encryption or exotic mechanisms like steganography are used depending on the configuration.

```
crypt(String plainText)
decrypt(String cipherText)
```
→ En- and decryption (symmetric or asymmetric).

```
cryptSym(String plainText)
decryptSym(String cipherText)
```
→ Symmetric en- and decryption.

```
cryptAsym(String plainText, String recipient)
decryptAsym(String cipherText)
```
→ Asymmetric en- and decryption. The parameter is optional. If it is not provided, the recipient (and the according key) is chosen from the current connection configuration. Pure asymmetric encryption will be used only in rare occasions, hybrid encryption is used instead.

```
cryptAsymHybrid(String plainText, String recipient)
decryptAsymHybrid(String cipherText)
```
→ Asymmetric cryptography, explicitly hybrid.

```
embed(String plainText, String coverData)
extract(String stegoText)
```
→ Steganography - a rather exotic mechanism which can provide confidentiality.

Fig. 8. An excerpt of the API

Further parts of the API deal with key exchange, certificates and secure storage (local to the application as well as in the central configuration).

4 Application example: Teleshopping

As an evaluation example we chose an integrated teleshopping scenario. There are two main applications which are important: a catalogue system and an electronic payment system (which will be an on-line system like ecash [ecash]). This brings at least three parties into action: a customer, a merchant and a bank.

Imagine the following scenario:

Merchant M offers videotapes of different categories, Customer C can browse a catalogue, order tapes and pay with electronic cash. Figure 9 shows excerpts of their application configurations considering the transactions `<request catalogue>` and `<sending catalogue>`.

```
customer application configuration
application teleshopping

transaction <request catalogue>
confidentiality   yes  DES-CBC
                       DES-OFB
anonymity
   sender         yes  anonymizer
                       Mix
   recipient      no
   communication  -
accountability    no   RSA
integrity         yes  RSA


transaction <sending catalogue>
confidentiality   yes  DES
anonymity
   sender         no
   recipient      yes  anonymizer
                       Mix
   communication  -
accountability    yes  RSA
integrity         yes  RSA
```

```
merchant application configuration
application teleshopping

transaction <request catalogue>
confidentiality   -
anonymity
   sender         no
   recipient      no
   communication  -
accountability    yes  DSS
                       RSA
integrity         yes  DES-CFB
                       DSS
                       RSA


transaction <sending catalogue>
confidentiality   no
anonymity
   sender         no
   recipient      no
   communication  -
accountability    yes  DSS
                       RSA
integrity         yes  DES-CFB
                       DSS
                       RSA
```

Fig. 9. Application configuration of customer C and merchant M

In the proceeded negotiation phase 1 (compare Chapter 3.2, Figure 5), customer C and merchant M agreed on the security requirements confidentiality, accountability and integrity.

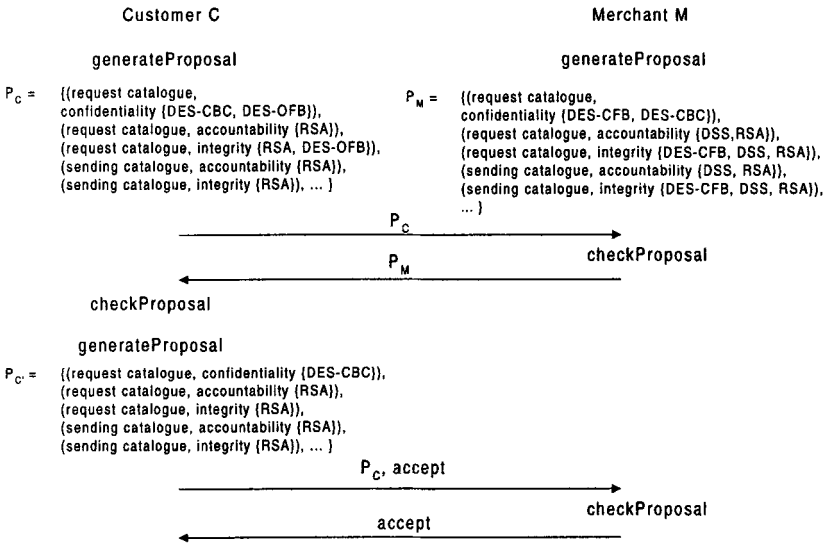

Fig. 10. Negotiation phase 2 for the teleshopping scenario

Figure 10 shows the protocol steps of negotiation phase 2. For all three security requirements, both participants prefer a distinct mechanism, e.g. DES-CBC and DES-CFB for confidentiality. Therefore the negotiation of customer C automatically generates the new proposal $P_{C'}$ considering the offer from merchant M. Our example describes that they agree on using DES in CBC mode to encrypt the transaction `<request catalogue>` and RSA for accountability and integrity of both transactions. Figure 11 shows the resulting connection configuration.

```
customer C - merchant M connection configuration
application teleshopping

transaction <request catalogue>
confidentiality   yes  DES-CBC
anonymity
   sender         no
   recipient      no
   communication  -
accountability    yes  RSA
integrity         yes  RSA

transaction <sending catalogue>
confidentiality   no
anonymity
   sender         no
   recipient      no
   communication  -
accountability    yes  RSA
integrity         yes  RSA
```

Fig. 11. Connection configuration of customer C and merchant M

We chose this scenario because it has numerous features we would like to evaluate our security platform with, such as:
- multiple parties with different interests,
- needs for confidentiality and authenticity,
- desire to stay anonymous in certain actions,
- contracts which have to be fulfilled and can be checked by third parties,
- different levels of security (the information about the account standings does not necessarily have to be secured as much as electronic money itself, for instance).

5 Experiences and Conclusions

Our work attempted to demonstrate that multilateral security is a valuable extension to the standard centralized approach to security in distributed systems. According to current experiences, this concept is also accepted by people unfamiliar with the field of security in information systems [MüPf_97]. Moreover, we have shown that generic platform support for multilateral security is possible.

Particular experiences with the conceptual approach are as follows. (1) The selected object-oriented design and implementation has proven to be a valuable approach for organizing and presenting security mechanisms. (2) As opposed to lower-level facilities such as sockets or RPC (Remote Procedure Call), object interactions with Java RMI are a powerful and flexible method to implement distributed interactions as found within our negotiation protocols. (3) Based on the Java implementation, integration of existing mechanisms, also provided in other programming languages such as C, was relatively easy. This is of particular importance due to the wide availability of basic security implementations, for example for supporting confidentiality or integrity. (4) Finally, it was possible to implement a concrete application with multilateral security features easily by exploiting our basic platform. Generic support will prove even more important when addressing a wide field of different applications that may at least partially share security mechanisms and negotiation protocols.

Future work will focus on the implementation of such alternative applications; possible areas might be extended teleshopping / electronic market scenarios or dedicated teleworking facilities. Moreover, the platform itself will be enhanced at the implementation level, for example by integrating further security libraries, by extending the negotiation protocols and by providing additional management tools. Rather specific security facilities such as mixes [Chau_81] for anonymity might also be imbedded. Furthermore, a concept of so-called transforming agents has been developed and will be further refined; such agents act as mediators between users with heterogeneous roles and requirements in case they are not able to completely agree to a common set of security mechanisms.

References

Chau_81 David Chaum: Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. Communications of the ACM 24/2 (1981) 84-88

CORB_97 Security, Transactions,...and More, 97-07-04
See: http://www.omg.org/corba/sectran1.htm

Cryp_97 Cryptix - Cryptographic extensions for Java.
See: http://www.systemics.com/software/cryptix-java/index.html

DCE_92 Open Software Foundation: Introduction to OSF TM DCE. Prentice Hall, Englew. Cliffs, 1992.

Ecash DigiCash: Solutions for Security and Privacy.
See: http://www.digicash.com/index_e.html

Gehr_94 M. Gehrke: Eine Sicherheitsarchitektur für kooperative und offene Umgebungen. Dissertation. Berichte der GMD Nr. 239, R. Oldenbourg Verlag, 1994

Java_97 Java Security, 19.11.97.
See: http://www.javasoft.com/security/

Kran_96 A. Krannig: „PLASMA - Platform for Secure Multimedia Applications". Proceedings: Communications and Multimedia Security II, Essen, 1996

MüPf_97 G. Müller, A. Pfitzmann (Hrsg.): Mehrseitige Sicherheit in der Kommunikationstechnik: Komponenten, Verfahren, Integration. Addison Wesley, Bonn 1997.

PLASMA „PLASMA - Platform for Secure Multimedia Applications". In: DeTeBerkom: Security - a Cornerstone of the Information Society
See: http://www.deteberkom.de/projekte/texte/Sec.eng.html

SSL The SSL Protocol, Version 3.0, 03/1996;
See: http://home.netscape.com/engl/ssl3/3-SPEC.HTM

StWi_97 S. Staamann, U. Wilhelm: CORBA as the Core of the TINA-DPE: A View from the Security Perspective. Object World Frankfurt '97, special track Distributed Object Computing in Telecommunications (DOCT'97), Frankfurt a.M., Germany, October 7-10, 1997.

Wiew_96 E. Wiewall: „Secure Your Applications with the Microsoft CryptoAPI". In: Microsoft Developer Network News, 5 (1996) 3/4, 1

WSWZ_97 U. G. Wilhelm, S. Staamann, G. Wolf, J. Zöllner: "Sicherheit in CORBA und TINA". In: G. Müller, A. Pfitzmann (Hrsg.): Mehrseitige Sicherheit in der Kommunikationstechnik: Komponenten, Verfahren, Integration. Addison Wesley, Bonn 1997.