Distributed-Application Development Tools for DCE/OSF

Uri Shani and Israel Gold

Haifa Research Laboratory
IBM Israel Science and Technology
Matam — Advanced Technology Center
Haifa 31905, Israel
E-mail: shani@vnet.ibm.com

Abstract

DCE/OSF offers a comprehensive RPC-based solution for the development of open distributed applications across network of heterogeneous multiple-vendor machines. However, even simple DCE applications may have a rather complicated structure, requiring good understanding of the elaborate DCE technology. In this paper, we examine the DCE application development path, and introduce two new tools, ***idlgen*** *and* ***gluegen*** *(collectively called* **MakeDCE***), which help to separate application logic from DCE aspects.*

The development of simple DCE applications using ***idlgen*** *and* ***gluegen*** *remains a rather simple task, requiring very little knowledge of DCE, and of the DCE toolkit. These tools are particularly useful for splitting existing monolithic programs into clients and servers, with almost no change to the original application code. Complex DCE applications are supported as well, even though the splitting approach may not be fully applicable.*

Keywords: Distributed computing, application development, CASE, DCE, wrapper, source matching, source generation, client, server, IDL

Introduction

Distributed Programming Environment (DCE) by the Open Software Foundation (OSF) offers a comprehensive solution for distributed client/server applications over network of heterogeneous multi-vendor machines [9]. The DCE services and development toolkit come in several levels based at the bottom on *threads* and *Remote Procedure Call* (RPC) [1]. Threads allow concurrent threads of control to execute within the same application process. DCE RPC is based on Apollo's NCS [6, 7], providing a synchronous communication mechanism which appears syntactically the same at the application level as a local procedure call.

Above these basic layers come additional services: *Directory services* maintain a global symbolic identification system of servers on the network. *Security services* provide authentication and validation services on an open network environment. *Time services* maintain global time synchronization, and a *Distributed file system.* Each component of DCE comes from a well established and proven technology, all integrated into a single coherent environment (see references in [9]).

The DCE run-time complexity can be appreciated by the size of its library of support functions — a few hundreds of functions to support all the above services. Close to a hundred to support the fundamental RPC level. The full comprehension of DCE is a significant task, realizing that DCE applications may have a rather complicated structure, even for simple cases.

To develop a DCE application, you first have to design its distributed architecture, and write DCE code to perform RPC operations. On the server side this amounts to *registering* and *unregistering* its remote services. On the client side this amounts to locating and *binding* to servers and calling remote services. In order to perform RPC to remote services, they must be defined as operations in an *Interface Definition Language* (IDL) file. The recommended DCE development path is first to write an IDL file. The file is compiled by the DCE stub-compiler ***idl***, generating stubs and an *.h* header file. The header file needs to be *#include*-d in the client and server sources, and the stubs need to be compiled and linked with them. Therefore, the application becomes very DCE dependant — in structure as well as in source code.

We look at the situation where the application can have a monolithic version — where no elements of distributed programming exist. The program is then

0-8186-5835-5/94 $03.00 © 1994 IEEE

split, or *partitioned* into several partitions playing the role of clients, servers, or both. If properly done, the application can coexist in its monolithic version as well as its split version. DCE cannot support this approach to its fullest since there are aspects of program partitioning [10] which DCE cannot comply with. For instance, how to split a global variable. However, a significant simplification of DCE application development can be achieved with program splitting.

The two main aspects of splitting existing applications is that the IDL file can be *extracted* from the original C source, and that the original code from the monolithic application will remain unchanged as much as possible. Therefore, the interface to the DCE run-time has to be separated from the application.

We report here of tools to achieve these goals. ***idlgen*** is a tool that generates IDL files from C files where certain remote services are implemented. Since the IDL file is derived from C sources, there is no need to import (include) the *.h* file which results from stub-compiling the IDL file. For this reason, ***idlgen*** also maintains the compatibility between IDL and C sources. ***gluegen*** as its name alludes to, is a tool which generates *glue-code* to bind together the client and server. ***gluegen*** generates code which performs all network and DCE run-time management so that when RPC occurs, it is properly *bound* and eventually succeeds. Glue-code is generated based on a small high-level specification language we term *Application Profile* (APF), which frees the programmer from getting into the gory details of DCE run-time. We only give a limited account of ***gluegen*** in this paper. For a detailed report see [4].

idlgen is a compiler which reads IDL and C sources and produces a new, modified IDL file. ***idlgen*** extracts the interface-relevant information from its input files, checks that all relevant declarations are present and do not introduce any conflicts. Wherever needed, ***idlgen*** generates IDL attributes and declarations, and when needed, it converts IDL-*incompatible* constructs in C into equivalent IDL-*compatible* constructs. ***idlgen*** incorporates into IDL files, changes made to their corresponding C sources; and vice-versa, it analyzes implications of changes made to the IDL file by the user.

Using ***idlgen*** to develop an IDL file from a collection of C sources can be as easy as

1. Decide which functions will become IDL operations
2. Invoke ***idlgen*** to generate an IDL file
3. Verify that the generated file answers the requirements

The following chapters introduce the ***idlgen*** and ***gluegen*** tools, using examples rather than formal descriptions. We conclude the paper with a short comparison to other approaches in this field.

idlgen — an IDL extractor

The ***idlgen*** tool analyzes C and IDL files thus enables to update old IDL files to accommodate changes in existing applications, as well as to generate IDL files for new applications. It is a language translator that takes source files (C and IDL) as input, and outputs a new IDL source file — ready for the ***idl*** compiler.

idlgen is invoked from the command line. A number of command line options and switches are available for tailoring ***idlgen*** operations. A most simple C program such as "f(a,b){return a+b;}" can make an immediate demonstration of generating a legal IDL file by ***idlgen*** as follows,

```
User:
echo 'f(a,b){return a+b;}' | idlgen -id +v

System:
interface noname {

/*@;**************************************************
   * This file built with the MakeDCE facility ver 1.0 *
   *     - A DCE Application-Development Enabling Tool *
   *                                                   *
   * Initially generated on Sun Jan 31 08:45:12 1993   *
   *         Last update on Sun Jan 31 08:45:12 1993   *
   *                                                   *
   * IBM Corporation 1992                              *
   ****************************************************/
/*@[export] f ;  file -stdin */

long int f (
   [  in] long int a,
   [  in] long int b
);

}
```

The ***-id*** switches tell ***idlgen*** to generate a full IDL file using built-in default attributes; the switch $+v$ tells ***idlgen*** to turn off verbose mode. The resulting file is written to the standard output since no output file name appears in the command line. No input file name is given neither, so standard-input is taken — piped in from the *echo* command.

The resulting IDL file in the example above, reveals how ***idlgen*** maintains its own information in IDL files.

We term this method *Meta-Comments* since we hide the information inside C comments.

Meta-comments have the following general format:

```
/*@ [ meta-statement ] ; [ comment ] */
```

When parsing IDL files, ***idlgen*** checks the *meta-statement* for correct syntax, while ignoring the *comment* part.

In the above example, meta-comments are used for two purposes. One is to place a disclaimer to the fact that **MakeDCE** generated this file. The second is to state that the function *f()* is exported from the file *"-stdin"* (although "stdin" is not a very useful filename). When ***idlgen*** is invoked on multiple input files, one of which is an IDL file, and the others are C files, it will only process *exported* functions and other declarations *used* by them.

In the example above, there is only one C input file, and all its exportable functions are considered for export by default. Alternatively, we can proceed in steps. Our initial step will be to *select* which of the exportable symbols of C sources are to be considered for the IDL file. To help accomplish this task, ***idlgen*** will extract from the C sources all exportable symbols and list them within meta-comments in an *initial IDL file*.

Our next step will be to go over this list, using a text editor, and alter the status of functions we do not want to export. We will do that by changing the attribute [***export***] to [***noexport***]. Than, invoking ***idlgen*** again taking this time the initial IDL file as input, together with the original C sources, producing a new modified IDL file. The resulting file can be a separate new file, or override the initial IDL file, and will be a legal and operational IDL file.

Comparing and modifying C and IDL files

The general invocation command line for ***idlgen*** is,

```
idlgen <input file>... <switches>...
```

Whenever the input files consist of an IDL file and C files, the declarations in the IDL file, and those relevant declarations in the C files are compared and checked for consistency.

An initial IDL file contains only a list of names of functions to be exported or not-exported. In other cases, existing IDL files may be edited so that status of functions is inverted. In both cases new declarations may be extracted from C sources and added to the IDL file. In the later case, some declarations in an existing IDL file may become redundant, and will not be reproduced in the resulting output file.

The new declarations are converted to meet the syntax and semantics of IDL which are not fully compatible with those of the C language. In particular, IDL attributes are added based on some analysis of the C sources. For instance, the [***input***] and [***output***] attributes as in the simple example above (for the parameters of *f()*).

In subsequent invocations of ***idlgen***, the proper aspects of the corresponding declarations in the IDL file and the C files will be compared and verified.

It is important to understand that although an IDL file can export functions from many C sources, ***idlgen*** can process portions of the IDL file, depending on which of the C files are input to ***idlgen***. It could also process the IDL file and all of the relevant C files in one step.

We will discuss now some of the conversions done by ***idlgen***.

Matching unions.

A union in an input C file will be converted to the appropriate variant in IDL. For instance, the C union:

```
union u { int i; float f; }
```

Is converted to the following IDL typedef by ***idlgen***,

```
/*Manufactured typedef for an aggregate by MakeDCE */
typedef union u switch (long MK_TEMP) {
     case 0: long int i;
     case 1: float f;
} u_MKGEN;
```

As the program evolves, changes may occur in the declaration of *union u*. ***idlgen*** will match the unions in C and IDL even when their fields are reordered. In our example, if we add the field *char c*, and shuffle the fields,

```
union u { float f; char c; int i; }
```

idlgen will produce a modified IDL declaration,

```
/*Manufactured typedef for an aggregate by MakeDCE */
typedef union u switch (long MK_TEMP) {
     case 1: float f;
     case 2: char c;
     case 0: long int i;
} u_MKGEN;
```

Enum tags.

The IDL language does not accept **enum** with tags. All enum declarations must be enclosed within a **typedef**, and have no tags. To maintain affinity between the IDL file and its C source, the tag is entered into the IDL declaration inside a meta-comment (this is a third situation where meta-

comments are used by ***idlgen***). The ***idl*** compiler ignores this as a comment, while ***idlgen*** reads the tag name and is able to associate it with the corresponding C source construct for type matching. Carrying the tag name into the IDL file will also be a helpful piece of information for the developer.

For instance, the C enum

```
enum e { a,b,c };
```

Is converted to an IDL enum as follows:

```
typedef [ transmit_as(long) ] enum /*@ e ;*/ {
   a,
   b,
   c
} e_MKGEN;
```

Long identifiers.

The IDL language limits identifiers' length to below the length permitted in C. ***idlgen*** has no limitation on identifier-length, but will check identifiers in input C files against the IDL limit. Moreover, ***idlgen*** generates new identifiers in output IDL files which can be very long since they are generated as a combination of C identifiers.

To prevent IDL syntax errors, ***idlgen*** generates **#define** statements in the output IDL file where long names are defined as short names.

```
#ifndef IDLGEN
#   define this_is_a_very_long_name_indeed MKSHORT_0
#endif
```

When the file is processed by ***idl*** compiler, only the short names are "seen." When processed by ***idlgen***, the long names are used. This is needed to properly identify and match C and IDL sources.

Few other cases.

Although not all ***idlgen*** features are shown here, few of the interesting ones are:

- IDL reserved words differ from C reserved words. ***idlgen*** checks C identifiers which are IDL reserved words and renames them to prevent ***idl*** compiler errors. ***idlgen*** reports these cases in warning messages.
- Bit-fields in structs are not allowed in IDL. ***idlgen*** will generate errors for them. Alignment (0-length) fields are removed.
- Long indirections can be "broken" to multiple typedefs where IDL attributes can be defined.
- The choice and limitations in using the indirection attributes [ptr] and [ref] when related to unions, arrays, and function return values are considered by ***idlgen*** when generating the attributes in the output IDL file.
- aggregates in function header are not very useful since their scope is the function head and body alone. Yet, if the C files do that, ***idlgen*** will carefully redefine all these aggregates so that a legal IDL file is generated.

gluegen — glue-code generator

gluegen generates binding code which allows DCE applications to startup, establish a handle, and perform RPC. In Figure 2, the code generated by ***gluegen*** plays parallel role to the stubs generated by the ***idl*** compiler when "wrapping" an application to operate properly in the DCE environment.

To clarify how ***gluegen*** works, we present a simplified view of the possible DCE application models which ***gluegen*** can handle. See [4] for more.

DCE application models

DCE building blocks for distributed applications allow the development of rather complicated and intricate solutions. An operational DCE application is best represented in the general case as a dynamic graph consisting of nodes and links. Nodes are Client and Server programs, each executing on some machine, and links are bindings between Clients and Servers. A link can represent a potential binding between two nodes, a connection, or an on-going RPC. It is essential to emphasize that in DCE, a link is represented via an interface, as described in a particular IDL file. Figure 1 depicts a schematic representation of this idea.

The topology of an application can change during execution. Nodes may appear and disappear, move (i.e., be assigned to different hosts at different times), connect and disconnect in various ways, have multiple connections at a time, gain parallel access or be serialized, and so on. The most general situation can only be implemented by coding a program in a general-purpose language which uses the DCE/RPC primitives.

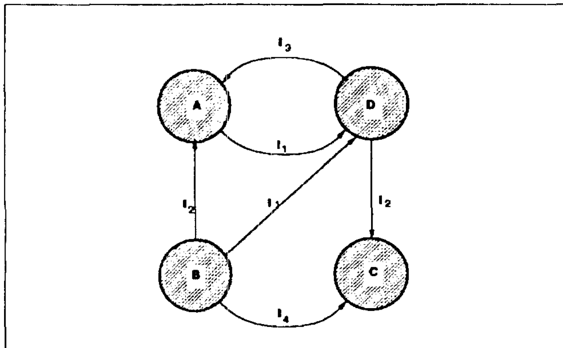

Figure 1. Distributed Application Graph. An edge in the graph represents a collection of RPC-able operations as defined in a **DCE Interface**. The edge arrowhead represents the direction of an RPC. For instance, the edge I_2 directed from B to A specifies that interface I_2 is *imported* by node B and *exported* by node A.

Although the general task of DCE programming is very broad, we are looking for a formal description of this task so that generic templates can be prepared ahead of time and be used to easily build useful DCE applications in a short time.

When an application node imports a given interface, it is said to be a *Client* of that interface. And when it exports a given interface, it is said to be a *Server* of that interface. An application node cannot import and export the *same* interface. Observe, however, that a single application node may import and/or export multiple interfaces. Moreover, a particular interface may be exported by several application nodes. In our example, interface I_2 is exported by nodes A and C. Node A imports and exports interfaces, playing thus both client and server roles, making it a *chaining-server*.

To summarize, we have two major goals: To define the details of a single interface (binding) between DCE nodes of which the IDL file is a major component, and second to combine multiple interfaces into DCE applications.

The DCE environment is more complicated and flexible [9]. Interfaces are uniquely identified by a unique universal identifier (uuid). This identifier is written within the IDL file defining the interface. Additional differentiation comes on the basis of major and minor version numbers in the interface (IDL) file head. Further differentiation is provided by introducing *objects*, which are also uniquely identified by a uuid. An object can be considered as an *implementation* of an interface, and there may be several different implementation of the same interface which are distinguished by an object uuid. Moreover, multiple identical implementations of the same interface may coexist on several machines on the network. In order to bind, a client needs to identify which of the objects of an interface it wishes to bind to, and than select one of the competing servers on the network according to some criteria. Interface uuid-s are written in the IDL file header. Objects uuid-s are not stored in IDL files, but are used in the code which binds a client to a particular server that implements an interface. This complexity is encapsulated within the code generated by ***gluegen***.

Directory and security services are strongly related to the client/server binding issue but will not be included in our discussion of ***gluegen*** glue-code in the scope of this paper (see [4]).

gluegen takes as input the application's topology defined by a set of attributes. ***gluegen*** uses a simple syntax in an *Application Profile* (APF) file. This is a formal language which controls the generation of the appropriate glue-code. Some of the attributes are borrowed from the DCE documented terminology, and some are specific to ***gluegen***.

gluegen profile objects.

Rather than presenting the ***gluegen*** language, we will show an example of defining a Client/Server application using APF.

The APF language identifies an *interface profile*, which defines the binding methods for a particular interface. An interface is associated with a particular IDL file, and thus represents a set of operations. In the DCE application model graphs discussed above, an interface represents an edge between two application nodes. The version and interface name are automatically extracted by ***gluegen*** from the corresponding IDL file.

A collection of interfaces may be associated with an application in an object we term *application profile*. In an application profile, an interface can either be exported, or imported. The application profile also defines how will the application be initialized (start up), and how many concurrent threads of service will coexist in a server.

The interface profile attributes fall into two categories: *compile-time* attributes and *run-time* attributes. Values of compile-time attributes must be specified in the APF. Values of run-time attributes may be specified in the APF, or later at run-time upon invocation of the Client and Server applications. Run-time attributes can come from the command-line, the standard-input, or another file — depending on the user's choice in the APF. Likewise, the values picked for attributes

at run-time, and which are needed to bind to an interface exported by a server, can be reported by the server application to the standard output streams, or to a file. This is also done according to the user's choice in the APF.

When the server reports its binding attribute values to a file, the file can then be fed to the client. As a result, the two applications will bind in a very simple and direct method.

Example. Consider the distributed application graph in Figure 1. Application node A imports interface I_1 and exports interfaces I_2 and I_3. Definition of application node A using the APF language might look as follows (keywords are in bold letters):

```
/* Application Profile for Node A */

interface I1 {
     protseq = ncadg_ip_udp;
     host = nodeD;
     bindtype = repm;
     handle = explicit;
     idl = "I1.idl";
  }
interface I2 {
     protseq = ncadg_ip_udp;
     bindtype = lepm;
     handle = explicit;
     idl = "I2.idl";
}
interface I3 like I2 {
     idl = "I3.idl";
}
application appA {
    finput = null;
    foutput = stdout;
    nthreads = 1;
    import I1;
    export I2;
    export I3;
}
```

Explanation:

The application profile above defines an application named *appA*. The (interface) binding attributes for *appA*, some provided with command arguments and some resolved at run-time, will be written to the standard output (***foutput***= stdout). No binding information will be read from input file (***finput***= null). The application will work serially using a single thread (***nthreads*** = 1).

All three interfaces use the same protocol sequence (***protseq*** = ncadg_ip_udp). The exported interfaces *I2* and *I3* will be registered in the local entry-point mapper — the ***rpcd*** component of DCE (***bindtype*** = lepm). The imported interface *I1* uses the same method on its own host, which can be a remote host (nodeD) to the *AppA* application (***bindtype*** = repm, and ***host*** = nodeD). All three interfaces use an explicit handle (***handle*** = explicit). Each interface also has its own IDL file.

Note the **like** feature allowing I3 to *inherit* most of its attributes from I2. Additional effectiveness can be obtained via C preprocessing:

```
/* Application Profile for Node A  using
** I1.ipf, I2.ipf and the LIKE statement
*/
#include "I1.ipf"
#include "I2.ipf"

interface I3 like I2 { idl = "I3.idl" }

...
}
```

The APF language model is object-oriented for convenience and ease of comprehension, even though it is based solely on attributes and has no user-defined methods. The internal representation of the model is object-oriented, where the application, interface and DCE objects (not shown in the example) are predefined classes with predefined methods. The ***gluegen*** library is not exposed to the programmer and is used only via the APF file and run-time attributes as described above. Exposing more of the library and allowing access to more of the built-in methods will make the combination of compile-time APF, and run-time library very powerful.

Putting it all together

To demonstrate the relationships and origin of components which make up a DCE application using ***idlgen*** and ***gluegen***, Figure 2 on page 7 depicts a situation where a monolithic application (top) is *split* to two programs serving in the roles of a Client and a Server (bottom).

The components of each application node (Client and Server) are as follows:

1. Glue-code and glue-stub are generated by ***gluegen*** from an APF file. Separate pairs of glue-code and glue-stub files are generated for each application — one for the Client, and one for the Server. The glue-codes include the *main()* entry-points of the Client and Server programs.

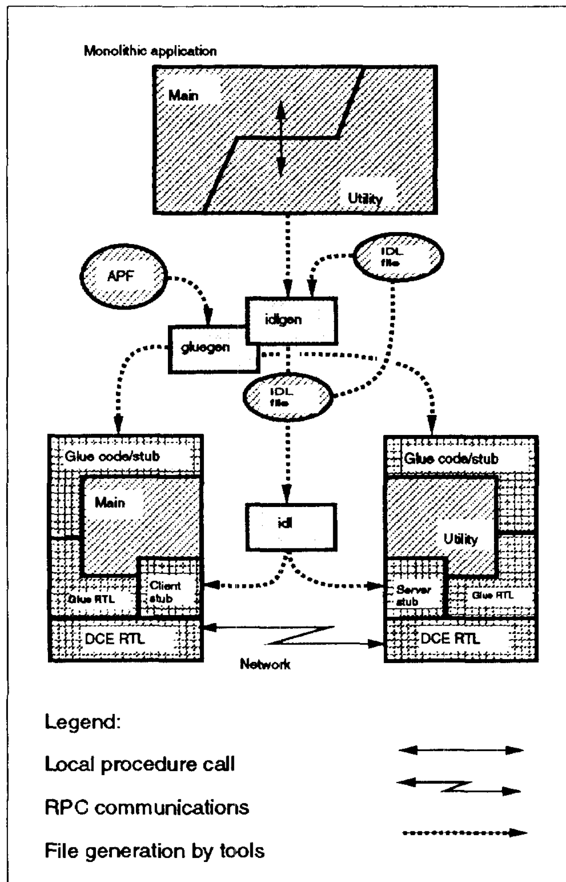

Figure 2. Putting it all together

2. The glue-library is linked with each program and implements the API used in the glue-code. Application code may also use the API of the glue-lib.
3. Application code in the Client is the *Main* part of the original application — which performs an RPC to the Server.
4. Application code in the Server is the *Utility* part of the original application — which implements the RPC performed by the Client.
5. Server-stub is a code generated by the ***idl*** compiler (part of DCE) for the Server. The code is generated from an IDL file which represents the interface between the Client and the Server.
6. Client-stub is a code generated by the ***idl*** compiler for the Client.
7. The IDL file used to generate the stubs in steps 6, and 5, is extracted from the C source by ***idlgen***.
8. DCE RTL is the DCE run-time library which supports the DCE execution environment of DCE applications. This layer uses other lower-level communication support layers in the operating systems of the respective platforms on which the application nodes execute.

Directory and security services

Directory services add another whole world of possibilities, specifically in hiding gory details of objects within symbolic names in a *global* and *cell* directory data bases. When dealing with directory services, one has to distinguish management from access. The management of directory services requires suitable tools which are not part of application development. Within this aspect of management, we can also include many elements of security (see below). In the application development aspects, directory services allow to define an interface via its symbolic name in the directory, reducing the amount of information entered to ***gluegen*** via the APF file.

Security services are used to control access to services via authentication and validation. Although issues of security can be rather complicated, one can observe however, that some aspects, like *Access Control Lists* (ACL) are part of directory services management. To fully exploit security in the application, certain DCE activities may have to be interleaved within the application code.

The extension of the APF language to include objects, directory and security services is being incorporated into ***gluegen*** and will simplifies many aspects of the tool as well as increases its utility. The extensions to APF and to the ***gluegen*** run-time library are not described in this paper (see [4]).

Conclusions

This paper introduces the **MakeDCE** family including the tools ***idlgen*** and ***gluegen*** for the development of DCE applications, with respect to splitting monolithic applications into clients and servers. The tools can be applied to the general case of DCE applications as well, and serve two goals: Reduce code dependency on DCE, and relieve the programmer from being deeply familiar with the details of DCE development toolkit.

There are many alternative approaches to distributed application development. DCE is not an object-oriented system, even though its internal architecture is such. DCE is intended to develop procedural applications on a distributed environment. Object oriented alternatives [8] may or may not use DCE as an imple-

mentation base. For instance, there are efforts to enrich DCE with objects [5], or to build a C++ library encapsulating its run-time [2, 3]. ***gluegen*** uses a library which introduces a higher-level of abstraction above the DCE run-time, but keeps it at the procedural format. The run-time support in ***gluegen*** strongly relates to the elements of the APF language, where *applications*, *interfaces*, and *DCE objects* are treated -- at the language and in the internal representation -- as objects in the "OOP" sense. ***gluegen*** run-time library, maintains this model and gives access to it through a limited API.

Our approach keeps DCE aspects separated from the application logic and makes it much less dependent not only on DCE, but also on the fact that the application is distributed.

A totally different approach is to introduce a new language [11] where aspects of distribution are language elements integrated with the application logic. The application is than independent of DCE, which now is a choice of an implementation vehicle in the language. We believe that the approach adopted by DCE of providing support for distribution via functions is preferred. Independence of the application from DCE can be achieved by separating application logic from aspects of distribution as two orthogonal implementation efforts. Our tools offer an essential instrumentation in this direction.

Acknowledgements

Thanks to Ran Canetti, Karen Laster, and Arie Tal who contributed to the tools reported here.

References

[1] Birrell, A. and B., Nelson, "Implementing Remote Procedure Calls," *ACM Trans. on Computer Systems*, vol. 2, pp. 39-59, Feb. 1984.

[2] Citibank Distributed Processing Technology, Objtran Programmer's Guide (Available via internet from lcp@fig.citib.com), 1993.

[3] Dilley, J., "Object_Oriented Distributied Computing With C++ and OSF DCE," *Proceedings of the International DCE Workshop, A. Schill (ed.)*, pp. 256-266, Karlsruhe WG: Springer-Verlag, October 1993.

[4] Gold, I. and U., Shani, "Wrapping DCE/OSF Client/Server Applications," *Proceedings of USENIX UNIX Applications Development Symposium*, Toronto, Canada, April 25-28 1994.

[5] Mock, M. U., "DCE++: Distributing C++ Objects using OSF DCE," *Proceedings of the International DCE Workshop, A. Schill (ed.)*, pp. 242-255, Karlsruhe, WG: Springer-Verlag, October 1993.

[6] Network Computing Architecture, Apollo Computer Inc., Prentice Hall, 1991.

[7] Network Computing System Reference Manual, Apollo Computer Inc., Prentice Hall, 1991.

[8] Object Management Group, The Common Object Request Broker: Architecture and Specification, 1991.

[9] Open Software Foundation, DCE Application Development Guide, 1993.

[10] Shani, U., N., Amit, I., Boldo, M., Kaplan, J., Marberg, R. Y., Pinter and M., Rodeh, "Program Partitioning for Heterogeneous Machines," *Proceedings, The Sixth Israeli Conference on Computer Systems and Software Engineering*, pp. 136-145, Herzliah, Israel, June 2-3 1992.

[11] Yemini, S., G., Goldszmidt, A., Stoyenko, Y. Wei and L., Beeck, "Concert: A Heterogeneous High-Level-Language Approach to Heterogeneous Distributed Systems," *Proceedings of the 9th International Conference on Distributed Computing Systems*, 1989.