3rd Int. Conf. Accel. Large Exp. Phys. Control Syst. ICALEPCS1991, Tsukuba, Japan JACoW Publishing
ISBN: 978-3-95450-254-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS1991-S08NC04

Synchronous Message-Based Communication For Distributed Heterogeneous Systems

N. Wilkinson
TRIUMF
4004 Wesbrook Mall
Vancouver, B.C
Canada V6T 2A3

D. Dohan
SSC
2550 Beckleymeade Avenue
Dallas, Texas
USA 75237

Abstract

The use of a synchronous, message-based real-time operating system (Unison) as the basis of transparent inter-process and inter-processor communication over VME-bus is described. The implementation of a synchronous, message-based protocol for network communication between heterogeneous systems is discussed. In particular, the design and implementation of a message-based session layer over a virtual circuit transport layer protocol using UDP/IP is described. Inter-process communication is achieved via a message-based semantic which is portable by virtue of its ease of implementation in other operating system environments. Protocol performance for network communication among heterogeneous architectures is presented, including VMS, Unix, Mach and Unison.

1 Introduction

The use of domain-driven object modeling techniques in the specification of the KAON Factory Control System[1] was in contrast to the more traditional emphasis on implementation and technology details and the consequent imposition of the technology on the requirements. When these object-oriented methods were used to model the KAON Factory Control System and to allocate the requirements specification to processor units[2], a *logical* architecture was derived which consisted of a network of distributed processors connected by two specialized communications buses: the *control* bus, a fast communication link responsible for the deterministic transport of control information and the *data* bus, a wide bandwidth link responsible for non-deterministic, data-intensive communication.

Since no generic processor platform can economically perform all of the functions required, the distributed network of computing platforms utilized by the KAON Factory Central Control System (KF CCS) will be non-homogeneous and will undoubtedly consist of both real-time and non real-time platforms. It is therefore important that a consistent software architecture be employed to implement communication among these platforms.

2 Message Based Architecture

The message-based semantic is a candidate for implementing the transparent, high performance inter-process and inter-processor communication required for the KF CCS. *Message passing* can elegantly encapsulate both task synchronization and data transfer into a small set of simple primitives having well-defined semantics[3]. Since a message header can identify a particular *method* to be used by a task *instance*, the use of the message-based semantic provides a convenient means of implementing an object-oriented architecture in a distributed environment.

In the *synchronous* message-based semantic, three primitives are employed for inter-process communication and synchronization: *send()*, *receive()* and *reply()*.

The *send* primitive implements the dispatch of a message to a destination task followed by the receipt of a reply from that task. Once it has called the *send* primitive, a task remains blocked until the receipt of a reply from the destination task.

The *receive* primitive is used to receive messages from other tasks and, typically, to wait for signals from interrupt service routines. Tasks which use the *receive* primitive remain blocked until a message or signal arrives, or until a user-determined timeout occurs. The use of the *receive* primitive for the receipt of both messages and signals at a single point of execution considerably simplifies the structure and design of tasks which must simultaneously deal with external events and communicate with other tasks.

The *reply* primitive is non-blocking. It is used by tasks, such as servers, which cannot block while dispatching a message. A *reply* is always made to a task which has used the *send* primitive to send a message to a given task and is waiting for that task to reply.

Task synchronization is implicit in the communications primitives employed. For instance, a *server* object should never block when posting a message to another task. For this reason, servers employ the *receive* primitive to receive messages from clients and the *reply* primitive to respond to clients. *Courier* objects are used to carry messages between servers, as a server would block if it employed the *send* primitive to communicate directly with another server.

A synchronous message-based semantic may be constructed from the more primitive inter-process communications semantics offered by operating systems such as VMS or VxWorks, or it may be obtained as the native semantic

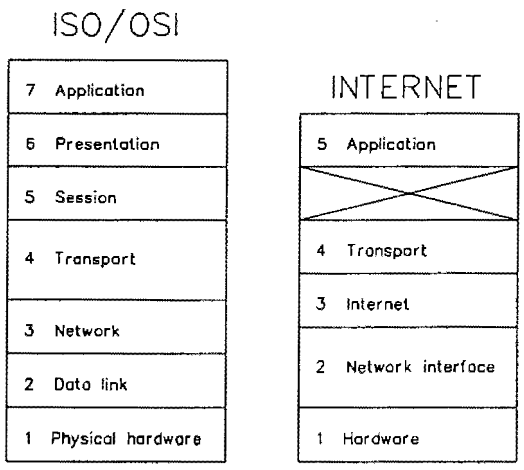

Figure 1: Protocol Stacks

of message-based operating systems, such as the Harmony real-time kernel. For instance, the Unison real-time operating system from Multiprocessor Toolsmiths implements the synchronous message-based semantic using the native semantics of the Reliant kernel, a commercial variant of Harmony, or as a messaging layer added to the pSOS+ real-time kernel.

3 Network Architecture

Common to all modern network protocols is the separation of *logical* elements of the protocol into several *layers*. The five layer Internet protocol stack and the seven layer ISO/OSI protocol stack are illustrated in Figure 1.

Proponents of the Internet suite of protocols, which has been widely implemented in the Unix and other domains, claim that it is the de-facto network standard. In the realm of networked real-time operating systems, the Internet protocol suite is almost exclusively used.

3.1 Potential Implementations

Ethernet, the IBM token ring and emerging technologies such as the Fibre Distributed Data Interface (FDDI), are prime candidates for implementation of *control* and/or *data* bus architectures. FDDI, with its high bandwidth and deterministic response, could perform both *control* and *data* bus functions. In some situations, provided that its bandwidth and non-deterministic response is acceptable, ethernet could be used for both functions at a considerably lower cost.

3.2 Transport Layer Issues

The Internet *virtual circuit* transport layer protocol, TCP, employs a *sliding window* protocol. When used for real-time control on an unreliable network, this protocol may exacerbate non-deterministic response due to the retransmission of unnecessary data segments. In addition, since a sliding window protocol transmits a number of segments before waiting for an acknowledge, unacceptable time delays may be introduced in waiting for urgent control information to be acknowledged. For this reason, the more rudimentary UDP transport protocol, which implements the unreliable transfer of datagrams, is preferable. However, the use of UDP requires additional transport layer functionality in order to ensure the reliable transfer of datagrams.

3.3 Session/Presentation Layer Issues

As shown on Figure 1, the Internet protocol stack has no session layer. For the reasons described in Section 2, a session layer employing a synchronous message-based semantic is considered desirable.

The Internet protocol stack also lacks a presentation layer. For a message-based session layer, the critical presentation layer service is one which ensures that the representation of data is the same regardless of the platform employed. The use of such a service ensures that a distributed network of heterogeneous processors can understand each other's messages. The *External Data Representation* (XDR), which has been designated RFB1014 by the ARPA Network Information Center, provides the required presentation layer functionality.

4 The Socket Server

During the KAON Factory Project Definition Study, a session layer protocol for network communication was developed which implements network communication using a synchronous, message-based semantic. In addition, a reliable virtual circuit transport layer which does not employ a multiple segment sliding window protocol was developed over UDP. This protocol suite is known collectively as the *socket server*[4] and consists of three co-operating objects which communicate via a synchronous message-based semantic.

The session layer of the socket server is embodied in the *socket_server* task and a reliable virtual circuit transport layer is embodied in multiple paired instances of *socket_task* and *recvfrom_task*. Socket server clients use the *ss_open* primitive to open a network connection and the *ss_close* primitive to close a network connection. Once a connection is open, socket server clients communicate with each other via the synchronous, message-passing primitives *ss_send*, *ss_receive* and *ss_reply*.

The socket server has been successfully implemented on Unison, VMS, Mach and SunOS[1] platforms. Ethernet message-passing performance between a Unison platform and VMS, NeXt/Mach and SunOS platforms has been measured. Socket server ports to RISC platforms are forthcoming.

5 Performance Evaluation

The hardware configuration of Figure 2 has been used to evaluate socket server performance. The transceiver fanout was used to connect two or more systems for performance measurements. The VMEbus Unison platform consisted of a 25 MHz Motorola 68030 processor board and a

[1] Sun 3/60

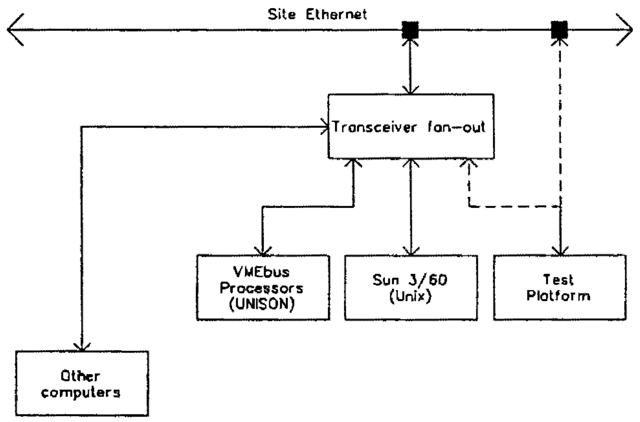

Figure 2: Equipment Block Diagram

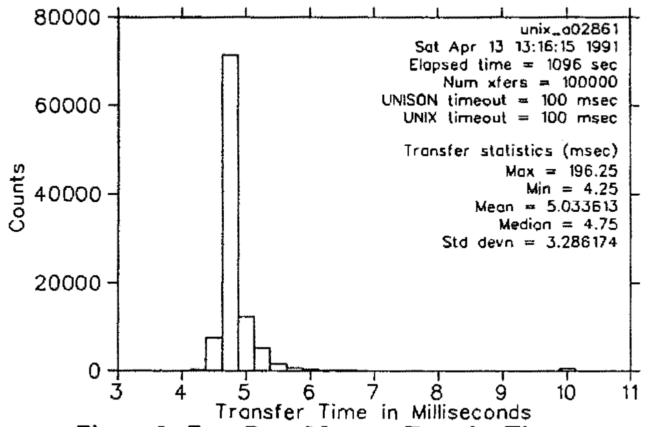

Figure 3: Four Byte Message Transfer Times

low-cost ethernet processor equipped with a 10 MHz Motorola 68010 processor. The Sun 3/60 served as the software development host for the Unison system and was the first test platform for the socket server performance.

Round-trip message transfer times were measured by the Unison client, which communicated with the non-Unison client using the ss_send primitive. The Unison client also employed ss_send to periodically transfer the acquired data to the non-Unison client for storage in a file. The one-way message transfer time was taken as half of the round-trip time. Figure 3 illustrates a typical histogram of one-way message transfer times obtained during performance tests for a Sun 3/60.[2]

For the purpose of performance measurement, only the Unison platform used the native synchronous messaging primitives for inter-process communication among the socket server tasks. The VMS, Mach and SunOS platforms were evaluated using a single-threaded variant of the socket server which emulated message passing using message copying and function calls.

[2]The 10 millisecond bin of Figure 3 contains the sum of all counts for which the one-way message transfer time was 10 milliseconds or longer.

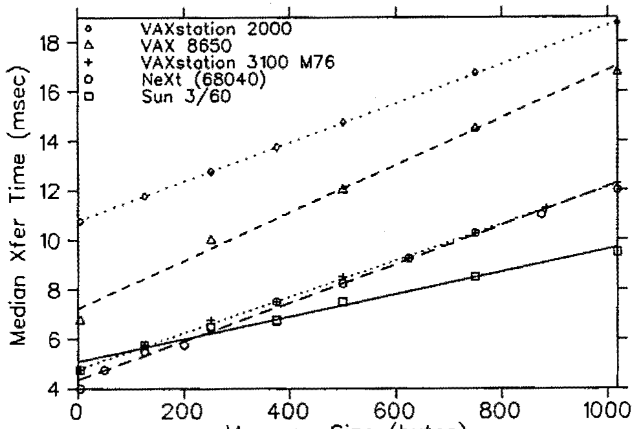

Figure 4: Socket Server Performance

6 Results

Figure 4 presents graphs of message size versus median transfer time for each configuration tested.[3] For each set of points, the data was fitted to a line, which is also shown on Figure 4. Since the Unison platform was common to all measurements, Figure 4 shows the relative performance of the non-Unison platforms tested. Figure 4 indicates that the NeXt platform obtained the best performance for small message sizes. However, for message sizes greater than about 200 bytes, the Sun 3/60 performance was superior to all others. The VAXstation 3100 M76 and the NeXt platform were quite similar in performance, despite the large difference in their costs.

As only the round-trip transfer time was measured, it is impossible to separate the Unison and non-Unison protocol overheads. However, the round-trip transfer time was recently measured between the Sun 3/60 platform and a high performance Unison platform using a single 25 MHz Motorola 68030 processor with on-board ethernet. The median transfer time between these platforms showed an improvement of only 0.75 milliseconds as compared to the transfer time measured between the Sun 3/60 and the two-processor Unison platform which was employed for the rest of the tests. This suggests that the low performance ethernet processor in the two-processor Unison platform does not make a significant contribution to protocol overheads. The Unison overheads will be established by a forthcoming measurement of the round-trip transfer times between two identical Unison platforms.

Table 1 presents the median message transfer rates for each platform tested. The NeXt achieved the highest message transfer rate of 250 messages/second for 4 byte messages. An ethernet implementation of the *control* bus would be limited to this maximum message transfer rate for present day CISC processors. However, a performance improvement is anticipated upon completion of the forthcoming socket server port to RISC platforms

[3]The median transfer time has been chosen as it provides a better indication of the peak on the corresponding histogram.

Platform	Data transfer rate (messages/second)	
	4 byte messages	1018 byte messages
VAXstation 2000	93	53
VAX 8650	148	60
VAXstation 3100 M76	211	82
NeXt	250	83
Sun 3/60	211	105

Table 1: Median Message Transfer Rates

Platform	Data transfer rate (kbytes/second)	
	4 byte messages	1018 byte messages
VAXstation 2000	0.36	53
VAX 8650	0.58	59
VAXstation 3100 M76	0.82	81
NeXt	0.98	83
Sun 3/60	0.82	105

Table 2: Median Data Transfer Rates

(SPARCstation, DECstation). Although ethernet is non-deterministic, the histogram of Figure 3 is sharply peaked at the median transfer time. Very few one-way transfers required more than 6 milliseconds and the maximum transfer time was 196 milliseconds. Provided that the ethernet bandwidth is adequate and that ethernet saturation can be avoided, this level of determinism may be acceptable for some *control* bus applications.

Table 2 presents the median data transfer rates obtained for each platform. For a 1018 byte message, the data transfer rate for the Sun 3/60 platform was in excess of 100 kbytes/second. If ethernet were employed for the *data* bus, the maximum data rate would be of the order of 100 kbytes/second for CISC platforms. It is expected that superior data transfer performance would be obtained if the socket server employed a multiple segment sliding window protocol (TCP) for *data* bus transfers and reserved the non-sliding window protocol for *control* bus applications.

Since it is suspected that the largest contribution to socket server protocol overheads is made by the non-Unison processor, the forthcoming socket server port to RISC platforms may considerably enhance socket server ethernet performance. However, FDDI is the prime candidate for truly high performance *control* and *data* bus applications.[4] Due to its redundant counter-rotating token ring scheme, FDDI is both reliable and deterministic. Since the FDDI standard incorporates a layered software and hardware architecture, it will easily accommodate change and platform heterogeneity.

7 Conclusion

The socket server is well suited for communication among distributed heterogeneous systems. For instance, the synchronous message-based semantic has recently been ported to VMS platforms and this enables a multi-process VMS socket server. Since the socket server uses Unison messaging primitives, it will function in both the Unison/Reliant and the Unison/pSOS+ environments. A socket server port to the VxWorks real-time operating system is also anticipated.

[4] A Unison TCP/IP and UDP/IP port to FDDI is anticipated by Fourth Quarter this year.

If the protocol overheads of a layered network implementation are unacceptable and present day technology must be employed, the KF CCS *control* and *data* buses must be implemented using custom/proprietary systems. Such systems may be networked systems using custom hardware and software, they may be distributed shared memory systems or they may be hybrid shared memory/networked systems. In general, such systems are more awkward to implement and expensive to maintain because they do not easily accommodate change. In addition, typical custom/proprietary systems do not easily support an operational environment which consists of heterogeneous computing platforms.

The ethernet socket server is suitable for *control* and *data* bus applications provided that its limitations (250 messages/second, 100 kbytes/second, non-deterministic) are acceptable. If these limitations are not acceptable then RISC/FDDI socket server ports or custom systems are required.

References

[1] C Inwood, G A Ludgate, D A Dohan, E A Osberg, and S Koscielniak. "Domain-driven specification techniques simplify the analysis of requirements for the KAON factory central control system". *Nuclear Instruments and Methods in Physics Research*, A293(1,2):390–393, 1990.

[2] E A Osberg, G A Ludgate, S Koscielniak, and D A Dohan. "Dynamic object modelling as applied to the KAON control system". *Nuclear Instruments and Methods in Physics Research*, A293(1,2):394–401, 1990.

[3] Peter J M Baker. "A programming paradigm based on the send-receive-reply task communication primitives". Master's thesis, Department of Electrical and Computer Engineering, University of Victoria, Victoria, B.C., Canada, 1988.

[4] Neil A. Wilkinson. "Message-based network communication between heterogeneous systems". KAON Factory Project Definition Study TRI-DN-91-K164, TRIUMF, 4004 Wesbrook Mall, Vancouver, B.C., Canada V6T 2A3, June 1991.