Transparent Filtering of Streams in GLADE

Laurent Pautet
Computer Science Department
ENST Paris
75634 Paris cedex 13, France
pautet@inf.enst.fr

Thomas Wolf
Software Engineering Laboratory
Swiss Federal Institute of Technology
CH–1015 Lausanne, Switzerland
wolf@di.epfl.ch

Abstract

In the context of GLADE, the implementation of Annex E of the Ada Reference Manual for the GNAT compiler, a transparent extension of the PCS for performing arbitrary data transformations on the marshaled arguments of remote calls is presented. The concepts underlying the implementation and its implications on the configuration language are shown. A small example elucidates the chosen approach and illustrates at the same time the use of remote accesses to class–wide types for implementing distributed objects in Ada 95.

Keywords: *Ada 95, Compression, Cryptography, Distributed Objects, Distributed Systems, Security*

1 Introduction

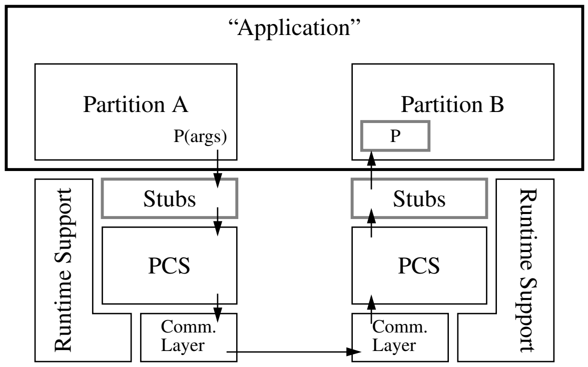
Ada 95 integrates a model for building distributed applications within the language itself (Annex E of the Ada Reference Manual [8]). Distribution introduces a whole new set of problems in applications, e.g., having to pass data across a network may raise performance or security issues. In this paper, we present a solution to these problems allowing arbitrary data transformations, such as compression or encryption, to be applied transparently to data sent over the network.

Section 2 of this paper briefly recalls the basic principles of distribution in Ada 95, before we give an overview of GLADE, the implementation of Annex E for the GNAT [7] compiler, in section 3. In section 4, we describe our filtering addition to GLADE, a transparent solution for applying arbitrary data transformations to RPC parameters. Section 5 describes how to configure filtering. The example in section 6 illustrates these concepts and also shows nicely how to use class–wide programming to implement distributed objects in Ada 95. Section 7 finally concludes the paper and points out some future directions of work.

2 Distributed Systems in Ada 95

In the model described by the language standard, a distributed Ada program is viewed as a composition of so–called *partitions* (virtual nodes), which communicate primarily through remote procedure calls (RPC, [2]). Each partition is a collection of Ada library units, some of which are marked by means of categorization pragmas as the interface of the partition towards other partitions. Such interface packages may contain remotely callable subprograms (marked by pragma `Remote_Call_Interface`, called RCI for short) as well as types used for their arguments (pragmas `Remote_Types` (RT) and `Pure`). A fourth pragma `Shared_Passive` is intended for modeling memory that is shared between partitions, either physically or through a distributed shared memory implementation. RT and RCI packages may contain in their visible parts access to subprogram types as well as access types to class–wide limited private types. These implicitly become remote access types, allowing dynamically bound remote calls as well as remote dispatching on the primitive operations of tagged types.

In order to send the parameters of a remote call across a network, they must be flattened into a simple stream of bytes. On the receiving partition, they then have to be reconstructed from this linearized representation again. This process is called *marshaling* and *unmarshaling* and is done transparently to the application by code inserted automatically by the compiler on both the sender and receiver partition for each remotely callable subprogram. These *stubs* use the language–defined attributes `'Write` and `'Read`, which append/retrieve data items to/from a `Stream`, to do the marshaling and unmarshaling. They then use the standardized interface (`System.RPC`) of the Partition Communication Subsystem (PCS) that is part of the Ada runtime support to actually do the remote call, i.e., to transport the data across the network to the destination partition (and back, if the remote call is not asynchronous). This flow of control is summarized in fig. 1:



The result of the call `P (args)` follows the inverse route.

Fig. 1 Simplified Control Flow During an RPC

The stubs for remotely callable procedures take care of argument marshaling and unmarshaling. The PCS is responsible for sending the request to the correct partition, to call the right subprogram on the receiver, and to correctly return results.

3 GLADE

The basic idea of the Distributed Systems Annex is to allow a user to develop his application the same way whether this application will be executed as several programs communicating on a distributed system or as a single program on a non–distributed system. The annex has been designed to minimize the changes to the source code of an ordinary centralized program when it is converted to a distributed program.

3.1 Development

The first step in the development of a distributed application is to create a non–distributed version of it. Of course, the design of this application should already take into account that some units ultimately will be accessed remotely. The application should be structured in a modular way such that it can be distributed at all—a monolithic program is not well suited for distribution. Also, the categorization pragmas (see section 2 above) should already be put in the appropriate library level compilation units. The units that require categorization are typically those that will be called remotely, or those providing types used in remote calls. With these pragmas, the compiler can verify the constraints that apply for remote calls and their parameters even in this centralized prototype of the final distributed application.

Using a centralized prototype has the advantage that most of the testing and debugging can be done with any off–the–shelf software development environment; specialized tools like debuggers for distributed applications are not needed. Of course, no calls will truly be remote calls yet. Before really distributing the application, one might even introduce remote calls by way of pragma `All_Calls_Remote`, which makes invocations of remotely callable subprograms go through the PCS even if the called subprogram resides on the same partition.

3.2 Partitioning and Configuration

Finally, the centralized application is partitioned and configured into a truly distributed application, i.e., into multiple partitions working cooperatively as part of a single Ada program. The process of mapping the partitions of a program to the physical nodes in a distributed system is called configuring the partitions of the program. The Distributed Systems Annex does not describe *how* a distributed application should be configured. It is up to the user to define the partitions in his program and on which machines they should be executed.

GLADE (Gnat Library for Ada Distributed Execution) is an implementation of the Distributed System Annex for the GNAT compiler. It is composed of two main parts:

- GARLIC [4] is an implementation of the PCS, providing the run-time support for executing a distributed Ada 95 application, and
- GNATDIST [3] is a configuration tool allowing users, by means of a configuration language, to partition their programs and to specify the physical nodes (machines) where the individual partitions are to be executed.

GNATDIST reads a configuration file and builds several executables, one for each partition. It generates the caller and receiver stubs when an Ada unit has been categorized as a remote call interface, and it adds some code to launch the different partitions, to invoke the communication layer of GLADE, called GARLIC, and to map Ada units or Ada unit stubs on each partition. All these operations—except writing the configuration file, of course—are transparent to the user.

The configuration language is designed to be simple and convenient to use. Its syntax is in some respects similar to the Ada syntax in order to ease the task of an Ada programmer: the keywords are mostly the same as in Ada, though they are sometimes used in a slightly different context.

3.3 Client/Server Applications using the Distributed Systems Annex

The "single distributed application" model of the Ada 95 language standard [8, E(1)] is not well suited to the development of client/server applications, which have a different structure: they usually comprise two programs (client and server), and cannot be defined easily as a partitioning of a single program.

The language standard gives an implementation enough leeway to accommodate other models of distributed applications. [8, 10.2(7)] for instance gives the permission to define several main procedures, one per partition, which departs from the basic model of a single distributed application. The Distributed Systems Annex relaxes some other aspects of the single application model.

Note that if we reconsider the development process of a distributed application, it is clear that the initial, centralized application must be structured accordingly if these steps are to be used for client/server applications. E.g., as a centralized application comprises one partition only, it can have only one main procedure, and the developer has to mimic other main procedures by executing them in separate tasks. Usually, the overhead of having to do this still pays off because the centralized prototype of the final distributed client/server application can be developed using the usual software engineering environment. If the distributed application were developed directly, without creating a centralized version first and using tasking to "simulate" the later distribution, special–purpose distributed development tools would have to be used (e.g. for debugging the distributed application).

GLADE has been designed to allow such development. Some features of its configuration language are specially dedicated to it. We will discuss configuration in detail in section 5.

4 Filtering

We define *filtering* as the transformation of data sent over a network to accommodate (non–functional) requirements such as performance or security issues introduced by distribution. As has been pointed out in [5], the mechanisms defined in the Ada language allow several levels of data filtering for parameters of remote calls. Besides ad–hoc filtering of the parameters on the application level with its obvious disadvantages there are two other possibilities:

- The application may redefine the `'Write` and `'Read` attributes of a type to do the filtering based on the type of each parameter. This allows tailoring the use of filtering to the application's needs, but may not be appropriate for all kinds of data transformations, e.g., compression would be done more efficiently on all parameters together *after* they have been marshaled.
- Filtering can be applied within the PCS after marshaling and before unmarshaling. This has several advantages; in particular, filtering becomes completely transparent to the user program: the filtering properties must be defined only when the application is configured. On the downside, *all* data sent between any two partitions will be filtered.

Which method is appropriate in a particular context depends on the application's semantics and requirements. The first method must be implemented necessarily in the application itself since only it can know how certain types should be marshaled, if a behavior other than the default one is desired.

Our filtering extension of GLADE provides ample support for the latter, transparent method. It consists of two parts: an addition to GARLIC providing the hooks allowing adding arbitrary filters, and some changes in the configuration language of GNATDIST to specify filtering properties. Filtering is specified for *channels*, which are logical connection between two partitions.

4.1 Implementation

GARLIC itself is structured as a subsystem implementing the interface specified by `System.RPC` in the language standard, providing synchronous and asynchronous remote calls. The actual network protocols to be used are decoupled from this higher–level functionality: different protocols may be used, and new ones may be added without recompiling GARLIC. This extensibility is obtained through the use of object–oriented techniques.

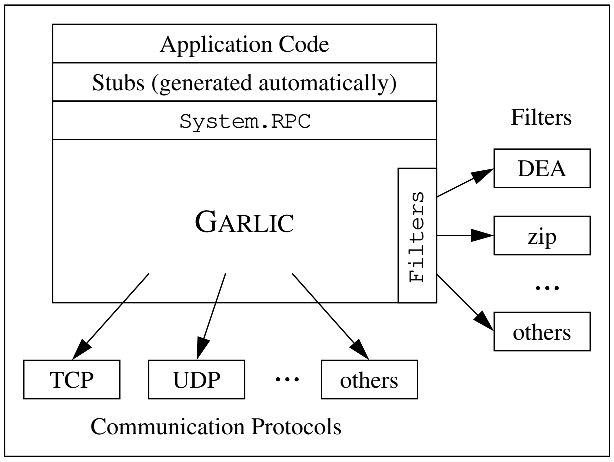


Fig. 2 Module Structure

Just as GARLIC is extensible on the protocol level, so is the filtering extension on the filter level. Filters are completely decoupled from the communication protocols used: the two are unrelated in all ways and can be thought of as two different dimensions of message processing. Filtering is transparent to the application, and nearly transparent to GARLIC itself: `System.Garlic.Filters` offers two very simple subroutines for GARLIC to call just before it passes the message on to a protocol package upon sending, and just after having received an as yet uninterpreted message from a protocol, respectively. The communication protocols, the RPC stubs, the application: all have no knowledge of filtering.

Object–oriented programming techniques are used to allow plug'n'play extensibility in the filter dimension. The module structure (shown in fig. 2) is simple: there is one package for filter management in GARLIC, `System.Garlic.Filters`. The interface of package `System.Garlic.Filters` towards the rest of GARLIC is very simple; there are only two routines for filtering streams:

```
package System.Garlic.Filters is

   function Filter_Outgoing
      (To     : in System.RPC.Partition_ID;
       Params : in Ada.Streams.Stream_Element_Array)
      return Ada.Streams.Stream_Element_Array;

   function Filter_Incoming
      (From   : in System.RPC.Partition_ID;
       Params : in Ada.Streams.Stream_Element_Array)
      return Ada.Streams.Stream_Element_Array;

private
   ...
end System.Garlic.Filters;
```

(GARLIC uses `Ada.Streams.Stream_Element_Array` as the internal representation of marshaled arguments. The interface as shown is slightly simplified, in reality, there is a third parameter to be able to distinguish messages corresponding to remote calls from GARLIC's internal housekeeping messages, which have to be handled differently in some cases.)

The above subprograms are called by GARLIC just before sending a message and just after having received a message. As can be seen from this interface, the filter to use for a particular message depends upon the partition the message shall go to or arrived from. This uniquely identifies the channel from the point of view of a partition. Package `System.Garlic.Filters` maintains this relation between channels and filters.

Individual filters are implemented in child packages of this package as objects of types derived from a root filter type, a tagged type declared in `System.Garlic.Filters`. At elaboration time, each filter registers itself with the top package, so that the latter can invoke the filters through dispatching calls on the registered object reference. This root filter type constitutes the interface towards the actual filters; it is declared in the private part of the filter manager's specification:

```
package System.Garlic.Filters is

   ...

private

   type Filter_Type is
      abstract tagged limited null record;
   type Filter_Access is
      access all Filter_Type'Class;

   type Filter_Params_Type is
      abstract tagged null record;
   type Filter_Params_Access is
      access all Filter_Params'Class;

   function Filter_Outgoing
      (Filter : in Filter_Type;
       Params : in Filter_Params_Access;
       Stream : in Ada.Streams.Stream_Element_Array)
      return Ada.Streams.Stream_Element_Array
   is abstract;

   function Filter_Incoming
      (Filter : in Filter_Type;
       Params : in Filter_Params_Access;
       Stream : in Ada.Streams.Stream_Element_Array)
      return Ada.Streams.Stream_Element_Array
   is abstract;
   -- Filter_Outgoing filters the Stream before it is sent, and
   -- Filter_Incoming performs the inverse filtering on a Stream
   -- that has been received.

   procedure Generate_Params
      (Filter          : in  Filter_Type;
       Public_Params   : out Filter_Params_Access;
       Private_Params  : out Filter_Params_Access;
       Needs_Exchange  : out Boolean)
   is abstract;
   -- This procedure must generate new filter parameters. It may always
   -- deliver the same parameters (hard–coded constant) or generate
   -- new ones each time it is called.
   --       If the algorithm is symmetric, its parameters are returned in
   -- Public_Params, Private_Params is null, and the last argu-
   -- ment Needs_Exchange is True.

   function Name
      (Filter : Filter_Type)
      return String is abstract;
   -- Has to return the filter package's name (e.g. "LZH").

   procedure Register (Filter : in Filter_Access);
   -- Called in the elaboration of a filter, so this package knows which
   -- filters are available.

end System.Garlic.Filters;
```

Each filter may have parameters associated with a channel; such parameters may be frequency tables for compression algorithms, or encryption keys (both public and secret key encryption is possible), or whatever is appropriate for the filtering algorithm. Upon the first communication on a channel, the filter manager calls `Generate_Params` for the filter to use on this channel. The parameters returned will then be used for all communication on this channel. If necessary, the filter manager will also perform a key exchange (see subsection 4.3 below).

Once the filter parameters are known and have been exchanged between the two partitions, the filter manager invokes the primitive operations `Filter_Incoming` or `Filter_Outgoing` of the appropriate filter through a dispatching call and then returns the result to GARLIC.

4.2 Adding a new Filter

It is easy to add new filters to an existing system due to the strict decoupling of filter management and the filtering algorithms themselves. Child packages implementing filters must simply register their filter objects with GARLIC's filter manager at elaboration. Using a new filter is equally simple: if the configuration language specifies a filter for a certain channel, it suffices to link the appropriate filter child package to the partitions' executables—the application does not have to be recompiled.

A sample filter implementation is shown below:

```
package System.Garlic.Filters.Example is
private
   type My_Filter_Type is new Filter_Type
      with record ... end record;

   type My_Parameter_Type is new Filter_Params_Type
      with record
         -- Whatever is needed (keys, tables, etc.)
      end record;

   -- Primitive operations as given by System.Garlic.Filters.

end System.Garlic.Filters.Example;
```

The new filter must be implemented as a derivation of the root filter type `System.Garlic.Filters.Filter_Type`. As each filter also needs its own filter parameters, an additional derived type for those is needed, too. The implementation must then implement the abstract primitive operations of the root filter type and finally register itself with the filter manager as shown below:

```
package body System.Garlic.Filters.Example is

   Example_Dummy : aliased My_Filter_Type;
   -- This object will be registered with the filter manager at elaboration.

   -- Primitive operations as given in System.Garlic.Filters:

   function Name
      (Filter : in My_Filter_Type)
      return String is
   begin
      return "EXAMPLE";
   end Name;

   -- Other prim. operations: Filter_Incoming, Filter_Outgoing,
   -- and Generate_Params.

begin
   Register (Example_Dummy'Access);
end System.Garlic.Filters.Example;
```

Which filter to use on a particular channel is defined during configuration (see also section 5). To pass configuration information from the (off–line) configuration phase to the running application, GLADE uses a mechanism based on the generation of a configuration package `System.Garlic.Elaboration`. The configuration tool GNATDIST generates a different body for this package for each partition, containing configuration information, e.g. filtering information[1]. During configuration, filters are identified using names (strings). At run–time, the filter manager has to map these names to filter types, which is done through the primitive operation `Name`.

4.3 Key Exchange

If filtering is used for transparently encrypting the messages during a remote procedure call and a secret key algorithm is used, this key first has to be made known to the receiver partition of the message. This mechanism is encapsulated within the filter manager (package `System.Garlic.Filters`), since it is the same for all filters.

For the key exchange to work, we require that there be one public key algorithm in the configuration. This algorithm is designated in the configuration as the "registration filter", which will be used to safely send secret keys across the network. The public key of a partition in the distributed application for this algorithm must be known to all other partitions that want to communicate with it. This is assured dynamically, as depicted in figure 3.

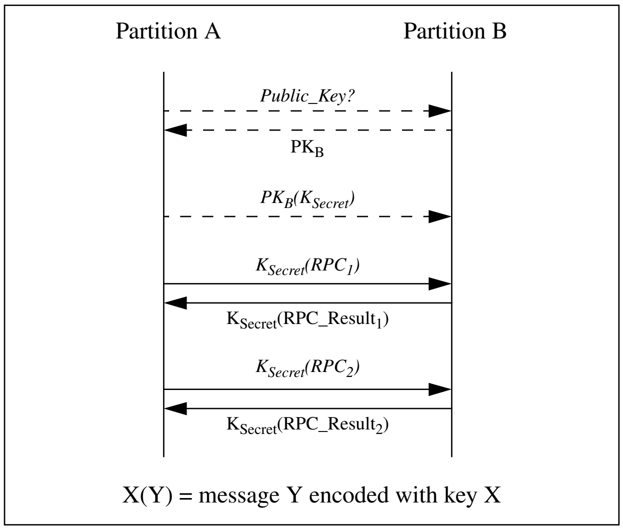


Fig. 3 Key Exchange

In this figure, partition A is about to make its first remote call to partition B. Before it can do so, it first asks the target partition for its public key for the registration filter (PK_B). Once it has received that, partition A uses it to encrypt the secret key K_{Secret} for the algorithm to be used for subsequent communication between these two partitions and sends that encrypted key to B. Only B is able to decrypt this message and to recover the session key K_{Secret}. The algorithm to use this key with is known to both partitions a priori as part of the configuration information. Once the session key has been exchanged in this manner, remote calls between these two partitions using secret key encryption are possible. (Note that this method is different from our original key exchange as described in [5]: the lead partition no longer acts as a global repository for registration keys.)

The use of this feature of course is not limited to encryption algorithms, one might also use the default null filter, which does nothing, as the registration filter and use the key exchange facility to share some arbitrary filtering parameters (e.g. compression levels) that must be known to both partitions. A filter indicates that a

1. The filter packages cannot be *private* child packages because they are `with`ed by the configuration package `System.Garlic.Elaboration` so they will be linked to the application.

parameter exchange is needed by returning `True` for the `Needs_Exchange` parameter of its primitive operation `Generate_Params`.

The current implementation of filtering lets each partition generate its own keys for the registration filter. Other approaches are also feasible, e.g., one partition in the application could be used as an application–wide trusted key server.

5 Configuration

Configuring a distributed application is done with a configuration file in GLADE. In this section, we present the essentials of the configuration language, with an emphasis on its support for filtering and for client/server applications. For a detailed description of other aspects of configuration in GLADE, the reader is referred to [3].

5.1 Overview

The distribution of an Ada program is described by a single configuration unit. This configuration unit has a specification part and an optional body part.

```
Configuration_Unit ::=
  configuration Identifier is
    Declarative_Part
  [begin
    Sequence_Of_Statements]
  end Identifier;
```

The configuration language provides two predefined types `Partition` and `Channel`, which describe the objects a configuration operates upon. A channel is a communication link between two partitions. In the declarative part of a configuration unit, the user declares all the partitions as well as the channels between them, and defines which Ada units are to be assigned to which partition. GNATDIST checks the consistency of the configuration, e.g., it makes sure that no RCI unit is assigned to multiple partitions, and it also checks that the identifiers given in the definition of a partition really correspond to existing Ada units.

The user can also declare a list of channels, each consisting of a pair of identifiers of previously declared partitions.

```
Declarative_Part ::= {Declarative_Item}

Declarative_Item ::=
    Partition_Declaration
  | Channel_Declaration
  | Representation_Clause

Partition_Declaration ::=
  Defining_Identifier_List : Partition
    [:= Enumeration_Of_Identifiers];

Channel_Declaration ::=
  Defining_Identifier_List : Channel
    [:= (Defining_Identifier, Defining_Identifier)];

Enumeration_Of_Identifiers ::=
  ([Defining_Identifier {, Defining_Identifier}])

Defining_Identifier_List ::=
  Defining_Identifier {, Defining_Identifier}
```

This syntax specifies a partition as an arbitrarily long list of Ada units and a channel a pair of partitions. The user can declare a list of partitions initialized with a list of Ada units. Once declared (and possibly initialized), each partition can be extended individually with additional Ada units in the statement part of the configuration unit.

The configuration language provides a way to declare the main procedure of the non–distributed application:

```
procedure Main_Unit_Name is in Partition;
```

Representation clauses are widely used in the configuration language of GLADE to change the default behavior. For instance, partitions can be mapped onto physical nodes, and GNATDIST's default behavior is to ask interactively (when the distributed application is run) for the physical node a certain partition should be run on. This behavior can be modified via an attribute clause for the attribute `'Host` giving the machine a partition should be executed on. This attribute can be applied either to the type `Partition` (i.e., to all partitions), or to specific individual partitions.

5.2 Filter Configuration

The configuration language also allows developers to modify the behavior of a channel via representation clauses, e.g., the `'Filter` attribute of a channel can be set to a given filter. (The default behavior is not to filter at all.) In this way, the user can modify the behavior of a given channel, or of any channel when the attribute is applied to the predefined type `Channel`.

```
Representation_Clause ::=
    for Identifier'Host use String_Literal;
  | for Identifier'Host use Subprogram;
  | for Identifier'Filter use String_Literal;
  | for Partition'Host use String_Literal;
  | for Partition'Host use Subprogram;
  | for Partition'Filter use String_Literal;
  | for Channel'Filter use String_Literal;
```

A partition host can be determined statically by giving a string literal. But it can also be determined at run-time. Therefore, an attribute clause for attribute `'Host` can be the result of a subprogram call. As filtering code has to be present for the linker, the attribute clause for attribute `'Filter` cannot be the result of a subprogram call.

It is not necessary to declare all possible channels between partitions. The default filter, which does nothing, is used on any channel that has not been declared or whose `'Filter` attribute has not been set.

A configuration unit might look like the following example (we have commented only the new features; for more information on the configuration language, see [3]):

```
configuration Example is

  Partition_1 : Partition;
  procedure Main is in Partition_1;

  Partition_2, Partition_3 : Partition
    := (Normal);

  function Choose_Best_Node (P : String)
    return String;
  pragma Import
    (Shell, Choose_Best_Node, "./best");
  for Partition'Host use Choose_Best_Node;
  --  Use function "Choose_Best_Node" to determine on which
  --  machine a certain partition should be executed. Here, this
  --  function is supposed to be a shell script "./best".

  Partition_4 : Partition := (RCI_B5);

  Channel_1_2 : Channel
    := (Partition_1, Partition_2);
  --  A logical link between two partitions. The link is
  --  symmetric.

  for Channel_1_2'Filter use "zip";
  --  This representation clause defines the behavior of the
  --  channel. All data exchanged through it will be compressed
  --  using the filter "zip".
```

```
   Channel_1_3 : Channel;
   -- The full specification of this channel will occur later in the
   -- configuration body.

   for Channel'Filter use "pgp";
   -- This representation clause affects all declared channels
   -- unless overridden by channel–specific clauses (as
   -- above for Channel_1_2).

begin
   Partition_2 := (RCI_B2, RCI_B4);
   Partition_3 := (RCI_B3);
   Channel_1_3 := (Partition_1, Partition_3);
end Example;
```

The above configuration unit results in the distributed application shown in fig 4:

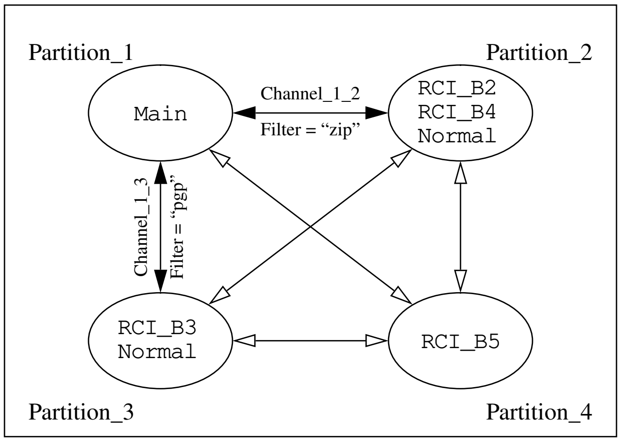


Fig. 4 The resulting distributed application

The channels depicted by the hollow arrows (——▷) have not been declared in the configuration unit and thus will use the default filter, which does not modify data in any way, if data is sent on them. This default behavior can also be changed by setting the `Filter` attribute for type `Partition`:

```
for Partition'Filter use "xor";
```

will use the filter "xor" as the default filter. In this way, the user has complete control of the ways streams are exchanged on communication channels.

Note that `Partition'Filter` forces all communication channels to use a given filter even if these communication channels have not been declared in the configuration unit, whereas `Channel'Filter` forces all *declared* communication channels to use a given filter. In the example, the filter for the declared `Channel_1_3` between `Partition_1` and `Partition_3` is determined by the `Channel'Filter` attribute. It is overridden for the other declared channel by a channel-specific attribute.

These attributes form a hierarchy: The default filter for all channels (`Partition'Filter`) can be overridden with a filter for all declared channels (`Channel'Filter`), which again can be overridden by a channel–specific filter.

Several filters are already available in GLADE. The "zip" filter allows data compression on the fly. This could be useful when network performance is an issue. Encryption filters are also available for secure applications. In the next section, we present a banking example where confidentiality is required.

5.3 Support for Client/Server Applications in GLADE

GLADE offers several features intended to facilitate the development of client/server applications with the Distributed Systems Annex, making use of the standard's provisions for other models than the "single application" one.

5.3.1 Partition Main Subprogram

As described in section 3, when the user configures its centralized application into a distributed application, several partitions (i.e. programs) are built. Each partition may have its own main subprogram [8, 10.2(7)] and the main subprogram of the initial application is mapped onto only one of them (the lead partition). In GLADE, the main procedure of a particular partition can be set using the partition attribute `'Main`:

```
Client : Partition;
procedure Another_Main;
for Client'Main use Another_Main;
```

5.3.2 Partition Launching Procedure

A centralized application is started with one single program invocation. In a distributed application, each partition must be started to execute the distributed program. To maintain the distribution transparency, GLADE build a single program to allow the user to execute the distributed application with a single command invocation, too. This single program is the lead partition; it launches the remaining partitions on their associated physical nodes. This behavior is not reasonable in a client/server context as any number of clients can be running at the same time on any physical nodes. To change this default behavior, GLADE provides the pragma `Starter` which can force the generation of a specific launching procedure. To build a client/server application, no starter should be provided:

```
pragma Starter (None);
```

5.3.3 Program Termination

A centralized application terminates when all its tasks can terminate. GLADE supports the "single distributed application" paradigm by providing a distributed termination detection which allows a partition to terminate only when *all* the tasks of *all* partitions in the distributed application can terminate. Unfortunately, this behavior is not sensible in a client/server application. A client partition may want to terminate once its own tasks can terminate, irrespective of the state of tasks in the server partition, and the server partitions should never terminate. GLADE supports this case through the attribute `'Termination`, which can be used to chose a particular termination algorithm:

```
Client : Partition;
for Client'Termination use Local_Termination;

Server : Partition;
for Server'Termination use Global_Termination;
```

6 An Example: A Secure Client/Server Application

In this section, we present a small and simple banking application demonstrating two things: first, we show how easy filtering is to use, and second, we demonstrate how to program client/server applications in Ada 95.

In this banking example, a client can perform several basic requests on his bank account. This interface allows to check balance, deposit money and withdraw cash. As these requests can be performed remotely, the `Server` package is categorized as a `Remote_Call_Interface` (RCI package).

```
package Server is
   pragma Remote_Call_Interface;

   function Balance
      (Customer : in Customer_Type;
       Password : in Password_Type)
     return Integer;

   procedure Deposit
      (Customer : in Customer_Type;
       Amount   : in Positive);

   procedure Withdraw
      (Customer : in Customer_Type;
       Password : in Password_Type;
       Amount   : in Positive);

   ...

end Server;
```

For security reasons, this package is the front–end of a more general package Bank which allows privileged requests like account creation. Note that accesses to accounts are protected by a password on the application level. However, on the network level this banking system may still be vulnerable if the communication is insecure and passwords are sent in cleartext. In this case, the bank will require the encryption of any transaction. Therefore, an encryption filter should be applied on the communication channel between the bank server and its clients.

Using the features presented in the previous paragraphs, the developer will write the following configuration unit:

```
configuration Simcity is

   pragma Starter (None);
   -- Bank_Client and Bank_Server are launched manually.

   Bank_Server : Partition := (Server);
   -- There is only one RCI package. Note that this RCI package is
   -- similar to the Bank package with only the limited features allowed
   -- to a client.

   Bank_Client : Partition;
   for Bank_Client'Termination use Local_Termination;
   -- A client should not wait for the termination of the whole
   -- distributed application to terminate.

   procedure Manager is in Bank_Server;

   procedure Client;
   for Bank_Client'Main use Client;
   -- The partition Bank_Client uses a given main subprogram.

   Channel_1 : Channel := (Bank_Client, Bank_Server);
   for Channel_1'Filter use "dea";
   -- Defines the filtering (here: encryption) to use between client and
   -- server.

   pragma Registration_Filter ("rsa");
   -- Since DEA is a secret key encryption, we need a public key filter for
   -- exchanging the DEA keys between clients and servers.

end Simcity;
```

Note that the only difference from an insecure version of this banking application is in the configuration file. The application's code itself is not aware of filtering.

Note that normal packages are included in the partitions as needed: if a normal package is withed by one of the library units assigned to a partition, it is included automatically in that partition, too. In the example, this is the case for the normal package Bank, which is supposed to implement the account handling on partition Bank_Server. The pragma Registration_Filter defines the application–wide public key algorithm that will be used to perform key exchanges as discussed in subsection 4.3 above.

The above configuration is depicted in fig. 5 below:

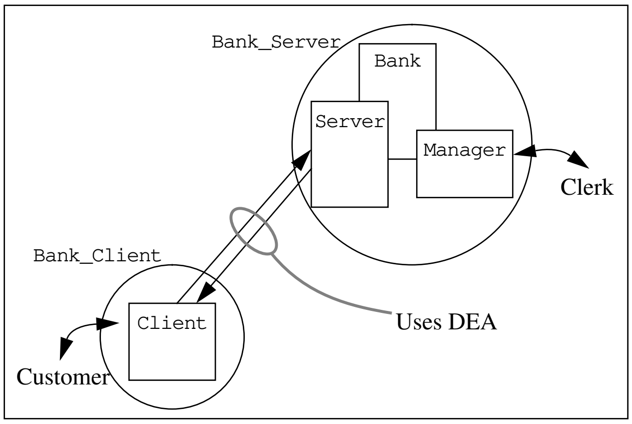


Fig. 5 The Configuration of the Banking Application

6.1 Controlling a Given Client

In the pure client/server paradigm, the client sends requests to its server. In some client/server applications, however, the server needs to have some control over its clients. E.g., a server might want to notify its clients of some events or to identify its clients in order to force some operations on them later on. Of course, a server can identify an individual client by its partition ID, but in this case, he has no way to control its client or to perform any classical remote operation proposed by the Distributed Systems Annex. The mechanism in Ada 95 for controlling particular instances of a partition are distributed objects.

Distributed objects are provided in Ada 95 by the concept of remote accesses to class–wide types. An object and its derived type are instantiated on a client, which then passes a reference to this object to a server. When the server wants to notify the client, it can use the registered remote object reference to invoke a primitive operation (i.e., method) of the object. A call of a primitive operation on a remote object reference will invoke the operation on the partition where the object resides, i.e., where it had been created. This dynamically resolved operation is called a *remote dispatching call*. The remote object reference can be described as a structure that contains obviously the local address of the object and its internal tag, but also its partition ID.

When the same client partition is executed several times simultaneously in a distributed application, each copy of the partition is identified in GLADE by a unique partition ID (different for each copy). When a remote object reference is registered on the server by each of these client partition instantiations, these references are different even if they come from the same code, because of their different partition IDs.

In our banking example, the server wants to notify a given client that his account has been credited a given amount of money by a given customer. To make this possible, each client registers a remote reference to an object of his at the server, which saves them all and later notifies a given client by invoking a remote dispatching operation on the registered reference. Subsequently, we'll show how to implement this in Ada 95.

In a pure package Alarm, we declare an abstract type Terminal_Type and one of its abstract subprogram Notify. This pure package Alarm will be replicated on both client and server side as it is stateless.

```
package Alarm is
   pragma Pure;

   type Terminal_Type is
      abstract tagged limited private;
```

```
   procedure Notify
      (Terminal : access  Terminal_Type;
       Donator  : in      Customer_Type;
       Amount   : in      Integer)
      is abstract;
   -- Note the use of mode access here to allow dispatching calls on a
   -- reference to Terminal_Type.

   ...

end Alarm;
```

In the RCI package `Server`, we declare an access type on any derived type of `Terminal_Type`. We provide a way for the client to register its object reference (parameter `Terminal`) on the server. We also declare a new customer operation `Transfer` to transfer money between two accounts.

```
package Server is
   pragma Remote_Call_Interface;

   type Terminal_Access is
      access all Terminal_Type'Class;

   procedure Register
      (Terminal : in Terminal_Access;
       Customer : in Customer_Type;
       Password : in Password_Type);

   procedure Transfer
      (Donator  : in Customer_Type;
       Password : in Password_Type;
       Amount   : in Positive;
       Customer : in Customer_Type);

   ...

end Server;
```

The server saves each registered client terminal reference in a table `Accounts` and performs the following operations during a `Transfer` operation:

```
   ID_1 := Find (Donator);
   ID_2 := Find (Customer);
   Accounts (ID_1).Withdraw (Amount);
   Accounts (ID_2).Deposit  (Amount);

   -- If Customer is present, notify him through a remote dispatching
   -- operation.
   Term := Accounts (ID_2).Terminal;
   if Term /= null then
      Notify (Term, Donator, Amount);
   end if;
```

The client derives the abstract type `Terminal_Type` and creates an aliased object of this derived type. Since the derivation and the object are in a normal package, each client gets its own version of this package and hence its own object.

```
package Message is

   type Alarm_Terminal is
      new Terminal_Type with null record;

   procedure Notify
      (Terminal : access  Alarm_Terminal;
       Donator  : in      Customer_Type;
       Amount   : in      Integer);
   -- Implements a client's response to a notification from the server.

   My_Terminal : aliased Alarm_Terminal;

end Message;
```

The primitive operation `Notify` implements the client's handling of a notification from the server. In our example, it might be the following:

```
   procedure Notify
      (Terminal : access  Alarm_Terminal;
       Donator  : in      Customer_Type;
       Amount   : in      Integer);
   begin
      Put_Line
         ("NOTIFICATION: Received"  &
          Integer'Image (Amount)    &
          " from "                  &
          String (Donator));
   end Notify;
```

The reference to the client's aliased object is registered at the server to provide it with a handle for use in the `Transfer` operation:

```
   Register (My_Terminal'Access, Customer, Password);
```

Note that the server partition does not have a copy of the package `Message`, only the client partition uses it. Also note that the dispatching in a remote dispatching call is done on the receiver side (in our case, at the client). The server uses the partition ID of the reference the client registered to determine where it should send the remote call request. When that particular client receives the request, it uses a *locally* dispatching call to invoke the right primitive operation. The new application's configuration is shown in figure 6 below:

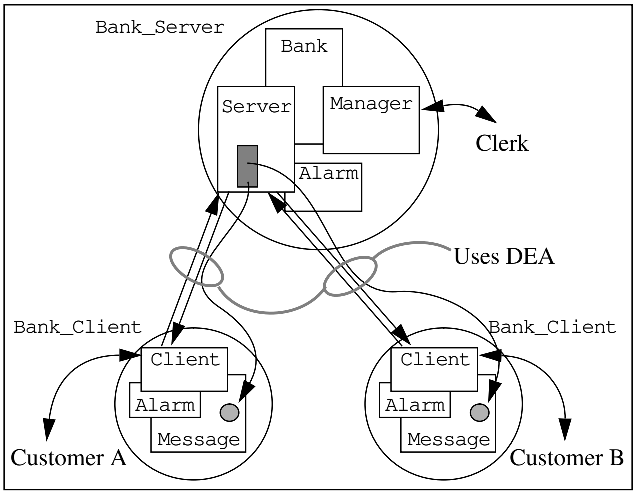


Fig. 6 The Banking Application with Notification

Note that a channel is identified during configuration by the *names* of the two partitions it connects. In this example, however, there are several instances of partition `Bank_Client`. The logical channel between partitions `Bank_Server` and `Bank_Client` comprises in this case several physical links. As they are considered part of the *same* logical channel, they all use the same filter. On the other hand, the filter *parameters* are different for each instance of `Bank_Client` (i.e., for each physical link), just as each instance has its own, different partition ID. In the example, DEA is used on all physical links that are part of the logical channel `Channel_1`; but the secret keys are different for each link.

7 Conclusion & Future Work

Filtering is a transparent solution for some performance and security problems introduced by distribution, where the filtering aspects are separated from functional concerns. The use of Ada's facilities for object–oriented programming renders the implementation of filtering within GARLIC highly modular and easily extensible.

As filtering is transparent to the application, it becomes a pure configuration concern; prior development phases do not have to deal with filtering aspects. It also makes filtering very easy to use, even in highly dynamic client/server configurations.

GLADE is available on all the usual GNAT mirror sites. As of version 1.02, GLADE supports filtering and comes with a filter based on the algorithms used in gzip for transparently compressing and decompressing streams. Additional filter packages implementing the DEA [1] and RSA [6] encryption schemes are available via the Web from URL <http://lglwww.epfl.ch/Ada/filters.html> or via ftp from directory <ftp://lglftp.epfl.ch/pub/Ada/filters>.

Future work on the filter extension to GLADE will deal with enhancing the internal protocols to allow dynamic changes of keys, providing an application–level interface for defining filtering parameters (e.g., the compression level of the gzip–filter), the development of a binding to the IDEA cipher algorithm, and the development of a method to automatically combine filters by simply specifying a combination in the configuration file.

Acknowledgements

We would like to thank Andy Wellings for having read a draft version of this paper. His comments helped improve the presentation substantially.

8 References

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

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

[3] Kermarrec, Y.; Nana, L.; Pautet, L.: “GNATDIST: a configuration language for distributed Ada 95 applications”. In *Proceedings of Tri-Ada ’96*, pp. 63 – 72, Philadelphia PA, USA, Dec. 1996.

[4] Kermarrec, Y.; Pautet, L.; Tardieu, S.: “GARLIC: Generic Ada Reusable Library for Interpartition Communication”. In Charles B. Engle, J. (Ed.), *Proceedings of Tri-Ada ’95*, pp. 263 – 269, Anaheim CA, USA, Nov. 1995.

[5] Kienzle, J.; Wolf, T.; Strohmeier, A.: “Secure Communication in Distributed Ada”. In Strohmeier, A. (Ed.), *Proceedings of Ada–Europe ’96*, pp. 198 – 210, Montreux, Switzerland, June 1996, Lecture Notes in Computer Science **1088**, Springer, 1996.

[6] Rivest, R. L.; Shamir, A.; Adleman, L.: “A method of obtaining digital signatures and public key cryptosystems”. *Communications of the ACM* ***21**(2)*, pp. 120 – 126, Feb. 1978.

[7] Schonberg, E.; Banner, B.: “The GNAT Project: A GNU–Ada 9X Compiler”. In *Proceedings of Tri-Ada ’94*, pp. 48 – 57, Baltimore MD, USA, 1994.

[8] Taft, S. T.; Duff, R. A. (Eds.): *International Standard ISO/IEC 8652:1995(E): Ada Reference Manual*. Lecture Notes in Computer Science **1246**, Springer, 1997.