The Neutral Message Language: A Model and Method for Message Passing in Heterogeneous Environments*

William P. Shackleford, Frederick M. Proctor, and John L. Michaloski
National Institute of Standards and Technology

ABSTRACT

To achieve efficient communication between distributed real-time processes, it is desirable to both choose the best protocol for each communication path and limit variation to improve software portability. These divergent goals can be satisfied through the use of a uniform application programming interface (API) that hides the details of specific protocols from programmers. The Neutral Message Language (NML) is a uniform API to communication functions that includes many popular protocols: interprocess shared memory; interprocessor backplane global memory; and Internet networking. NML implements a mailbox model for communication, with both queued- and non-queued access, blocking- and non-blocking reads and writes, and multiple readers and writers. NML provides language bindings for both C++ and Java. The protocol parameters are contained in configuration files that are read at run time, so that a system's allocation of processes to processors can be deferred as late as desired and modified dynamically. NML handles mutual exclusion for data integrity, and converts between native machine format and neutral data encoding when necessary.

NML has been used in a variety of applications, from manufacturing to vehicle control. This paper details how NML works; how NML applications are built, tested, and fielded; and compares NML with other communication mechanisms.

KEYWORDS: communication, real-time, distributed control, network, C++, Java, shared memory

---

* No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied. Certain commercial equipment, instruments, or materials are identified in this report in order to facilitate understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

INTRODUCTION

Background

Achieving efficient communication between real-time processes in a distributed environment is a difficult task. The difficulty arises due to the tradeoff between performance and the degree of distribution. A system designer is faced with two alternatives. One can select a single communication protocol supported throughout the distributed system, and pay a performance penalty over what could be provided by faster protocols that may exist for specific communication paths. Alternatively, one can select the individual best protocol for each communication path, and pay a software development penalty when system components are moved between dissimilar platforms.

The software development penalty can be significantly reduced through the use of a uniform application programming interface (API) to communication functions. The use of an API hides the details of specific protocols from the programmers, with a resulting increase in portability. Additional cost is incurred at the beginning in order to implement the API for various protocols, but this is a one-time cost for each protocol. Defining a uniform API is complicated by the need to decide which features of each protocol to include. Limiting the API to only those features that are common to all protocols results in a specification that may be too limited. Attempting to define options that include each feature in each protocol eliminates the benefit of uniformity originally intended. A useful API is one that strikes a balance between rigid adherence to common functionality and the flexibility required to realize efficient communication.

The Neutral Message Language (NML) is a uniform API to communication functions that includes many popular protocols: interprocess shared memory; backplane global memory; and UDP and TCP/IP networking. NML implements a mailbox model for communication, with both queued and non-queued access, blocking- and non-blocking reads and writes, and multiple readers and writers. NML provides language bindings for both C++ and Java. The protocol parameters are contained in configuration files that are read at run time, so that a system's allocation of processes to processors can be deferred as late as desired and modified dynamically. NML handles mutual exclusion for data integrity, and converts between native machine format and neutral data encoding when necessary.

Application Domain

NML originated as a concept for a neutral manufacturing language in the U. S. Air Force’s Next Generation Controller (NGC) program, which developed a specification for APIs to open architecture machine tool controllers [1]. The NGC specification was never formally standardized, but led to several implementations based on its communication API. One of these implementations formed the basis for communication between cyclic control processes in the Real-time Control Systems (RCS) Architecture [2]. The processes in such systems typically execute at fixed periods in the millisecond or greater range, forming a hierarchy of supervisors and subordinates in which cycle times are longer for processes at the top. These processes do not fit naturally into a client/server model and are usually designed so that they can execute in parallel.

NML forms the basis for a larger library of software and development tools used to build RCS applications [3]. It has been applied to controllers for a wide range of applications, including robotic welding, machining, inspection, vehicle control, and process control.

MODES OF COMMUNICATION

NML implements a mailbox model of communication, in which messages of varying size are written into buffers of a fixed maximum size. Messages contain only data, not executable programs. The sender and receiver of an NML message need to agree ahead of time on the format of the message and what behavior should result from its transmission. There is no restriction on what messages can mean, but typically they are either commands, status, or world model information. Messages typically range in size from a few bytes to hundreds of kilobytes, although there is no limit to message size other than practical memory limits.

NML also includes message queues, blocking communications, and a query-reply service.

Queued Vs. Non-Queued

In RCS applications, status and world model information is understood to be most recent information available, and therefore these buffers are not queued. Each new message erases any previous message, even if it were never read. The same is true for commands. Only the most recent command is active, and any previous commands are obsolete. Acknowledgement of command receipt by subordinates through their status ensures that commands are not lost.

However, there are circumstances in which queued buffers are desirable. This includes logged data or error messages. For a queued buffer, each write appends a message to the queue until the buffer is full. The writing process can detect a full buffer and choose to hold its message internally until the queue empties, drop the message, or clear the buffer and erase the unread messages. Each time a message is read it is removed from the queue, so it is not normally desirable to have multiple readers of a queued buffer.

Blocking Vs. Non-Blocking

Blocking a process on a read (or write) operation until data has been received (or sent) is an efficient alternative to constantly polling a channel until the operation is completed. When a blocking operation is performed, the operating system is informed that it should put the calling process in an efficient wait state using minimal system resources until either the operation has completed or a timeout period has passed.

RCS applications continually monitor supervisor commands, subordinate status, sensor inputs, and internal states based on periodic timer events. This precludes blocking on the arrival of a message, which would delay control an arbitrary amount of time. However, blocking communication is suitable for processes, like data loggers or message displays, whose only purpose is to handle data written to a channel. Alternatively, a control process could be split into several parallel threads of execution, one of which handles communication. Because of the general usefulness of blocking communication, it is supported by NML.

Some limitations apply to the use of blocking communication. It is not possible to simultaneously block on read or write operations on multiple NML buffers, as is possible on Unix systems, for example, with the `select()` system call. On platforms whose underlying operating systems do not support blocking, the effect is simulated through polling. This eliminates the advantage of blocking communication, but results in the same behavior across platforms.

Query-Reply Service

Two-way communication with NML is easily achieved using two separate buffers, one for each reading/writing relationship. NML will not coordinate the messages in any way. If coordination is desired, the NML query-reply service may be used. This is an additional software layer atop a pair of normal NML buffers that allows one process to post a query and wait for the corresponding reply.

APPLICATION PROGRAMMING INTERFACE (API)

We have found that it is very important to design the application interface for each language separately in order to take advantage of the strengths and mitigate the weaknesses of each language.

Interoperability between languages is accomplished within the implementation of each language's API and is not particularly served by efforts to make the APIs identical. In each case however, the API was dramatically simplified by the use of configuration files that specify all the protocol and system specific information.

The C++ Language API

C++ allows for very flexible and efficient use of pointers. However, exception handling was not universally supported when the API was developed, and correctly allocating and freeing memory can be difficult for beginners. The details of the API are outside the scope of this paper, and fully documented in [4]. Here is a list of primary functions, which includes creation, reading, writing, checking status, and update functions that act on specific data types:

```
   /* Create NML channel */
   NML::NML(NML_FORMAT_PTR f_ptr, char * buf, char *proc, char *file);

/* Read 1 message  non-blocking */
   NMLTYPE NML::read();
/* Wait for and read one message. */
   NMLTYPE NML::blocking_read(double timeout);
/* Read message  non-blocking but don't mark the message as was read.*/
   NMLTYPE NML::peek();
/* Get the address where messages are stored */
   NMLmsg *NML::get_address();
/* Write one message. (Use reference) */
   int NML::write(NMLmsg &nml_msg);
/* Write one message.  (Use pointer) */
   int NML::write(NMLmsg nml_msg);
```

```
/* Write the message only if last one was read. (Use reference) */
   int NML::write_if_read(NMLmsg &nml_msg);
/* Write the message only if last one was read. */
   int NML::write_if_read(NMLmsg nml_msg);
/* Check if last message was read.*/
   int NML::check_if_read();
/* Delete all messages and reset buffer state. */
   int NML::clear();
/* Format message in ASCII text string  for debug/display */
    const char * NML::msg2str(NMLmsg *);
/* Update functions are used by CodeGen tool to convert data between
different processor architecture formats. */
   void CMS::update(char &);  /* convert char */
   void CMS::update(short &); /* convert short  */
   void CMS::update(int &); /* convert int */
   void CMS::update(long &); /* convert long */
   void CMS::update(float &); /* convert float */
   void CMS::update(double &); /*convert double */
/* start server */
   void run_nml_servers();
```

The Java Language API

Java is not as efficient as C++, and there are no true pointers; however, exception handling and run-time type checking are widely used. As with the C++ API, a complete description of the functions is beyond this scope. They are fully documented in [5]. Like C++, they are encapsulated in a class, in this case with Java syntax:

```
   public class NMLConnection {
      public NMLmsg peek() throws NMLException
      public NMLmsg read() throws NMLException
      public int write(NMLmsg );
   };

   public class NMLFormatConverter {
      char update(char); throws NMLexception
         short update(short ); throws NMLexception
         int update(int); throws NMLexception
   };
```

COMPARISONS WITH OTHER COMMUNICATIONS SOFTWARE

CORBA

The Common Object Request Broker Architecture (CORBA) is one of many standards for distributed object computing created by the Object Management Group (OMG) [6]. CORBA defines the “backbone” that supports communications between distributed objects. CORBA enables client objects to send requests to and receive responses from server objects. CORBA automates many common network programming tasks such as object registration, location, and activation; framing and error handling; parameter marshalling and demarshalling; and operation dispatching. CORBA adopts source-level interoperability by having a CORBA Interface Definition Language (IDL) compiler

transform object interfaces defined with IDL into target programming language class definitions.

NML is designed to operate in a distributed real-time environment. By contrast, CORBA works well in a totally distributed environment where applications communicate across a network, but it fails to achieve success in a hard real-time environment. Further, CORBA has no standard to support local object interaction; however, most vendors implement this feature in a non-standard manner. The local object support omission also prevents the exploiting of symmetric multiprocessing power. Another problem with CORBA has been the lack of integrated development and testing tools to assist in server development, which makes the development process quite difficult. The NML package offers an integrated interface design tool and run-time communication monitor to measure communication traffic.

CORBA originally was unsuitable for real-time communication. This led to the creation of a OMG Special Interest Group to develop Realtime CORBA comprising: preemptive priority scheduling, control over CORBA resources for end-to-end predictibility, and flexible communications. Recently, several products have been released that comply to this standard. Although inherently real-time, the products implementation of hard-real-time vary since CORBA has no standard for pluggable tranports.

COM/DCOM

The Component Object Model (COM) is a Microsoft product integrated into all Windows operating systems that defines a framework for creating and using components that is ubiquitous in the PC marketplace. COM, like CORBA, defines many additional object life cycle services. COM is a client-server technology that allows objects to transparently interact, be it within a single process or across process and machine boundaries. COM enables object interaction by specifying that the only way to manipulate data associated with an object is through an interface on the object. COM, like CORBA, defines interfaces using IDL. COM, unlike CORBA, offers “local” distributed object interaction, component interoperability at the binary level and independence from the source interface definition [7].

NML supports hard real-time communication. COM performance depends on the linkage between the client and server. COM has a “zero” sacrifice for in-process performance and works very well in a symmetric multiprocessing environment. Distributed COM (DCOM) applies to across-platform communication. The DCOM wire-protocol is based on DCE RPC and is primarily network bound. Although not hard-real time, DCOM only adds a 35% overhead over raw TCP/IP performance (2 ms roundtrip time for TCP/IP) [8]. However, COM does support a pluggable transport layer so it is possible to realize hard-real-time assuming the underlying network supports it. NML provides a communication protocol using global backplane memory shared by a bus-connector, which enables real-time inter-platform communication.

NML is open-source and freely available from NIST. COM, on the other hand, is a proprietary Microsoft product found on any Microsoft Windows platform. COM is available as a commercial product for most other major operating systems, but not on any non-Microsoft real-time operating systems. Full source code for COM is licensable from The Open Group (formerly OSF and X/Open). Porting COM to a new real-time environment would be a major undertaking. Porting NML would not be nearly as arduous as it is Real-time POSIX-compliant [9].

CONCLUSIONS

NML is an efficient and flexible mechanism of communications, suitable for embedded real-time applications and the specification of standard interfaces. The use of configuration files allows users to optimize and select the most efficient underlying communication protocol without limiting the platforms on which applications can be developed.

REFERENCES

1. National Center for Manufacturing Sciences, “Next Generation Controller (NGC) Specifications for an Open System Architecture Controller (SOSAS),” Revision 2.0, August 1994.
2. Albus, J.S., Meystel, A.M., “A Reference Model Architecture for Design and Implementation of Intelligent Control in Large and Complex Systems,” International Journal of Intelligent Control and Systems, Vol. 1, No. 1, pp. 15-30.
3. Mathew L. Moore, Veysel Gazi, Kevin M. Passino, Will P. Shackleford, and Frederick M. Proctor, “Complex Control System Design and Implementation Using the NIST RCS Software Library,” IEEE Control Systems, Volume 19, Number 6, pp. 12-28, December 1999.
4. Shackleford, W.P., “The NML C++ Programmer's Guide,” www.isd.mel.nist.gov/projects/rcs_lib/NMLcpp.html.
5. Shackleford, W.P., “The NML Java Programmer's Guide,” www.isd.mel.nist.gov/projects/rcs_lib/NMLjava.html.
6. Common Object Request Broker Architecture, Object Management Group, Framingham, MA, 1995.
7. COM Specification, Microsoft Corporation, Redmond WA, 1995.
8. DCOM Technical Overview, Microsoft Corporation, Redmond WA, 1996
9. POSIX 1003.1b Real Time Extensions, IEEE, 1993.