Design for a New Untyped IPC for Mach

Franklin Reynolds, Franco Travostino, Rod MacDonald,
David Ellistion, Keith Loepere

OSF Research Institute
One Cambridge Center
Cambridge, MA 02142

September, 1993

Abstract

Mach has enjoyed considerable success as a vehicle for Operating Systems research and development. This success has led to the development of different versions of Mach. OSF in collaboration with ARPA has embarked on a effort to unify the micro-kernel's interface and to make an implementation of this interface available to the Mach community. In this paper we discuss a series of changes made to Mach IPC. These changes include the simplification of the the message format, additional copy semantics for Out-Of-Line regions, an extensible message trailer and a new MIG. This collection of changes is referred to as "untyped IPC".

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Material Command (AFMC).

Introduction

Mach has enjoyed wide spread success within the research community and industry. A variety of different research and commercial organizations have used Mach as a basis of their operating system research and development projects. This has resulted in a certain amount of inevitable divergence between the versions of Mach used by the different organizations. In the first quarter of 1992, an effort to capture the best work done by these different groups and to coordinate the continued evolution of Mach was begun by OSF and ARPA. The goal was to provide a kernel interface specification that unified the work being done in the areas of realtime, distribution and security. A draft of the new kernel interface document is available from OSF and has been circulated within the Mach user community [1]. The "kernel interface unification" project is funded by OSF and ARPA and is being conducted in close collaboration with the Mach research community and OSF's member companies.

One of the projects undertaken by OSF as part of the kernel interface unification effort is the design and implementation of several changes and enhancements to Mach IPC. These modifications, collectively referred to as "untyped IPC", include the following:

- replacement of self describing message data with untyped message data
- new copy semantics for out-of-line regions to improve server performance and security and provide initial support for real-time applications
- a security token for Manditory Access Control (MAC) and Discretionary Access Control (DAC)
- extensible trailers for kernel supplied data and a new version of the Mach Interface Generator (MIG).

Untyped IPC is intended to perform as well or better than typed IPC depending on the features being used. While the changes to the message format preclude backwards binary compatibility, another goal of the project is to preserve MIG source compatibility. This paper discusses the design, implementation and rationale of these new features.

Untyped Messages

Mach IPC provides network transparent communication between heterogeneous nodes, e.g. different architectures. The previous version of Mach IPC (typed IPC) supported communication between heterogeneous nodes by the use of a self describing message format [2].

A typed message consists of a message header and a body. The body is contiguous with the header and contains a list of parameters which comprise the user-specified data. The header, which has a fixed size and format, contains the destination and reply ports, various flags, the message size, a sequence number and a user-controlled field. The collection of flags is used to indicate message properties such as simple or complex message formats and what types of port rights are being passed as destination and reply ports. The message size is the length of the header and the body. The sequence number is generated by the kernel when the message is received. It is used by multi-threaded applications such as an external pager to track the order that messages are delivered. The user field is typically used by interface generators. For instance, MIG uses the destination port to identify the target object and the user field to identify the target method to be invoked on the target object.

Each parameter in a typed message includes type and length information. Types include integers, floats and arrays of integers as well as kernel-managed types such as ports and out-of-line regions. Since each parameter is self-describing, kernel-managed types can appear in any portion of the message body. The kernel parses and makes any necessary data transformations. Remote IPC is implemented by interposing a task, the netmsgserver, on all remote ports. This task has the responsibility for sending and receiving messages across network boundaries. Any data transformations that must take place between network nodes is the responsibility of the netmsgserver. In general, extending IPC semantics (network communication, group or multicast communication, etc.) is accomplished by interposing other tasks on any ports that need to exhibit the new behaviors. The kernel and each of these servers must separately implement the data encoding and transformation rules.

The new, untyped IPC service does not use self-describing data. A message consists of a message header, an optional array of kernel-managed data descriptors,the message body and the message trailer. The untyped message header is almost identical to the typed message header. This allowed the reuse of a significant amount of the typed IPC implementation. The one difference is that the sequence number has been moved to the trailer and its slot in the header has been reserved for future use. Port rights, Out-Of-Line (OOL) regions and OOL arrays of port rights are passed in the descriptor array. The kernel does not understand the format of the message body. It does not parse or transform data in the message body. It is the responsibility of the end-to-end layers, such as the new untyped MIG, to provide data encoding and any necessary transformation. The trailer is used by the kernel to append additional data to messages. This kernel-generated data is sometimes called implicit data to differentiate it from the data that is explicitly sent by the sending task in the header, descriptor array or message body.

Data representation and transformation is best treated as an end-to-end problem. Intervening layers, protocols or kernel participation are not necessary for supporting heterogeneous networks for a micro-kernel. OSF Distributed Computing Environment (DCE) RPC and other heterogeneous communication systems do not use underlying kernel support to deal with data encoding and transformation. DCE RPC uses the Network Data Representation (NDR) to encode data in messages. The untyped MIG uses NDR to encode its messages. This is a first step towards our goal of eventually integrating Mach and DCE.

Rather than requiring support for encoding arbitrary user specified data in the kernel and the netmsgserver, we have consolidated the responsibility for data encoding and transformation in MIG. This both simplifies maintenance and future improvements.

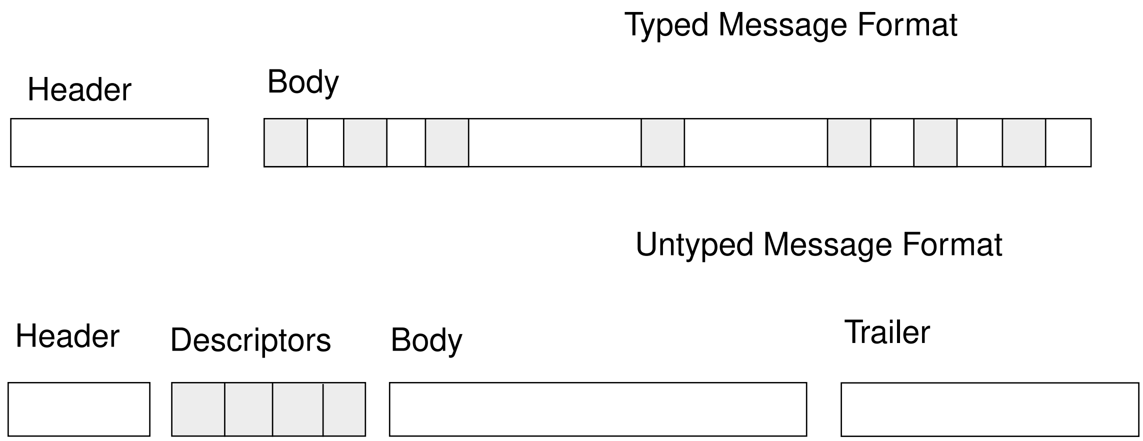

figure 1.

The data encoding used in typed IPC is sufficient describe the data needed by the kernel interfaces on most architectures. However, it does not support architectures or applications with unusual formats such as Cray floating point format or EBCDIC characters. Marshaling and unmarshaling these kinds of data require additional work by the runtime environments by both the sender and the receiver of a message.

Both the typed and the untyped kernels exports trap-based interfaces for local services and message-based interfaces for location independent services. Both kernels use MIG to manage the message-based interfaces it exports. Rather than sharing the responsibility between tools, servers and the kernel, the untyped kernel consolidates the responsibility for handling data encoding and transformation in one tool (MIG). Experiments and tests can be easily performed without modification to the kernel. Maintanence and enhancement of the data encoding algorithms are significantly simplified.

Other stub generators can be used and are currently under development by different organizations. To facilitate the development of other stub generators, the MIG protocol has been published in a separate technical report [12]. Stub generators that understand this protocol can interoperate with Untyped IPC tasks and the kernel.

Typed IPC has good local performance [3]; however, as has been observed by other researchers [4], the format of typed messages introduces some unnecessary overhead. The costs of marshaling and parsing self describing messages are unnecessary in the common case when communication is between two tasks on the same machine or two machines of the same type.

Another expense is incurred when ports or OOL regions are passed in a message. After the kernel or the netmsgserver copy the bits associated with the message into a buffer, the buffer must be scanned again to look for kernel data types. It does not help to place the kernel-managed types at the front of the message because there is no way to know how many types there are in a message without examining the entire message. The format of untyped messages defines a count and a well known location for all kernel-managed parameters. This simplifies and streamlines the processing of complex messages by the kernel and MIG.

Copy Semantics

Changes have been made to the copy semantics of OOL regions to enhance the real-time characteristics, performance and security of servers. A physical copy option has been added to message-send to complement the virtual copy semantics that already exist. Scatter lists for OOL regions have been added to message-receive. This allows a task to receive or overwrite OOL regions into buffers that are already allocated.

The typed IPC OOL regions are copied in units of entire pages. The untyped service allows the sender to specify precisely which bytes will be copied.

Physical Copy

Physical Copy is a new copy attribute associated with sending an OOL region. An OOL region is specified in a message using an OOL descriptor in the array of kernel descriptors. A bit in each descriptor is used to indicate whether physical or virtual copy semantics are desired. Virtual copy descriptors and physical copy descriptors can be specified in a single message. Virtual copy semantics allow the lazy evaluation of the copy. The pages backing the OOL region are shared between the sender and receiver. The OOL regions in the sender and the receiver are marked copy-on-write (COW). The data is not actually copied until the sender or the receiver writes into the OOL region. This is the behavior of Mach 3.0 OOL regions.

In addition to virtual copy, the new IPC service supports the new physical copy semantics. Unlike virtual copy, physical copy copies the OOL data into the kernel at send time. While physical copy has better performance than virtual copy for some common kinds of messages which will be discussed later, it is primarily a real-time feature. When an OOL region is virtually copied it is marked copy-on-write. This is done even if the page is wired. Even in the case where the receiver takes the first COW fault and consequently copies the region the sender's OOL region remains marked for COW faults. If the sender subsequently writes into its OOL region it will take the COW fault, discover that the copy has already been made and return. This is not cheap. This sort of behavior is hard to predict and complicates the programming of real-time applications. It is also a violation of POSIX wired memory semantics.

To avoid these problems, the physical copy of an OOL region eagerly evaluates the copy. The cost of sending a region is relatively easy to understand and predict. The sender's region is not marked copy-on-write. Subsequent action or lack of action by the receiver does not interfere with the sender.

Overwrite and Scatter Lists

Typed IPC has scatter/gather support. A list of OOL regions can be specified at the time a message is sent. The kernel will *gather* the OOL regions and send them with the messages. When the message is received, the list of OOL regions is *scattered* into unallocated memory in the receiving task's address space. The receiving task has no control over the placement of the OOL regions. This can be a serious problem for the receiving task. Heaps, stacks, mapped file regions or other dynamic regions of memory have to take into account the possibility that the kernel may allocate random areas of memory to receive OOL regions.

Untyped IPC has a method for a receiving task to specify a scatter list. The scatter list contains the address and length of a list of buffers or regions of allocated memory. Any OOL regions that are received in a message are written into the buffers in the scatter list. The previous contents of the buffer are *overwritten*. Each OOL region is copied into a different buffer in the scatter list. This allows the preservation of alignment and the structure of complex structures sent in OOL regions. Scatter lists, by allowing OOL regions to overwrite currently allocated memory, have performance advantages that will be discussed later.

Physical copy and overwrite are orthogonal, though complementary, features of untyped IPC. The kernel attempts to reduce actual copies whenever possible by the use of VM mechanisms and hints, such as copy-on-write, detection of page aligned transfers and the deallocate option

Physical copy is a property of the message being sent. It indicates to the kernel that the specified, OOL regions should be eagerly copied from the sender to the kernel. If the deallocate bit is also specified the kernel may be able to avoid the copy by removing the page from the sender. Overwrite is used to receive a message containing OOL regions into explicitly named buffers, thus avoiding the complications associated with receiving OOL regions into dynamically allocated buffers. As indicated in **figure 2.**, messages can be sent and received using any combination of physical copy and overwrite.

Send \ Receive	Overwrite, Align	Overwrite, NotAlign	Dynamic allocation
Physical Copy			
Phy,deallocate			
Virtual Copy			
Vir,deallocate			

Figure 2.

The new interface to IPC defines separate send and receive message buffers. Typed IPC uses the same-buffer for sending and receiving a message. The act of receiving a message writes over the send buffer. A separate receive header and buffer are provided by untyped IPC. This separates the side effects of receiving a message from sending a message and allows multiple send operations to use the same send header. The receive buffer is used to provide the scatter list.

Granularity of Out-Of-Line Region Copies

The untyped service moves arbitrary amounts of OOL data without restrictions associated with data alignment or page boundaries. The OOL region specifies exactly the bytes that are copied. The previous IPC service moved OOL data in page-sized chunks. This has some unfortunate performance and security characteristics. Consider what happens when a small OOL region that straddles a page boundary is sent in a message. In addition to the specified OOL region, the entire page associated with the beginning of the region and the entire page associated with the end or the region will be sent.

Clearly, the previous implementation of OOL regions are an expensive way to move small amounts of data, but it is also a potential source of attacks on secure servers. A secure server, such as a file server or an authentication server, would have to take special care when allocating and placing data structures that might be copied via OOL regions. It would be a serious flaw for a file server or authentication server to return information to a user that the user was not entitled to read.

Performance

We were interested in improving the performance of certain types of messages and preserving the level of performance for the rest. The use of physical copy and overwrite can be combined to provide eager copy behavior. Eager copy of OOL regions can provide better performance than inline data or virtual copy of OOL data for some types of applications. In-line data, which usually involves marshaling and unmarshaling copies as well as a copy into the kernel and a copy out of the kernel, is typically used for small messages. The virtual copy semantics of OOL regions are optimized for the case where large amounts of data sent but the COW faults and subsequent copies do not occur. Our experience building operating system servers suggests that regardless of the size of the message, data is copied more often than not. The effectiveness of lazy copy optimizations is limited in applications that touch or change the data in the OOL region.

A virtual copy requires the sender's data to be marked copy-on-write and data structures in the VM subsystem necessary to support page sharing and copying have to be set up. When the message is received, a new region of memory is allocated - but not mapped. The first time a page in the new OOL region is referenced a read fault occurs - this maps the shared page. The first time either the sender or the receiver writes to the shared page a COW fault and copy of the entire page occurs. Once the receiver is finished with the OOL region it must consider deallocating the buffer to avoid the undue fragmentation or exhaustion of its address space. The sender will trip a COW fault the next time it writes to the OOL region regardless of whether the receiver tripped a COW fault and made a copy of the pages. It the copy has already been made then the fault is processed quickly, otherwise this will trigger a full copy. The performance of lazy evaluating the copy of the OOL region varies among architectures but it is never cheap.

The cost of lazily evaluating copies versus using physical copy and overwrite to force eager copies varies with different architectures. In some cases, the VM architecture is such that lazy copies are only faster than eager copies if the receiver can avoid most of the read faults and all of the write faults. In our experience, even those processors that have relatively friendly VM designs don't show an advantage unless several pages of data are sent. The cost of setting up the COW structures and executing the copy via the VM fault path are significant costs. Since the virtual copy semantics of OOL regions dynamically allocate random memory, additional work must be done to manage the task's address space.

Consider, as an example, a simple network server that implements read and write operations. When data is sent to the server to be transmitted over the network, protocol headers may have to be added or trailers appended to the message. The message may be too large to be sent in a single transfer across the network interface, resulting in the message being fragmented. Headers may have to be associated with each fragment. A checksum may have to be calculated and added to header. Perhaps the message will have to be encrypted. If the protocol is reliable, it may not be possible to deallocate the message data until some future time because it may have to be retransmitted.

If the message was sent to the server as an OOL region, the pages associated with it were marked copy on write. No copy of the pages is made until the sender or the receiver write into the OOL regions that share the COW protected pages. The OOL region is delivered to the server by allocating new space. If it is important for the protocol header to be contiguous with the message, either new memory in front of the message will have to be allocated to contain the header (which may not be possible if the OOL region was allocated adjacent to memory that was already allocated - the common case), or the message will have to be copied to another location that has room for the message and the header. Allocating memory for the header is not cheap if it requires a system call (vm_allocate) or a page fault (to grow the heap for malloc). After the space for the header and the message is allocated, the copy of the protocol header begins. If the memory was allocated then the first write will cause a zero-fill fault which causes a physical page to be allocated and filled with zeros. After all of this, the copy of the protocol header can proceed.

If the message must be directly manipulated for any reason (message fragmentation or encryption), then the copy on write fault will trip, causing the VM processing and the copy of the data. The COW costs will be incurred for each page, rather than once for the entire OOL region, because the VM subsystem attempts to defer any copies as long as possible. It has no way of knowing if the entire OOL region will eventually be copied and consequently can not optimize for that case.

Once the message is safely transmitted across the network, the OOL memory that was associated with the message must be deallocated (otherwise the address space of the task will eventually be consumed). Only after all this is done can the server signal the completion of the RPC. Regardless of whether the server was able to process the message without tripping over the COW fault and causing the bytes to be copied, the sending pages are still marked for COW. The next time those pages are written, a COW fault will occur.

Care was taken to add these new features without impacting the performance of other types of messages. Micro benchmarks and system tests (AIM and SPEC) show the untyped IPC service and MIG to be as fast or faster than the typed service. The one exception is transfer of non-page aligned buffers using virtual copy and dynamic allocation. Untyped IPC only transfers the data specified whereas the typed service forces OOL transfers to be page aligned. As a result, the typed service can avoid some zero-fill page faults. This was a decision we made in order to simplify the constructions of secure servers. Note that in many cases, the new OOL semantics out-perform the old OOL semantics despite the cost of the zero-fill page faults.

Special Kernel Interfaces

Two special kernel interfaces were modified to take advantage of overwrite semantics: **vm_read_overwrite**() and **device_read_overwrite**(). The **vm_read**() interface allows task A to read a specified portion of the address space of task B. The **vm_read()** function returns a dynamically allocated OOL region containing a virtual copy of the region read from task B's address space. The VM costs associated with virtual copy and dynamic allocation make the use of **vm_read()** for small to medium sized amounts of data expensive. Since the OOL regions are dynamically allocated with each use of **vm_read()**, a server must arrange for those regions to be eventually deallocated or risk running out of address space. The **vm_read_overwrite**() kernel service permit servers to read relatively small amounts of data, such as user stacks, without incurring the costs of page size copies. Because the results of the **vm_read_overwrite**() operation write into an existing buffer rather than allocating a new OOL region, it is not necessary for the server to garbage collect the OOL regions.

The **device_read**() interface returns its results in dynamically allocated OOL regions. **The device_read_overwrite()** kernel service is used to place data read from a device, such as a disk, precisely where it is desired. The **device_read_overwrite**() function has advantages similar to those of **vm_read_overwrite**(). The size of disk operations does not always correspond to the size of a page. Ethernet interfaces, other network interfaces, the buffer cache, byte streams and other data structures associated with I/O are not necessarily well suited to data presented in page-sized chunks. This lack of page granularity diminishes the value of OOL region virtual copy and dynamic allocation semantics. The OSF/1 server uses both of the new kernel services to improve I/O performance.

Security Token

Different groups have begun to seriously study the problem of building a secure system that could pass a B3 [5] evaluation using Mach. It has become clear that more implicit data containing security related information is necessary. Groups at IBM, Trusted Information Systems and OSF have observed that trusted servers need to be able to determine the identity of the task that sent a message in order to implement Mandatory Access Control (MAC) and Discretionary Access Control (DAC) [6]. There is no way to securely ascertain the identity of the sending task in a system using typed IPC. This is an intentional characteristic of the system and is part of the design that permits arbitrary interposition on ports and the implementation of remote IPC via the netmsgserver.

In the new system, each task has a security token. When a task is created, it inherits its token from its parent. Only a trusted task can change the security token of a task. The sending task's security token is associated with each message it sends. Messages sent by the kernel, such as notifications, have the kernel's security token. While not used by the kernel to implement security policies, this token can be used by the servers to implement MAC and DAC policies. It is necessary that the kernel provide the identity of the sender because the sender cannot be trusted to accurately identify itself. Tasks that have sufficient authority to interpose on communication channels, i.e. ports, between tasks and secure servers have the authority to masquerade as other tasks. The netmsgserver is an example of such a trusted server. This ability to masquerade permits trusted tasks to interpose on ports and forward the identity/authentication of the original sending task.

Trailers

The last new feature added to messages is an extensible trailer for implicit data. Implicit data is information added to a message by the kernel rather than data sent by the sending task. An example of implicit data in the Mach 3.0 typed IPC service is the sequence number (msgh_seqno) in the header. The sequence number indicates the order in which messages are dequeued from a port. These numbers are generated by the kernel and provided to the receiver in the message header. They were implemented in an early version of 3.0 by Richard Draves [3] to simplify the construction of multi-threaded external pagers. Pagers communicate via asynchronous messages, as opposed to RPCs, and need to process the messages in order. The field that was used for the sequence number had previously been msgh_kind and was used as a hint to servers such as the netmsgserver. The hint was intended to be used to indicate that the message was part of an RPC, etc. Rather than add a field to the message and break all existing programs it as decided to change the meaning of the msgh_kind field. At that time, this was the only example of implicit data and even though it was rarely used it made sense to use a field in the header.

Since then, we have discovered the need for other types of implicit data. The security token is an example of implicit data and other types of implicit data are anticipated. Real-time projects at OSF and other institutions are beginning to investigate time constraint propagation with RPCs [7,8,9]. The security token and the time constraint are examples of new types of implicit data that are necessary to solve new problem domains. There were no more fields to cannibalize in the message header. Rather than a series of ad hoc and potentially incompatible message format changes, we created an extensible message trailer for implicit data.

The message trailer immediately follows the message. It begins on a naturally aligned boundary. It contains a small header and any implicit data associated with the message. Currently there are two instances of implicit data: the security token and the message sequence number. The sequence number has been moved from the header to the trailer. The field in the header that had been used as the sequence number is reserved for future use.

The trailer is always generated by the kernel and the trailer is used exclusively for implicit data. It contains no information explicitly sent by a task. The one exception is the case where a trusted task masquerades as a different task. It is possible for a trusted task, such as the netmsgserver, to receive a message and forward it, including the original trailer, to another port.

The receiving task can control how much of the trailer is actually delivered. In this way a task need not contend with implicit data it is not interested in or prepared to deal with. This mechanism also permits future backwards compatibility. Programs need not anticipate what new kinds of implicit data will be added by future kernels. The task can tell the kernel exactly what kinds of implicit data it is willing to accept. It is possible to ask for empty trailers, i.e., trailers that do not have any implicit data. Of course, the kernel returns an error if the task asks for a type of implicit data that does not exist.

Each trailer begins with a header that contains type and size fields. The type field indicates the format of the trailer and the size indicates the size of the trailer. The message header also contains a size field (msgh_size). The message size field is the size of the message header plus array of descriptors plus the message body. It does not include the message trailer. This separates the processing of explicit data that is sent by a task from implicit data that is attached to the message by the kernel. In this way we reserve compatibility with programs that receive a message in a buffer and then, without modification, attempt to send it.

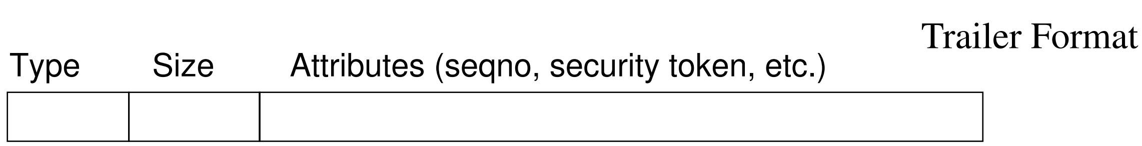

Figure 3.

MIG

A new version of MIG was necessary to support all of the changes to the format of Mach messages and the semantics of IPC. The MIG interface language was extended to expose the new copy semantics, trailers and complex types. The new format simplifies the parsing of messages. Changes have been made to speed up operation dispatch and to support dispatching to multiple subsystems. Optional message logging and profiling support has been added to aid in the debugging of clients and servers. Except for a handful of obsolete and unused features that have been removed, backwards source compatibility with the earlier MIG is preserved.

A new technical report documents the MIG Protocol [11]. This document describes in detail the rules MIG uses for formatting and parsing messages. This information can be used by other stub generators to interoperate with MIG programs. This is invaluable information because the kernel interfaces are defined by MIG.

Untyped Support

With the elimination of self describing data and the kernel's ability to perform data transformations, MIG must provide data encoding and transformation services. We have elected to use the Network Data Representation (NDR) and receiver-makes-right techniques from OSF's Distributed Computing Environment (DCE) [10] for encoding and transforming data.

The message header, the array of descriptors and the trailer have defined formats that are understood by the kernel. It is the responsibility of the kernel to perform any necessary data transformations. If the message is empty, nothing need be done; otherwise, MIG uses enhanced version of NDR to encode the body of the message. This is done by placing a 64 bit header at the beginning of the message that describes the data format used by the sender.

NDR Header

```
typedef struct {
            unsigned8  mig_vers;     /* version of MIG */
            unsigned8  if_vers;      /* version of the target interface */
            unsigned8  reserved1;
            unsigned8  migencoding; /* encoding rules being used */
            unsigned8  int_rep;      /* big endian or little endian */
            unsigned8  char_rep;     /* ASCII or EBCDIC */
            unsigned8  float_rep;    /* IEEE or Cray floating point */
            unsigned8  reserved2;
} ndr_format_t;
```

figure 4.

The first four octets are used by MIG to determine the version of MIG and the target interface. These enhancements are intended to help detect and control interface version skew as systems evolve. The last four octets are the standard NDR values for encoding data. When a message is received the header is examined to determine which interface is the target of the message. Once the target interface is identified and it isdetermined that the version of the interface expected by the sender is compatible with receiver's version, the stub can use the information about the parameters the stub is expecting and the NDR record in the message to perform any necessary data transformations.

Language Extensions

MIG stubs are generated from interfaces defined in the MIG Interface Description Language. This language was extended to support the new copy semantics of OOL regions and variable arrays of variable arrays. Some obsolete components of the grammar have been removed. Except for the elimination of the obsolete, features the new MIG Interface Language is backwards compatible with the typed MIG. Old programs can use the new untyped IPC and MIG by being recompiled and in most cases no changes are necessary. The new grammar is described in the Mach 3 Server Writer's Guide [13].

The obsolete features are:

- The ability to create new fundamental types has been removed. Types must be constructed from a set of predefined built-in types.
- Polymorphic types can only be used for different types of port rights.
- The Servercopy, IsLong and IsNotLong keywords have been eliminated.

Debugging Support

MIG has new features to aid in the debugging of new programs. An optional per interface version number has been added. This allows the stubs to dynamically detect interface version skew.

A build option will generate stubs that have additional instrumentation code. This code makes detailed error logging possible. It is also possible to gather traffic metrics on a per interface, per stub or per application basis. Support for this information gathering exists in both the user and the kernel stubs.

Project Status

The first implementations of untyped IPC and MIG are complete and are part of the OSF version of the micro-kernel. This technology is available to member companies. It is available as part of the Research Institute OSF/1 single server technology release, OSF MK 5.0. This technology is being adopted by OSF Engineering as the basis for the next general release of OSF/1. The kernel and sever documentation has been updated to include untyped IPC. The technical reports describing untyped IPC, the protocol of the new MIG [11] and a conversion guide [12] are available.

The performance of untyped IPC is comparable with typed IPC but additional performance improvements are possible. The new copy semantics make greater use of the in-kernel, per processor IPC buffer cache. In the near future we hope to experiment with the IPC buffer cache and some improvements to the VM support for IPC.

Though the techniques used to support heterogeneous computing are well understood and widely used by DCE, the implementation in MIG has not been tested. This is because currently neither NORMA IPC nor a Mach 3.0 netmsgserver was available and usable in our heterogeneous networking environment. Work is currently underway to convert both NORMA and the University of Arizona's netmsgserver to use untyped IPC and MIG.

The changes to the format of Mach messages are not backwards binary compatible with old Mach programs. However, except for a few insignificant exceptions, MIG Interface Description Language compatibility has been preserved. Programs that use MIG can be recompiled, usually with no change, and run on the new system. Programs that use raw IPC must be changed. Our experience is that this is a simple procedure.

Acknowledgements

Many people and organizations played a role in defining the new kernel interface; however, the authors would like to thank the following people for their efforts on the new IPC service: Brian Bershad, Marc Auslander, Ken Birman, Larry Peterson, Sean O'Malley, Homayoon Tajalli and our colleagues at OSF. Special thanks to Richard Draves and his colleagues at CMU who created the 3.0 IPC service which served as the basis for this work.

Bibliography

[1] "OSF Mach Final Draft Kernel Interfaces"; Keith Loepere; OSF publication, May 1993

[2] "Mach Kernel Interfaces"; Keith Loepere; OSF publication, January 1992

[3] "A Revised IPC Interface"; Richard Draves; Usenix Mach Workshop, Autumn 1990

[4] "A Fast and General Implementation of MachIPC in a Network"; Hilarie Orman, Edwin Menze III, Sean OMalley, Larry Peterson; Proceedings of the USENIX Mach III Symposium, April 1993

[5] Department of Defense Trusted Computer System Evaluation Criteria, DoD 500.28-STD

[6] "Trusted Mach TM Philosopy of Protection"; NIST Document No: TMACH/93-014

[7] "OSF* Real-Time Mach: Initial Technology Distribution"; Douglas M. Wells, Jack A. Test, Franklin D. Reynolds; OSF Member's Meeting, November 1992

[8] "Threads: A Programming Construct For Reliable Real-Time Distributed Computing"; F. D. Reynolds, J. D. Northcutt, E.D. Jensen, R.K. Clark, S.E. Shipman, B. Dasarathy, and D.P. Maynard; International Journal of Mini and Microcomputers, Vol 12, No.3, 1990

[9] "Experiments with Real-Time Servers in Real-Time Mach"; Tatsuo Nakajima, Takuro Kitayama, Hideyuki Tokuda; USENIX Mach Symposium Proceedings, April 1993

[10] "Network Computing Architecture"; M. Kong; version 2.0, Cooperative Computing Division - Hewlett/Packard Company, 1991

[11] MIG protocol tech report; Franco Travostino

[12] “Untyped MIG: What Has Changed and Migration Guide”; Franco Travostino

[13] “Mach 3 Server’s Writer’s Guide”; Keith Loepere