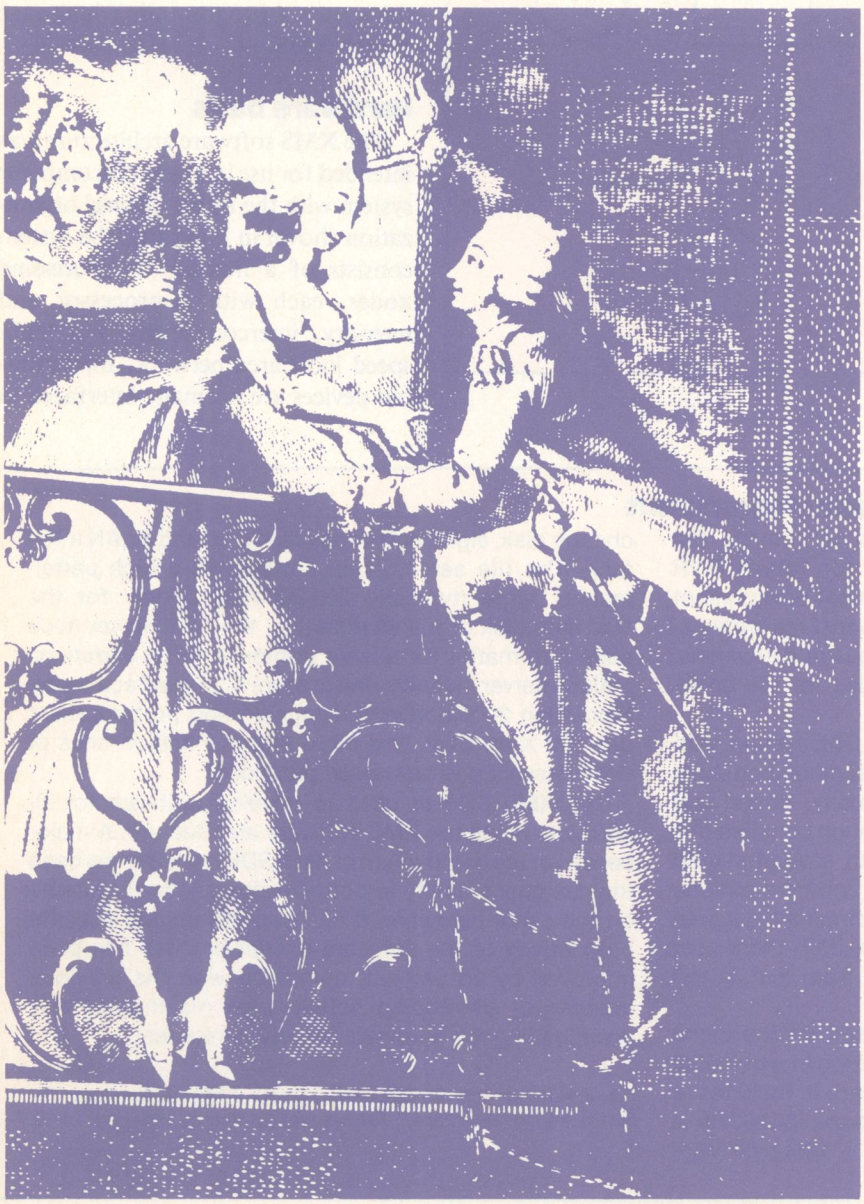
XMS: A Rendezvous-Based Distributed System Software Architecture

XMS creates a single, powerful system from loosely coupled microcomputers. Programs work together across nodes, making systemwide resource management transparent and distributed-system design simpler.

Neil Gammage and Liam Casey, BNR, Ottawa



The Bettmann Archive

A distributed processing architecture has been developed at BNR for use as the base computing system for a wide range of future Northern Telecom products. The system, known as XMS, has been implemented on a number of different hardware organizations. The first implementation was for the XMS software development environment (see box p. 10), and it has subsequently been implemented for NT's Meridian product family.

XMS is based on the use of multiple loosely coupled microcomputers. The interconnection of many inexpensive computers to construct a single powerful computing system offers many potential benefits, including good cost/performance due to the use of cheap processors, improved fault tolerance as a result of multiple computing units, and the ability to tailor the power of a system to meet the processing needs of the application by adding or removing computing units.

Hardware alone does not make a distributed processing system; neither does the ad hoc grafting of a communications mechanism to an existing system. To fully realize the potential benefits of a multicomputer architecture, it is necessary to have a software architecture that makes effective use of the hardware—an architecture that provides a uniform and systematic way of handling concurrency, managing resources, and recovering from errors, both within a computing module and between computing modules.

The XMS software architecture follows closely the concurrency model of Ada.[1] It is based on the use of many concurrently executing tasks that communicate using rendezvous. The rendezvous mechanism has been uni-

0740-7459/85/0500/0009$01.00 © 1985 IEEE

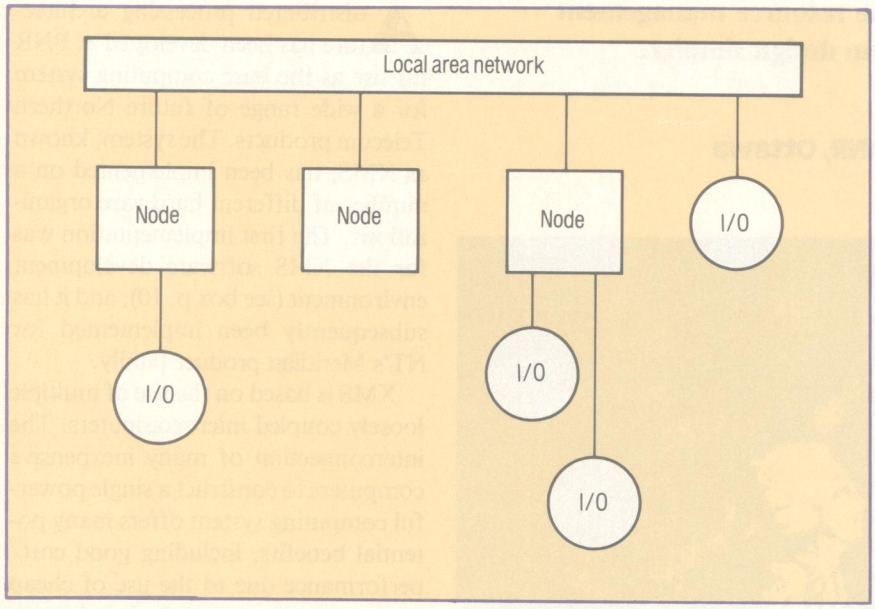


Figure 1. XMS hardware configuration.

formly extended to provide communication between tasks in different programs, either in the same node or in different nodes. The concurrency features implemented by the XMS operating system kernel are incorporated in the design of BNR Pascal, the systems programming language for the XMS.

Hardware basis

The XMS software architecture was intended for use in any multicomputer system with the rather general organization shown in Figure 1. The system consists of a number of processing nodes, each with a processor and memory, interconnected by a high-speed local area network. Input/output devices are normally interfaced to

The XMS software development environment

The intention behind the XMS is to provide the distributed computing technology for future BNR products. The XMS software development environment was constructed with two purposes: to test the architectural constructs and BNR Pascal language extensions, and to provide the computing facilities for the development of XMS-based projects.

The first XMS multiple workstation SDE system, with a shared file server, became operational more than three years ago. The SDE is being continually refined and enhanced, and a typical SDE configuration consists of a number of personal workstations with several shared servers such as Felix file servers, communications servers, and print servers accessed across a local area network (see figure near right). Within BNR there now exist over 25 LANs having anywhere from five to 150 workstations attached.

All SDE processing nodes are based on the same standard computing unit. The organization of a personal workstation nodes is shown far right. It includes a computing model (based on the Motorola MC68010), a 1M-byte memory module, a terminal, a 10M-byte Winchester disk, eight inch floppy disk unit, and a LAN interface. The file server node substitutes a high performance, large (typically 200-300M-byte) disk for the 10M-byte disk of the workstation, the print server node adds a formatter for a laser printer, and the communications server includes one or more X.25 interface cards. More than 2000 software designers and programmers use the XMS SDE, and more than 1.5 million lines of BNR Pascal code have been produced.

The XMS SDE in used to develop software for all products using the XMS software architecture. A major benefit of the deployment of XMS SDE systems has been the designer's ability to both develop and test systems on the same hardware. It has been estimated that the productivity of the software design process has been improved by 25 percent, compared with the previous process of developing software on mainframes and downloading the hardware prototypes for test.*

*A. LaMarca and G. Stewart, "Using XMS Technology for Software Development," *Telesis*, Vol. 11, No. 3, 1984, pp. 14-19.

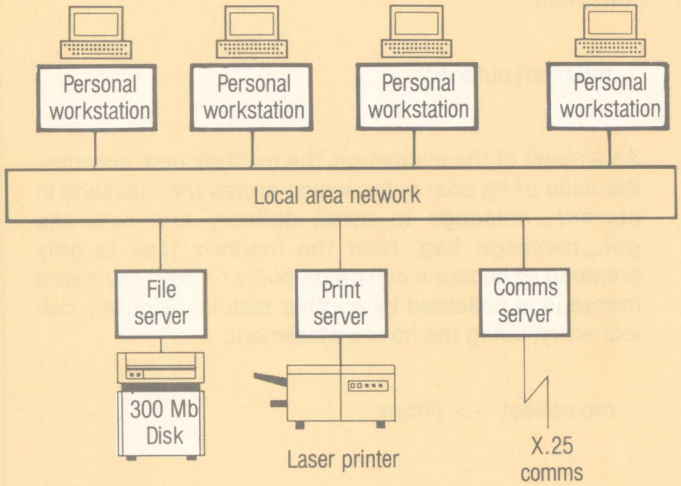
The Bettmann Archive

one of the processing nodes of the system, but intelligent peripherals may also be directly interfaced to the local area network and may communicate with more than one processing node. Processors communicate by exchanging data packets across the local area network.

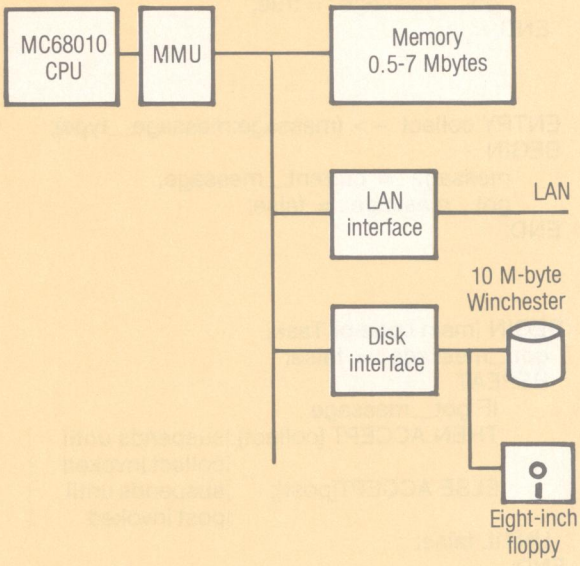
A node may support a number of virtual address spaces. The computing module used in the software development environment includes a memory management unit supporting segmented virtual addressing. The 16M-byte virtual address space is divided into 64 segments of up to 256K bytes each. Each segment can be individually write-protected or marked for supervisor (kernel) access only, and each can be positioned anywhere in memory. Up to 256 independent address spaces may be in use simultaneously on any node.

Software architecture

The XMS software architecture will be the base for many different types of distributed systems. It provides features that support real-time operation, high availabilty, dynamic reconfiguration, and load sharing. How these features will be exploited is left to the designer of individual systems. At the same time, the complexity of the design, programming, and maintenance of distributed systems has to be addressed. Software engineering considerations led to the adoption of the following features for XMS: separate compilation, language support for parallelism and communication, blocking communication primitives, and the dynamic binding of programs.



XMS SDE configuration.



XMS SDE workstation node.

A tasking example

The following example shows a simple task whose function is to buffer messages that are transmitted between two other tasks. The code and data of the task, and the interface that it presents to other tasks, are defined by the task type definition:

```
      {Interface of Task}

TASK TYPE mailbox;
PROVIDES
   ENTRY post(message:message_type);
   ENTRY collect -> (message:message_type);
END; {of interface part}


      {Body of Task}

 VAR current_message:message_type; {private data}
      got_message:BOOLEAN;          {of Task      }

 ENTRY post(message:message_type);
 BEGIN {action performed when ENTRY accepted}
      current_message := message;
      got_message := true;
 END;


ENTRY collect -> (message:message_type);
BEGIN
     message := current_message;
     got_message := false;
END;


BEGIN {main code of Task}
 got_message := false;
 REPEAT
      IF got_message
      THEN ACCEPT [collect] {suspends until  }
                            {collect invoked }
      ELSE ACCEPT[post];    {suspends until  }
                            {post invoked    }
 UNTIL false;
END;
```

An instance of this task type may be declared as a variable:

```
VAR mb : TASK mailbox;
```

The task is instantiated, and its taskid is assigned to the task variable by use of the initiate statement

```
INITIATE mb SPACE 200;
```

which creates a new copy of a mailbox task with a space of 200 bytes for its own stack and variables *current_message* and *got_message.*

After it is initiated, the task executes its main code to initialize itself and then executes an endless repeat loop in which by using the accept statement, it waits for other tasks to invoke it. Initially it has no message stored and is only prepared to accept a call on its post entry. A task trying to collect a message will be delayed until there is a message ready for collection. A message is posted by some other task invoking the post entry, using the invoke statement:

```
mb.post{outbox};
```

As a result of the invocation, the mailbox task executes the code of its post entry, which stores the message in *current_message* to await delivery and sets the *got_message* flag. Now the mailbox task is only prepared to accept a call on its collect entry. The saved message is collected by another task invoking the collect entry, using the invoke statement:

```
mb.collect -> {inbox};
```

The code of the collect entry is then executed, extracting the message and delivering it to the calling task, and resetting the *got_message* flag.

The Bettmann Archive

BNR Pascal. BNR Pascal is the systems programming language for XMS. It was designed to meet three fundamental requirements:

- support for the features of a modern high-level language such as a rich set of control constructs, strong type checking, and data abstraction mechanisms;
- provision of extensive support for modularity and separate compilation along the lines of Ada packages; and
- support for concurrency and intertask communication at the language level.

We decided to extend Pascal with modularity and concurrency features. The use of Pascal as a base met the first requirement. Pascal provides a good set of looping and control constructs, strong type checking, and user-defined types to support data abstraction. Support for modularity and separate compilation is based on the use of units, which are similar to Ada packages. A unit is used to specify related types, variables, and the procedures that operate on them. The procedures may be called from outside the unit, but their inner workings are concealed from outside users. Units are the basic parcels of compilation in BNR Pascal. Support for concurrency and intertask communication closely follows the Ada tasking model.[1] Tasks are the unit of execution in BNR Pascal. They communicate using a form of rendezvous, extended to allow two tasks anywhere in a system to communicate.

Programs. An XMS system is constructed from a set of independent but cooperating programs. The program is the addressing context; a variable declared in one program may not be read or written by the code of another program, nor may a procedure in one program be directly called from another program. Within a system, there may be several instances of a given program executing at the same time, and the mix of program instances in execution may vary dynamically, according to the needs of the application. The program instance is the unit of distribution. Each program instance executes in its own virtual address space on a node of the system and cannot be moved to another node while it is executing.

Tasks. A BNR Pascal program consists of one or more tasks that execute in parallel with each other and in parallel with tasks in other programs. The synchronization of tasks and communication between tasks is carried out by rendezvous. Every task is an instance of some task type. Any number of instances of a given task type may exist at any one time. These instances may be created and destroyed dynamically.

A task type consists of an interface and a body. The interface defines the functions the task can perform. These functions are defined by the entries the task type provides for rendezvous by other tasks.The body defines the task type's dynamic behavior. The accompanying sidebar gives an example of a simple task type—a mailbox for messages—and shows how new tasks of this type can be created and destroyed through rendezvous.

All tasks of a given task type provide the same interface and execute the same code, but each task has its own set of variables. A task is identified by a taskid that is generated when the task is created and invalidated when the task is destroyed. The taskid can be thought of as a handle that uniquely identifies a task in a locale.

Local rendezvous. The local rendezvous is used for communication between two tasks in the same program executing in the same address space. A rendezvous occurs as a consequence of one task invoking an entry declared in the interface of a second task, and the second task executing an accept statement enabling a call on that entry. The entry may have both input and output parameters. The input parameters are passed from the invoking to the invoked task (by value) when the rendezvous begins, and the output parameters are passed back to the invoking from the invoked task when the rendezvous ends.

Figure 2 shows the sequence of events during a rendezvous. If the accept is issued before the rendezvous call from the invoker, the accepter is suspended. When the rendezvous call occurs, the input parameters are copied from the invoker to the accepter, the invoker is suspended, and the accepter executes the code of the entry procedure. When the entry procedure completes execution, the output parameters are copied from the accepter to the invoker, which resumes execution from the point that it made the rendezvous call. If the rendezvous is invoked before the accept has been issued, the invoker is suspended as before. If, when the accept is issued, there is more than one acceptable invocation waiting, the one that has been waiting longest will be accepted; the accepter continues running, executing the code of the entry procedure.

Remote rendezvous. The remote rendezvous is a uniform extension of the local rendezvous. It may be used between any two tasks, whether they are in the same or different programs, executing on the same or different nodes. The local rendezvous can be considered an optimized version of the more general remote rendezvous, for use only where the communicating tasks are part of the same program. The semantics of the remote rendezvous are, as far as possible, identical to the semantics of the local rendezvous. The three cases of remote rendezvous—between tasks in the same program, between tasks in different programs executing on the same node, and between tasks in programs executing on different nodes—are mechanized differently but have identical semantics.

To invoke a remote rendezvous, a task must have both visibility to the in-

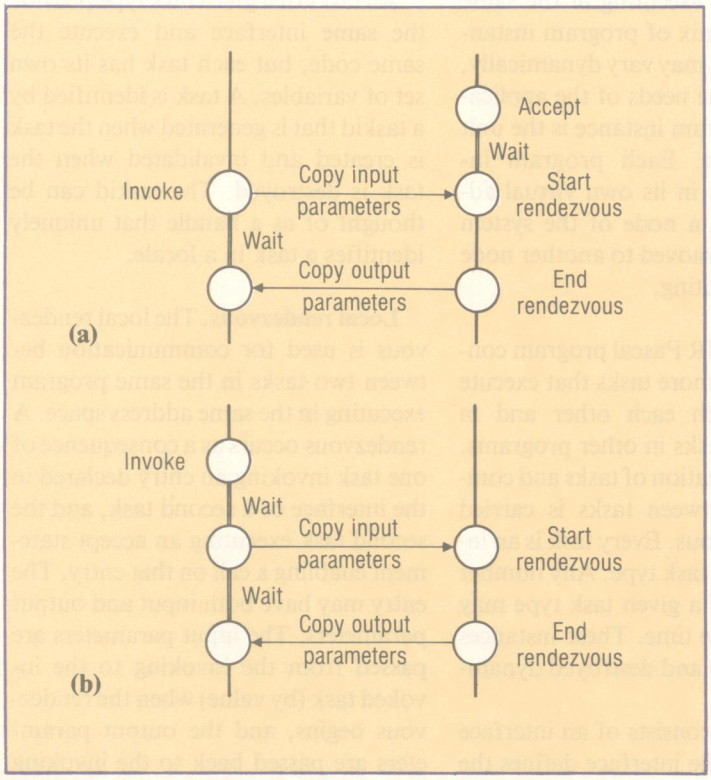


Figure 2. Rendezvous event sequence: (a) accepter first; (b) invoke first.

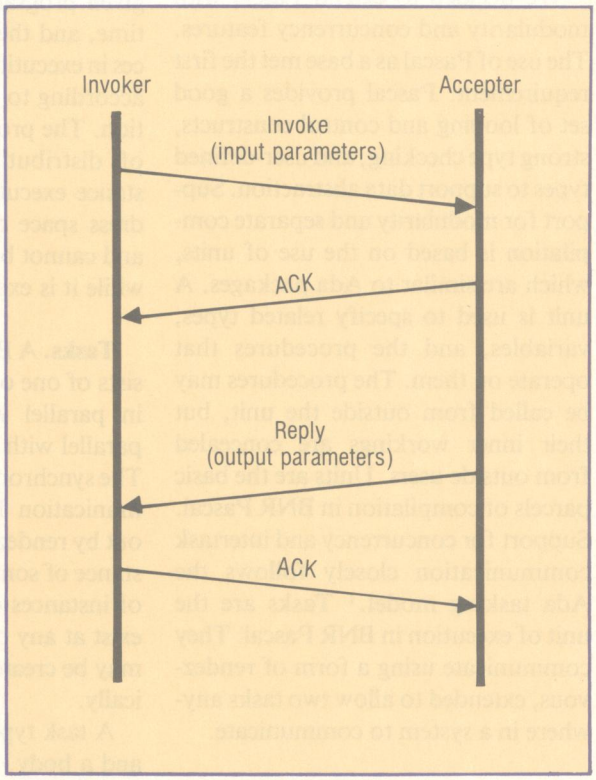


Figure 3. Remote rendezvous protocol.

The Bettmann Archive

terface of the remote task and a remote taskid for it. The interface is provided by copying the interface definition of the target task into the text of any program that wishes to invoke it. The remote taskid consists of the taskid of the target task and the identity of the locale and node in which it is executing. The sidebar below right shows the declarations required in a program so that a task can remotely access the mailbox task defined in the sidebar on p. 12. The same task can be invoked by either local or remote rendezvous; there is no way that the invoked task can distinguish a local from a remote invocation.

A remote rendezvous is invoked in exactly the same way as a local rendezvous. The input parameters are copied from the address space of the invoker to the address space of the invoked task before the rendezvous begins, and the output parameters are copied back when the rendezvous is completed. Memory-to-memory parameter copies are used if the two tasks are executing on the same node. If the tasks are executing on different nodes, the parameters are transmitted as data packets over the local area network. Figure 3 illustrates the simple four-packet protocol used to implement remote rendezvous between nodes.

Distribution of remote taskids. For a remote rendezvous to take place, a task must obtain the remote taskid of the task with which it wishes to communicate. Remote taskids may be obtained from an operating system task known as the name server. An instance of the name server is created on each node of a system and is given a special, well-known taskid.

A task that is prepared to be invoked remotely can register itself with the name server on its node, supplying its taskid and a unique text string name. Another task can obtain the remote taskid of this task by presenting the same text string to the name server on its node. All the name server tasks communicate, using remote rendezvous, to exchange the names and taskids of registered tasks so that they may be invoked from any program on any node. Remote taskids may also be exchanged between tasks as parameters of a rendezvous. A program therefore only needs to register one task with the name server, and the registered task can then hand out the taskids of other tasks of the program.

Dynamic binding of programs. Dynamic loading of new programs into a system is facilitated by requiring only the interface part of remote task types to be available when compiling any program that wishes to perform remote rendezvous, and by the remote taskid distribution mechanisms described above. Any program can be replaced by another that has the same task interfaces. Recompilation of other programs that perform remote

Defining a remote task

The mailbox task of the previous box can be accessed remotely by a task in a program that includes the remote task definition and remote task declaration shown below.

```
TASK TYPE mailbox;
PROVIDES {definition of remote task interface}
    ENTRY post(message;message_type);
    ENTRY collect -> (message:message_type);
END; REMOTE;

VAR rmb : REMOTE TASK mailbox
```

The remote task definition is simply a copy of the interface of the mailbox task, with the addition of the remote qualifier, and the task is declared in the same way as the declaration of a local task, with the addition of the remote keyword. Once the remote taskid for the mailbox task has been assigned to the remote task variable *rmb*, then the remote rendezvous can be invoked just like local rendezvous.

```
rmb.post(outbox);

rmb.collect ->(inbox);
```

rendezvous to it is not required, nor does the whole system have to be reloaded. These are important considerations for high-availability real-time systems.

Application architecture

Using the facilities of BNR Pascal and the XMS operating system, an application is structured as a set of cooperating programs that may be distributed over the nodes of an XMS system. The division of the application into programs is not arbitrary, being determined by a number of factors. First, since each program executes in its own protected virtual address space, the program is the unit of protection in the system. Protection of operating systems code and data from tampering by applications code is ensured by dividing the operating system and applications into separate sets of programs. The real-time overheads for a remote rendezvous are higher than those for a local rendezvous (typically double for an intranode and five times for an internode rendezvous), and data may not be shared between tasks in different programs. These factors largely dictate the partitioning of the application into programs. It is expected that programs will be of moderate size and will present relatively narrow interfaces to other programs. Narrow interfaces permit independent development of each of the programs in a distributed application.

While the division of an application into programs is fairly rigid (once programmed it is not easy to change), the distribution of programs over the nodes of a system is very flexible. Any number of programs, each executing in its own locale, may be assigned to a given node at load time, up to the real-time and/or memory limits of the node in question. If the application is small enough, it may be possible to assign all the programs to a single XMS node, but in many cases multiple nodes must be used for real-time or reliability reasons. This flexibility ensures that a multinode system can be viewed as a single computing system, with hardware structural details such as numbers of processing nodes and the use of the local area network abstracted away by the network operating system, making them invisible to the application software.

The preferred approach for XMS is to use a client-server model of the application.

The fact that the underlying hardware consists of a number of independent nodes may, however, be exploited by the application to increase system integrity and fault resilience. The failure of the hardware of one node, or a fault in the software it is running, can be prevented from causing the entire system to fail, provided that the assignment of programs to nodes is properly controlled. For example, in a software development system, the failure of a single workstation node will not cause the entire system to fail. The actual assignment of programs to nodes is quite static for the software development environment: Each user's programs run on his or her own workstation while each server has a dedicated node. Other XMS-based products, with more stringent real-time and availability constraints, dynamically determine where programs should be loaded.

The client-server model. An application may be divided into programs according to a number of different structuring principles. The preferred approach for XMS is to use a client-server model of the application. This model divides the application into layers, where each layer provides some set of services to the layers above it, building on and in some sense improving on the facilities provided by lower layers. Each layer is regarded as both a server providing services to higher layers and a client of the services provided by lower layers. For example, the Felix file server[2] is a client of the operating system and provides a basic file storage service. The DBMS layer is a client of Felix and provides a database service to the application layers.

In XMS, all services are implemented by programs, and services are accessed by rendezvous with public tasks of the service program. At least one public task in each service program must be registered with the operating system's name server. Other public tasks may be made known to users by communicating with the registered task.

Resource managers. A particularly important class of servers are resource managers. A resource manager provides a number of resources that may be allocated to and used by client layers. For example, the Felix file server is a resource manager whose resources are open files. When a file is

The Bettmann Archive

opened, a task is allocated to perform all operations (read, write, etc.) on the file, and the taskid of this file handler task is passed back to the client, who may then read or write the file by performing rendezvous directly with it.

Felix is an example of a more general organization. Each resource manager provides a public task (the allocator) which is responsible for allocating resources to clients. When a client needs a resource, it performs a rendezvous with the allocator; the allocator passes back to the client the (remote) taskid of a task (a resource handler), which executes the code to implement all operations that may be performed on the resource (for example, the read and write operations in the case of an open file). The private data of the resource handler task defines the current state of the resource, and all operations on the resource are performed by the invoking entries of the resource handler.

A number of significant resource management issues are dealt with by this approach. They are resource integrity, protection, revocation, and recovery.

Resource integrity. Since the state of the resource object is only accessible to the resource handler and cannot be directly accessed by the client, the integrity of the resource is ensured.

Resource protection. In order to use a resource, a client task must possess a valid taskid for the resource handler. Since taskids are only generated by the XMS kernel and are difficult to forge, accidental or even malicious use of a resource by an unauthorized client can be detected and prevented.

Resource revocation. It is sometimes necessary to revoke rights to access a resource. For example, if a hardware or software fault kills the resource handler, it must be ensured that clients cannot continue to access the resource. This is achieved by the system automatically invalidating the taskid of a task when it dies and ensuring that the same taskid will not be reissued for a long time. The same mechanism can be used if the resource allocator wishes to revoke access to a resource; all that is necessary is to destroy the resource handler task, thereby invalidating its taskid. If the resource manager as a whole crashes, all taskids for all its resource handlers will be invalidated, thus revoking access to all of its resources automatically.

When a client needs a resource, it rendezvouses with the allocator, who passes back the taskid of a resource handler, which executes the code.

Resource recovery. If a client crashes or is destroyed, it is necessary to recover all of the previously allocated resources (open files, etc.). In XMS this is the resource manager's responsibility. Garbage collection of unused resources may be based on active (audit) or passive (timeout) principles. When a resource is allocated, the taskid of the client is recorded. A periodic check on the client task status is performed by the resource manager, and the resource is deallocated if the client is found to be dead. The resource handler applies a timeout between uses of the resource object. If the resource remains unused for some predefined time, it is assumed that the client has lost interest in it, and the resource is is revoked. A combined technique may also be used, with the timeout being used as a warning of a potential problem, and a client audit being used to determine whether or not the client is in fact still alive. It should be noted that these techniques do not rely on messages from dying clients, or on operating system messages informing resource managers that clients have died. In a distributed system, there are no guarantees that such messages will be generated or delivered correctly.

Networking architecture

So far we have examined the properties of a set of XMS nodes interconnected by a single local area network. We shall call such a set of nodes an XMS cluster. We now turn to the architecture appropriate to the interconnection of a number of XMS clusters. A cluster consists of a number of homogeneous nodes on a single LAN.

By "homogeneous" we mean that any two nodes are capable of sending, receiving, and correctly interpreting the packets used to implement remote rendezvous. This does not mean that the nodes are of identical design, but it does imply a minimal level of internal consistency—for example, that their data representations are identical, and that their byte and word numbering schemes (the so-called byte sex) are the same. By "single LAN" we mean a uniform interconnect such that any two nodes can communicate in the same way, with similar real-time performance. Thus, two LAN segments connected by a transparent, nonblocking gateway can be regarded as form-

ing a single LAN for the purpose of defining a cluster.

From the architectural point of view, the most important property of a cluster is that it defines a uniform task address space. Any task in the cluster can potentially communicate with any other task in the cluster without needing to know precisely where the other task is executing, but only whether it is executing in the same or in a different program. Interchange of data between clusters is provided by communications servers, which use the LAN to communicate with client tasks in the same cluster and a data network (such as Datapac, Telenet, or other public or private equivalents) to communicate with a communications server on another cluster. XMS communications servers use a standard network protocol, based on the ISO Open Systems Interconnection Model, for session-layer communication over the data network.

It was decided not to extend the general remote rendezvous to operate between clusters. We felt that to do so would impact the efficiency of remote rendezvous within a cluster. Problems we foresaw included managing virtual circuits, handling different timeouts, and dealing with a potenially unlimited task address space. Although we do not provide network transparency at the task communications level, services may be built on the basic communications server facilities to provide network transparency at higher levels in the OSI sense. An example is the Helix distributed file system,[3] which gives uniform access to files distributed over nodes within a cluster and may be extended by the networking service to provide uniform access to files geographically distributed over clusters.

Experience

Development of the XMS continues as we refine and extend features in light of our experiences using the XMS software development environment and providing the base technology for other products.

One particular advantage is that the whole system state is captured in the state of its tasks.

One of the distinguishing features of the XMS is the remote rendezvous. Some designers who had a message-passing background had initial difficulties in accepting the blocking nature of rendezvous-based communications. But almost all applications fit the server-client model, and its use has provided overall benefits in terms of system structuring and engineering ease. One particular advantage is that the whole system state is captured in the state of its tasks, instead of being an uncertain combination of task states plus outstanding messages.

There is a small category of application where not all task interactions are of the server-client variety. Some distributed applications require non-blocking interactions between co-equals. This can always he handled by using a communications subtask that performs the interaction on behalf of its parent and then rendezvous with the parent to inform it of the results. This process has been mechanized with a new BNR Pascal language feature called Send.

Users of the XMS software development environment do not in general have to be aware of the distributed nature of their system. This is due in no small part to the uniform access to files provided by Helix.[3] However, users do see a delay when they start up their workstations. This arises from name server initialization. It has proved difficult to implement a distributed name server that is both fast at informing a newly started workstation of available services and at the same time avoids partitioning the system. The LAN technology used for the software development environment system does not support quick polling of all workstations in the system. We are researching possible solutions to this problem.

In current implementations, remote rendezvous has not proved to be an efficient method for transferring large amounts of data between locales, although special hardware support could overcome this. An more direct memory-to-memory transfer mechanism called Push/Pull has been provided. In order to provide some protection against random writing into a locale's memory, the push and pull operations can only be invoked by a task that is in rendezvous. Push copies data from the task's locale to that of the invoker, while Pull copies data in the reverse direction.

In the early stages there was some debate as to how the binding of actual parameters to formal parameters of remote rendezvous entries would be checked. The solution—runtime checking when the parameters arrive at the invoked task—has proved to be a good compromise. Copying the target task's interface into programs has not proved

The Bettmann Archive

difficult to organize. In certain environments, such as the software development environment, it is impossible to recompile all the programs in the system together every time there is an interface change. This loose binding has allowed us to introduce new entries into a server's interface. Any programs that don't invoke that particular entry are unaffected. Changing the parameters to an existing entry can be done by giving the changed entry a new name and supporting both old and new entries for a transition period.

A wide range of functional requirements can be met by systems composed of multiple loosely coupled microprocessors. The XMS software architecture described in this article is well suited to many applications, ranging from office products through software development systems to embedded subsystems. It has been designed to minimize the complexity of designing and programming distributed systems.

The tasking and rendezvous supported by XMS are important concepts for the structuring of systems. They directly support a server-client model and facilitate robust and resilient resource management. The remote rendezvous is a straightforward and uniform extension of the rendezvous. It permits a totally flexible assignment of programs to nodes and makes systemwide resource management transparent.

The XMS software architecture is meeting its goal of providing the distributed computing base for future Northern Telecom products. The first implementation of the XMS software architecture, the software development environment, has been operational for more than three years. It is widely used within BNR for the development of XMS-based projects and has been credited with a 25-percent productivity increase in the software design process. □

References

1. "Ada Programming Language," Military Standard ANSI/MIL-STD-1815A, Feb. 1983.
2. M. Fridrich and W. Older, "The Felix File Server," *Proc. 8th Symp. Operating Systems Principles,* Dec. 1981, pp. 37-44.
3. M. Fridrich and W. Older, "Helix: The Architecture of the XMS Distributed File System," *Proc. 8th Symp. Operating Systems Principles,* Dec. 1981, pp. 37-44.

Additional reading

A. Birrel and B. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Comp. Syst.,* Vol. 2, No. 1, Feb. 1984.

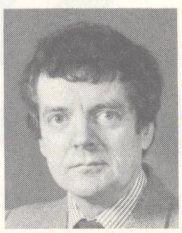
L. Casey and N. Shelness, "A Domain Structure for Distributed Systems," *Proc. 6th Symp. Operating Systems Principles,* Nov. 1977, pp. 101-108.

M. Dowson, B. Collins, and B. McBride, "Software Strategy for Multiprocessors," *Microprocessors and Microsystems,* Vol. 3., No. 6, July/Aug. 1979, pp. 263-266.

G. Shoja, F. Halsall, and R. Grimsdale, "A Control Kernel to Support Ada Intertask Communication on a Distributed Multiprocessor Computer System," *Software and Microsystems,* Vol. 1, No. 5 Aug. 1982, pp. 128-134.

S. Shrivastava and F. Panzieri, "The Design of a Reliable Remote Procedure Call Mechanism," *IEEE Trans. Comp.,* Vol. C-31, No. 7, July 1982, pp. 692-697.

BNR *Telesis,* Vol. 11, No. 3, 1984.



Neil Gammage is director of computing technology development at BNR in Ottawa, Canada. His main interests lie in computer architectures, operating systems, and programming languages, and he has been involved in the development of the XMS distributed computing system for the past five years. Gammage worked in the British computing industry for ten years before joining BNR. He received his BA in physics from Oxford University, England.

Liam Casey manages file system development for the XMS system at BNR in Ottawa, Canada. Before joining BNR, Casey worked on software projects for the New Zealand DSIR. His research interests lie in all aspects of distributed computing. Casey received his PhD from the University of Edinburgh, Scotland, where he also undertook SRC-supported postdoctoral research. He is a member of the British Computer Society.

The authors may be contacted at BNR, PO Box 3511 Station C, Ottawa, Ontario, Canada K1Y 4H7; (613) 727-2000.