Q: A Multi-lingual Interprocess Communications System for Software Environment Implementation

Mark Maybee
Leon J. Osterweil[1]
Stephen D. Sykes[2]

CU-CS-476-90 June 1990

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430

This research was supported by DARPA grant CCR 8996102

[1]Department of Information and Computer Science, University of California, Irvine, CA 92715
[2]TRW, One Space Park, Redondo Beach, CA 90278

Abstract

Q is a set of matched C and Ada interfaces designed to support interprocess communication between these two languages. It is a first step toward a more general notion of a multi-lingual interprocess communication model. The Q interfaces are adapted from the remote procedure call interface model. The need for modification was imposed by the unavailability of certain C language features in the Ada language. Q attempts to define an interprocess communication model common to both languages, and a type space common to both languages.

1 Introduction

Background

Large systems often need to be able to pass typed objects between seperate subcomponents written in different languages. For example, the Rebus system in the Arcadia environment project [8] uses an Ada process which interacts with the user, and a C process which provides database services. The Ada and C processes must communicate by passing typed objects back and forth. The same need has arisen in the case of a user interface front end that interacts with a database back end. Due to differences in debuggers, run-time systems, and difficulties in managing the complex system as a single process, the front end was seperated into one Ada based process, and the back end into a different database process.

In the separate process architecture, there is a need to pass typed objects between the two processes. The Unix inter-process communications capability — XDR/RPC[3] — provides a limited way to do this. XDR and RPC were designed to allow one process to make a procedure call to another process — *independently of the different computer architecture the other process lived in* — and have XDR/RPC handle the details of inter-process communication.

However, XDR/RPC only supports procedure calls between two processes written in the C language. Hence its interfaces are written in C and make use of semantic constructs which are not supported in Ada (such as procedure variables). Additionally, objects do not migrate between different type systems as they do when they are passed between an Ada and a C process. XDR does not deal with issues relating to consistency of types in the different type models.

As a temporary solution to these problems, we developed a primitive Ada interface to the Unix XDR/RPC services. This interface implements a slightly different model of communication than the Unix XDR/RPC model, mainly because of the lack of procedure variables in Ada. To use this interface for communication between Ada processes one need only understand the interface specifications. However, when it is used to communicate between Ada and C processes, detailed knowledge of the implementation is required. No analogous C interface had been constructed, so the standard Unix RPC/XDR interfaces had to be used. Although the differences between the Ada and C interprocess communication models were minor, they were subtle and confusing to users.

[3]eXternal Data Representation/Remote Procedure Call

Q

Q addresses these difficulties in using by providing a C interface which is directly analogous to our Ada interface, and simpler than the standard XDR/RPC interface. We call these new interfaces QDR and QPC. Additionally, several simplifications of our Ada interface were made, mainly to remove lower level RPC services[4].

This paper is a description of the Q system. It includes:

- a discussion of work related to Q being done at CMU and MIT,
- a discussion of the current implementation's inter-process communications model and its derivation from the Unix XDR/RPC model,
- a specification of the current Q software implementation,
- a discussion of relevant experience using Q, and
- the long term goals of Q and the limited scope of Q activities as described herein,

2 Related Work

Our model of interprocess communication is derived from the remote procedure call model [2]. We have adapted this model for use in a multi-typespace environment. In doing so we have identified a common type model that represents an intersection between the C and Ada languages which we utilize as the foundation for data exchange between these languages. Similar work in this area has been done at Carnegie Mellon University with the *Matchmaker* language [5], designed to support the construction of multi-lingual interprocess communication interfaces, and at the Massachusetts Institute of Technology in the form of a value transmission method for abstract data types [4].

Matchmaker is a language for specifying and automating the generation of multi-lingual interprocess communication interfaces. When supported by the capability-based interprocess communications found in the Mach kernel [1] it provides a heterogeneous, distributed, object-oriented programming environment. Currently the Mach/Matchmaker system supports interface generation for the C, Common Lisp,

[4]The standard Unix RPC library provides several levels of client services. This allows the communications protocol to be precisely controlled, if so desired, by using low level services. In the long term, Q will most likely support the same multiple levels of services the Unix RPC services provide, or more sophisticated services.

Ada, and Pascal languages. The Matchmaker language defines the type model within which the supported languages may exchange data objects. This type model is similar to that loosely defined by the Q interfaces. From a Matchmaker interface specification, which includes both the object type declarations as well as the interface function specifications, the system is capable of automatically generating both client and server interfaces for any of the supported languages.

The automatic generation of the client and server interfaces represents a significant advantage over the Q system. However, the use of this technique also introduces a certain amount of inflexibility into the system as well. By providing the *building blocks* for type encoding, Q allows users to pass quite sophisticated objects between client and server.[5] Matchmaker only supports a single level of pointer dereference for example. The servers generated by the Matchmaker system also suffer from inflexibility. Matchmaker supports only a single, object-oriented, client/server model of communication. The notion of an *active* server is unknown in the Matchmaker system, while in Q this notion is supported specifically for the Ada language where modeling the server as a task seems natural.

The work done at MIT on a value transmission method for abstract data types is designed to support communicating abstract data types between regions of a system using different data value representations. This method defines call-by-value semantics for communicating values over a network of different computers. A canonical representation for each type used in communications is defined. Each implementation of the type must define a translation between its internal representation and the canonical representation. Automated support is provided to construct language interfaces from the canonical type specifications and interface specifications. Currently only the C and Lisp languages are supported.

As with Matchmaker; the notion of an external type representation being used to accomplish the automatic generation of interprocess communication interfaces forms the core of the system. Although the system currently only supports communications within a single language space (i.e. only C-to-C and Lisp-to-Lisp) it is clear that the techniques are applicable to inter-language communication. However, this method appears to suffer from the same inflexibility as the CMU system. Since the routines for translating from the source language representation to the canonical representation are generated automatically, there is little control over this mapping. It is unclear how mapping of sophisticated data types would achieved.

[5]For example, it is possible to produce a data encode routine to pass an entire linked graph structure (such as a binary tree) between client and server.

3 The Q Implementation

The Q communications model is derived from Sun's eXternal Data Representation (XDR) standard [6] and their Remote Procedure Call (RPC) protocol [7]. The existing implementation of XDR is designed for use with the C programming language, offering support for the standard C base data types. The existing implementation of RPC is also designed for use with the C programming language, utilizing standard C call semantics, and is also restricted in its implementation to versions of Unix with IPC sockets available. The communications semantics for Q were abstracted from these models for use between the Ada and C languages. The Q implementations, QDR and QPC, are intended to provide communications between Ada and C programs, including support for the transfer of objects built from Ada and C base types.

The Model

The communications model was derived from Sun's Remote Procedure Call protocol specification. As the name suggests, this model is based on the concept of emulating procedure calls across process (and processor) boundaries. Although the actual communication is achieved via a message passing subsystem, the RPC layers abstract this into a procedure call interface. The system uses a client/server model to achieve procedure call-like semantics. A client process initiates a remote procedure call which is then "serviced" by a specified server process. The server process need not reside on the same processor.

In order to successfully communicate data types between differing machine architectures the RPC model is combined with the XDR protocol. The XDR protocol defines data encodings for a set of *base* types as well as the ability to build up the base types into more sophisticated abstract data types. Before transmitting the procedure call data from the client to the server (or vice versa) the data is first encoded using XDR and then decoded on the receiving side, thus ensuring data type integrity.

It is considered the responsibility of all type implementors to provide encode/decode functions for types to be used in interprocess communication. Private types without provided encapsulation functions can be arbitrarily encoded as opaque data — but this implies that the data cannot be interpreted on the receiving side of the communication and that instances may even be incompletely represented. For example, private data types involving complex structure (i.e. pointers) cannot be properly encoded/decoded as opaque data.

The RPC model

Server side: Each server procedure (*service*) is "registered" with a procedure number, argument list, and encode/decode routines for each argument. Additionally, the server process itself is registered on the machine it is running on with a process identifier (program number).

When a server receives a service request it (1) decodes the argument list using the routines defined during registration, (2) invokes the procedure, (3) encodes return arguments, and (4) returns a message to the client process with the encoded argument list.

The RPC mechanism automatically applies the encode/decode routines which were provided during registration of the server procedures.

Client side: Remote procedures are invoked via a special function called `callrpc`. This function takes a variable number of arguments: the machine name and program number, remote procedure number, the argument list, and the procedures to encode/decode the argument list. It is the sole responsibility of the persons coding the client and server to ensure that the encode/decode routines registered with the server are compatible with those provided during a `callrpc` function call.

Upon invocation of this function the argument list is encoded, and a message is sent to the server with the encoded argument list. Upon receipt of a response message the return argument list is decoded and control is returned from the function.

As with the server process, the RPC mechanism automatically applies the encode/decode routines to the arguments.

The QPC model

Ada's refusal to accept variable length procedure argument lists and procedure parameters make the standard model for remote procedure calling impossible to implement. Hence the communications model has been modified to accommodate Ada's restrictions.

Server side: Instead of registering multiple service procedures, only a single service dispatch procedure is registered. A "service type" argument is provided to this procedure to distinguish service requests. The service dispatch procedure can then call an appropriate local procedure to handle the particular service being requested.

The service arguments are also passed in a different manner than in the standard model. Only a single "argument" is passed — a QDR buffer. This buffer must

then be explicitly unpacked by the server — the buffer may contain multiple arguments that can differ according to the type of service being requested. Each service procedure unpacks the buffer as appropriate for the arguments to that procedure.

Client side: For the reasons just stated for the server, the client side must explicitly encode and pack its arguments into a QDR argument buffer before invoking the `callrpc` special function.

As with the standard model, it is the sole responsibility of the persons coding the client and server to ensure that the arguments in the QDR buffer are consistent between the client and server.

Convention Orientation of Q

One way of viewing Q is that it supports the ability to take objects from some type schema, defined via a type model, and pass those objects into a "pipeline". The objects are subsequently removed from the pipeline, and placed in a *new* schema, defined via a (potentially) *new* type model. Examples of type models are those provided by C and Ada; an example of a type schema is a set of types defined by type declarations in a particular C or Ada program.

There are two extreme approaches that can be taken in supporting the migration of objects between these two type schemas. One is to provide mechanisms for defining a unifying type model, and a means for automatically migrating objects between type schemas which have been defined in terms of that type model, or in models which are unified by that type model. It would then be the responsibility of the Q system to automatically ensure that the objects are placed in the destination schema in a manner consistent with their position in the originating schema.

The other extreme is to provide low level capabilities for placing objects into the pipeline, and for removing them from the pipeline. It is then the responsibility of the people writing applications to agree upon conventions for passing objects through the pipeline such that their positions in the originating and destination type schemas are consistent.

The second, convention oriented, approach is the one that this version of Q takes.

The Implementation

The overall structure of Q is shown in Figure 1.

QDR is separate from QPC because it is sometimes desirable for a system to pack and unpack data without passing it to another process via QPC. Hence the

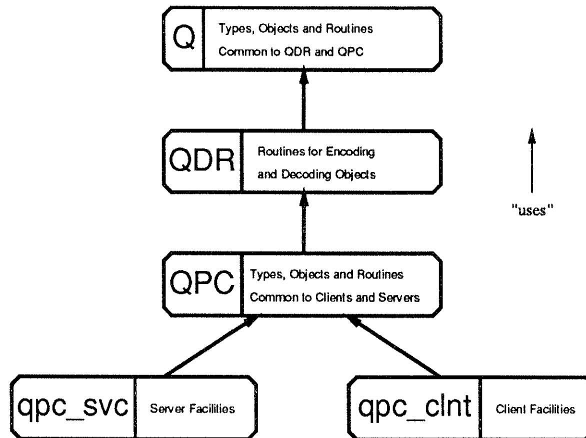

Figure 1: The Q system architecture

QDR routines are "stand alone". However, QPC relies on the QDR routines, and hence anyone who uses QPC will need to understand QDR.

The Q Interface

The top level interface to Q defines types and objects which are necessary for use of either QDR or QPC. A set of base types are defined here which are guaranteed to have the following property: if an instance of a particular C or Ada type derived from these base types is placed into a Q buffer, and then removed as another instance of that C or Ada type, then all properties of the instance that were true prior to its being encoded, will be true following its being decoded. Properties of interest include

- satisfaction of Ada constraints,
- values of Ada attributes, and
- representability of value by type in C (*i.e.* whether or not the object's value is within the range of the type).

Integer, floating point and character base types are supported.

This guarantees the sizes and ranges of numbers that can be passed between programs using Q: any Q integer (or floating point number) in Ada is guaranteed to map to a legal Q integer (or floating point number) in C, and *vice versa*, and a C character is guaranteed to map properly to an Ada character.

These types define the *base* types that are guaranteed to be handled properly. Any types which

- can in C be safely cast into these types, and then back again, and
- in Ada are subtypes of these types or are derived from these types

can be safely passed between programs via Q.

The QDR Interface

The QDR interface provides routines which *encode* and *decode* base data types in the C and Ada languages. Encoded data values are stored as a *linearization* in a buffer which is referred to, in the implementation, as a *handle.* These buffers are used as arguments to the other QDR routines, which pack and unpack data to and from a buffer. The buffers are also passed between clients and servers in the QPC model — hence the buffers are also used as arguments to various QPC routines.

The QDR interface routines are symmetric. Each routine is capable of acting as both a value encoder and value decoder. Therefore, the calls required to encode a set of data items is identical to the set of calls necessary to decode it. New encode/decode routines for aggregate types built from the QDR supported base types will in general retain this symmetric property. Figure 2 gives an example of such a user-defined routine.

Although, of necessity, there are differences between the QDR interface specifications for C and Ada, there exists a well-defined mapping between the interface routines. Mapped pairs of routines define the set of types which are "consistent" between the Ada and C languages. When passing data between C and Ada processes, data which is packed using a member of a paired set may be unpacked by a member of that paired set. See Figure 3 for the defined mapping.

Most of the QDR procedures are self-explanatory. The procedure *bool*, for example, packs and unpacks a Boolean value into the supplied buffer. Some, however, require more detail. This detail follows.

Ada Fixed Point Types: C has no type which is analogous to the Ada fixed point types. Nevertheless, one may wish to to pass fixed point objects to other Ada programs, or to C programs simply for storage. For this reason, there is a routine on the Ada side for encoding fixed point typed objects. C routines cannot decode these objects, so they should be passed in linearized buffers that are not decoded on the C side.

Ada Private Types: We do not provide any means for encoding objects which are Ada `private`. We believe that the implementation of the private type contains

```
TYPE    TestType IS RECORD
                int_field        : Integer;
                float_field      : Float;
                fixed_field      : Money;
                bool_field       : Boolean;
                slice_field      : String(1..10);
                string_field     : String_ref;
                END RECORD;

PROCEDURE qdr_testtype (
      qdrs            : IN QDR.Handle;
      s               : IN OUT TestType
      ) IS
      PROCEDURE qdr_integer IS NEW
                      generic_integer (Integer);
      PROCEDURE qdr_float IS NEW
                      generic_floating (Float);
      PROCEDURE qdr_fixed IS NEW
                      generic_fixed (Money);
BEGIN
      qdr_integer (qdrs, s.int_field);
      qdr_float (qdrs, s.float_field);
      qdr_fixed (qdrs, s.fixed_field);
      bool (qdrs, s.bool_field);
      string_slice (qdrs, s.slice_field);
      strings (qdrs, s.string_field);
END qdr_testtype;
```

Figure 2: Example QDR encode/decode routine

Ada procedure	C function
QDR.generic_integer	qdr_integer
QDR.generic_floating	qdr_floating
QDR.generic_fixed	qdr_bytes
QDR.generic_enumeration	qdr_integer
QDR.bool	qdr_bool
QDR.strings *or*	qdr_string *or*
QDR.string_slice	qdr_bytes
QDR.generic_pointer	qdr_pointer
QDR.generic_array	qdr_array

Figure 3: QDR interface routine mapping

semantic information which is necessary for successfully defining the meaning of encoding objects of that type, and building the routine to do so. For example, a private type named `stack` might simply be a pointer to a linked list of stack elements. In a proper abstraction, only the implementation would have the information necessary to encode the whole stack, and define exactly what it means to do so. Providing a QDR routine to encode a stack object would simply encode the pointer variable — while the user could believe they were encoding the whole stack.

Enumeration Types Within the C language, enumeration types are represented as integers. The Ada routine for enumeration objects is therefore designed to be compatible with the C integer routine. The value encoded is the integer value returned by Ada's `T'POS` operation for the specific enumeration type [9].

Strings It should be noted that while there are several different interface routines defined for the string type in both the C and Ada interfaces, they all utilize the same underlying data encoding technique. They have been provided for user convenience and are all compatible and interchangeable.

The QPC Interface

Again, the C and Ada interfaces are paired, with semantically equivalent routines in each interface pair. Systems that use QPC have two types of processes: **servers** and **clients**. Servers define and register procedures for handling client calls, utilizing the server interface. Clients call server procedures that have been registered, using the

```
TYPE    Service_ID      IS (PING, INC);

PROCEDURE service (
      id               : IN Service_ID;
      argblk           : IN QDR.Handle;
      result           : IN QDR.Handle
      ) IS
      PROCEDURE qdr_integer IS NEW
                QDR.generic_integer(Integer);
      value    : Integer;
BEGIN
      IF id = INC THEN
              qdr_integer (argblk, value);
                value := value + 1;
              qdr_integer (result, value);
      END IF;
END service;
```

Figure 4: Example service dispatch routine

client interface. A process may be both a server and a client, in which case it uses both interfaces. Finally, since parameters to server procedures must be encoded and decoded using QDR, both servers and clients use the QDR interface.

The Server: Servers are created by defining a dispatch procedure that will process QPC calls for a particular set of services, and then registering it. Figure 4 presents an example service dispatch routine. The intention is that any one process should be registered under only one program number at any one time, and hence may not function simultaneously as different servers. Hence an Ada program with several tasks, for example, should not have separate tasks acting as different servers. Figure 5 presents an example server which registers the dispatch routine from figure 4.

The Client: The client specifies a server (via a machine, version, and server id), the type of service requested, (simply an id which can be used by the server to discriminate between requests when one server provides several services), and a QDR buffer which contains packed parameters. On return, this buffer contains results which the server placed in the second, results, buffer. As was mentioned above, it is the responsibility of the system designers to ensure that the type and

```
WITH QDR;
WITH QPC;
WITH QPC_SVC;

PROCEDURE server IS
        .
        .
        .

        PACKAGE ThisService IS NEW
                QPC_SVC(QPC.program_number(1),
                        QPC.DEFAULT_VERSION,
                        Service_ID, service);
BEGIN
        Put_Line ("Starting server...");
        ThisService.SVC_TASK.svc_start;
        Put ("Started - return to terminate: ");
        Get_Line (input, last);
        Put_Line ("Terminating server");
        ThisService.SVC_TASK.svc_stop;
        ThisService.SVC_TASK.svc_terminate;
END server;
```

Figure 5: Example server

order of data in the buffer on call and return are consistent with those expected by the server. Figure 6 presents an example client process.

4 Experience with Q

To date Q has played a major part in two significant software projects. It is supporting the interprocess communication needs of the Arcadia Chiron project [14, 3]. Chiron is a user interface management system developed to support the UIMS needs of the Arcadia environment. It relies heavily on the notion of separating the application program from the display process in a client/server split. Q is also supporting the use of interprocess communication in the Triton data management system. Triton is a highly extensible and flexible database management system being developed to compliment the PGRAPHITE [10], PIC [13], and SLI [12] systems to support many of the Arcadia persistent and shared data management requirements. It also relies on a client/server split between the application program and the database management process.

Chiron

Q is being used in the development of Chiron-1 because of its ability to cleanly separate conceptual boundaries thus preventing unwanted compiler dependencies, its ability to dynamically add and interact with new software modules, and its ability to describe arbitrary data structures in a language independent (between C and Ada) fashion.

The separation of concerns between the application domain (model) and the presentation domain (view) in Chiron-1 require a sharp split between a Chiron Client (application) and a Chiron Server (presentation). Separating the server code from the client code reduces the size and complexity of client applications. This also allows modifications to be made without having to recompile both modules. Because Q allows the server process to reside on a separate processor from the client process, Chiron's flexibility, portability, and efficiency are all increased. The client and server rely heavily upon Q to link and maintain interprocess communication between the Client Protocol and Server Protocol Managers which are the gateways between the two modules.

Because of the Client/Server split, Chiron needs to send various Ada structures using Q protocols. Chiron uses the QDR functions to encode and decode the classes and Ada records which are passed between the client and server. QDR functions are also useful in other areas where communication between different languages might corrupt the data. For example, in Chiron's mapper, where events from the server

```
WITH QDR;
WITH QPC;
WITH QPC_CLNT;
WITH Text_IO;   USE Text_IO;

PROCEDURE client IS

      value            : Integer := 3;
      wagram           : QPC_CLNT.Svc_server;

      TYPE Services   IS (Ping,Inc);
      PROCEDURE qdr_integer IS NEW
                QDR.generic_integer (Integer);
      PROCEDURE qpc_call IS NEW
                QPC_CLNT.call (Services);

BEGIN
      wagram := QPC_CLNT.establish
                ("wagram", QPC.program_number(1));
      Put_Line ("Value before: " &
                Integer'IMAGE(value));
      QDR.set_write (QPC.Handle);
      qdr_integer (QPC.Handle, value);
      qpc_call (wagram, Inc);
      QDR.set_read (QPC.Handle);
      qdr_integer (QPC.HANDLE, value);
      Put_Line ("Value after: " &
                Integer'IMAGE(value));
END client;
```

Figure 6: Example client

are mapped to procedures in the client, QDR functions are used to encode and decode the parameters which will be used when the Ada procedure's address is dereferenced in C. The QDR functions prevent the corruption of the data passed from server to client and then from Ada to C and finally back to Ada.

Triton

Triton utilizes Q to handle interprocess communication needs. Triton is designed to provide object management capabilities to a wide variety of tools and processes that will populate the Arcadia environment. Triton will provide dynamic binding of types that may be shared among processes and will provide persistent storage of instances of those types. Because of the diverse nature of the tools and processes within the Arcadia environment, there is a need for a general mechanism for handling communication between processes instantiated with diverse programming languages.

Abstraction of the interprocess communication capabilities helps Triton focus on object management issues. Q provides such an abstraction and fits well into the Triton model. In Triton, we envision object management capabilities to be provided on a serverized basis such that any process within the environment could request services asynchronously as a client of the server. Q was designed with specific support for the client/server model. A process may pass data to the Triton server by encoding and decoding the data using the QDR data representation scheme on both the client side and the server side of an interprocess communication channel. QPC then transmits the data without relying on an understanding of the underlying interprocess communication mechanisms. Since Triton will serve processes dynamically, Q provides the ideal interface mechanism for passing data.

5 Future Work

Although Q contains relatively minor modifications and extensions to the Unix XDR/RPC software, and is implemented almost directly on the UNIX XDR/RPC software platform, the philosophy of Q contains some significant additions to the XDR/RPC model. It is not our intent to address these extensions in detail here, but only to mention them briefly as long term goals of the work.

The ultimate goal of this work is a language and architecture independent interprocess communication capability. To provide such a capability, we must first address issues with regard to the transfer of data between differing programming languages, and hence differing type systems. This is tantamount to the quest for a single universal type model, and is clearly a non-trivial problem. We do not intend to address this issue here. Our means of skirting this issue was to limit our scope to

only provide communications capabilities for Ada and C processes[6]. Furthermore, data in our system which is passed between an Ada and a C process migrate between *two type systems.* Hence each object has two types, one in each process.

We only allow data to be passed between these processes when we can determine that the two *different* types of the object are consistent. Consistency is defined via a pairwise listing of Ada and C base types. Two types are consistent if they are, or are derived from, a pair in the list. Our definition of consistent is limited in that we have not fully unified the two type systems, but only convinced ourselves that it seems reasonable to pass objects between each pair of types in the different type models. The ultimate responsibility for deciding whether two types are consistent will rest with the creator of the software that uses Q: Q will transfer objects between the two type systems when the object's types are a pair in the list.

The result of this limited scope is that we did not need to find a type model which will unify programming language type models in general, and the Ada and C type models in specific. We believe, however, that later versions of Q must provide strongly typed communications where consistency is rigorously defined and automatically verified.

Another thorny issue (which we likewise sidestep) crops up when we consider what it means to take an object from a process and *store* it elsewhere for some duration, change the state of the process, and then re-introduce the object into the process. The exact meaning of an object (i.e. its *value*) may well be dependent on the state of the process; hence its value may implicitly change as the state of the process changes.

As an example of such an object consider a *stack.* If a stack is defined as an Ada private type, then we have at least two possibilities when we take a stack object from a process and then return it at a later time. First, the object could contain all of the stack values, and returning it to the process would return the stack in its original state. Second, the stack could be a reference to the stack values. Assuming that this reference is valid on return to the process, we will get whatever values the referred-to stack has on return. If there were aliases to the referred-to stack, and those aliases were used to push or pop the stack, then the value of the saved stack may be different upon its return. A more extreme example occurs when an object persists past the life of the process that created it. In this version of Q, we simply ignored this issue. We expect people who create software using Q to ensure to their own satisfaction that their system behaves in a reasonable manner. Clearly we must address these issues in future elaborations upon this work.

[6] I.e. Ada↔Ada, C↔C, Ada↔C

6 Conclusions

Q represents the first step toward a more general muti-lingual interprocess communication model. Through Q we have investigated, in a microcosm of the C and Ada languages, the difficulty involved in attempting to unify diverse type models and identify a communications paradigm applicable to a variety of language models. Q has also turned out to be surprisingly useful. The fact that its ability to facilitate communications between C and Ada processes was immediately utilized by several projects implies that a more general multi-lingual facility may be even more desirable and useful.

Although Q does not explicitly define a unified type model for interprocess communication, it does identify implicitly a type model intersection between the C and Ada languages. Through an explicit set of mapped type encoding/decoding procedures, Q provides a base upon which typed objects may be exchanged meaningfully between these two languages. Work on the development of a more general unified type model is currently underway as part of the Arcadia project [11]. We hope to introduce this model as the intermediate representation for types in the Q system, thereby reducing the n^2 problem of identifying the type intersection of n languages to a n problem of defining the mapping between each language type system and the unified type model.

Although the remote procedure call model is a natural model for most computer languages, it is not always straightforward to implement in an arbitrary languages. The remote procedure call model tends to impose an object-oriented flavor to the resulting IPC interfaces. For most procedural languages and particularly for object-oriented languages this is extremely useful. Q attempts to retain this basic model while adapting itself to the limitations and peculiarities of arbitrary languages. We hope to broaden Q's ability to deal with arbitrary languages by expanding on the current fixed remote procedure call model to include general message passing and broadcast models of communication.

The Q implementation has proven to be immediately useful. Its desirability as a key component in several large software projects has demonstrated the need for such a facility in the software community. As software systems grow in complexity and as computational power continues to be distributed in larger and more diverse networks, the importance of being able to distribute the system as a set of communicating processes increases. Often individual components of the system will be best implemented in diverse languages. A general mechanism which will allow these components to communicate at a meaningful level of type abstraction would clearly be of tremendous use.

Acknowledgments

We would like to thank Greg Bolcer of the Chiron development team and Harry Yessayan of the Triton development team for their contributions. Their patience in endeavoring to use a new system and their invaluable feedback is appreciated. We would also like to acknowledge Dennis Heimbigner who had tremendous influence on the development of the Q system.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. In *Procedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.

[2] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. Technical Report CSL-83-7, XEROX, October 1983.

[3] Gregory Alan Bolcer, Mary Cameron, M. Gregory James, Rudolf K. Keller, Richard N. Taylor, and Dennis B. Troup. Chiron-1: Concept and design. Arcadia Technical Report UCI-89-12, University of California, Irvine, October 1989.

[4] M. Herlihy and B. H. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[5] Michael B. Jones and Richard F. Rashid. Mach and matchmaker: Kernel and language support for object-oriented distributed systems. Technical Report CMU-CS-87-150, Carnegie Mellon University, September 1986.

[6] Sun Microsystems. XDR: External data representation standard. Technical Report RFC-1014, Sun Microsystems, Inc., June 1987.

[7] Sun Microsystems. RPC: Remote procedure call protocol specification. Technical Report RFC-1057, Sun Microsystems, Inc., June 1988.

[8] Richard N. Taylor, Lori Clarke, Leon J. Osterweil, Jack C. Wileden, and Michal Young. Arcadia: A software development environment research project. In *Proceedings of the IEEE Computer Society Second International Conference on Ada Applications and Environments*, pages 137–149, Miami, Florida, April 1986.

[9] United States Department of Defense. *Reference Manual for the Ada Programming Language*, 1983. ANSI/MIL-STD-1815A-1983.

[10] Jack C. Wileden, Alexander L. Wolf, Charles D. Fisher, and Peri L. Tarr. PGRAPHITE: An experiment in persistent typed object management. In *Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments*, pages 130–142, Boston, November 1988.

[11] Jack C. Wileden, Alexander L. Wolf, William R. Rosenblatt, and Peri L. Tarr. UTM-0: Initial proposal for a unified type model for arcadia environments. Arcadia Technical Report UM–89–01, University of Massachusetts, Amherst, 1989.

[12] Jack C. Wileden, Alexander L. Wolf, William R. Rosenblatt, and Peri L. Tarr. Specification level interoperability. In *Proceedings of the Twelfth International Conference on Software Engineering*, Nice, March 1990. To appear.

[13] Alexander L. Wolf, Lori A. Clarke, and Jack C. Wileden. Ada-based support for programming-in-the-large. *IEEE Software*, 2(2):58–71, March 1985.

[14] Michal Young, Richard N. Taylor, and Dennis B. Troup. Software environment architectures and user interface facilities. *IEEE Transactions on Software Engineering*, June 1988.