Secure Communication in Distributed Ada

Jörg Kienzle, Thomas Wolf, Alfred Strohmeier

Swiss Federal Institute of Technology
Software Engineering Laboratory
1015 Lausanne Ecublens
Switzerland

email: {jkienzle, wolf, strohmeier}@di.epfl.ch
phone: +41 (21) 693.42.37 – fax: +41 (21) 693.50.79

Abstract. This document describes an implementation of secure communication for distributed applications in Ada 95. It explains the basics of cryptography and introduces the problem of key exchange. After a brief description of distributed systems in Ada 95, the modular structure of the implementation and the way it handles key distribution is presented.

Keywords. Distributed Systems, Security, Ada 95, Object–Oriented Programming, Communication, Cryptography.

1 Introduction

The advent of low–cost, yet powerful workstations combined with fast networks has made the distributed execution of applications over a network a viable alternative to the use of one centralized large computer. To fully exploit the resources of such a distributed system, software has to be designed and implemented in a radically different way than for a centralized system.

The main differences are the inherent parallelism of distributed execution and the fact that the different parts of a distributed application have to communicate with each other to accomplish their task [10]. Various programming models on different levels of abstraction are used to deal with these issues.

Examples for communication paradigms are simple message passing, or on a higher level remote procedure calls (RPC) [2] or remote method invocation. Whatever way is used, potentially confidential data has to be transported from one node of the distributed system to another one. Data travelling over a network can be eavesdropped upon without great effort, provided that the attacker has physical access to the communication lines. One solution to this problem is to encrypt sensitive data.

This paper describes an implementation of secure communication for distributed systems in Ada 95. It is an open implementation in the sense that it allows any cryptographic algorithm to be used.

The outline of this paper is as follows: in section 2 we give a short review of private and public key algorithms, section 3 gives an introduction to distributed systems in Ada 95, section 4 shows different levels of using encryption, followed by an overview of the implementation of the distributed systems annex of the Ada 95 reference manual that we worked with in section 5, and section 6 explains our approach of providing user–transparent secure communication for this implementation.

2 Private and Public Key Algorithms

In modern cryptography *plaintext* is transformed into *ciphertext* using an encryption function E and a key k (see fig. 1). The inverse function D expresses decryption of the ciphertext to reproduce the plaintext, again using the same key k. This kind of algorithm is called symmetric or private key algorithm [3]. The Data Encryption Algorithm (DEA, [1]) for example is a very popular cryptographic algorithm of this category.

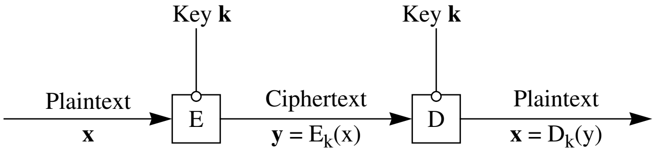

Fig. 1 Encryption and decryption

The key is an essential feature of a cipher. Keeping the functions E and D secret would suffice to conceal the plaintext, even without a key. But there are two major problems in this case: if the deciphering function D becomes known to a third party, all subsequent communications using E can also be read by that third party; and an encryption function E not using a key is more vulnerable to cryptanalytic attacks than one that does employ a key. Parametrizing the functions with a key k actually defines a class of ciphers: if one key is compromised, it can be changed, while the cipher functions E and D can remain in use.

When encryption is used in communication, there must be prior agreement between the communicating parties about all aspects of the procedure. An algorithm and the method of using it must be agreed upon. The most difficult requirement is that a key must be chosen and made available at both ends of the communication path. Before encrypted data can flow over the line, the key must make a similar journey. Keys can be encrypted using other keys, but in the end at least one key has to be exchanged by some means.

This problem can be solved using public key ciphers, also called asymmetric ciphers, e.g. RSA [8]. Although these cannot be implemented as efficiently as private key ciphers, they can be used to exchange small amounts of data, e.g. a private key that is to be used for subsequent communication.

In public key algorithms, the keys used for encrypting and decrypting are not the same. The decryption key kd is kept secret by the receiver. The other key ke is made public, enabling anyone to encrypt data for the one receiver to whom that key belongs. When A wants to establish a secure communication with B, all he has to do is create a private key k_A, encrypt it with the public key ke_B of B and send it to B (see fig. 2). B is the only

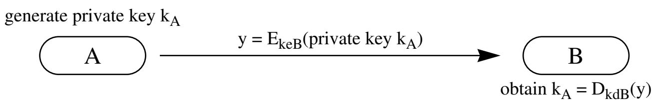

Fig. 2 Private key exchange using a public key

one who knows the decryption key kd_B and therefore the only one who can successfully decrypt the message to obtain k_A. Now, both A and B know k_A and they can exchange data using an efficient private key algorithm. This method is used for key exchange in the cryptography implementation for Ada 95 presented in this document.

3 Distributed Systems in Ada 95

As distributed systems are getting more and more widespread, many programming languages provide libraries or precompilers to facilitate the creation of distributed applications, such as `rpcgen` for the C language. Ada 95 goes even a step further by incorporating a model for programming distributed systems into the language itself.

A distributed program in Ada 95 is comprised of one or more virtual nodes (called *partitions* in Ada 95) that execute independently. Partitions themselves can hold several library units, some of which constitute the interface towards other partitions. The process of mapping the partitions of a program to the actual physical nodes in a distributed system is called configuration and is done after compilation. This makes it possible to create different distributed configurations of the same program without recompilation.

The communication paradigm between these partitions is the remote procedure call (RPC). Remote procedure calls are highly transparent to the application: only the interface packages of a partition have to be marked as such by pragmas. Pragma REMOTE_CALL_INTERFACE marks packages declaring subprograms that may be called remotely; these packages are called *RCI packages*. In order to send the parameters of an RPC over a network, they have to be transformed into a flat stream of bytes and converted back to the original structure on the receiving side. This *marshalling* and *unmarshalling* is performed by code inserted by the compiler on both the sender and the receiver partition for every subprogram that can be accessed remotely. These pieces of code are called *calling stub* and *receiving stub*, respectively. How data gets actually sent over the net is handled by the Partition Communication Subsystem (PCS). The Ada Reference Manual only specifies the interface to the PCS.

Our implementation of secure communication for the PCS was built using the GNAT [9] compiler and the GARLIC (Generic Ada Reusable Library for Interpartition Communication, [7]) Partition Communication Subsystem.

4 Some Levels of Abstraction

Given the availability of a library implementing a cryptographic algorithm, we can identify different levels of abstraction for the use of encryption, which we will discuss in this section.

4.1 The Encryption Library

The encryption library itself lies at a very low level of abstraction. It doesn't have to know about data types or any other application–specific things. Its sole purpose is to encrypt and decrypt sequences of unstructured data, again producing sequences of equally unstructured data. This view immediately leads to the following, very simple interface.

```
with Ada.Streams;
package Crypt is

   subtype SEA is Ada.Streams.Stream_Element_Array;
   --  for brevity

   type Key_Type is private;
```

```
    function Encrypt (Key        : in Key_Type;
                      Plain_Text : in SEA) return SEA;

    function Decrypt (Key        : in Key_Type;
                      Cipher_Text: in SEA) return SEA;

    function Get_Key (...) return Key_Type;

private
    ...
end Crypt;
```

This interface is slightly simplified. In particular, there might be more functions to generate keys, or pairs of public and private keys in the case of an asymmetric algorithm. Yet the above interface is fully sufficient for our discussion.

4.2 Ad–hoc Encryption

Of course the application programmer always has the possibility to encrypt "by hand" any arbitrary data item. The main advantage of this approach is that it allows to tailor the use of encryption precisely to the application's needs, thus minimizing the overhead incurred by the use of encryption. This applies to centralized applications as well as to a distributed setting, in particular if the configuration of the distribution is known at design time. The approach is not really suitable for post–partitioning, especially if the configuration is subject to changes.

The programmer has full control over the application of cryptography, but he is also responsible for using it correctly: in a distributed system, the application itself must deal with an eventual key exchange, and both the caller and the callee of a remote procedure call must be aware of the fact that some arguments are encrypted. In particular, the interface for the remote subprogram must be changed, e.g. from

```
function Verify (Id : in Credit_Card_Number) return Boolean;
```

to

```
function Verify (Coded_Id : in Crypt.SEA) return Boolean;
```

For increased type safety, ad–hoc encryption requires the creation of a strongly typed wrapping layer around the very general low–level interface to the encryption library. Using this interface directly on the application level would require the use of unsafe type conversions, and as a consequence the advantages offered by Ada's strong type system would be lost.

Since a strongly typed intermediate layer is recommended anyway, the following, more structured approach for user–level encryption is certainly the preferred one.

4.3 Encryption per Type

Ada 95 defines for each type the attributes `'Write` and `'Read` as well as `'Output` and `'Input` to linearize objects into a flat `Stream` and vice versa. These attributes can be redefined by the application. We can redefine them to automatically encrypt (for `'Write`) or decrypt (for `'Read`) any value of the given type. In this way, the use of cryptography is hidden from the rest of the application. All references to the (potentially unsafe) interface of the encryption library as well as the complexities of employing encryption are thus localized in one place (namely, the implementation of these attributes) instead of cluttering the whole application.

Using this method works particularly well in a distributed system. Marshalling and unmarshalling of parameters of remote procedure calls in Ada 95 is done transparently in the stubs which are automatically generated by the compiler. Within these stubs, the parameters are transformed into a linearized representation in the form of a `Stream` using attributes `'Write` or `'Output`, while at the callee's side, parameters are re–transformed into an internal representation using `'Read` or `'Input`. Redefining these attributes to use encryption thus gives secure transmission for all objects of that type.

The following simplified example illustrates this approach:

```
with Ada.Streams;
package Credit_Card_Numbers is

   type Credit_Card_Number is private;

   --  Operations on type Credit_Card_Number elided...

private

   type Credit_Card_Number is ...;

   procedure Read_Credit_Card_Number
              (Stream : access Ada.Streams.Stream'Class;
               Id     : out    Credit_Card_Number);

   for Credit_Card_Number'Read use Read_Credit_Card_Number;

   procedure Write_Credit_Card_Number
              (Stream : access Ada.Streams.Stream'Class;
               Id     : in     Credit_Card_Number);

   for Credit_Card_Number'Write use Write_Credit_Card_Number;

end Credit_Card_Numbers;
```

In this example, we ignore the precise definition of the type `Credit_Card_Number` because it is of no interest for this discussion.

In the body of this package, we can implement the two procedures using the cryptographic algorithm of our choice; even if a key exchange is necessary, it may also be hidden within the implementation of this package:

```
with Crypt;
with Ada.Unchecked_Conversion;
package body Credit_Card_Numbers is

   --  Operations on type Credit_Card_Number elided...

   key : Crypt.Key_Type;  --  The encryption/decryption key used.

   procedure Read_Credit_Card_Number
              (stream : access Ada.Streams.Stream'Class;
               id     : out    Credit_Card_Number) is

      clear_id : Crypt.SEA :=
                 Crypt.Decrypt (key, Crypt.SEA'Input (stream));

      subtype Buff_Type is Crypt.SEA (clear_id'Range);

      function From_SEA is new Ada.Unchecked_Conversion
                                    (Buff_Type, Credit_Card_Number);
   begin
      id := From_SEA (clear_id);
   end Read_Credit_Card_Number;
```

```
   procedure Write_Credit_Card_Number
             (stream : access Ada.Streams.Stream'Class;
              id     : in     Credit_Card_Number) is

      subtype Buff_Type is Crypt.SEA (1 .. id'Size / 8);

      function To_SEA is new Ada.Unchecked_Conversion
                                 (Credit_Card_Number, Buff_Type);
   begin
      Crypt.SEA'Output (stream, Crypt.Encrypt (key, To_SEA (id)));
   end Write_Credit_Card_Number;

begin -- body of package
   key := Crypt.Get_Key (...);
   --  Do key exchange here, if necessary
end Credit_Card_Numbers;
```

In this way, we retain the main advantage of the first approach – minimizing the overhead of using a cryptographic scheme – while also minimizing its disadvantages. However, the application programmer still has to write the above package, and he still has to deal himself with all the complexities of an eventual key exchange, which may involve the creation of a key server and other entities that basically are unrelated to the application domain.

4.4 Transparent Encryption

Finally, there is a third level of abstraction: the PCS might encrypt everything that is sent to another partition, moving the concern for employing cryptography completely out of the application programmer's realm. It now has become a configuration concern – so we would like to have a way to specify the use of encryption on this level that is transparent to the application. This becomes especially interesting if it is possible to specify the encryption scheme to be used on a per–link basis, i.e. if we can configure the application to use different algorithms for secure communication between different pairs of partitions. Consider a distributed system where some links are safety critical and others are not: we would like to encrypt data only on the vulnerable links. Two things are needed to accomplish this:

- a configuration language powerful enough to allow the specification of such special characteristics of links between partitions, and
- an implementation of the PCS flexible enough to support these characteristics.

The rest of this paper will detail our extension of the PCS to provide flexible and totally transparent encryption on links.

5 GARLIC

GARLIC [7] is an implementation of the PCS as specified by annex E of the Ada Reference Manual [4] for the GNAT compiler. It implements and encapsulates the protocols needed to provide the semantics of remote procedure calls. Simultaneous RPCs (i.e. from different tasks) are possible, and the called node also can handle RPCs in parallel: for each RPC request, a task is started.

GARLIC itself is independent from the actual communication protocol used – several different protocols may be employed, e.g. TCP/IP or UDP or something totally different (see fig. 3). This flexibility is achieved by dynamic binding using object–oriented programming techniques. We will detail the approach later on, since it is basically the same mechanism that is used within our secure communication implementation to make it independent from the cryptographic algorithms actually used.

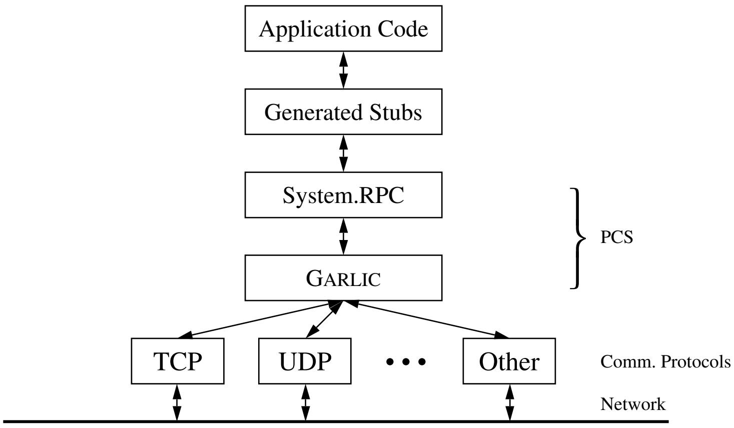

Fig. 3 The structure of GARLIC

The challenge in a distributed system is to make all the components work together. In Ada, each partition must receive a unique partition ID. GARLIC handles this by providing a partition ID server at the *lead partition* (the one executing the main program). At elaboration time each partition contacts this server and registers its own network address, receiving back its partition ID (steps 1 and 2 in fig. 4). The network address of the lead partition is given by an environment variable. Once all instances of GARLIC have elaborated, each RCI package registers its name and partition ID on the name server, which also runs on the lead partition (step 3).

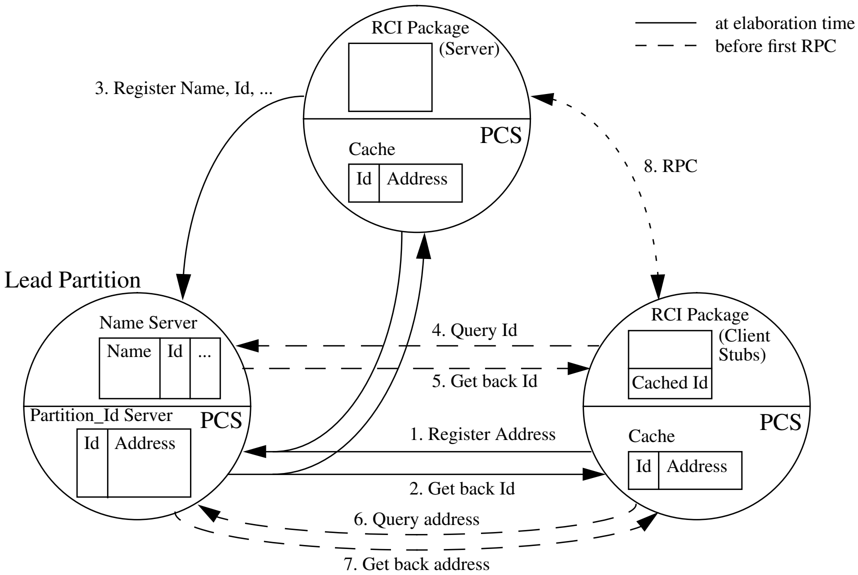

Fig. 4 Data Structures within GARLIC

To obtain the address of a partition before issuing the first RPC, the lead partition is contacted to look up the partition ID of the RCI package (steps 4 and 5). Then, the corresponding address is queried (steps 6 and 7). Partition ID and network address are cached so that later RPCs to that same partition can reuse the information obtained the first time. All of this happens inside the PCS and the stubs, totally transparent to the higher application layers.

6 Security Implementation

There were two basic requirements for our implementation of secure communication: it should be flexible, and the interface of the PCS defined in the Ada Reference Manual was not to be changed. The latter implies that our implementation must be transparent to the application and hence the specification of the cryptographic methods to be employed becomes a configuration concern.

6.1 Modular Structure

To make the security library independent of the cryptographic algorithms actually used to encrypt the data, it was designed in a very modular way. The new hierarchical library structure in Ada 95 made it possible to group all configuration and key management routines in a top level library unit (called `Security`), whereas different cryptographic algorithms are implemented using child units. This top–level unit itself is a child unit of `System.RPC`.

The following figure shows the security library structure:

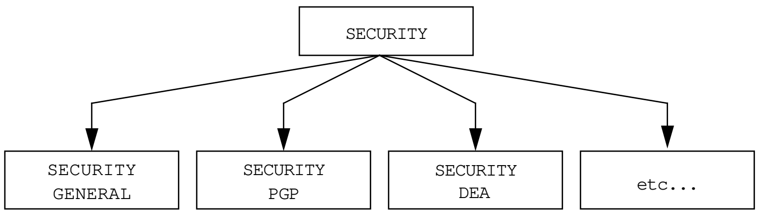

Fig. 5 Security library structure

The top unit contains no direct references to its child units and therefore does not depend on any particular cryptographic algorithm. This is accomplished using the new object–oriented features in Ada 95.

The `Security` unit exports an abstract type called `Encryption_Type`. Each child unit must derive from this type and create its own type (e.g. `DEA_Encryption_Type`). During elaboration each child unit must register a dummy instance of this newly created type at the `Security` package by calling the procedure `Register_Encryption` in the top unit. There, the dummy instance gets stored in an internal table. From now on, this dummy instance can be used to make dispatching calls to the corresponding child unit.

The following source code shows the interface of the abstract type declared in the top level library:

```
package System.RPC.Security
   type Encryption_Type   is abstract tagged limited private;
   type Encryption_Access is access all Encrypition_Type'Class;
   type Key_Type          is abstract tagged private;
   type Key_Access        is access all Key_Type'Class;
```

```ada
private
   type Encryption_Type is abstract tagged limited null record;
   type Key_Type        is abstract tagged null record;
   type Encryption_Name is String;

   procedure Encrypt (Crypt  : in     Encryption_Type;
                      Key    : in     Key_Access;
                      Stream : in out Stream_Element_Array_Access)
      is abstract;

   procedure Decrypt (Crypt  : in     Encryption_Type;
                      Key    : in     Key_Access;
                      Stream : in out Stream_Element_Array_Access)
      is abstract;

   function Get_Name (Encryption : in Encryption_Type)
      return Encryption_Name is abstract;

   procedure Register_Encryption (Encryption : in Encryption_Access);
      -- Note: this is not a primitive operation!

end System.RPC.Security;
```

To implement a particular encryption algorithm a child unit must declare a type derived from this abstract tagged type:

```ada
package System.RPC.Security.Example_Encryption is
   type Example_Encryption_Type is new Encryption_Type with private;
   type Example_Key_Type is new Key_Type with private;
   ...
private
   type Example_Encryption_Type is new Encryption_Type with null record;
   type Example_Key_Type is new Key_Type with
      record
         --  (data structures for the key)
      end record;
end System.RPC.Security.Example_Encryption;
```

In the statement sequence of the body of this child unit, the new encryption scheme must be registered during elaboration with our extension of GARLIC (i.e. with the `System.RPC.Security` package).

```ada
package body System.RPC.Security.Example_Encryption is

   Example_Dummy : aliased Example_Encryption_Type;

   function Get_Name (Encryption : in Example_Encryption_Type)
      return Encryption_Name is
   begin
      return ("EXAMPLE");
   end Get_Name;

   -- Other primitive operations, such as Encrypt, Decrypt, ...

begin
   Register_Encryption (Example_Dummy'Access);
end System.RPC.Security.Example_Encryption;
```

Since the `Security` package is completely decoupled from its children, the linker needs some additional information to be able to produce an executable. This information is given in the unit `Security.General`. Created by the (yet to be implemented) configuration tool, this unit contains a "with" clause for each encryption algorithm unit needed for a particular configuration. It also incorporates a table that specifies which cryptographic algorithm to use when communicating with a specific partition. This makes it possible to use a different encryption method for each link.

The different algorithms are identified using a type `Encryption_Name`, which is a short string of characters.

A user can implement a new encryption algorithm or use the ones already implemented. These are:

- NONE: No Encryption
- XOR: Xor every byte with a key (provides no real protection, but good for testing)
- DEA: Data Encryption Algorithm (a good private key algorithm)
- PGP: Pretty Good Privacy (a freeware public key algorithm)

6.2 Key Exchange

The `Security` library also provides a key exchange facility. It allows private key algorithms to exchange keys automatically. In order to use this facility, there must be a public key algorithm available as one of the child libraries.

During elaboration the different partitions of a distributed Ada program contact the lead partition to register their network address at the partition ID server and to get their unique partition ID. At the same time, they also register their public keys at the lead partition (see fig. 6). This first message is encrypted using the lead partition's public key *keLead*, which is made available to all partitions through an environment variable just like the lead partition's network address.

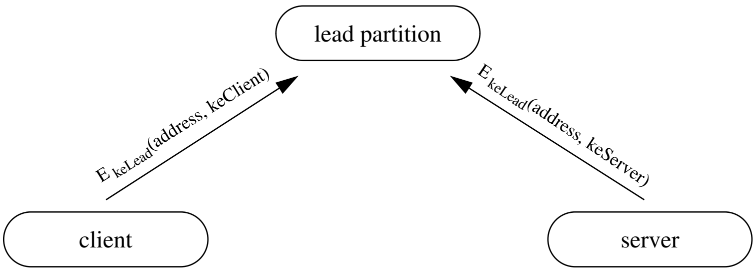

Fig. 6 Registering the partitions' public keys during elaboration

The first time a client wants to make an RPC to a server partition, it has to look up the network address of that partition. This is accomplished by sending the desired partition ID to the lead partition. Of course this call is also encrypted with the public key of the lead partition *keLead* (fig. 7).

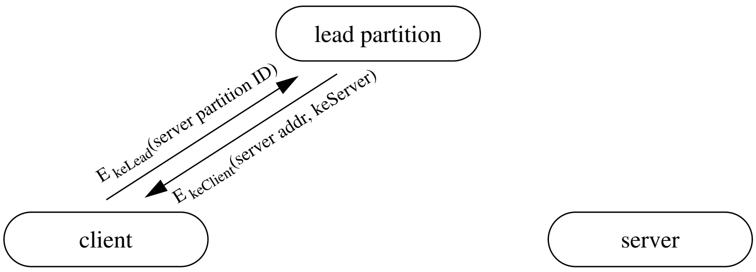

Fig. 7 Querying the server's network address and public key

The lead partition then looks up the corresponding network address in a table that has been created during start–up and sends back this address and the public key of the server partition encrypted with the public key of the client partition.

Next, the client generates a new private key *kClientPrivate*, encrypts it using the newly received public key *keServer*, and sends it to the server (fig. 8). Now that they both know the key *kClientPrivate* they can communicate securely.

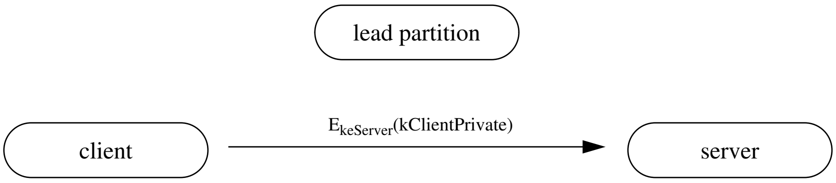

Fig. 8 Exchanging the key with the server partition

On both sides, the private key gets stored in a table so that it can be reused in subsequent RPCs. If a key becomes invalid (e.g. time–stamped keys that are only valid for a given time), a new key exchange is performed.

All this is encapsulated inside the top `Security` module. Child units only have to provide the following subprograms:

- `Generate_Key` — generates a new key (or two keys for public key algorithms)
- `Is_Valid` — returns true if a key is still valid

6.3 Writing a Distributed Application Using Secure Communication

Writing a distributed application using secure communication does not differ from writing any other distributed application in Ada 95. The following example shows a distributed 'Hello World' program using secure communication:

```
package P is
   pragma Remote_Call_Interface;
   procedure T(s : in String);
end P;

with Text_Io;
package body P is
   procedure T(s : in String) is
   begin
      Text_Io.Put_Line (s);
   end T;
end P;

with P;
procedure Main is
begin
   P.T("Hello, World!");
end Main;
```

The only difference is that the configuration tool must now create a `Security.General` package, telling each partition which link to encrypt using which algorithm and which public key algorithm to use for the key exchange. Although this configuration tool is not implemented yet, and the configuration language is still to be defined (currently, we are configuring manually), a hypothetical configuration program for our 'Hello World' example might look like this:

```
partition p1 is ();
partition p2 is (P);

procedure Main is in p1;

link l1 is (p1, p2);

for l1'Encryption use DEA;

key_exchange uses PGP;
```

7 Conclusion and Future Work

We have seen that there are different levels for employing cryptography. While letting the application take the responsibility for encryption may make sense in some cases, this invariably increases the complexity of the application. If private keys have to be exchanged (using a public key algorithm), the application even has to contain some kind of key server.

A transparent implementation of encryption frees the application from these mundane administrative tasks, but the granularity of encryption becomes coarser. Moving encryption and decryption into the lower layers of the PCS gives the desired transparency. Using clear modularization and object–oriented techniques, our extension becomes independent of the actual encryption schemes used; for specifying the encryption parameters only at configuration time, this kind of plug'n'play situation is mandatory.

This flexibility also opens the road towards other kinds of data transformations besides cryptography. One can think of transparent error detection in communications using some kind of checksum (e.g. CRC, or MD5), or transparently compressing and decompressing data, or even a combination of these. Our future work in this direction includes the implementation of new data transformations as mentioned above and research on the configuration language.

8 References

[1] ANSI, *Data Encryption Algorithm*, ANSI X3.92 – 1981

[2] A.D. Birrell, B.J. Nelson, Implementing Remote Procedure Calls, *ACM Transactions on Computer Systems* **2**(1), pp. 39 – 59, 1984.

[3] D.W. Davies and W. L. Price, *Security for Computer Networks*, John Wiley & Sons Ltd., 1992.

[4] ISO: International Standard ISO/IEC 8652:1995(E), *Ada Reference Manual*, Intermetrics, Inc., 1995.

[5] Intermetrics, *Ada 95 Rationale,* Intermetrics, Inc., Cambridge, Massachusetts, Jan. 1995.

[6] Y. Kermarrec, L. Pautet, and E. Schonberg, Design Document: Implementation of Distributed Systems Annex of Ada 9X in GNAT, *Technical report*, New York University, Courant Institute, June 1995.

[7] Y. Kermarrec, L. Pautet, S. Tardieu, GARLIC: Generic Ada Reusable Library for Interpartition Communication, *Proc. TRI–Ada '95*, Anaheim (CA), USA, 1995.

[8] R.L. Rivest, A. Shamir, L. Adleman, A method of obtaining digital signatures and public key cryptosystems, *Comm. ACM* **21**(2), pp. 120 – 126, Feb. 1978.

[9] E. Schonberg et al., GNAT: the GNU – NYU Ada Translator, a compiler for everyone, *Proc. TRI–Ada '94*, Baltimore (MD), USA, 1994.

[10] Andrew S. Tanenbaum, *Communication in Distributed Systems*, in *Modern Operating Systems*, pp. 395 – 462, Prentice–Hall, 1992.