A Portable, Network-Transparent Communication System for Message-Based Applications

Michael P. Zeleznik

Department of Computer Science
University of Utah
Salt Lake City UT 84112

Abstract

A portable, network-transparent communication system (NTCS) has been developed, which supports the message passing requirements of a distributed information retrieval system testbed. The NTCS provides interprocess communication, while isolating the application from issues such as physical location, underlying communication details, and internetting. It also allows for dynamic reconfiguration; processes can be transparently distributed across different machines while running.

Full implementations of this NTCS have been used extensively during its development over the last two years. It has proven to be both highly portable, and relatively insensitive to the underlying operating systems, with processes currently distributed among Apollo, VAX, and Sun systems, across both TCP and Apollo MBX communication support.

This paper emphasizes the more novel features of the NTCS, which have proved valuable in its design: a portable dynamic naming service is built recursively on top of the communication services it supports; the underlying portable internet scheme uses the naming service which it supports, to determine routing; and the method of inter-machine data conversion dynamically adapts to the environment, and is independent of the conversion scheme. Finally, some of the difficulties with supporting the necessary recursion in a communication system are discussed.

1. Introduction

We have developed a portable, network-transparent communication system (NTCS), which supports the message-passing requirements of a large class of message-based, distributed applications: those of a large-grain, loosely-coupled nature, distributed at the process level [29]. All interprocess communication takes place through the NTCS. There is no shared memory among processes. It forms the foundation on which all other aspects of the application are built; from the application-specific programs themselves, to the necessary distributed run-time support (DRTS) services, such as process control and time service.

The NTCS supports these requirements with a complete set of message passing primitives allowing independent processes to communicate, while isolating them from issues such as physical location, underlying communication details, and internetting. It also allows for dynamic system reconfiguration; application processes can be distributed across multiple machines and networks, while running, transparent at the application interface. These aspects make the NTCS approach ideal for testbed environments, for which our particular implementation was targeted.

Implementations of this NTCS have been used extensively for the past two years, as the foundation of a distributed information retrieval system [5]. The NTCS is built on top of the existing interprocess communication system (IPCS) on each machine, following a cleanly layered design. All communication dependencies are localized to a conversion layer, with portable upper layers providing internet support, dynamic reconfiguration, and a naming service. The NTCS has proven to be both highly portable, and relatively insensitive to the underlying operating system. It was implemented in the C programming language, and currently runs under both Unix TCP and Apollo MBX communication support, with processes distributed among Apollo, VAX, and Sun systems.

1.1. Purpose of this Paper

While a brief overview of the entire architecture is given, the goal of this paper is to emphasize the more novel aspects of the NTCS architecture, which have also proved valuable in our implementations. We feel that these can be advantageously applied in any similar system. First, the NTCS addressing and naming is supported by a completely portable dynamic naming service, which is recursively built on top of the communication services it supports. It is also responsible for handling dynamic reconfiguration. Second, the underlying portable internet scheme, which supports this naming service directly, also uses it to determine routing. Third, in support of a heterogeneous environment, a simple yet effective method of inter-machine data conversion is employed, which results in no needless data conversions, and adapts dynamically to the environment as modules are relocated. Further, it does not force a particular conversion scheme; this can be determined entirely by the application. Finally, some of the difficulties associated with supporting the necessary recursion in this environment are discussed. Not only is the NTCS itself recursive, but it also uses some of the DRTS services which it supports (e.g., monitoring, time, and error logging).

1.2. Motivation

This work evolved from the development of a message-based architecture for a distributed information retrieval system testbed: the Utah Retrieval System Architecture (URSA^{tm}) project, underway since early 1983 [5]. The URSA system is based on a number of backend servers (e.g., for index lookup, searching, or retrieval of documents), handling requests from host processors or user workstations. A fundamental URSA requirement was transparent distribution across many, possibly different processors and communication networks. Implementations would consist of multiple computer systems, personal workstations, and specialized backend search hardware, while communication systems would range from local to long-haul networks. In addition, the URSA testbed requirements dictated the need to dynamically add, modify, or replace system modules, while in operation. From the NTCS perspective, the URSA requirements were: support for a message-based architecture, distribution across multiple machines and networks, portability, support for dynamic reconfiguration, and a high degree of network transparency.

Although distributed support environments of many flavors are in existence [28, 25, 2, 4, 15], the stringent portability requirements, the unpredictable target environment, and the need for dynamic

Presented at the
6th International Conference on Distributed Computing Systems
May 19-23, 1986, Cambridge MA

reconfiguration, obviates them [29]. The chosen solution was to develop our own support, in a highly portable manner, built on top of the most stable base we could find; the native IPCS of each system. The NTCS was developed explicitly for this purpose. However, software support for any distributed system involves more than simply grafting on a communication mechanism to whatever already exists [4]. Aside from the obvious issue of intermodule communication, a second, less obvious issue is the necessary distributed run-time support (DRTS). This includes such services as distributed process management, file service, time service, and monitoring. The required amount and type of such support depends largely on the application. In the limit, this comprises a distributed operating system (DOS) such as the LOCUS [17] or Apollo DOMAIN [12] systems. Thus, on top of both the NTCS and the native operating system at each machine, various DRTS services have been added as required [27, 22].

1.3. Design Concepts

The NTCS architecture is cleanly layered. This was chosen purely for design integrity and maintainability, and not for compatibility with external standards, such as the ISO [3]. Since the NTCS was viewed as an integral system, replacement of internal layers with 'off the shelf' systems was never anticipated. Also, we were somewhat insensitive to any possible layering inefficiencies, due to the loosely-coupled nature of the application.

The requirement for portability was the primary motivation behind the architecture design. Consequently, the architecture supports this with relative ease. Portability was handled in the conventional manner of writing in a largely portable language, and localizing the machine and network dependencies into a conversion layer. The C programming language [8] was chosen since it was largely portable, available on all target systems, and provided the the necessary low-level interfaces.

Another major goal of the NTCS was to provide network-transparency to the application. In this work, network-transparent implies that neither the physical location of a module nor the details of the underlying communication mechanism, need be visible at the application interface. This transparency is limited to communications only, and does not extend to the operating system level [12, 17]. Such mechanisms would be built on top of the NTCS, part of the DRTS system, much as conventional operating system functions rely on the basic IPCS.

In addition to network transparency, the communication system was required to support dynamic reconfiguration; that is, allow the replacement, removal or addition of modules while the system is in operation. This is primarily to allow module replacement and upgrade, as opposed to something like fast-response, dynamic load balancing. Like transparency, this also has limitations. If careless, conversations in progress may be adversely affected by removing a participant, or messages may be lost. Recovery from this type of failure belongs in the area of transaction management [23, 14], and not in the NTCS (Section 3.5).

The entire NTCS is based on virtual circuits. From the application viewpoint, these were appropriate since interactions among application modules would stabilize in a set of extended conversations, rather than random, unpredictable messages. Further, the effect of dynamic reconfiguration would be a minor perturbation on these conversations. In terms of underlying communication support, many existing IPCSs and network protocols provide the fundamental semantics of virtual circuits, and often much more. Thus, the virtual circuit paradigm mapped cleanly across the layers.

As for the services provided, the application level interface (described in detail in [29]) consists of three classes of primitives; the basic *communication primitives*, the *resource location primitives*, and *utilities*. The NTCS provides both asynchronous and synchronous (send/receive/reply) forms of communication primitives. While application processes refer to each other through logical names, all communication primitives are based on NTCS-assigned addresses. The application first calls on the resource location primitives to dynamically map names to addresses. An application module need only obtain an address once; module relocation will then occur as required, during all communication, transparent at this interface.

Lastly, a distributed network monitor and precision time corrector have been developed by another project member, on top of the NTCS [27]. Since the NTCS itself utilizes both of these services, recursive operation in addition to that of the naming service is observed.

2. The NTCS Architecture

2.1. Overview

The NTCS is a cleanly layered architecture, built entirely on top of the IPCS at each machine, and utilizing one or more internal support modules (processes). Each application process must bind with a passive communication module (ComMod), which is the only aspect of the NTCS visible to the application. To the application, the ComMod *is* the NTCS. This is shown in Fig. 2-1.

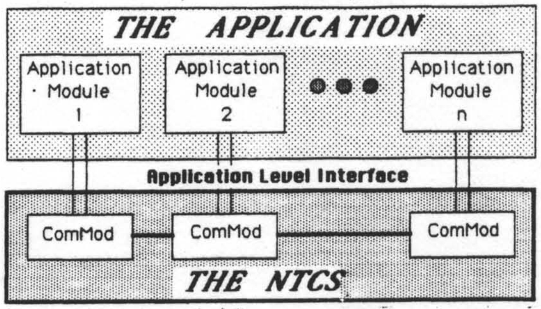

Figure 2-1: The Applications View of the NTCS

Internally, the NTCS is designed around a single communication *Nucleus*, which provides a fundamental set of protocols and access points supporting all NTCS functions. The Nucleus is bound with every NTCS module, just as the ComMod is bound with every application module. Both the ComMod and Nucleus each consist of multiple internal layers, and both are completely passive; they do not exist as separate processes. The internal NTCS layers are now described from the bottom up, with the aid of a hypothetical machine configuration in the figures.

2.2. The Nucleus Layers

The lowest layer in the NTCS is the *Network Dependent Layer* (ND-Layer) (Fig. 2-2). All machine and network communication dependencies are localized here, providing a uniform virtual circuit interface (STD-IF) for the remainder of the NTCS. Everything above the ND-Layer is portable, in terms of the communication interface. Other system level interfaces exist in separate mapping layers, or as separate subsystems of the DRTS (e.g., our process control and time service).

A simple STD-IF was desired, and since direct compatibility with external standards was not required, a custom interface was specified. This incorporates only those features necessary for the NTCS, while maintaining a high degree of compatibility with anticipated underlying IPCSs (thus reducing the porting costs). These ND-Layer *local virtual circuits* (LVCs) are limited to destinations supported directly by the local IPCS, which may be limited to processes on the same machine, or at best, among machines on the same local network. The ND-Layer is not capable of communicating between machines on networks which are not supported directly by the endpoint IPCSs. There is no automatic relocation or recovery from failed channels (except for retry on open); notification is simply passed upward. The goal was to hide the machine/network dependent details and provide a uniform interface across all systems, with minimal complexity.

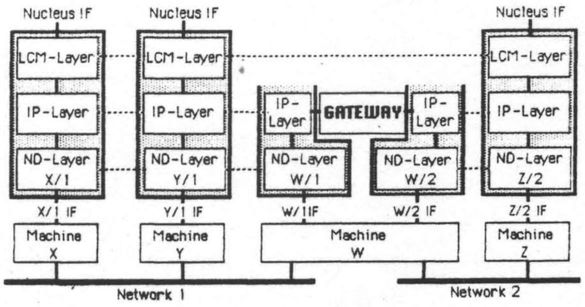

Figure 2-2: The Nucleus Internal Layering

The *Internet Protocol Layer* (IP-Layer), in conjunction with one or more Gateway modules, provides *internet virtual circuits* (IVCs) across disjoint networks and machines (Fig. 2-2). IVCs are established either as a single LVC on the local network, or as a chained set of LVCs linked through one or more Gateways as required (Section 4). Thus, above the IP-Layer, the interface as well as access all points in the system is now uniform. Since the Gateway and IP-layers exist above the ND-Layer, they are entirely portable. This not only simplified their design, but allows the *same* Gateway module to be used for all networks and machines. As with the ND-Layer, there is no automatic relocation or recovery from failed channels; notification is simply passed upward. The goal was to hide the internetting details with minimal complexity.

Support for dynamic reconfiguration is handled by the *Logical Connection Maintenance Layer* (LCM-Layer) (Fig. 2-2). Its primary function is to relocate modules which may have moved, and to recover from broken connections, though it also provides a connectionless protocol. No explicit open or close primitives are provided at the Nucleus interface; messages are simply sent/received directly to/from the desired destinations, with the underlying IVCs being established as needed.

2.3. Addressing Levels

While the Nucleus layers provide the necessary communication mechanics, the obvious remaining issue is addressing. The NTCS employs two levels of *internal addressing*, and one level of *logical naming*. At the lowest level are network-dependent *physical addresses*, such as TCP/IP 32-bit integers or Apollo MBX pathnames, over which we have no control. Above this are the internal *UAdds* (Unique ADDresses), which comprise a flat, network- and location-independent address space, forming the foundation of the NTCS. UAdds are identical to the UIDs of many file systems [11, 7, 20], and proved useful for their intrinsic location independence, the ease of passing them around, and the simplicity of handling unstructured values. Two drawbacks, the difficulty of generating them, and the problem of locating objects, are both handled by the naming service. At the top level are *logical names*. While currently limited to character strings, naming schemes can be very application dependent [24, 1, 16, 6], and through the naming service design, this can be readily changed.

2.4. The ComMod Layers

A *naming service*, built entirely on top of the Nucleus, supports all of the dynamic address and name resolution in the NTCS, providing the logical naming and the internal addressing services required by both the application and the internal Nucleus layers (Section 3). Building this on top of the Nucleus was quite advantageous. Its design is greatly simplified (being completely portable), the system integrity is enhanced, and the application level naming scheme, or the naming service implementation, can be easily altered. For all practical purposes, the naming service is nothing more than an application built on the Nucleus; however, it is also used *by* the Nucleus, forcing the Nucleus to operate recursively.

The Name Service Protocol Layer (NSP-Layer) is the single naming service access point for all layers within the ComMod (Fig. 2-3). Its purpose is to fully isolate the ComMod from the naming service implementation. Currently, the NSP-Layer communicates with a single Name Server module, which maintains the name/address database. However, other implementations are certainly possible, with no direct impact on the NTCS.

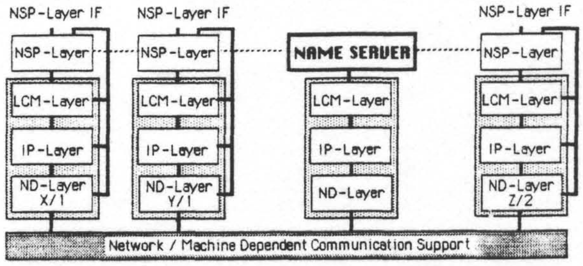

Figure 2-3: The Naming Service Protocol (NSP) Layer

The application interface primitives are provided by the Application Level Interface Layer (ALI-Layer), forming the topmost layer in the ComMod (Fig. 2-4). It simply provides the application interface primitives from the Nucleus and NSP-Layer services, tailors the error returns, and performs parameter checking. It may be better described as a thin *veneer*.

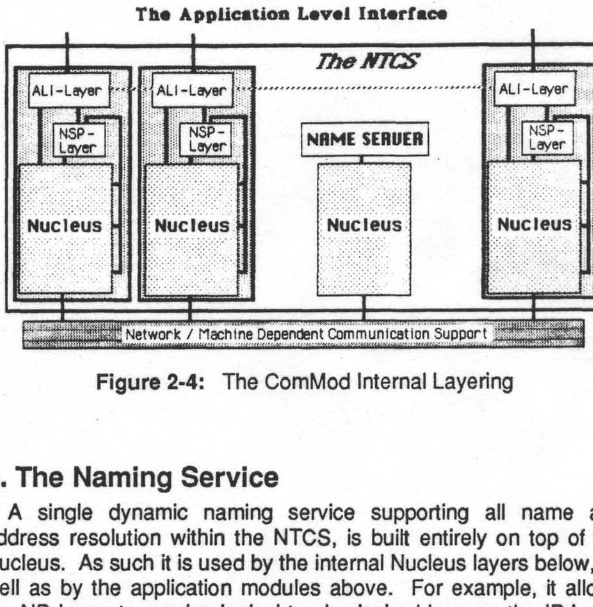

Figure 2-4: The ComMod Internal Layering

3. The Naming Service

A single dynamic naming service supporting all name and address resolution within the NTCS, is built entirely on top of the Nucleus. As such it is used by the internal Nucleus layers below, as well as by the application modules above. For example, it allows the ND-Layer to resolve logical to physical addresses, the IP-Layer to determine destination networks, the LCM-layer to determine forwarding addresses, and the ALI-Layer to map between logical names and addresses. The NSP-Layer completely isolates the implementation of this service from the rest of the NTCS.

In the current implementation, the NSP-Layer communicates with a single Name Server module, which maintains the name/address database. The database could, however, be partially distributed across two or more such modules, or fully distributed among the NSP-Layers themselves, without affecting the rest of the NTCS. This flexibility is a direct result of having built this service on top of the Nucleus, and of isolating it with the NSP-Layer.

3.1. A Recursive Design

Building the naming service on top of the Nucleus proves to be quite advantageous. First, it becomes completely portable, a feature especially desirable in replicated or distributed implementations of this service. Additionally, it eliminates internetting considerations. The NSP-layers talk across multiple networks in the identical manner as application modules do, further simplifying its design. Also, by utilizing the same communication services as all other modules, the system integrity is enhanced. Lastly, the logical naming scheme (seen to be very application-dependent), or the naming service implementation, can be changed independently of the basic communication system.

There is one difficulty with this scheme; it forces the nucleus to support recursion. Recursion, however, is also necessary for some DRTS functions (e.g., monitoring and distributed time), and the interaction can be quite interesting. Some problems with recursion are discussed in Sections 3.4 and 6.

3.2. Operation

Regardless of the particular logical naming scheme, each module's name information is registered with the naming service when that module comes on-line. At that time, the naming service generates a *UAdd* for the module, and maintains a mapping from the module name to this address. UAdds are currently generated by a simple monotonically increasing counter (in a distributed implementation, a unique Name Server identifier would be appended). Also at this time, the module creates any necessary communication resources (e.g., a TCP/IP port, or an Apollo MBX server mailbox), and informs the naming service of this, along with its logical network identifier. This *physical address* information is also maintained, uninterpreted, in the naming service, along with the module's logical name and UAdd.

Thus, module names can be resolved to UAdds, and UAdds can be resolved to the physical address (location) information necessary for communication. Through the naming service, the two major problems with UAdds have been handled: generating them, and locating the modules they refer to.

3.3. Address Resolution

Two address mappings must occur for communication to take place: one from logical name to UAdd, and the other from UAdd to the physical address. The initiator is responsible for obtaining the UAdd from the logical name. This may be an application module, one of the internal NTCS layers, or any of the DRTS modules. Application and DRTS modules use the resource location primitives while internal NTCS layers talk directly to the NSP-layer.

The ND-Layer maps from UAdd to physical address, either through the NSP-layer services, or by information exchanged between modules during the channel open protocol. This information is then locally cached for future reference. In fact, once all necessary addresses have been resolved (e.g., after the system has been heavily used for a while), the Name Server can be removed with no consequence, unless the system is reconfigured.

3.4. Temporary Addresses (TAdds)

There is one catch in the recursive naming service which occurs during initial connection with the Name Server. All communication with the Name Server is through the LCM-layer primitives, which are based entirely on UAdds. However, since all UAdds are assigned *by* the Name Server, how does the initial datacom with the Name Server take place, before the UAdd is assigned?

In fact, a more fundamental problem is that of obtaining the physical address of the Name Server, since it obviously can not provide its own, prior to connection. Gateway addresses pose a similar problem, since they may be required to reach the Name Server. Thus, a small number of 'well known' addresses are loaded into the ComMod address tables when each module is initialized; those of the Name Server and of certain 'prime' gateways. Once in operation, other (non-prime) gateways can be located through the naming service.

In terms of the UAdd problem, in order to avoid special 'initial connection' protocols, the concept of a temporary address (TAdd) was adopted. TAdds are identical to UAdds, except they are only unique *locally* to the module that assigned them. Each module assigns itself one initially, and each Nucleus layer assigns its own TAdd to each incoming connection from a TAdd source, since the source TAdd is not unique to the receiver. TAdds are handled like UAdds, except that they are replaced in local tables when the real UAdd is available. That is, upon receipt of a message from a UAdd source, if the local tables still refer to an old TAdd, this is replaced with the new UAdd. In this manner, TAdds for any given module will be purged from all layers within the first two communications with the Name Server, after which time the Name Server will be referring to the module by its real UAdd. Note, there is nothing *magical* about TAdds. They are of no use in locating objects, but simply allow all of the internal protocols to work cleanly in this special case situation. The actual connection establishment is handled through the physical addresses, which in this case happen to be 'well known'.

The use of TAdds resulted in only minor modification to the internal Nucleus connection protocols. Further, by passing them out of the Nucleus (e.g., to the Name Server, NSP-layer, and Gateway), it allowed these external protocols to work unmodified at this boundary condition. Thus, TAdds provide a way of gracefully dealing with initialization, without special protocols, and then disappear very quickly. They are not useful on a permanent basis, since they have only local significance.

3.5. Dynamic Reconfiguration

A previously resolved address may be invalid (e.g., the module was moved, or the communication link failed). An attempt to communicate with that address results in a simple address fault in the ND-Layer, which will close the channel and eventually return to the LCM-Layer. The LCM-Layer will query a local forwarding address (UAdd) table, to no avail since this just occurred, followed by an address fault handler which calls the NSP-layer to obtain a forwarding UAdd. This requires some intelligence in the naming service, first determining whether the old UAdd is really inactive, mapping the old UAdd to its name, and then looking for a similar name in a newer module. With our new attribute-based naming, this is more involved.

If a new UAdd is obtained, it is entered in the local forwarding address table, and control is returned to the calling routine. It will now find this forwarding UAdd, observe that no connection exists, and establish a connection in exactly the same manner as during an initial connection. This ability allows the system to be dynamically reconfigured, with the communication automatically reaching the correct destination.

A new UAdd may not be available for two reasons: no replacement module was located, or the original module is still alive. In the first case, the call will simply return with an error. In the second, depending on the type of call, it will attempt to reestablish what appears to be a broken communication link.

While the NTCS, can not lose messages in a static environment, they can be dropped due to the nature of dynamic reconfiguration. However, even if the NTCS could guarantee that no messages were lost *due to itself* (e.g., with a modified sliding window protocol), problems could still occur. The NTCS could not recover if a failed module had buffered messages internally, or was involved in incomplete transactions. The first case would necessitate a module-level recovery mechanism, while the latter would require some type of transaction roll-back procedure. Both of these can be viewed as part of a transaction management system [23, 14]. Only in rare cases would the NTCS recovery be of use. Since any more complex case would require a transaction management system (also capable of handling the rare cases), the NTCS recovery would be largely redundant. Given that NTCS recovery would be insufficient in most cases, and that the higher level recovery must eventually be built, there was no great motivation to build such recovery into the NTCS [29]. To date, this decision has posed no problems. This problem of redundant recovery mechanisms appears to be common in layered designs [21].

4. Portable Internet Support

The IP-Layer, in conjunction with one or more Gateway modules, provides (IVCs) across disjoint networks, either as a single LVC on the local network, or as a chained set of LVCs linked through one or more Gateways. Internetting was viewed as integral to the NTCS. That is, unlike the naming service built on top of the nucleus, internetting was built directly into the nucleus. It is precisely for this reason that the naming service *could* be built on top.

4.1. A Recursive Design

While Gateways exist below, and *support* the naming service, their logical name and connected networks are *registered with* the naming service; the same as any application module. In this manner, the internet scheme is simplified. The same resource location protocols already available from the NSP-Layer, are used in determining the appropriate Gateway(s) through which to establish communication links, by both the IP-layer and the Gateways themselves.

Further, the Gateway and IP-layers are both entirely portable. This not only simplified their design, but allows the *same* Gateway module to be used for all networks and machines. The ability for each Gateway module to communicate with different networks is handled by the independent ComMods with which it binds (Fig. 2-2). Each ComMod is bound with an ND-Layer designed for one of the networks. Thus, no network-dependent issues are visible within the Gateway.

4.2. The Approach

The virtual circuit-based internet scheme was dictated by both the application and lower layer NTCS requirements. The only advantage of a packet switched approach would be to handle very frequent module relocation, or reliability under extreme failure conditions, neither of which was envisioned. Our solution combines ideas from both centralized and decentralized internet schemes [26, 18].

The compromise was to decentralize the circuit rooting and establishment, while centralizing the topological information in the naming service. Primarily, this eliminates any complex inter-gateway protocols; in fact, *no* inter-gateway communication ever takes place. Establishment of circuits at each point in the system proceeds autonomously. Further, this eliminates all of the problems of database distribution. The centralized topology was tolerable since this information is only required at circuit establishment time, which is relatively rare, and locally cached values will likely be correct since reconfiguration is infrequent. Such would not be the case if each packet were routed individually (e.g., the ARPAnet).

4.3. Dynamic Reconfiguration

Dynamic relocation in the internet environment is handled by simply aborting the IVC; the LCM-Layer handles it from there. Module death is detected by the ND-layer in any connected module and the physical channel is closed. When the IP-layer attempts to use the associated LVC, an error will result. It will then close down the LVC and the associated IVC. If this is an originating module, the error is passed up to the LCM-layer, where a new connection (or relocation) will be attempted. If this is a Gateway, the error will be passed out of the IP-layer to the Gateway. The Gateway will instruct the IP-layer on the other side of the link to close the associated IVC. This results in the closing of the lower LVC, which will be detected by the ND-layer of the connected module. This process continues until the originating module is eventually reached, at which time the error will be passed up to the LCM-layer, where a new connection (or relocation) will be attempted. While messages may get lost in Gateway queues during this reconfiguration, for all practical purposes, this is indistinguishable from the issues already discussed due to dynamic reconfiguration.

5. Data Conversions

Data type conversions may be necessary [25, 10, 19] when moving data among different machines. For example, the byte ordering of long integers differs between the VAX and the Sun systems. We chose a relatively simple solution which has worked well; it results in no needless conversions, and adapts dynamically to the environment as modules are relocated. Furthermore, the transport format is not dictated by the NTCS; it is determined entirely by the application.

Messages between identical machines are simply byte-copied (image mode) while those between incompatible machines are transmitted in a converted representation (packed mode). The NTCS determines the correct mode based on the source and destination machine types, thus avoiding needless conversions. Placing data conversions at the upper communication layers (e.g., the OSI Model Presentation Level) made little sense. While the highest layer (the application) is responsible for providing the conversion routines, the decision to apply them is left to the lowest layers, where the destination machine type is visible.

5.1. Image and Packed Modes

In all cases, the original application message must consist of a contiguous block of memory (e.g., linked lists are not allowed). In *image mode*, a byte-copy of the memory image is simply deposited at the destination. In *packed mode*, the NTCS applies conversion functions at each end, while transporting the message as a simple byte stream. Each application module provides these conversion functions to pack/unpack its messages into/from a standard byte-stream transport format.

The particular transport format is of no consequence, as long as it is based on a packed message; it can be entirely application dependent. For example, embedding structure information in the byte-stream, defining it in a 'header', or simply inferring it through a message 'type', are all viable options.

A character representation transport format was chosen for the current implementation, purely for simplicity. The NTCS guarantees correct character representation across machines (reasonable since most all are the same). Although the characters used are machine *dependent*, the pack/unpack functions are built with language constructs which are machine representation independent (e.g., sprintf or sscanf in C), or by custom routines (such as NTCS shift mode, below). Standard problems with byte orderings do not arise, since the message is viewed as a byte stream. One member of the URSA project implemented an automatic code generating mechanism which builds these pack/unpack routines directly from the message structure definitions [22].

5.2. Shift Mode

For internal message headers, a mode efficient enough to be used for *all* transfers, regardless of destination, was desired. Character conversion was viewed as excessive overhead, and results in undesirable variable length (or worst-case-long) messages. In *shift mode*, all message headers are built with structures of four byte integers, which can be bit field divided as required. Any necessary data field in an NTCS control message is built in packed mode. Since these data fields are relatively rare, this conversion overhead is not bothersome. Message header information is transferred by byte shifting each header integer sequentially into the final message, using standard high level shift and mask routines. The remainder of the message, in packed or image format, is transferred directly as a byte stream. At the destination, the shift mode bytes are shifted back into the header integers. Byte ordering problems are hidden by the high level shift/mask routines, and by transmitting the values as a byte stream.

6. Recursion in the NTCS

Building the NTCS and DRTS support services on top of the Nucleus introduced problems due to recursion. While not bad for the traditional reason of speed (recursive calls are rare under normal operation), it posed difficulties with debugging and exception handling, largely due to its non-deterministic nature. It also forced the notion of TAdds (Section 3.4).

6.1. A Scenario

The amount of recursion occurring within the NTCS may not be obvious; the following simplified scenario applies to sending a message to a destination for the first time, with monitoring and time correction enabled.

As the application level Send is initiated, control passes to the LCM-layer, which generates a time stamp for monitor data. A distributed time primitive is called, which *may* recursively call on the ComMod to communicate with its support module. If this is the first such communication, it will call the resource location primitives to locate the module, invoking the ComMod recursively again. Once resolved, it will send the message, at which time the entire process we are describing will occur recursively for *that* send (time correction and monitoring are disabled here, to avoid the obvious infinite recursion). As a further complication, a time correction may involve multiple messages to multiple modules. With the time stamp now generated, the *original* send passes to the IP-layer. This contacts the naming service for network resolution, invoking the NSP-layer recursively again. Once complete, the ND-layer opens the channel and sends the message. Upon success, the LCM-layer sends data to the monitor by calling itself. If this is the first such communication, the monitor is first located, and the connection established, as with the time service above. Control then returns to the application.

6.2. Debugging

Outside of the conventional problems of distributed debugging [13], recursion added considerably to the difficulties. It became difficult to simply *understand* what the system was doing at run-time, as the above scenario attempted to demonstrate. The NTCS and DRTS functions often call on each other in unpredictable ways (e.g., time service data communication only occurs periodically, and Name Server calls depend on the local NTCS state), resulting in non-deterministic, multiple-level recursion. Furthermore, we have ignored the housekeeping which must occur every time the passive Nucleus is called. None of this is deterministic from the programmer's point of view. Further, one is usually looking at two or more such modules, concurrently.

Thus, the need for large amounts of debug information became apparent. For example, simple tracebacks are largely inadequate. One must also know *why* a layer is being called, and *who* is calling it. However, adequate *selectivity* in observing this information is equally important. We have not yet devised an adequate mechanism for dealing with this problem.

6.3. Exception Handling

The NTCS (like any communication system), quickly became inundated with the handling of unlikely exceptional conditions, attempting to gracefully recover from unexpected situations. Most of these are not errors, but are simply due to the non-deterministic nature of this type of system; at any point in time, one can be certain of very little.

One negative side effect of recovering from these conditions is that the better the system is at it, the less one may know about how it is actually running. Logic, or even coding errors, may be covered up by layers of relentless exception handlers. While a running table of errors could be maintained and monitored, the clean layering and recursive nature of this system complicate the matter.

In a cleanly layered system, a given layer is usually unaware of why it was called, or which layer called it, and has little knowledge of the external state (in addition to the classic difficulty of assessing the *distributed* state [9]). Thus, it is difficult for it to decide whether a particular exception is, or is not, an error. In many cases, such conditions are reasonable all but a small fraction of the time.

Recursion within the NTCS only complicates this problem by adding to the unknowns. As the previous scenario demonstrated, a given layer can be called from above or below, often while it is in the middle of some other action. The following experience is offered as an example of the difficulty.

The virtual circuit between a module and the Name Server may break. During the next Name Server communication, the following will occur. The ND-Layer will detect the failed circuit, and clear it out. Control will be returned to the LCM-Layer, where the forwarding address table will be checked, to no avail. This results in an LCM-Layer address trap, which automatically queries the NSP-Layer for the status of this UAdd. This, of course, attempts to talk to the Name Server, which ends up at the ND-Layer *where this all started.* It will see the dead circuit, and recursively run through this whole thing until either the stack overflows, or the connection can be reestablished with the Name Server, whichever occurs first. In operation, both were observed.

Since layers below the NSP-Layer know nothing of the Name Server, they are unable to stop this problem. The only layer with explicit knowledge of the Name Server, the NSP-Layer, knows nothing of connections or forwarding addresses. This problem was eventually patched in the LCM-Layer address fault handler, although it also should not know of the Name Server. As an interesting note, the exception which caused this address trap is reasonable *in all cases but this one.*

7. Results

The current NTCS implementation has demonstrated the feasibility of our original design concepts, in a number of large, working systems. The ability to design the entire NTCS around a single portable Nucleus, and that in turn, around clean internal layers, both simplified the development and added to its integrity. As a result, both the naming service and internet support could then be developed in a completely portable fashion, reducing their development time and allowing them to easily distribute. In addition, the complete separation of the naming service from the Nucleus has been quite advantageous. Both the naming scheme and the naming service implementation are currently being replaced, with minimal impact on the rest of the system. The former will be attribute-value based, while the latter will be replicated for failure resiliency. Lastly, the inter-machine data conversion scheme has worked well, eliminating needless conversions and dynamically adapting to the environment as modules are relocated.

However, while the layered architecture has proven its utility, we largely underestimated the difficulty of designing the system *initially* in this manner; it added considerably to the development time. In a similar sense, the recursive nature of this architecture is a double edged issue; while it simplified the design of many support modules (both internal to the NTCS (the naming service and internet scheme), and external (monitoring and time service), it added considerably to the already difficult task of debugging and exception handling. From our experiences, however, the benefits of both of these approaches have far outweighed the problems.

From the application viewpoint, the NTCS has proved to be a reasonable mechanism for developing this class of distributed application. It has been successfully employed in three generations of distributed information retrieval systems, during its development over the last two years. In addition, the concept of a distributed run-time support (DRTS) system, built on top of the NTCS, has been effective method of providing distributed system level support. It does, however, add to the recursion complexity, since the NTCS must also use some of these services.

8. Conclusion

The NTCS has provided a reasonable foundation for this class of distributed application. It has been successfully employed in three generations of distributed information retrieval systems, during its development over the last two years. The current implementation has demonstrated the feasibility of a number of novel design concepts, while also clearly demonstrating where additional work is required. It has provided both a working communication system for the URSA development, as well as a flexible framework for future research.

Acknowledgments

This work would not have been possible without the support of the entire URSA project staff, and the project director, Lee Hollaar.

References

[1] Birrell, A., Levin, R., Needham, R.M., Schroeder, M.D.
Grapevine: An Exercise in Distributed Computing.
Communications of the ACM 25(4):260-74, April, 1982.

[2] Cheriton, D.
The V Kernel: a Software Base for Distributed Systems.
I.E.E.E. Software 1(2):19-42, April, 1984.

[3] Day, J.D., Zimmermann, H.
The OSI Reference Model.
Proceedings of the IEEE 71(12):1334-40, December, 1983.

[4] Gammage, N., Casey, L.
XMS: A Rendezvous-Based Distributed System Software Architecture.
Software 2(3):9-19, May, 1985.

[5] Hollaar, L.A.
A Testbed for Information Retrieval Research: The Utah Retrieval System Architecture.
In *Eighth Annual International ACM SIGIR Conference.* ACM, June, 1985.

[6] International Federation of Information Processing Societies.
A User-friendly Naming Convention for Use in Communication Networks.
Working Paper, IFIP Working Group 6.5, International Computer Message Systems, March, 1984.
Version 3.

[7] Kernighan, B.W., Pike, R.
Software Series: The UNIX Programming Environment.
Prentice-Hall, 1984.

[8] Kernighan, B.W., Ritchie D.M.
Software Series: The C Programming Language.
Prentice-Hall, 1978.

[9] Lampson, B.W. (ed.).
Lecture Notes in Computer Science 105: Distributed Systems - Architecture and Implementation.
Springer-Verlag, 1981.

[10] Lantz, K.A., Gradischnig, K.D., Feldman, J.A., Rashid, R.F.
Rochester's Intelligent Gateway.
Computer 15(10):54-68, October, 1982.

[11] Leach, P.J., Stumpf, B.L., Hamilton, J.A., Levine, P.H.
UIDs as Internal Names in a Distributed File System.
In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 34-41. ACM, Ottawa, Canada, August, 1982.

[12] Leach, P.J., Levine, P.H., Douros, B.P., Hamilton, J.A., Nelson, D.L., Stumpf, B.L.
The Architecture of an Integrated Local Network.
I.E.E.E. Journal on Selected Areas in Communications SAC-1(5):842-56, November, 1983.

[13] Leblanc, R.
Event-Driven Monitoring of Distributed Programs.
In *Proc. 5th International Conference on Distributed Computing Systems*, pages 515-522. IEEE Computer Society, 1985.

[14] Liskov, B.
On Linguistic Support for Distributed Programs.
I.E.E.E. Transactions on Software Engineering SE-8(3):203-10, May, 1982.

[15] McDonald, W.
A Flexible Distributed Testbed for Real-Time Applications.
Computer 15(10):25-39, October, 1982.

[16] Oppen, D.C., Dalal, Y.K.
The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment.
ACM Transactions on Office Information Systems 1(3):230-253, July, 1983.

[17] Popek, G., et al.
LOCUS: A Network Transparent, High Reliability Distributed System.
Proc. 8th ACM Symp. on Operating Systems Principles :169-177, 1981.

[18] Postel, J.
Internet Protocol, DARPA Internet Program Protocol Specification.
RFC 791, University of Southern California, Information Sciences Institute, 1981.

[19] Rashid, R.F.
An Inter-Process Communication Facility for UNIX.
Technical Report CMU-CS-80-124, Department of Comupter Science, Carnegie-Mellon University, 1980.

[20] Redell, D., et al.
Pilot: An Operating System for a Personal Computer.
Communications of the ACM 23(2):81-91, February, 1980.

[21] Saltzer, J.H., et al.
End-To-End Arguments in System Design.
ACM Transactions on Computer Systems 2(4):277-288, November, 1984.

[22] Schlegel, D.
A Portable Window Manager for Message-Based Systems.
Master's thesis, University of Utah, 1985.

[23] Spector, A.Z., et al.
Support for Distributed Transactions in the TABS Prototype.
Technical Report Carnegie-Mellon University-CS-84-132, Carneige Mellon University, July, 1984.

[24] Su, Z., Postel, J.
The Domain Naming Convention for Internet User Applications.
RFC 819, SRI International, and University of Southern California, Information Sciences Institute, 1982.

[25] Sun Microsystems, Inc.
Sun Network File System Documentation.
November, 1984.
DRAFT: contains RPC and XDR specifications.

[26] Tymes, L.
Routing and Flow Control in TYMNET.
I.E.E.E. Transactions on Communications COM-29(4):87-93, April, 1981.

[27] Wang, K.
Performance Monitoring and Projection for an Information Retrieval System.
Master's thesis, University of Utah, 1985.

[28] Xerox Corporation.
Courier: The Remote Procedure Call Protocol.
Technical Report XSIS-038112, Xerox System and Integration Standard, Stamford Conn., December, 1981.

[29] Zeleznik, M.P.
A Portable Network-Transparent Communication System.
Master's thesis, University of Utah, 1985.