A Portable Implementation of the Distributed Systems Annex in Java

Yoav Tzruya
Department of Computer Science
Tel-Aviv University
Ramat-Aviv 69978 Israel
tzruya@math.tau.ac.il

Mordechai Ben-Ari
Department of Science Teaching
Weizmann Institute of Science
Rehovot 76100 Israel
ntbenari@wis.weizmann.ac.il

1. ABSTRACT

This paper describes an implementation of the Distributed Systems Annex of Ada95. The use of Java to build the Partition Communication Subsystem provides portability and Internet compatibility of distributed applications while retaining the advantages of Ada at the application level. The implementation is done partially in Ada and partially in Java: stubs are generated by the GNAT compiler and the processing of the streams (marshalling and unmarshalling) is done in Ada, while the processing of the messages required to implement the PCS is done in Java for maximal portability.

1.1 Keywords

Ada95, Java, distributed systems.

2. INTRODUCTION

Annex E of the Ada95 Reference Manual (ARM) [3] specifies a model for writing distributed applications, but does not specify the underlying implementation details. The programming language Java supports platform-independent binary compatibility, which offers an opportunity to reduce significantly the cost of building a portable, distributed application. The purpose of this work is to use Java to implement the Partition Communication Subsystem (PCS) - the runtime environment of the distribution model specified in Ada 95. Thus an application can simultaneously benefit from the well-known advantages of Ada for programming large critical systems and from the suitability of Java for net-based applications.

In addition to a PCS, the ARM specifies that an implementation shall provide a partitioning and configuration tool (PCT) for post-compilation configuration of the application when you decide where each unit will reside. *Compilation units* such as packages are *placed* in *partitions* that are assigned to physical *nodes.* This work has not implemented a PCT, as we were more interested in establishing the feasibility of a portable PCS.

Both the Java part of the PCS and the application-specific process are incorporated into one operating system process. Another possibility that we considered was to compile the Ada application into J-code, but we preferred to stay within native Ada compilation where compilers are both efficient and comprehensive in their implementation of the language. (In an environment using the forthcoming GNAT Ada95 to J-code compiler, the design of our system should be re-examined.) Advantages of using Java for low-level programming include the availability of multithreading which is needed in the PCS in order to control the flow of information and the built-in standard library of communications over TCP/IP based networks.

Figure 1 shows the structure of a distributed application using our PCS. The box showing the Ada95 application includes the compiler-generated stubs; in our implementation the stubs were generated by the GNAT compiler. We developed the PCS "from scratch": some code in Ada95 is used in the application-to-agent interface; the main part of the system is the management of the partitions and the inter-partition communications, which was written in Java.

The Ada95 application communicates only with the Java PCS agent in its local process, which in turn communicates with a name server partition (see below), which supplies data based upon information received from other partitions. The name server gives each PCS agent sufficient data to establish a connection with other partitions and to issue remote procedure calls (RPC) and asynchronous procedure calls (APC).

3. RELATED WORK

Kermarrec, Pautet and Schonberg [15] have suggested a scheme for generating stubs required by the Distributed Systems Annex. These include static (hard-coded) calls, remote accessing and modification of objects, and passing and using an access to a remote procedure. Fat pointers (structures that include an access to an object or subprogram together with information regarding its distributed location) and streams are the fundamental building blocks of their approach. Their techniques were used for stub generation in the GNAT compiler, but their article does not give implementation details of the full requirements of the Annex.

The GNAT Library for Ada Distributed Execution (GLADE) was developed by Kermarrec, Nana, Pautet and Tardieu [10]. GLADE is an implementation of the Annex for the GNAT compiler. It offers an environment for developing distributed applications, including a PCS and a simple configuration language. The implementation is written in Ada95 and utilizes the stubs generated by the GNAT compiler. It provides its own implementation of the predefined packages required by the Annex. GLADE requires some platform-specific code for low-level communication and is available only on Unix machines (non-portable features of the OS are used by both the PCS and the PCT).

Another interesting tool is the Advanced Distributed Engineering and Programming Toolset (ADEPT) ([14], [1]), developed at Texas A&M University. ADEPT is also designed to work with the GNAT compiler. Its goal is to provide open, easy-to-use implementation of Annex E. ADEPT provides its own PCS and a GUI for assigning units to the different partitions. The specified configuration is then used to build the correct stubs and images for different partitions of the application. ADEPT also provides interfacing of the Ada95 distributed systems Annex with Java's Remote Method Interface (RMI) through an Ada95 to J-Code compiler, which generates RMI stubs from the Ada95 packages. This gives the ability to call an Ada95 distributed subprogram as a Java remote method, making use of remote access to class wide types and their primitive operations. The ADEPT tool makes use of the Java RMI registry to advertise Ada95 servers to Java clients.

Our implementation provides the same features as the ADEPT PCS without having to compile Ada code into J-code. Furthermore, it improves on the ADEPT PCS because it allows dynamic change of the partition topology and the location of each service. We did not use RMI as in ADEPT, because RMI is a high-level, J-code dependant

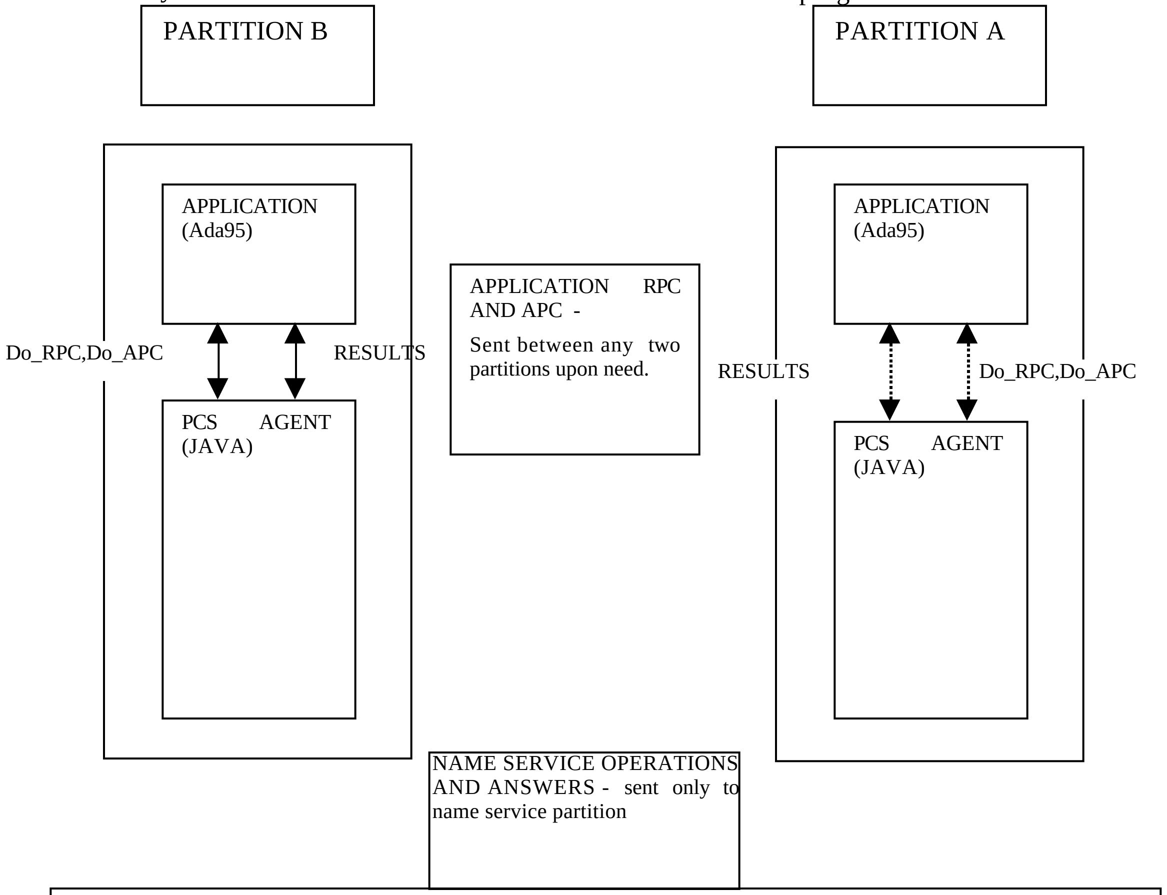
PARTITION B
PARTITION A
APPLICATION (Ada95)
APPLICATION (Ada95)
Do_RPC,Do_APC
RESULTS
PCS AGENT (JAVA)
APPLICATION RPC AND APC -
Sent between any two partitions upon need.
RESULTS
Do_RPC,Do_APC
PCS AGENT (JAVA)
NAME SERVICE OPERATIONS AND ANSWERS - sent only to name service partition

Figure 1 - General structure of two partitions using the Java PCS

facility that is not necessary in an implementation of the Ada Distributed Annex whose intended applications are compiled to native code. In addition, the RMI registry has some overhead and is superfluous given our direct implementation of a partition and service database.

Combining the easy-to-use graphical user interface and configuration language provided by ADEPT and the PCS proposed here could be used to enhance both the performance and the portability of distributed applications developed in Ada95.

The predefined library of the Java language (which is part of the standard language specification) and its binary compatibility features allows for more portable implementations of the annex then those who rely, for example, on OS dependant communication subsystems.

4. ARCHITECTURAL DESIGN

The Distributed Systems Annex requires the implementation of `System.RPC` and `System.Partition_Interface` packages. Here are the specifications of the subprograms for RPC and APC:

```
procedure Do_RPC -- Synchronous call
    (Partition: in Partition_ID;
     Params   : access Params_Stream_Type;
     Result   : access Params_Stream_Type);

procedure Do_APC -- Asynchronous call
    (Partition: in Partition_ID;
     Params   : access Params_Stream_Type);
```

It can be seen that calling an RPC or an APC involves knowing which partition we need to call and passing it parameters in the form of a stream. When issuing an RPC/APC one does not specify a reference to a specific target subprogram but rather the partition id of the target partition where the subprogram resides. Therefore the location of each subprogram must accessible through the `System.Partition_Interface` package.

Thus, there are three essential items of data that must be implemented in the PCS:

- Partition ID - Each the partitions participating in the distributed application has a unique identifier. It would have been possible to implement remote calls without knowing the partition id of a service or object, but instead to identify it by its name directly (not unlike the Java RMI registry mechanism). The partition ID must be predefined or allocated at run time.
- Partition information - Information about a partition's physical location and accessibility.
- Unit information - For every compilation unit we must have information on which partition it resides.

Note that the Annex itself treats a unit's location as static. The location of a unit does not change during the execution of the application. Furthermore, the ability of partitions to join and detach themselves dynamically from the application is not required. The model presented here allows this extension by maintaining a runtime dynamic repository for unit locations and partition information. This repository allows adding/removing RPC units, and adding and removing partitions dynamically.

We have chosen to build our new PCS in a way that most of the knowledge of the subsystem and the information regarding partitions and units would be stored on the Java agent level, to allow maximum independence of Ada95. A further design principle is that data required by the PCS reside on the lowest possible level and are accessible only to internal modules that need them. This is in keeping with the principle of encapsulation.

We must also decide on an appropriate architecture for the communications network and the flow of messages between the partitions. We will assume that any partition may offer and advertise services and may call a service on any other partition.

There is a requirement for some kind of "boot server" partition for two reasons. Firstly, when a new partition joins, it should advertise its location and the services it offers, and we decided that a central information repository is better than constant use of broadcast messages. Secondly, the server generates new partition IDs.

The design of the system is therefore based on a centralized boot server partition whose location is known in advance and which holds a repository for partition and unit information, in addition to generating partition IDs. It services queries on the locations of partitions and on the units included within each partition. A local cache can, and is, maintained in each partition to improve performance. No broadcast messages about configuration changes are sent. All information is sent upon need and the network is not flooded with possibly unnecessary messages. We acknowledge that this design introduces a single point of failure for configuration information (though not for distributed communication). Additional algorithms could be added to maintain a back-up boot server.

This decision influences the network architecture. The question that arises is what should the topology of the network be? A simple tree or star topology would be simple but not very efficient. Instead we assume a star graph: each node is connected to the boot server, but there are additional edges directly between partitions that need to communicate with each other. This topology creates a network that has a distance of one edge between any two partitions that needs to communicate. For simplicity we assume that all edges are reliable. This assumption is based on the use of TCP/IP protocol and the use of sockets, which are inherently reliable.

Another design decision concerns the degree of parallelism in handling messages. There are two types of messages: configuration messages and application messages arising from RPCs. Configuration messages are handled in parallel in the low-level Java code of the PCS. Applications messages are removed from the Java-Ada interface one by one, but handling the message is done in parallel via Ada tasks. This decision was taken in regards for the "correct" way a program should run (RPCs should be handled as though issued in no particular order, since they may have been issued from different tasks/partitions) while keeping the interface easy to handle.

5. DETAILED DESIGN

We now give an overview of the detailed implementation of the PCS; more information is available in the system documentation.

5.1 Initializing and terminating a partition

To initialize a partition, the following steps must be performed. (a) Inform the boot server that a new partition has joined. (b) Find out the partition ID of the new partition (either through a predefined partition ID or by querying the partition ID generator). (c) Pass the partition location to the boot server. (d) Let the boot server know what services this partition offers. When a partition comes up, it needs to know the following information: (a) the boot server physical location (IP address and port) and (b) its own partition id. (The second step is optional because the partition can get it from the id generator as well).

Initialization is performed by handshaking between the new partition and the boot server using two sets of configuration messages. The first establishes the addressing (partition ID and partition location) and the second advertises the services supplied by the partition and stores them in the central repository. This set of messages is also used later by other partitions to query the information from this repository. The queried information is stored in each partition in local, subset copies of the central repository.

To shut down a partition, a message must be sent to the boot server invalidating the information about the partition. Another reason for invalidation can be a failure to communicate with the partition. Messages regarding invalidation are flooded throughout the net. Invalidation of a partition/unit involves removing the relevant information from the local/central repositories and stopping all calls to that partition/unit. Several potential race conditions exist because other partitions might attempt communications with the terminated partition before they have received information that it is invalid. While we developed algorithms to solve these problems, they have not yet been implemented.

5.2 RPC and APC

It is important to emphasize that all calls to RPCs are done by the automatically generated stubs for the user-written code. Aside from writing the unit pragmas, the distributed application is completely transparent to the Ada applications programmer.

APC is an asynchronous call to a routine and thus does not return a value, while RPC is a synchronous call and may return a value. Both the parameters passed to the receiving partition and the result are passed using Ada streams. The specification of parameter passing using streams requires marshalling and unmarshalling of the parameters in the Ada part of the PCS.

The remote call procedures receive a specific partition id and not a specific unit or subprogram, so information on the location of each unit must be available in the Ada part of the *calling* partition. Stubs on the client side must therefore use messages to obtain the partition ID of the unit's partition. They perform the marshalling of the receiver identifier so that the server partition will be able to decide which specific unit is called. Note that the specific service that is called is not mentioned anywhere; it is up to the server and client to agree between them on the protocol which specifies how the identifier of the specific service is encoded (done in the generated stubs).

Once parameters have been marshalled onto a stream, the `Do_RPC` routine has to pass the call to the specified partition, wait for the result and return it to the caller. Note that the `Do_APC` routine only performs the first stage. Unless cached, the location of the called partition can be queried from the boot server partition – the only partition that holds the full topology. The boot server returns a message to the calling partition based on the information it has in its partition information repository; this is used to form a new link between the client and the server partition. An initialization stage between the two partitions is performed for verification.

Now that the caller (client) partition has formed a link to the called (server) partition, it can pass the information it needs in order to perform the RPC/APC. The information is passed using a message that consists solely of the parameters passed to the RPC. Upon receiving these messages the server partition knows only that it has been called by some partition. It now performs unmarshalling of (a) the receiver identifier of the specific unit called, (b) the identifier for the specific service that was called (performed by the unit) and (c) the rest of the parameters (performed by the service). This unmarshalling is done by the generated stubs, not by the applications code. Now the service is performed; upon termination, if the call is an RPC results must be returned. This is done in a similar manner using marshalling/unmarshalling and passing a message containing the result stream. Since the client side has to wait for the result and there might be more than one pending call on a certain partition, there is a need to identify each answer that arrives back in order to match it with the call that initiated the RPC.

6. IMPLEMENTATION

The Java part of the PCS contains a main thread (initialized from the Ada part) that is responsible for handling messages arriving from the Ada part. For each message received from the Ada part, a new thread is created to allow parallel processing of messages. There is also one thread for each remote partition that is linked to the local partition; the thread processes messages to and from the remote partition. Another thread is responsible for listening on a certain port for requests to form new links with other partitions. For each message arriving from another partition, there is a thread for handling the message in a parallel manner. Communication between Java parts of two partitions is done using a TCP/IP socket that is established during the handshake between them. To sum this up, each Java part of a partition consists of threads that listen to communication channels (a memory channel in the same process for Ada messages or a TCP/IP socket). When a message arrives, a new thread is created to handle it. These threads work on a common data structure and are synchronized using the Java constructs wake and notify.

Communication between the Ada and Java parts is done using data structures holding the data on the Java side of the PCS, such as queues and streams of messages (these constructs are portable and use only the standard language features). The Ada part of the PCS polls these queues to see if there is any message waiting to be processed. Upon receiving a message, it is translated and handled as explained above. The Java part of the PCS waits on these data structures to be notified when the Ada part has requested a service. Upon arrival of a message from Ada, the Java part handles it. Ada actually uses Java methods to access these data structures. The Ada part calls these Java routines using the Java Native Interface (JNI) capability that enables a non-Java program to call a Java method which will be executed on the JVM. This requires breaking messages into primitive data types in order to pass them.

The Ada part of the PCS handles messages in a parallel manner using a message-specific task that is created for each message that arrives from the Java part. This task is responsible for handling the message and, if needed, sending back an answer message. There is always a task that is waiting on the communication channel with the Java part for a new message. As soon as this task reads a message, it goes on to handle this message concurrently with the rest of the application. Upon termination, a new task is created to listen and read a message from the communication channel. Another task is responsible for setting up communications between Java and Ada95: setting up handles to JNI exported Java routines, creating the JVM to execute the Java part and calling the Java main routine.

Communications between the Ada part and Java part of the PCS are asynchronous, using queues of messages as explained above. We therefore need to provide the Ada part with a mechanism that will allow waiting for an answer. This is achieved by giving each message a unique identifier that can be waited upon using a mechanism implemented with a protected object; when an answer arrives it awakens the Ada task that issued the call.

A package that declares remote services must include the `Remote_Call_Interface` pragma (or another Annex E related pragma). During elaboration, the server-side stub calls a service to register the remote services it offers in the unit information repository on the boot server. The information regarding which services each of the partitions advertises is stored on the Java part of each partition in an efficient hash table. The information regarding a unit consists of its name, its version, which partition serves it (ID) and what is the address of the receiver procedure (identifier) for this unit. The boot server partition holds a full copy of this repository and each partition holds a subset that contains information regarding units it has already queried about. Another repository holds information regarding the location of each partition (physical location - IP address and listening port).

Each subprogram is identified by a unique code allocated for it by the stub generator. For each unit there is only one procedure (the receiver procedure) which serves as the sole entry point for all remote calls intended to be handled by this package. This is the unit's receiver kept as part of the unit information. The caller of a remote subprogram must have an access to the receiver procedure of the remote subprogram in order to perform an RPC or APC. This information is maintained at runtime by the PCS. The caller also needs the unique identifier to the specific subprogram called within the remote unit, which is maintained at compile time by the stub generator. With this information and the fact that the identifier of the partition that serves a certain unit is available to the Ada part of the partition, the stub can call the `System.RPC.Do_RPC` subprogram with the correct parameters. The PCS later takes the partition ID and–using configuration messages exchanged between the Java part of the calling partition and the boot server partition–forms a new socket with the called partition. This socket will serve from now on as the only means of communications between these two partitions.

During the building of the prototype a few points were noticed. Java threads and Ada95 (GNAT) can co-exist and interact with each other. However, not all platforms were tested for that compatibility and this issue can raise some problems depending on the different paradigms used to implement those mechanisms on a specific platform.

In regards to performance issues, this issue was not thoroughly examined at this prototype stage. Degradation of performance can arise from breaking down RPC calls parameters into primitive types. The usage of JNI is not expected to cause degradation of performance.

7. FUTURE WORK AND CONCLUSION

This work has demonstrated a platform-independent portable PCS that implements the Distributed Systems Annex of Ada95 using Java. Here are some suggestions for future work on this topic:

- **Adapting to other programming languages -** The method shown here for implementing remote procedures and remote data objects using the Java language, in particular the set of inter-partition messages that we defined, can be generalized for other languages. The use of the Java language makes it easier to use by other languages, more portable and standardized.
- **Persistent network structure and network recovery -** The network of partitions can be implemented as a persistent network and the PCS can be expanded to include failure recovery capabilities. The network structure of a static application can be read from a common source, activated and run without the need to form links on runtime upon need.

- **Educational value -** The runtime of the PCS can be expanded to include educational tracing messages for debugging and performance evaluation, and a graphical interface to demonstrate to students how a distributed system works.

8. REFERENCES

[1] Texas A&M University – various documents for the ADEPT project. http://www.cs.tamu.edu/research /ADEPT

[2] AdaMagic/AppletMagic compiler system by Intermetrics. http://www.appletmagic.com.

[3] *Intermetrics Inc.: "Ada95 Reference Manual: Language and Standard Libraries"*. 1995.

[4] Barnes J. *"Programming in Ada95"*. Addison Wesley. 1996.

[5] Ben Ari M. *"Principles of Concurrent and Distributed Programming"*. Prentice Hall International 1990.

[6] Booch. G. E. "Object-Oriented Development". IEEE Transactions on Software Engineering SE-12 (2), February 1986.

[7] Chandy K. M., Chelian A., Dimitrov B., Le H., Mandelson J., Richardson M., Rifkin A., Sivilotti P. A. G., Tanaka W. & Weisman L. *"A World-Wide Distributed System Using Java and the Internet"*. 1996. http://www.infospheres.caltech.edu/papers/chandy_etal/ hpdc.html)

[8] Gargaro A., Smith G., Theriault R. J., Volz R. A. & Waldrop R. *"Future Directions in Ada – Distributed Execution and Heterogeneous Language Interoperability Toolsets"* Ada Letters, Sep/Oct 1997 Volume XVII number 5. Pages 51-56.

[9] Gnat compiler documentation and sources. http://www.gnat.com

[10] *GLADE homepage.* http://www.act-europe.fr/ glade.html

[11] Gosling,J. and McGilton, H.*The Java™ Language Environment: A White Paper.* Sun Microsystems.

[12] *The Java™ Language Specification.* Addison Wesley, 1996

[13] (ISO/IEC 8652:1995): *"Information Technology -- Programming Languages – Ada"*. 1995.

[14] Jansen B. J. Maj., US Military Academy *"Adept – Building Distributed Systems in Ada95"*. 1997.

[15] Kermarrec Y., Pautet L. & Schonberg E. *"Design Document Implementation of Distributed Systems Annex of Ada9X in GNAT"* February 1995.

[16] Lippman S. B. *"C++ Primer 2nd Ed."* Addison Wesley 1991.

[17] *Java API Documentation.* Sun Microsystems.

[18] *Intermetrics Inc.: "Ada95 Rationale"*. 1995.

[19] Taft S. T. *Programming the Internet in Ada95.* http://www.appletmagic.com/ajpaper/index.html

9. APPENDIX A - EXAMPLE OF MESSAGES EXCHANGE

We give a here complete message transcript of a simple example application that uses RPCs and remote objects. In the figure below we can see the setup stage and a single iteration through the loop of the main routine of the example. For each message the Java side receives and handles, a new thread is created in order to process it in parallel. The same applies for the Ada side: with each message received, a new task is created in order to handle it. Additional calls used for obtaining unit information from Ada to Java are not drawn here for clarity, nor have we indicated the synchronization events.

9.1 Example Sources

```
package root is
        pragma pure;
        type root_type is abstract tagged limited private;
        procedure operate(t : access root_type) is
        abstract;
private
        type root_type is abstract tagged limited null
        record;
end root;


package root.typea is
        type type_a is new root_type with
        record
                content : string (1 .. 10);
        end record;
        procedure operate (t : access type_a);
end root.typea;


package root.typeb is
        type type_b is new root_type with
        record
                content : string (1 .. 10);
        end record;
        procedure operate (t : access type_b);
end root.typeb;


with root;
package provider is
        pragma remote_call_interface;
        type root_type_class_ptr is access all
        root.root_type'class;
        function get return root_type_class_ptr;
end provider;


with provider;
with root;
procedure main is
begin
```

```
      for I in 1 .. 5
      loop
              root.operate(provider.get);
      end loop;
```

9.2 Message Transcript

Follows here a figure illustrating the exchanged messages throughout a sample run of the above program.

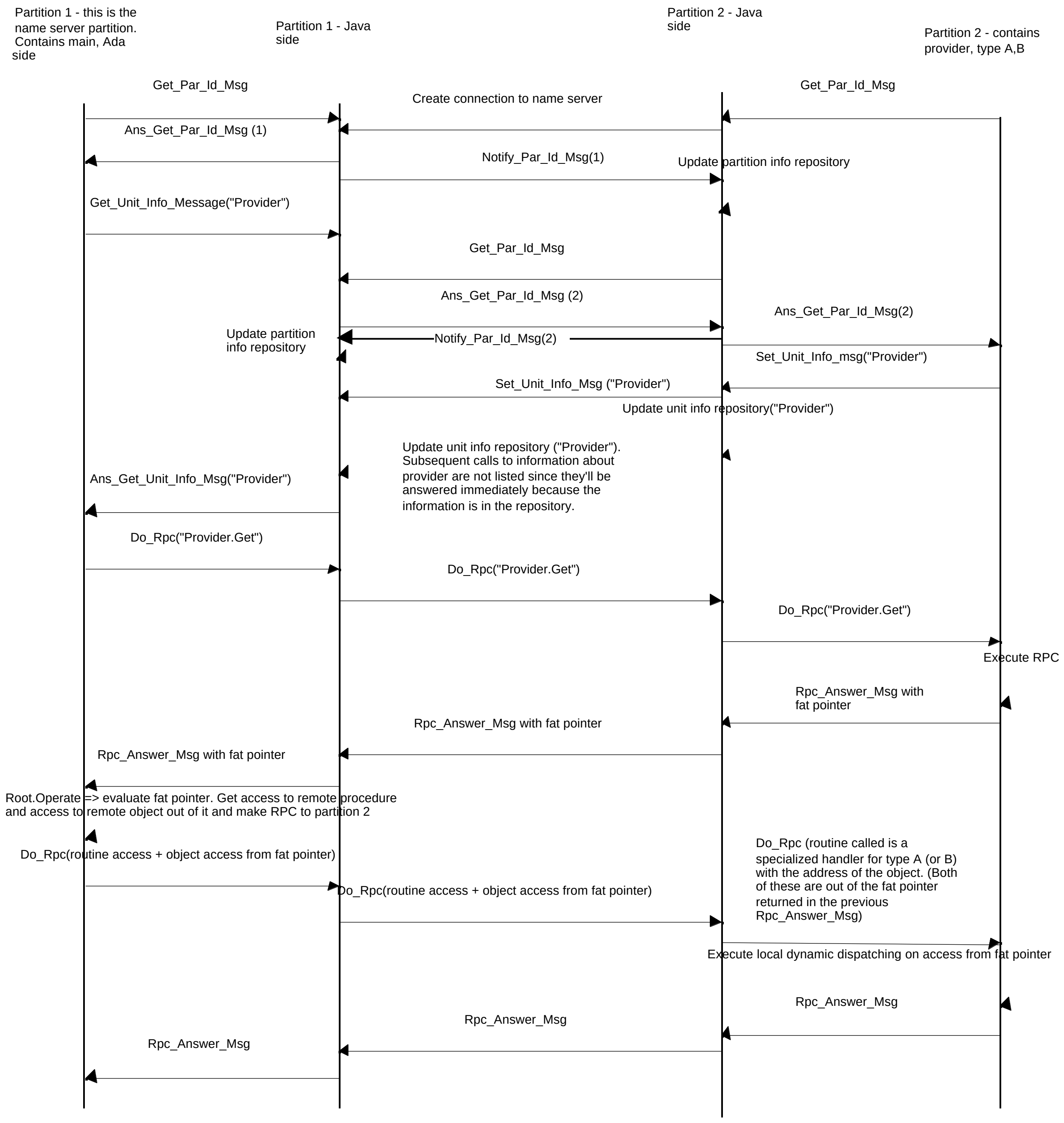
Partition 1 - this is the name server partition. Contains main, Ada side
Partition 1 - Java side
Partition 2 - Java side
Partition 2 - contains provider, type A,B
Get_Par_Id_Msg
Create connection to name server
Get_Par_Id_Msg
Ans_Get_Par_Id_Msg (1)
Notify_Par_Id_Msg(1)
Update partition info repository
Get_Unit_Info_Message("Provider")
Get_Par_Id_Msg
Ans_Get_Par_Id_Msg (2)
Ans_Get_Par_Id_Msg(2)
Update partition info repository
Notify_Par_Id_Msg(2)
Set_Unit_Info_msg("Provider")
Set_Unit_Info_Msg ("Provider")
Update unit info repository("Provider")
Update unit info repository ("Provider"). Subsequent calls to information about provider are not listed since they'll be answered immediately because the information is in the repository.
Ans_Get_Unit_Info_Msg("Provider")
Do_Rpc("Provider.Get")
Do_Rpc("Provider.Get")
Do_Rpc("Provider.Get")
Execute RPC
Rpc_Answer_Msg with fat pointer
Rpc_Answer_Msg with fat pointer
Rpc_Answer_Msg with fat pointer
Root.Operate => evaluate fat pointer. Get access to remote procedure and access to remote object out of it and make RPC to partition 2
Do_Rpc(routine access + object access from fat pointer)
Do_Rpc(routine access + object access from fat pointer)
Do_Rpc (routine called is a specialized handler for type A (or B) with the address of the object. (Both of these are out of the fat pointer returned in the previous Rpc_Answer_Msg)
Execute local dynamic dispatching on access from fat pointer
Rpc_Answer_Msg
Rpc_Answer_Msg
Rpc_Answer_Msg

Figure 2 - messages transcript

```
end main;
```