Printed on: Mon Feb 1 09:49:52 1988

Last saved on: Mon Feb 1 09:19:34 1988

Document: paper

For: mishkin.osdev.r_d.C366

pl2ps 1.4 Copyright 1986 Interleaf, Inc.

The Network Computing Architecture and System: An Environment for Developing Distributed Applications

Terence H. Dineen, Paul J. Leach, Nathaniel W. Mishkin,
Joseph N. Pato, Geoffrey L. Wyant
Apollo Computer Inc.
...!{wangins,yale,mit-eddie}!apollo!mishkin

1. Introduction

The Network Computing Architecture (NCA) is an object-oriented framework for developing distributed applications. The Network Computing System™ (NCS™) is a portable implementation of that architecture that runs on Unix® and other systems. By adopting an object-oriented approach, we encourage application designers to think in terms of what they want their applications to operate on, not what server they want the applications to make calls to or how those calls are implemented. This design increases robustness and flexibility in a changing environment.

NCS currently runs under Apollo's DOMAIN/IX™ [Leach 83], 4.2BSD and 4.3BSD, and Sun's version of Unix. Implementations are currently in progress for the IBM PC® and VAX/VMS®. Apollo Computer has placed NCA in the public domain.

In addition to its object orientation, some interesting features of the system are as follows. It supplies a transport-independent remote procedure call (RPC) facility using BSD sockets as the interface to any datagram facility. It provides at-most-once semantics over the datagram layer, with optimizations if an operation is declared to be idempotent. It is built on top of a concurrent programming support package that provides multiple threads of execution in a single address space, although versions can be made for machines that just have asynchronous timer interrupts. The data representation supports multiple scalar data formats, so that similar machines do not have to convert data to a canonical form, but can instead use their common data formats. The RPC interface definition compiler is extensible. Procedures to do the client/server binding can be attached to data types defined in the interface. Also, complex data types can be marshalled by user-supplied procedures which convert such types to data types the compiler understands. There is a replicated global location database: Using it, the locations of an object can be determined given its object ID, its type, or one of its supported interfaces.

There are several motivations for NCA. Large, heterogeneous networks are becoming more common. Users of systems in such networks are often frustrated by the fact that they can't get those systems to work cooperatively. Over the last few years, advances have been made in allowing *data sharing* to occur between the systems, but not *compute sharing*. Tools to allow the effective use of the aggregate compute power have not been available. The inability to share computing resources has become even more aggravating as more specialized processors (e.g. ones designed to run numerical applications fast) have become more widespread. Current technology obliges users of those processors to resort to FTP and Telnet. Even in an environment of systems of relatively similar power, a network computing architecture is called for: There are applications that can take advantage of many systems in parallel. (Parallel make is the most obvious example.) Also, replicating resources over a number of machines increases the reliability seen by users of the network.

It is important to understand that there is almost no network application that can't be implemented with*out* NCA/NCS. However, the implementation is bound to be more difficult, less general, and harder to install on a variety of systems. Further, experience has shown that some obviously useful network applications simply don't get written because of these problems. The existence of NCA/NCS helps to solve these problems and as a result, expand the set of network applications.

2. Architecture

Figure (1) illustrates NCA's overall structure.

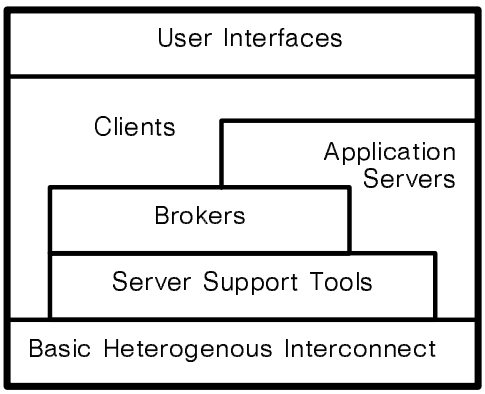

Figure 1. NCA's overall structure.

2.1 Heterogeneous Interconnect

The lowest level provides the basic interconnection to heterogenous computing systems. At this layer NCA currently defines a remote procedure call protocol (NCA/RPC), a Network Interface Definition Language (NIDL), and a Network Data Representation (NDR). RPC is a mechanism that allows programs to make calls to subroutines where the caller and the subroutine run in different processes, most commonly on different machines. The RPC approach and an implementation similar to ours is described in detail by Birrell and Nelson [Birrell 84]. NIDL is a high-level language used to specify the interfaces to procedures that are to be invoked through the RPC mechanism. NCS includes a portable NIDL compiler that takes NIDL interfaces as input and produces stub procedures that, among other things, handle data representation issues and connect program calls to the NCS RPC runtime environment that implements the NCA/RPC protocol. The relationships among the client (i.e. the caller of a remoted procedure), server, stubs, and NCS runtime is shown in figure (2).

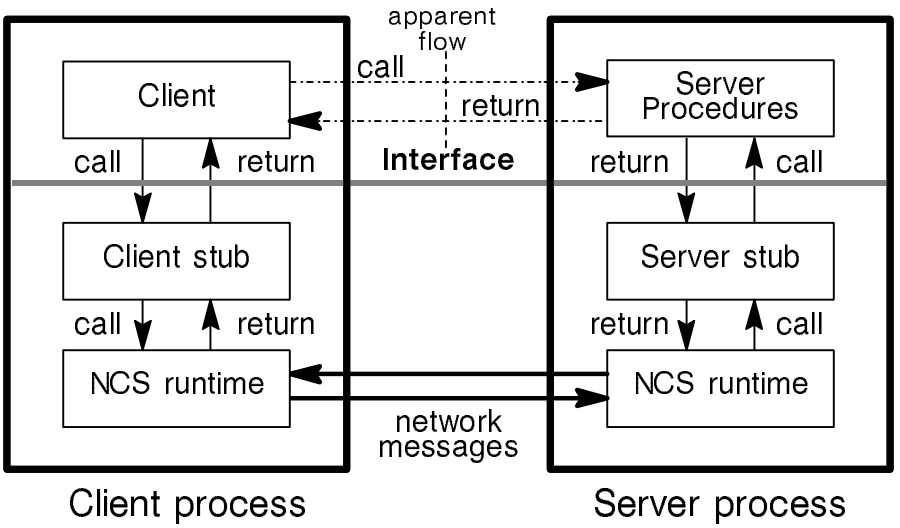

Figure 2. Relationships among client, server, stubs and NCS runtime

2.2 Server Support Tools

Augmenting the heterogenous interconnect layer are the server support tools. These tools simplify the writing of complex applications in a distributed environment. Currently these consist of the

Data Replication Manager (DRM) and Concurrent Programming Support (CPS). DRM provides a weakly consistent, replicated database facility. It is useful for providing replicated objects when high availability is important and weak consistency can be tolerated. CPS provides integrated lightweight tasking facilities. CPS allows multi-threaded servers to written easily.

2.3 Brokers, Clients, Servers and User Interfaces

Built on top of the server-support tools are a set of *brokers*. A broker is a third party agent that facilitates transactions between principals. In a network computing environment brokers are primarily useful in determining object locations, but can also be used for establishing secure communications (i.e. authentication), associatively selecting objects, issuing software licenses, and a variety of other administrative chores not directly related to the operation of the principals. The role of brokers is shown in figure (3).

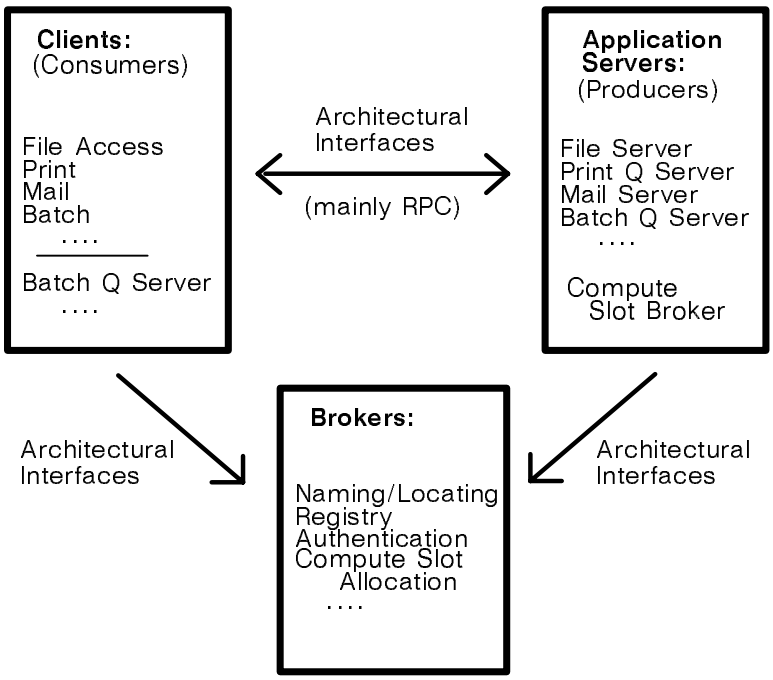

Figure 3. The role of brokers in NCA

Client programs and application servers make use of the three base layers. Application servers are the producers of services and clients the consumers . Servers invoke brokers to make their existence known. Clients can invoke brokers to locate application servers and then use the underlying RPC mechanism to make use of the services provided. The application server may be in turn a client of other distributed services.

From user's perspective, user interfaces tie all the pieces together. However, user interfaces are not part of NCA and will not be discussed in this paper.

2.4 Unique Identifiers

An important aspect of NCA is its use of *universal unique identifiers* (UUIDs) as the most primitive means of identifying NCA entities (e.g. objects, interfaces, operations). UUIDs are an extension of the unique identifiers (UIDs) already used throughout Apollo's system [Leach 82]. Both UIDs and UUIDs are fixed length identifiers that are guaranteed to refer to just one thing for all time. The principal advantages of using any kind of unique identifiers over using string names at the lowest level of the system include: small size, ease of embedding in data structures, location transparency, and the ability to layer various naming strategies on top of the primitive naming mechanism. Also, identifiers can be generated anywhere, without first having to contact some

other agent (e.g. a special server on the network, or a human representative of a company that hands out identifiers).

UIDs are 64 bits long and are guaranteed to be unique across all Apollo systems by embedding in them the node number of the system that generated the UID and the time on that system that the UID was generated. To make it possible to generate unique identifiers on non-Apollo system we defined UUIDs to be 128 bits and made the encoding of the identity of the system that generates the UUID more flexible.

The remainder of this paper discusses several aspects of NCA and NCS: NCA's object-oriented approach; NIDL; NDR; the NIDL compiler; the Location Broker used in connecting clients with servers; and the networking model and protocol used by NCS. We conclude with a description of future directions we expect NCA and NCS to follow.

3. The Object-Oriented Approach

NCA is object-oriented. By this we mean that it follows a paradigm established by systems such as Smalltalk [Goldberg 83], Eden [Almes 83, Lazowska 81], and Hydra [Wulf 75, Cohen 75]. The basic entity in an object-oriented system is the *object*. An object is a container of state (i.e. data) that can be accessed and modified only through a well-defined set of *operations* (what Smalltalk calls *messages*). The implementation of the operations is completely hidden from the client (i.e. caller) of the operations. Every object has some *type* (what Smalltalk calls a *class*). The implementation of a set of operations is called a *manager* (what Smalltalk calls a set of *methods*). Only the manager of a type knows the internal structure of objects of the type it manages. Sets of related operations are grouped into *interfaces*. Several types may support the same interface; a single type may support multiple interfaces.

For example, consider an interface called *directory* containing the operations *add_entry*, *drop_entry*, and *list_entries*. This interface might be supported by two types: *directory_of_files* and *print_queue*. There are potentially many objects of these two types. That there are many objects of the type *directory_of_files* should be obvious. By saying that there are many *print_queue* objects we mean that a system (or a network of connected systems) might have many print queues — say, one for each department in a large organization.

3.1 Motivation

The reason for using the object-oriented approach in the context of a network architecture is that this approach lets you concentrate on *what* you want done, instead of *where* it's going to be done and *how* it's going to be done: objects are the units of *distribution, abstraction, extension, reconfiguration,* and *reliability*.

Distribution. Distribution addresses the question of where an operation is performed. The answer to this question is that the operation is performed where the object resides. For example, if the print queue lives on system A, then an attempt to add an entry to the queue from system B must be implemented by making a remote procedure call from system B to system A. (This implementation fact is hidden from the program attempting to add the entry.)

Abstraction. Abstraction addresses the question of how an operation is performed. In NCA, the object's type manager knows how the operation is performed. For example, a single program *list_directory* could be used to list both the contents of a file system directory and the contents of a print queue. The program simply calls the *list_entries* operation. The type managers for the two types of objects might represent their information in completely different ways (because, say, of the different performance characteristics required). However, the *list_directory* program uses only the abstract operation and is insulated from the details of a particular type's implementation.

Extension. The object-oriented approach allows extension; i.e. it specifies how the system is enhanced. In NCA, there are two kinds of extensions allowed. The first is extension by creation of new types. For example, users can create new types of objects that support the *directory* interface; programs like *list_directory* that are clients of this interface simply work on objects of the new type, without modification. The second kind of extension is extension by creation of new interfaces. A new interface is the expression of new functionality.

Reconfiguration. Because of partial failures, or for load balancing, networked systems sometimes need to be reconfigured. In object-oriented terms, this reconfiguration takes place by moving

objects to new locations. For example, if the system that was the home for some print queue failed because of a hardware problem, the system would be reconfigured by moving the print queue object to a new system (and informing the network of the object's new location).

Reliability. The availability of many systems in a network should result in increased reliability. NCA's approach is to foster increased reliability by allowing objects to be replicated. Replication increases the probability that least one copy of the object will be available to users of the object. To make replication feasible, NCS provides tools to keep multiple replicas of an object in sync.

While NCA is object-oriented and we believe that applications that use the object-oriented capabilities of NCA will be more robust and general than those that don't, it is easy to use NCS as a conventional RPC system, ignoring its object-oriented features.

4. Network Interface Definition Language

The Network Interface Definition Language (NIDL) is the language used in the Network Computing Architecture to describe the remote interfaces called by clients and provided by servers. Interfaces described in NIDL are checked and translated by the NIDL compiler.

NIDL is strictly a *declarative* language it has no executable constructs. NIDL contains only constructs for defining the constants, types, and operations of an interface. NIDL is more than an interface definition language however. It is also a *network* interface definition language and, therefore, it enforces the restrictions inherent in a distributed computing model (e.g. lack of shared memory).

4.1 NIDL Language Constructs

A NIDL interface contains an header, constant and type definitions, and operation descriptions. The header provides the interface identification: its UUID, name, and version number. The UUID is the name by which an interface is known within NCA. It is similiar to the program number in other RPC systems, except that it is not centrally assigned. The interface name is a string name for the interface which is used by the NIDL compiler in naming certain publicly known variables. The version number is used to support compatible enhancements of interfaces.

A standard set of programming language types is provided. Integers (signed and unsigned) come in one, two, four, and eight byte sizes. Single (four byte) and double (eight byte) precision floating point numbers are available. Other scalars include signed and unsigned characters, as well as booleans and enumerations.

In addition to scalar types, NIDL provides the usual type constructors: structures, unions, pointers, and arrays. Unions must be discriminated. (I.e. non-discriminated unions are not permitted. The actual data values must be known at runtime so that it can be correctly transmitted to the remote server.) Pointers, in general, are restricted to being top-level. That is, pointers to other pointers, or records containing pointers are not permitted. Later, we'll see how this restriction can be relaxed. Arrays can be fixed in size or have their size determined at runtime.

Operation declarations are the heart of a remote interface definition. These define the procedures and functions that servers implement and to which clients make calls. All operations are strongly typed. This enables the NIDL compiler to generate the code to correctly copy parameters to and from the packet and to do any needed data conversions. Operation declarations can be optionally marked to have certain semantic properties, for example whether they are *idempotent*. (An idempotent procedure is one that can be executed many times with no ill-effect.)

All operations are required to have a *handle* as their first parameter. This parameter is similar to the implicit self argument of Smalltalk-80 or the this argument of C++ [Stroustrup 86]. The handle argument is used to determine what object and server is to receive the remote call. NIDL defines a primitive handle type named *handle_t*. An argument of this type can be used as an operation's handle parameter. Clients can obtain a *handle_t* by calling the NCS runtime, providing an object UUID and network location as input arguments. Use of more abstract kinds of handles is described below.

Handle arguments can be implicit. An interface definition can declare that a single global variable should be treated as the handle argument for all operations in the interface. While this style

conflicts with some of the goals of the object-oriented approach (e.g. it makes it harder to make calls on different objects using the same interface), it can be useful in cases where an existing local interface is being converted to work remotely.

4.2 NIDL Example

Figure (4) is a short example of an interface described in NIDL. The example is of an interface to a bank object that supports a single operation: deposit money into an account.

(1) Defines the UUID by which this interface is known. This the first version of this interface. If in the future, new operations are added, the version number should be incremented. (2) Declares the interfaces upon which this interface is dependent. The *import* statement is similiar to *#include*, except that the named interface is not textually included. The contents are made available for the importer to refer to types and constants defined in that interface. This allows factoring out a common set of types into a base interface. (3) Defines a set of types (account and account name types) that are used by the bank operations. Finally (4) defines the operation itself.

A variant of NIDL that looks Pascal-like (as opposed to the C-like version of which figure (4) is an example) is also available. Regardless of the variant used as input to the NIDL compiler, the output is the same.

```
[uuid(334033030000.0d.000.00.87.84.00.00.00), version(1)]     (1)
interface bank {
    import
          nbase.imp.idl  ;                                    (2)
    typedef                                                   (3)
        long int bank$acct_t;
    typedef
        char bank$acct_name_t[32];
    void bank$deposit(                                        (4)
        [in]    handle_t          h,
        [in]    bank$acct_t       acct,
        [in]    long int          amount,
        [out]   status_$t         *status
    );
};
```

Figure 4. Example interface

4.3 Object-Oriented Binding

One drawback of the language as described so far is that all operations are required to have a primitive *handle_t* as their first argument. This means clients need to embed these handles in their programs, and to manage the binding to servers themselves. We would like to achieve as much local-remote transparency as possible (i.e. to make programs insensitive to the location of the objects upon which they operate). Embedding primitive handles in client programs destroys much of this transparency. To relieve clients of the need to manage these handles, we introduced the notion of *object-oriented binding*.

Object-oriented binding comes into play when the first parameter to an operation is *not* a *handle_t*. In this case, the type is taken to represent some more abstract, client-oriented handle. Since to actually make remote calls, a *handle_t* is required, some way is needed to translate the abstract handle into a *handle_t*. The person who creates the abstract type is thus obliged to write a procedure to do the conversion. This procedure is assumed to have the name *type_bind* (where *type* is the type name of the abstract handle) and is automatically called from stubs when the remote call is made. You can view the abstract handle as an *object* (in the Smalltalk sense) which supports the *bind* operation.

To make this more concrete, we could reformulate the above bank example in terms of object-oriented binding. Instead of taking a *handle_t* as its first parameter, *bank$deposit* could take a bank name, of type *bank$name*. The NIDL compiler would generate a call to *bank$name_bind* to translate from a bank name to the primitive *handle_t*. This routine would probably call upon some

sort of naming server to look up the bank location. The bind routine might also choose to cache location information to make later translations faster.

Object-oriented binding hides the details of handle binding from the client and allows interfaces to be designed in a more abstract, client-oriented fashion. This provides a higher level of local-remote transparency than other systems which always require the client to manage handles or explicitly name the remote host on each call.

4.4 Marshalling Complex Types

In the section on NIDL language constructs, we stated that pointers could not be nested. The reason is that such nesting would require the NIDL compiler to generate code to transmit general graph structures. However, permitting only top-level, non-nested pointers can be a severe limitation in the design of an interface. For example, it excludes passing tree data structures to remote procedures.

To provide an escape from this restriction, NIDL allows a type to have an associated transmissible type. The transmissible type is a type that the NIDL compiler *does* know how to marshall. Any type that has an associated transmissible type must have a set of procedures to convert that type to and from its transmissible type. In the example of the binary tree, the transmissible type could be an array. The *tree$to_xmit_rep* procedure would walk the tree to build a representation of it in the array, and the *tree$from_xmit_rep* procedure would reconstruct the binary tree from the array.

Transmissible types may be associated with any type, not just types using nested pointers. Bitmaps are an example. It may be represented internally as a fixed size array of integers. Even though the NIDL compiler is capable of marshalling this, it may be more efficient to have it transmitted in a run-length encoded (RLE) form. So the bitmap type could have an associated *RLEBitmap* type, and a set of procedures for converting to and from the RLE form.

5. Network Data Representation

Communicating typed values in a heterogenous environment requires a data representation protocol. A data representation protocol defines a mapping between typed values and *byte streams*. A byte stream is a sequence of bytes indexed by nonnegative integers. Examples of data representation protocols are Courier [Xerox 81] and XDR [Sun 86]. A data representation protocol is needed because different machines represent data differently. For example, VAXes represent integers with the *least* significant byte at the low address and 68000s represent integers with the *most* significant byte at the low address. A data representation protocol defines the way data is represented so that machines with different local data representation can communicate typed values to each other.

NCA includes a data representation protocol called Network Data Representation (NDR). NDR defines a set of data types and type constructors which can be used to specify ordered sets of typed values. NDR also defines a mapping between ordered sets of values and their representations in messages.

Under NDR, the representation of a set of values consists of two items: a *format label* and a byte stream. The format label defines how scalar values are represented (e.g. VAX or IEEE floating point) in the byte stream; its representation is fixed by NDR as a data structure representable in four bytes.

NDR supports the scalar types *boolean, character, signed integer, unsigned integer,* and *floating point*. Booleans are represented in the byte stream with one byte; *false* is represented by a zero byte and *true* by a non-zero byte. Characters are represented in the byte stream with one byte; either ASCII or EBCDIC codes can be used. Four sizes of signed and unsigned integers are defined: *small, short, long,* and *hyper*. Small types are represented in the byte stream with one byte, short types with two bytes, long types with four bytes, and hyper types with eight bytes. Either big- or little-endian representation can be used for integers; two's complement is assumed for signed integers. The two sizes of floating point type are *single* and *double*. Single floating point

types are represented with four bytes and double floating point types use eight bytes. The supported floating point representations are IEEE, VAX, Cray, and IBM.

In addition to scalar types, NDR has a set of type constructors for defining aggregate types. These include *fixed size arrays, open arrays, zero terminated strings, records,* and *variant records*.

Fixed sized arrays have a known number of elements. Their values are represented in the byte stream simply as a sequence of representations of the values of the elements. Each element value is represented according to the element type of the array. Open array types have a fixed first index value and element type but their final index value is not known from their type. Therefore, it is necessary to represent the value of the index of the last element in the array immediately before the representation of the values of the array elements.

Zero terminated strings can be viewed as a special case of open arrays; they are open arrays of characters whose last index value is defined by a terminating zero byte. To support this common data type in an efficient manner, NDR represents such values with an explicit length value followed by the characters of the string including the terminating zero character.

Record values are represented in the byte stream by representations of the values of their fields in the order defined by the record type. Variant records are assumed to have an initial set of fixed fields which includes a tag field used to discriminate among the possible variants. Representations of the values of the fields of the selected variant follow the representations of the values of the fixed fields of a variant record value.

Some types may appear to be missing from NDR. NDR has no enumerated types, bit set types, or a pointer type constructor. The definition of a NIDL maps such types onto their representations in an NDR byte stream. For example, NIDL maps enumerated types and bit sets onto the NDR unsigned integer type of the appropriate size. Typed pointer values are mapped into the NDR type which represents the type that the pointer references.

NDR is abstract in that it does not define how the format label and the byte stream are represented in packets. The NIDL compiler and the NCA/RPC protocol are users of NDR: They work together to generate the format label and byte stream, encode the format label in packet headers, fragment the byte stream into packet-sized pieces, and put the fragments in packet bodies.

The important features of NDR are its flexible representation of scalar values, its use of *natural alignment*, and its extensibility.

By using a format label to specify an interpretation of the scalars in a byte stream NDR supports a recipient makes it right approach to data conversion in a heterogenous environment. A sending process can use its preferred encoding of scalars when constructing a byte stream providing that it is one of the defined options. A receiving process needs to convert data representations only when the format specified in the incoming format label differs from its own preferred format. Thus, two compatible machines can communicate efficiently without needing to convert to a conventional network format and back again on each transmission. NDR defines a broadly useful but not universal set of scalar formats. We believe that our choices are reasonable for promoting heterogenous network computing combining workstations and special purpose server machines. On the other hand, it is important to keep the space of possible formats to a reasonable size because each recipient needs to convert any incoming scalar format to its own.

NDR requires that values be natually aligned in the byte stream. Natural alignment means that all values of size 2^n are aligned at a byte stream index which is a multiple of 2^n, up to some limiting value of n; NDR choses this limit to be 3. (I.e. scalars of size up to eight bytes are naturally aligned.) This permits, but does not require, implementations of NCA to align buffers for the byte stream so that stub code can use natural operators to manipulate values in the byte stream effi-

ciently and without alignment faults. This also helps to promote communication ease between different kinds of machines in a heterogenous environment.

By its use of a format label NDR is an extensible data representation protcol. The format label could be extended to specify other aspects of the data representation such as packing disciplines, dynamic typing schemes, new encodings of scalars, or new classes of scalars.

6. The NCS NIDL Compiler and Stub Functions

NCS includes a compiler which mediates between NIDL on the one hand and NDR and the NCS runtime on the other. The functions of the compiler are: checking the syntax and semantics of interface definitions written in NIDL; translating NIDL definitions into declarations in implementation languages such as C; and generating client and server stubs for executing the remote operations of an interface.

The NIDL compiler is organized as a front-end component and a back-end component. The front-end parses and checks an interface definition and produces an abstract syntax tree (AST) intermediate form. If the interface definition is sound, the front-end then passes this tree to the back-end which generates implementation language include files and stub code files for the interface.

NCS's NIDL compiler is implemented for portability in C using YACC and LEX. It is available in source form to encourage its use and extension in heterogeneous networked environments.

6.1 NIDL Compiler Functions

Distributed object-oriented programming imposes certain restrictions on the semantics of interfaces. It is part of the compiler's job (along with the design of NIDL) to enforce these restrictions. We illustrate the front-end's semantic checks with some examples. All types used in a definition must be well defined. All parameters and fields whose type is an open array require the use of a *last_is* attribute to give their size at call time. Every remote interface requires a UUID. Every operation of an interface requires an implicit or explicit handle parameter to support object-oriented programming.

The second major function of the NIDL compiler is to derive files which declare the interface's constants, types, and operations in the languages in which client applications and servers are written. These files are included in client and server programs which use or implement the remote operations of an interface. For the current implementation the supported languages are C and Pascal. Generating these files is done by a fairly straightforward walk over the AST; adding the capability to generate include files in other Algol-like languages would be a simple exercise.

In addition to declaring the constants, types, and operations of an interface, the derived include files declare two important statically initialized variables defined for each interface. One is the *interface specification* (*ifspec*) which encapsulates the identity of the interface and its salient properties (number of operations, well known ports used, etc.). The ifspec variable is used in the binding and registering operations of the NCS runtime. The second variable is the server *Entry Point Vector* (*EPV*) which holds pointers to the server side's stub routines. This EPV variable is used by a server process when registering as a server for an interface; it is used by the NCS runtime to dispatch incoming calls.

The third major function of the NIDL compiler is to generate files of stub code for the operations defined in an interface. There are two such files one contains client side stub routines and the other contains server side stub routines. This emitted code is in standard C, which we use as a universal assembler to promote portability. Each operation in an interface gives rise to a client stub routine and a server stub routine. The following section discusses the functions of these routines.

6.2 Stub Functions

Client stub routines are called by clients of an interface; they have the same interface as the operation for which they stand in. Server stub routines are called by the server side NCS runtime; their interface is defined by NCS. Client stub routines call the client side NCS runtime to perform

remote calls. Server stubs call the manager's implementation of an operation to provide the actual service. Thus, the first function of stubs is to hide the NCS runtime from users and implementors of remote interfaces and to create the illusion of accessing a remote procedure as though it were local.

To communicate input and output arguments and function results between callers and called routines the stub must *marshall* and *unmarshall* argument values into call and reply packets. This is done in accordance with NDR and the conventions of NCS. Unmarshalling code is also responsible for detecting and performing necessary data conversions by comparing the incoming format label with the local formats. Data conversion is done by a combination of inline code and support operations in the NCS runtime.

The stubs also need to calculate the size requirements for call and reply packets based on the dynamic size of input and output arguments. The size information is used to determine whether or not a pre-declared packet on the stack is large enough. If not, the stubs need to allocate and free storage for packets. It is *not* the job of the stub to break up a large packet into pieces that can be sent over the network the NCS runtime provides the capability of handling arbitrarily sized packets.

Client side stubs map the operations of an interface to the operation number used by the NCS runtime to identify operations; they also pass options designating the desired calling semantics and the ifspec derived from the NIDL declaration of an operation to the NCS runtime's remote call primitive.

On the server side, the stub routines are responsible for managing storage to be used as the server side surrogates for dynamically sized arguments. This is necessary to support the server's illusion of large data structures passed to it by reference.

The stubs also manage the more elaborate features of NIDL described in section 3 above. Client stubs support automatic binding by calling users' binding and unbinding routines when necessary. Implicit handles are made explicit to the NCS runtime by client stub routines. Users' marshalling routines are invoked as necessary by both client and server stubs as part of marshalling input and output arguments of the appropriate types.

In summary, the stub generation function of the NIDL compiler automates the production of a large amount of protocol code based on a routine's interface defintion. This is important because the code is complex enough to make its hand coding very error prone and tedious. Hand producing this kind of code has been a major impediment to building distributed systems in the past.

7. Location Broker

A highly available location service is a fundamental component of a distributed system architecture. Objects representing people, resources, or services are transient and mobile in a network environment. Consumers of these entities cannot rely on a priori knowledge of their existence or location, but must consult a dynamic registry. When consumers rely solely on a location service for accessing objects, it becomes essential that the location server remain available in the face of partial network failures.

The NCA *Location Broker* (NCA/LB) protocol is designed to provide a reliable network-wide location broker. This protocol is defined by a NIDL interface and is thereby easily used by any NCA/RPC based application.

The NCA/LB, unlike location services like Xerox SDD's Clearinghouse [Oppen 83] or Berkeley's Internet Name Domain service (BIND) [Terry 84], yields location information based on UUIDs rather than on human readable string names. The advantages of using UUIDs were described earlier.

7.1 Locating

An object's type manager must first advertise its location with the Location Broker in order for that objected to locatable. A manager advertises itself by registering its location and its willingness to support some combination of specific objects, types of objects, or interfaces. A manager can

choose to advertise itself as a global service available to the entire network, or limit its registration to the local system. Managers that choose the latter form of registration do not make themselves unavailable, but rather limit their visibility to clients that specifically probe their system for location information.

Clients find objects by querying the Location Broker for appropriate registrations. A client can choose to query for a specific object, type, interface, or any combination of these characteristics. When operations are externally constrained to occur at a specific location, a client can choose to query the location broker at the required system for managers supporting the appropriate object.

7.2 Location Broker Organization

The Location Broker is divided into two components. The *Global Location Database* is a replicated object containing the registration information of all globally registered managers; the processes that manage this database are called the *Global Location Broker*. The NCS runtime implementation of the Global Location Broker uses the *Data Replication Manager* (*DRM*) to maintain the database. DRM provides a weakly consistent replicated KSAM package. Weak consistency implies that replicas of the Global Location Database object may be inconsistent at any time, but, in the absence of updates, that all replicas will converge to a consistent state within a finite amount of time. This form of consistency provides a high degree of both read and update availability to the Global Location Database. It is not necessary to be able to communicate with all replicas of the object to affect a change in the registration database. The DRM assumes the responsibility of propagating updates to the replicas in a timely fashion.

A *Local Location Broker* supports managers that wish to limit their registration to the local system. Access to these registrations if provided in two ways. A client can directly query the Location Broker at specific node to determine the objects and managers that are registered there. Alternately, a client can simply execute a remote operation while supplying an incompletely bound handle (i.e. one which specified only an object and system, not a particular server process). Remote calls made using such a handle are delivered to the Local Location Broker, which serves as a forwarding agent if an appropriate manager has registered itself locally. This mechanism obviates the need for users of the NCA to use well known ports.

The division of the Location Broker into two distinct entities is, to a large degree, an NCS runtime implementation decision. Logically the Local Location Database object and the Global Location Database object are a single partitioned object, and, in fact, access to these databases is provided through a common set of operations which select the target based on lookup keys.

8. The NCA/RPC Protocol and NCS Implementation

The NCA/RPC protocol is designed to be low cost for the common cases and independent of the underlying network protocols on top of which it is layered. The NCS runtime implementation of the NCA/RPC protocol is designed to be portable.

8.1 Protocol

The NCA/RPC protocol is designed so that a simple RPC call will result in as few network messages and have as little overhead as possible. It is well known that existing networking facilities designed to move long byte streams reliably (e.g. TCP/IP) are generally not well suited to being the underlying mechanism by which RPC runtimes exchanges messages. The primary reason for this is that the cost of setting up a connection using such facilities and the associated maintenance of that connection is quite high. Such a cost might be acceptable if, say, a client were to make 100 calls to one server. However, we don't want to preclude the possibility of one client making a call to 100 servers in turn. In general, we expect the number of calls made from a particular client to a particular server to be relatively small. The reliable connection solution is also unacceptable from the server's perspective: A popular server may need to handle calls from hundreds of clients over a relatively short period of time (say 1-2 minutes). The server does not want to bear the cost of maintaining network connections to all those clients.

The well-known way of getting around the well-known problem of using reliable network connections is to make the RPC protocol implement exactly the reliability it needs on top of an *unreliable* network service (e.g. UDP/IP). This approach has the additional advantage that some systems

(e.g. embedded microprocessors) can not or do not support *any* reliable network service; however, if they're connected to a network at all, you can be sure that they'll at least supply an unreliable service. Further, unreliable services tend to be more similar across protocol suites than do reliable services. (For example, some reliable protocols might return errors immediately if the network partitions even though a virtual circuit is currently idle, while others might defer until the next time I/O is attemped.) This similarity means that the RPC protocol can be accurately implemented in more protocol suites than if it would be possible if it assumed a reliable service.

All that the NCA/RPC protocol assumes is an underlying unreliable network service. The protocol is robust in the face of lost, duplicated, and long-delayed messages, messages arriving out of order, and server crashes. When necessary, the protocol ensures that no call is ever executed more than once. (Calls may execute zero or one times and, in the face of network partitions or server crashes, the client may not know which.)

The NCA/RPC protocol operates roughly as follows. The client side sends a packet describing the call (a *request* packet) and waits for a response. The server side receives and dispatches the request for execution, and sends a packet in response that describes the results of executing the call (the *response* packet). If the client doesn't receive a response to a request within a particular amount of time, it can inquire about the status of the request by sending a *ping* packet. The server either sends back a *working* packet, indicating that execution of the request is in progress, or a *nocall* packet, which means that the request has been lost (or that the server has crashed and rebooted) and the client needs to resend it. The protocol gets slightly more complicated if the input or output arguments do not fit into one packet.

If a called procedure is non-idempotent, the protocol ensures that the server executes the call at most once. To detect old (duplicate) requests, the server keeps track of the sequence number of the previous request for each client with which it has communicated. However, the server considers this information to be discardable and it may discard it if it hasn't heard from the client in a while. (I.e. there is no permanent connection between the client and server.) Thus, it is possible for a long-delayed duplicate request to arrive after the server has discarded the information about the requesting client. To handle this case, the server *calls back* to the client (using an idempotent remote procedure call) to ask the client for the client's current sequence number. The server then uses the returned sequence number to validate the request. Note then that for calls to non-idempotent procedures (with input and output arguments that fit in a single packet), a total of two message pairs will be exchanged between client and server for the simple case. Subsequent calls between the same client and server will require just one message pair. Note that the extra message pair in the first case could conceivably be eliminated if the server were willing to hold onto client sequence number information for long enough to ensure that all duplicate requests had been flushed from the network. We chose not to take this approach since any time interval we considered long enough (e.g. one minute or more) seemed too long to oblige the server to hold the information.

Also, for non-idempontent procedures, the server side saves and periodically retransmits the response packet until the client side has acknowledged receipt of the response. If the server side receives a retransmission of the request, it resends the saved response instead of re-executing the call. The client side acknowledges the response either implicitly, by sending a new request, or explicitly, by sending an *acknowledgement* packet. The protocol also handles the case in which the server has executed the non-idempotent call but, because of network partitions or a server crash, fails to send the response packet.

If a called procedure is idempotent, the protocol makes no guarantees about how many times the procedure is executed. On idempotent requests, the server side does not save the results of the operation once it has sent back the response packet. In addition, the client side is not required to acknowledge the receipt of responses to idempotent requests.

8.2 Runtime

The NCS RPC runtime is written in portable C and uses the BSD Unix *socket* abstraction. (In terms of the socket abstraction, it uses SOCK_DGRAM-style sockets.) This abstraction is intended to mask the details of various *protocol families* so that one can write protocol-independent networking code. (A protocol family is a suite of related protocols; e.g. TCP and UDP are part of the

DoD IP protocol family; PEP and SPP are part of the Xerox NS protocol family.) In practice, however, the socket abstraction has to be extended in several ways to make it possible to write truly protocol-independent code. We extended the socket abstraction via a set of operations implemented in a user-mode subroutine library; the NCS runtime uses these extensions so that it can be truly protocol-independent. Bringing up the NCS runtime on a new protocol family should *not* require any changes to the NCS runtime proper. All that should be required is to add some relatively trivial routines to the socket abstraction extension library.

NCS is careful about creating sockets. Sockets are a fairly scarce resource and tying lots of them up for a long period is not a good idea. NCS keeps of small private pool of sockets. One is pulled from the pool when a process makes a remote call. When the call completes, the socket is returned to the pool. The pool need contain only one socket for the entire process if the system supports only one thread of control per process (as is the case in standard Unix).

The use of the socket abstraction at all could be considered to be too much of a BSD-ism, thus reducing the portability of the runtime. Fortunately, two factors argue against this point of view: First, it appears that AT&T System V, Release 3 will support at least a sufficient subset of the socket calls (layered on top of their own networking model). Second, even if the target of a port doesn't have anything resembling the socket interface, NCS use of the interface is fairly simple and it wouldn't be too hard to implement the BSD calls in terms of whatever the target system supplies.

9. Future Directions

NCA and NCS represent the first step in a complete network computing environment. One of the guiding goals in the development of NCA has been *transparency*. This has a number of aspects: replication, failure, concurrency, location, and name transparency.

With replication transparency all copies of an object can be considered equivalent. The user of an object cannot tell whether it consists of a single copy or many. The DRM provides replication transparency in the case where some short-lived inconsistencies can be tolerated. Future versions of NCA will include support for strongly consistent replication.

Location transparency allows users to access objects without specifying where the objects are. Objects are free to be moved around the network to adapt to changing load conditions and the availability of new hardware. The Location Broker provides the ability to find the location of objects prior to their first use. We would like to be able to have objects move at any time during program execution.

Concurrency transparency supports the illusion that a given client is the sole user of an object. NCS addresses this partially through concurrent programming support which provides a simple locking facility. In the future, we would like to address this, and to some degree, failure transparency, through the use of an object-oriented atomic transaction facility.

Failure transparency, i.e. the ability of components of a distributed system to fail and recover transparently to their users, is largely a function of location and replication transparency. By replicating objects, when a given replica fails another is available to takes its place. Location transparency hides the switch from one replica to another from the user.

Neither NCA nor NCS address the issue of name transparency at this point. We anticipate building a general purpose name server in a future version of NCS. In addition, we intend to address a higher-level form of naming: In many instances, it is more convenient to find an object by attributes rather than by a text name. An *attribute broker* will provide this ability. Thus, a client will be able to query the attribute broker for a list of 26 page/sec laser printers rather than managing the mapping between machine names and attributes itself.

Most of the focus in the NCA development so far has been on getting the basic model right. Once the object-oriented model is in place, we feel that these higher level services will evolve naturally. Had we started with a more traditional process-oriented model, the level of integration and transparency we desire would be much more difficult to achieve.

References

[Almes 83]
Guy T. Almes. Integration and distribution in the Eden system. Technical Report 83-01-02, Department of Computer Science, University of Washington, 1983.

[Birrell 84]
Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, II(1):39-59, 1984.

[Cohen 75]
Ellis Cohen and David Jefferson. Protection in the Hydra operating system. In *Proceedings of the Fifth Symposium on Operating Systems Principles*, pages 141-160. ACM Special Interest Group on Operating Systems, 1975.

[Goldberg 83]
Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Lazowska 81]
Edward D. Lazowska, Henry M. Levy, Guy T. Almes, Michael J. Fischer, Robert J. Fowler, and Stephen C. Vestal. The architecture of the Eden system. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, 148-159. ACM Special Interest Group on Operating Systems, 1981.

[Leach 82]
Paul J. Leach, Bernard L. Stumpf, James A. Hamilton and Paul H. Levine. UIDs as Internal Names in a Distributed File System. In *Proceedings of the Symposium on Principles of Distributed Computing*, 34-41. Association for Computing Machinery, 1982.

[Leach 83]
Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A. Hamilton, David L. Nelson, and Bernard L. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, SAL-I(5):842-857, 1983.

[Oppen 83]
D. C. Oppen and Y. K. Dalal. The Clearinghouse: A decentralized agent for locating named objects in a distributed environment. *ACM Transactions on Office Information Systems* I(3):230-253, 1983.

[Stroustrup 86]
Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[Sun 86]
Sun Microsystems. Networking on the Sun workstation. Part no. 800-1324-03. 1986.

[Terry 84]
D. B. Terry, M. Painter, D. Riggle and S. Zhou. The Berkeley Internet Name Domain Server. In *Proceedings of the Usenix Association Summer Conference*, 21-31. 1984.

[Wulf 75]
W. Wulf, R. Levin, C. Pierson. Overview of the Hydra operating system development. *Proceedings of the Fifth Symposium on Operating Systems Principles*, pages 122-131. ACM Special Interest Group on Operating Systems, 1975.

[Xerox 81]
Xerox Corporation. Xerox System Integration Bulletin, OPD B018112. 1981.

APOLLO and DOMAIN are registered trademarks of Apollo Computer Inc. Network Computing System, NCS, and DOMAIN/IX are trademarks of Apollo Computer Inc.
IBM is a registered trademark of Internation Business Machines Corporation.
UNIX is a registered trademark of AT&T
XNS is a trademark, and ETHERNET is a registered trademark of Xerox Corporation.
VAX is a registered trademark of Digital Equipment Corporation.