Solaris MC: A Multi Computer OS

Yousef A. Khalidi, Jose M. Bernabeu, Vlada Matena,
Ken Shirriff, and Moti Thadani
Sun Microsystems Laboratories

Solaris MC: A Multi Computer OS

Yousef A. Khalidi, Jose M. Bernabeu, Vlada Matena, Ken Shirriff, Moti Thadani

Sun Microsystems Laboratories
2550 Garcia Avenue
Mountain View, CA 94043 USA.

Abstract

Solaris MC is a prototype distributed operating system for multi-computers (i.e. clusters of nodes) that provides a *single-system image*: a cluster appears to the user and applications as a single computer running the Solaris® operating system. Solaris MC is built as a set of extensions to the base Solaris UNIX® system and provides the same ABI/API as Solaris, running unmodified applications. The components of Solaris MC are implemented in C++ through a CORBA-compliant object oriented system with all new services defined by the IDL definition language. Objects communicate through a runtime system that borrows from Solaris doors and Spring subcontracts. Solaris MC is designed for high availability: if a node fails, the remaining nodes remain operational. Solaris MC has a distributed caching file system with Unix consistency semantics, based on the Spring virtual memory and file system architecture. Process operations are extended across the cluster, including remote process execution and a global /proc file system. The external networks is transparently accessible from any node in the cluster. The prototype is fairly complete—we regularly exercise the system by running multiple copies of an off-the-shelf commercial database system.

1. Introduction

Solaris MC[1] is a prototype operating system for a multi-computer, a cluster of computing nodes connected by a high-speed interconnect. The Solaris MC operating system provides a single system image, making the cluster look like a single machine to the user, to applications, and to the network. By extending operating system abstractions across the cluster, Solaris MC preserves the existing Solaris ABI/API and runs existing Solaris 2.x applications and device drivers without modification.

The decision to design a cluster operating system was motivated by trends in hardware technology. Traditional bus-based symmetric multiprocessors (SMP) are limited in the number of processors, memory, and I/O bandwidth that they can support. As processor speed increases, traditional SMPs will support an even smaller number of CPUs. Powerful, modular, and scalable computing systems can be built using inexpensive computing nodes coupled with high-speed interconnection networks. Such clustered systems can take the form of loosely-coupled systems, built out of workstations [1], massively-parallel systems (e.g. [24]), or perhaps as a collection of small SMPs interconnected through a low-latency high-bandwidth network.

The key to using clustered systems is to provide a single-system image operating system allowing them to be used as general purpose computers. Cluster systems in the past have been mostly used for custom-built parallel and distributed applications, and sometimes as specialized database systems. However, to fully exploit the potential of clustered systems, we believe that they have to be usable as *general purpose computers*, running existing applications without modification. Moreover, clustered systems have to be easy to administer and maintain. The fact that the computer is actually built out of multiple computing nodes should be invisible to the user. Finally, since clustered systems are built out of many components, the clustered system should be *highly-available* and should be able to tolerate the failure of any one component.

Our goals are to make a cluster of nodes that may or may not share memory appear as a single general purpose multiprocessor. It should be seen as a single machine by applications, users and administrators. We want this while preserving object code compatibility (the ABI), minimizing changes to kernel code,

1. Solaris MC is an internal name of a research project at Sun Microsystems Laboratories. More information on the project can be obtained from http://www.sunlabs.com/research/solaris-mc.

requiring minimal or no change to device drivers, and supporting high availability.

Solaris MC has several interesting features. It:

- Extends existing Solaris operating system

  Solaris MC is built on top of the Solaris operating system. Most of Solaris MC consists of loadable modules extending Solaris and minimizes the modifications to the existing Solaris kernel. Thus, Solaris MC shows how an existing, widely used operating system can be extended to support clusters.

- Maintains ABI/API compliance

  Existing application and device driver binaries run unmodified on Solaris MC. To provide this feature, Solaris MC has a global file system, extends process operations across all the nodes, allows transparent access to remote devices, and makes the cluster appear as a single machine on the network.

- Supports high availability

  The Solaris MC architecture provides fault-containment at the level of an individual node in the multi-computer. Solaris MC runs a separate kernel on each node. A failure of a node does not cause the whole system to fail. A failed node is detected and system services are reconfigured to use the remaining nodes. Only the programs that were using the resources of the failed node are affected by the failure. Solaris MC does not introduce new failure modes into UNIX.

- Uses C++, IDL, and CORBA in the kernel

  Solaris MC illustrates how the CORBA object model can be used to extend an existing UNIX to a distributed OS. At the same time it also shows the advantages of implementing strong interfaces for kernel components by using the IDL interface definition language. Finally, Solaris MC illustrates how C++ can be used for kernel development, coexisting with previous code.

- Leverages Spring technology

  Solaris MC illustrates how the distributed techniques developed by Spring OS [15] can be migrated into a commercial operating system. Solaris MC imports from Spring the idea of using a CORBA compliant object model [18] as the communication mechanism, the Spring virtual memory and file system architecture [7, 10, 9], and the use of C++ as the implementation language. One can view Solaris MC as a transition from the centralized Solaris operating system toward a more modular and distributed OS like Spring.

Solaris MC uses ideas from earlier distributed operating systems such as Sprite [19], LOCUS [20], OSF/1 AD TNC [26], MOS [2], and Spring. One key difference from other systems is that Solaris MC shows how a commercial operating system can be extended to a cluster while keeping the existing application base. In addition, Solaris MC uses an object-oriented approach to define new kernel components. Solaris MC also has a stronger emphasis on high availability. Finally, Solaris MC uses new techniques for making the cluster appear as a single machine to the external network.

The remainder of this paper is structured as follows. Section 2 explains the global file system. Section 3 describes how process management is globalized. Section 4 explains how I/O devices are made global and Section 5 explains how network operations are made transparent. Section 6 discusses the object-based communication model of Solaris MC, and explains CORBA and IDL. Section 7 briefly describes how Solaris MC provides high availability. Section 8 provides the current status of Solaris MC, Section 9 compares Solaris MC to other distributed operating systems, and Section 10 concludes the paper.

2. Global File System

Solaris MC uses a global file system to make file accesses location transparent—a process can open a file located anywhere in the system and processes on all nodes can use the same pathname to locate a file. The global file system uses coherency protocols to preserve the UNIX file access semantics even if the file is accessed concurrently from multiple nodes. This file system, called the proxy file system (PXFS), is built on top of the existing Solaris file system at the *vnode* [11] interface. This interface allows PXFS to be implemented without kernel modifications. The PXFS file system provides extensive caching for high performance using the caching approach from Spring [7], and provides zero-copy bulk I/O movement to move large data objects efficiently. This section discusses these features of PXFS in more detail.

PXFS interposes on file operations at the vnode/VFS interface and forwards them to the vnode layer where the file resides, as shown in Figure 1. Besides files, PXFS also provides access to other types of vnodes, such as directories, symbolic links, special devices, streams, swap files, fifos, and Solaris doors[2]. Because PXFS is built on top of the existing file

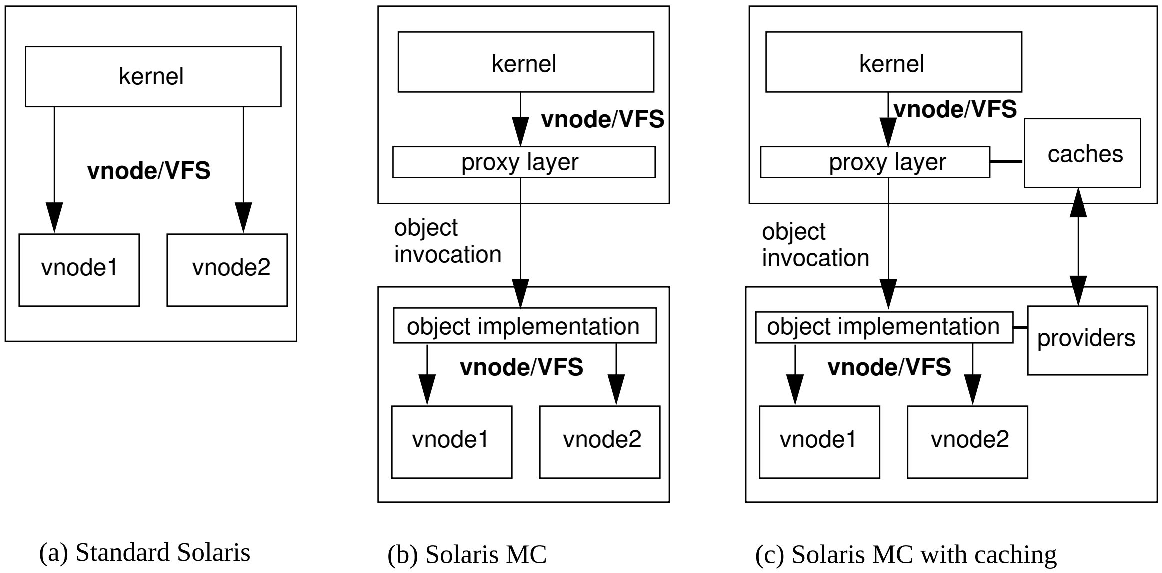

Figure 1. Extending File System Interfaces for Solaris MC. (a) In Solaris, the kernel accesses files through the VFS/vnode operations. (b) In Solaris MC, the VFS/vnode operations are converted by a proxy layer into *object invocations*. The invoked object may reside on any node in the system. The invoked object performs a local VFS/vnode operation on the underlying file system. Neither the kernel, nor the existing file systems have to be modified to run under Solaris MC. (c) Caching is used in Solaris MC to improve performance. Solaris MC supports caching of file pages, directory information, file attributes, and mount points.

system, it can leverage off the existing file system code. This is an important difference from distributed file systems such as Sprite or Spring that rewrite the entire file system.

PXFS uses extensive caching on the clients to reduce the number of remote object invocations. Figure 2 shows the objects used in the file paging and attribute caching protocols. The design of PXFS was influenced by the Spring file system and its caching architecture [7, 17, 16]. A client cache is implemented through a *cached* object on the client to manage the cached data and a *cacher* object on the server to maintain consistency. For data, the client has a *memcache* object and the server has a *mempager* object. For attributes, the client has a *attrcache* object and the server has a *attrprov* object.

As an example, suppose a process on Client 1 wishes to page in a page from a file. A *memcache* is a vnode in addition to being an IDL object, so it can accept GETPAGE and PUTPAGE operations from the Solaris virtual memory system. The memcache vnode is used as the paged vnode for the VOP_MAP operations on the proxy vnode. Memcache searches the local cache for the page. If it is not available, memcache requests the page from the associated mempager. The mempager checks the other mempagers to see if another client has the page, to maintain consistency. Finally, the page is obtained from the backing server vnode. Thus, PXFS has control over global page coherence.

The PXFS coherency protocol is token-based and allows a page to be cached read-only by multiple caches or read-write by a single cache. If a dirty page is transferred from one node to another, it is first written to the stable storage on the server to avoid losing updates due to crashes of unrelated nodes. Similarly, an attribute cache is also protected by a reader-writer token. The token is also used to enforce atomicity of read/write system calls on regular files. Token management is integrated with data transfer for better performance.

Directory caching and caching of mount points is done in a fashion similar to attribute caching. Directory operations that create or remove objects are implemented as write-through to be reflected synchronously in stable storage on the server.

PXFS has a “bulkio” object handler to perform zero-copy transfers between nodes of large data (file pages, uioread/uiowrite data) if the hardware interconnect has sufficient support. For example, if a

2. Solaris doors is a new IPC mechanism in Solaris 2.5 that is based on the Spring IPC mechanism [15].

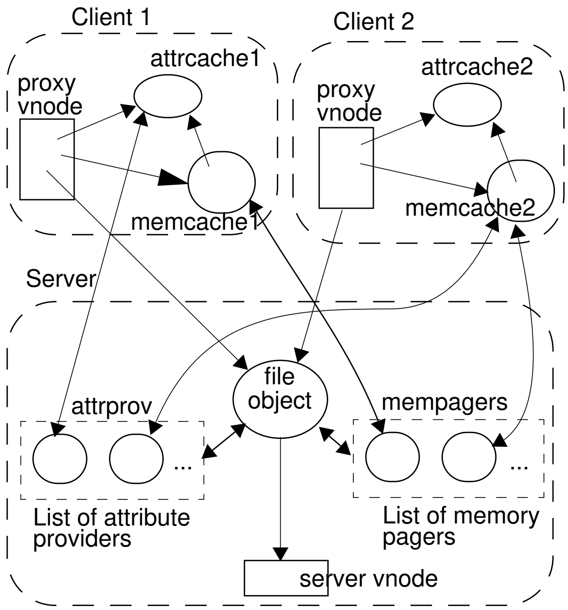

Figure 2. File Paging and Attribute Caching. Each client has a memcache object to cache data and an attrcache object to cache attributes. The server has corresponding mempager and attrprov objects to provide the data and attributes. The file object is an IDL object implementing the file protocol. The server vnode provides the underlying file storage.

process takes a page fault, it allocates a page in the local cache and invokes the *page_in* method on the mempager. The server then allocates a kernel buffer and reads the data from the disk into the buffer. The data is then transferred using the bulkio handler directly into the page on the client. If the underlying hardware supports shared memory, the server can map the client page and read data from the disk directly into the page without the need for an intermediate buffer on the server. By using a separate handler for bulk I/O, no changes to the PXFS client or server code are necessary to port PXFS to a different interconnect; only the bulkio handler has to be ported to take full advantage of the hardware.

3. Global Process Management

Global process management in Solaris MC extends OS process operations so that the location of a process is transparent to the user. While the threads of a single process must be on the same (possibly multiprocessor) node, a process can reside on any node. The design goals of process management are to support POSIX semantics for process operations while providing good performance, supplying high availability, and minimizing changes to the existing Solaris kernel. This section discusses the implementation of process management and how it transparently provide signals on global process ids, distributed waits, the /proc file system, and process migration.

Process management is implemented in a kernel module above the existing Solaris kernel code that manages the global view of processes. As illustrated in Figure 3, this layer consists of a virtual process (vproc) object for each local process and a node manager for the node. The vproc maintains state such as the parent and children of the process. The node manager keeps track of the local processes and the other nodes. Additional objects manage process groups and sessions.

The global process layer interacts with the rest of the system in several ways. First, process-related system calls are redirected to this layer. Second, a small number of hooks were added to the kernel to call this layer when appropriate. Finally, the vproc layers on different nodes communicate through IDL interfaces. Process management was made more difficult by the lack of an existing kernel interface (analogous to vnodes for the file system). We are exploring if the vproc interface can be extended to a flexible kernel interface useful for other system extensions.

Process identifiers (pids) in Solaris MC use a single global pid space and encode the home node of the process in the top bits. Thus, an arbitrary process can be located from its pid by contacting the home node, which knows the current location; this location can then be cached. The signal delivery code, for instance, uses the pid to deliver signals to a process no

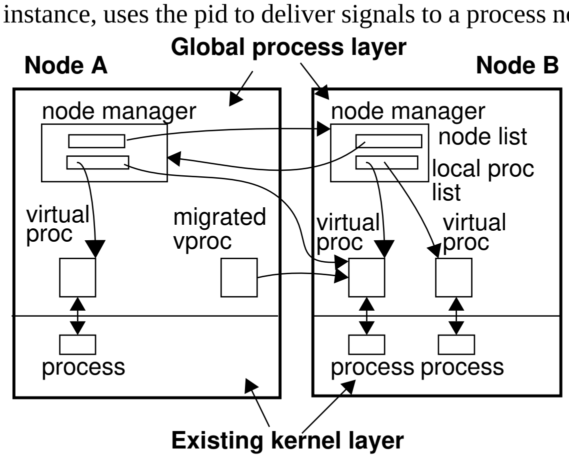

Figure 3. The data structures of the global process layer. Each node has a node manager object that has a list of all processes created or residing on the node and a list of the other nodes. Each process has a virtual process (vproc) object associated with it. When a process migrates, the old vproc is left behind to forward any operations. The vprocs keep track of the parent/child relationships of the processes.

matter where it resides. The pid encoding also ensures that processes on different nodes will not be created with the same pid. The same pid is used inside and outside the kernel; Solaris MC does not use distinct local (internal) pids and global (external) pids.

Waits pose problems for a cluster because a parent and child may be on separate nodes. In Solaris MC, distributed waits are implemented by having the child inform the parent of each state change (exit, stopped, continued, or debugged). The parent keeps track of the state of each child and wait operations use this local copy.

In Solaris, the /proc pseudo file system provides access to each process in the system; this is used by `ps` and the debugger, for instance. In Solaris MC, the /proc file system is extended to cover all processes in the cluster. Code for /proc in the pxfs file system merges together local /proc_local file systems into a global /proc. Thus, directory operations on /proc show the process entries in all the local /procs, and lookup operations are redirected to the appropriate node.

Solaris MC currently supports remote execution of processes and will soon support remote forks and migration of existing processes. For a remote fork or migration, most of the process's state will be moved automatically through the consistency mechanism of pxfs. A "shadow vproc" is left behind when a process migrates; any operations received by the shadow vproc are forwarded to the vproc on the node where the process resides. The policy decisions on load balancing will be built on top of the migration mechanisms; one possibility is to use a migration daemon, as in Sprite [5], that will decide which nodes should receive processes. However, we believe that the main use of process migration will be for planned shutdown of cluster nodes, rather than fine load balancing across the cluster; load balancing will largely be managed by placement of processes at exec time.

Global process management in Solaris MC will support high availability. That is, the failure of a node will not interfere with processes on another node. While the processes on a failed node will die, the rest of the system will continue, after a recovery phase. Parents and children will be notified appropriately of process failures. A new node will take over as home node for the failed node and migrated processes that originated on the failed node will now use the new node as home.

4. I/O Subsystem

The I/O subsystem makes it possible to access any I/O device from any node in the multi-computer without regard to the physical attachment of devices to nodes. Applications are able to access I/O devices as local devices even when the devices are physically attached to a node different from the one on which the application is running. Several areas require attention to ensure this access:

- Device configuration: Solaris provides dynamically loadable and configurable device drivers. Solaris MC transparently provides a consistent view of device configurations through a distributed device server that is notified when a new device is configured into the system on a particular node. When the device driver corresponding to the newly configured device is invoked on a different node, it is loaded on that node using the DDI/DKI device interfaces defined for Solaris. Different nodes in the system may have different devices attached and different sets of drivers/modules loaded in kernel memory at any point in time.

  The device server distributes the functionality of the Solaris *modctl()* interface, which handles the loading and unloading of dynamically loadable modules. Module configuration routines such as *make_devname()* add the new device names to the device server. Module control interfaces such as *mod_hold_dev_by_major(),* *ddi_name_to_major(),* and *ddi_major_to_name()* look-up the distributed device database rather than local data structures.

- Uniform device naming: Device numbers provide information about the location (i.e. node number) of the device in the system in addition to the type of device and the instance or unit number of the device. The operating system associates a location with every device special file. When a device is opened, the *open()* is directed to the node to which the physical device is attached.

- Providing process context to device drivers: Device drivers require access to process context for data transfer and credentials checking. In Solaris MC, the calling process may be on a different node than the node on which the driver executes. Consequently, the process context in which the driver runs is different from the process context of the calling process. The operating system provides a logical equivalence between the two processes in order for device drivers to be able to function without modification.

The Streams framework poses additional problems (which are not discussed in detail here due to space limitations). Solaris MC allows Streams device drivers and modules that use procedural interfaces to work unchanged in the new environment. Some modules, however, do not strictly obey the Streams interface; they may either be modified to run on Solaris MC, or they may be confined to one node in the cluster.

5. Networking

The networking subsystem in Solaris MC creates a single system image environment for networking applications. The operating system ensures that network connectivity is the same for every application regardless of which node the application runs on. This goal is achieved with minimal impact on the existing network subsystem implementation and without any changes to applications.

We considered three approaches for handling network traffic. The first approach was to perform all network protocol processing on a single node. This approach, however, is not scalable to large numbers of nodes. The second approach was to run network protocols over the interconnection backplane. This approach requires each node to have a separate network address, which prevents transparency. The third approach, which we took, was to use a packet filter to route packets to the proper node and perform protocol processing on that node.

Our approach creates the illusion that the set of real network interfaces available in the system is local to each node in the system. Applications are unaware of the real location of each network device and their view of the network is the same from every node in the system. When an application transmits data over an illusory network device on a node, the framework forwards the outgoing network packet to the real device. Similarly, on the input side, the framework forwards packets from the node on which the real network device is attached to the node where the appropriate application is running.

The advantages of our design are (a) protocol processing is not limited to those nodes that have network devices, (b) only one new module is written to handle networking for most protocol stacks, and (c) changes to the protocol stacks are minimized.

There are three key components of the Solaris MC networking subsystem:

- Demultiplexing of incoming packets to the "correct" node: Incoming packets are first received on the node that has the network adapter physically attached to it. The data may, however, be addressed to an application running on a different node. Solaris MC includes an enhanced implementation of the programmable Mach packet filter [14, 25], which extracts relevant information from each packet and matches it against state information maintained by the host system. Once the destination node within the multi-computer system is discovered, the packet is delivered to that node over the system interconnect.
- Multiplexing of outgoing packets from various nodes onto a network device: All protocol processing for outgoing packets is performed on the node on which the endpoint for the network connection exists. The layer that passes data to the device driver makes use of remote device access (transparently) to send data over the physical medium.
- Global management of the network name space: Network services are accessed through a service access point or sap. (For TCP/IP, the saps are simply ports.) Providing a single system image of the sap name space requires coordination between the various nodes. In Solaris MC, a database that maps service access points to nodes within the multi-computer is maintained by the SAPServer, which ensures that the same sap is not simultaneously allocated by different nodes in the system.

The structure of the networking system is shown in Figure 4. The *mc_net* module is the packet filter that creates the illusion of a local lower stream corresponding to a remote physical network device in the system. The *mc_net* module is pushed above the cloneable network device driver by the Solaris MC network configuration utilities. The network stack, with the exception of the *mc_net* module is oblivious of the location of the network device within the multicomputer system. In the figure, the SAPServer is shown independent of a node for clarity; in reality it is provided on one or more of the nodes of the system.

Solaris MC networking also provides the ability to replicate network services to provide higher throughput and lower response times. This is achieved by extending the API to allow multiple processes to register themselves as servers for a particular service. The network subsystem then chooses a particular process when a service request is received. For example, *rlogin*, *telnet*, and *http* servers are by default replicated on each node. Each new connection to these services is sent to a different node in the cluster based on a load balancing policy (currently, a simple round-robin load distribution policy). This allows the

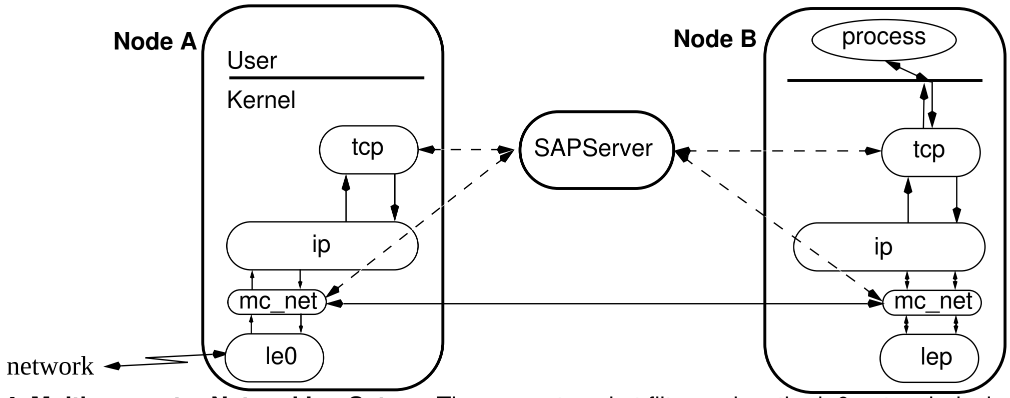

Figure 4. Multi-computer Networking Set-up. The mc_net packet filter makes the le0 network device appear local to the application process. TCP/IP protocol processing occurs on node B, preventing node A from becoming a bottleneck. Solid lines show data traffic and dotted lines show service access port control communication.

cluster to be used as a HTTP server, for example, with all nodes handling requests in parallel.

Other features of the Solaris MC networking subsystem are management of global state in the network protocols, such as network statistics maintained for network management agents and network state information acquired from routers or peers on the network. In the former case, the network management agents are modified to collect information from all the component nodes of the multi-computer, while in the latter case information collected on any node is broadcast to the other nodes.

6. Communication & Programming Infrastructure

Solaris MC is built from a set of components on top of the Solaris basic kernel. Those components include most OS services, from file system support to global process management and networking management. The programming and communications framework provides support for implementing the components and the communication between components. The framework includes a programming model, a compiler, and run time support for component implementation.

6.1 Programming model

Solaris MC components require a mechanism for accessing them both locally and remotely, and to determine when a component is no longer used by the rest of the system. At the same time, it is essential that each new component have a clearly specified interface, permitting its maintenance and evolution. These two requirements led us to decide on the adoption of an object oriented approach to the design of Solaris MC.

From the available possibilities we decided to adopt the CORBA (*Common Object Request Broker Architecture*) [18] object model, as the best suited for our purposes. CORBA is an architecture with mechanisms for objects to make requests and receive responses in a heterogeneous distributed environment, somewhat similar to RPCs. CORBA provides a strong separation between interfaces and implementations. In CORBA, an interface is basically a set of operations and each object accepts requests for the operations defined by the associated interface. How a given object implements an interface is up to the implementor of the particular object. CORBA also includes reference counting. In order to perform a request on an object, the client code must obtain a reference to that object, allowing the system to keep track of the number of references.

Interfaces are defined by using CORBA's *Interface Definition Language* (IDL) [23]. IDL allows the definition of interfaces by specifying the set of operations the interface accepts (similar to C function declarations), as well as the set of exceptions any given operation may raise. Interfaces can be composed using *interface inheritance* mechanisms, including multiple inheritance. Client and server object implementation code can be written using any programming language for which a mapping from IDL has been established. Currently there are a few such programming languages, including C and C++. We decided to use C++ as it provided the best match for the CORBA object model.

Every major component of Solaris MC is defined by one or more IDL specified object type. All interactions among the components are carried out by issuing requests for the operations defined in each component's interface. Such requests are carried out independently of the location of the object instance by using our own ORB *(Object Request Broker)*, or runtime. When the invocation is local (within the same address space), it proceeds like a procedure call. When the invocation crosses domains (across address spaces or nodes), the invocation proceeds essentially as an RPC, where the client code uses stubs, and the server (implementation) code uses skeleton code to handle the call.

The stubs used by the client code, as well as the skeletons used by the server code are generated automatically from the IDL interface definition by a CORBA IDL to C++ compiler. We currently use a modified version of the Fresco IDL to C++ compiler [13].

6.2 The Run time System

The different components of the system communicate using the services of the ORB. The main functions of the Solaris MC ORB are reference counting, marshaling/unmarshaling support, remote request support (RPC), and communication fault recovery. The three main goals of our ORB architecture are to provide an efficient object invocation mechanism, easy configuration of clusters, and support for high availability.

Solaris MC's ORB is composed of three layers: the handler, the xdoor, and the transport layer (Figure 5). Each object reference is associated with a handler. The handler is responsible for preparing inter-domain requests to the object whose reference it handles. A handler is also in charge of performing marshaling of its associated references, as well as of local (to an address space) reference counting. The handler layer implements the subcontract paradigm [21], providing flexible means of introducing multiple object manipulation mechanisms, such as zero-copy.

In order to perform an invocation on its object, a handler is associated with one or more *xdoors*. The xdoor layer implements an RPC-like inter process communication mechanism. This layer extends and builds on the functionality of the Solaris doors mechanism. With xdoors, it is possible to perform arbitrary inter-domain (intra- or inter-node) object invocations following an RPC scheme. References to xdoors are carried out with invocations and replies, permitting the ORB to locate particular instances of implementation objects. The xdoor layer implements a reference counting mechanism for ORB and application structures. Together, the handler and xdoor layers support

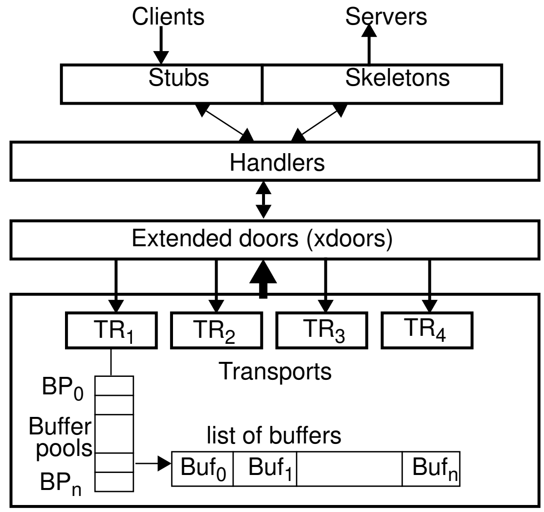

Figure 5. The different layers of the ORB.

efficient parameter passing for inter-address-space, inter- and intra-node invocations.

The transport layer defines the interface that a transport (such as Ethernet or Myrinet [4]) has to satisfy to be used by the xdoor layer. Transports implement reliable *sends* of arbitrary length messages. It is also possible to register receive functions with a transport. A transport has a set of buffer pools with lists of buffers, waiting to be filled by incoming messages. Arriving messages are stored in buffers from the buffer pools specified in the message's headers. Thus, the transport's interface is both simple and sufficiently powerful to support highly efficient object invocation, providing message delivery with scatter and gather capabilities through the buffer pools. A Solaris MC system can support several ORB transports at the same time. While our main goal is to support closely connected clusters, this capability makes it easy to configure the system to have some long distance links within the cluster.

Together, the three layers provide support for efficient parameter marshaling and passing, making it possible to implement zero-copy schemes for inter-node communications when the communications hardware supports it.

To support high availability, the xdoor layer never aborts an outstanding invocation unless a failure in the cluster is detected. In that case, outstanding invocations to failed nodes are aborted, and an exception is raised which can be caught by the handler layer, or by the component code itself. Services needing high availability make use of the information in the exception either directly or by means of special handlers. In addition to this failure reporting, the xdoor layer implements a reference counting algorithm that can recover after node failures. For efficiency, the algorithm is optimistic in nature, performing most of its work when an actual node failure is detected.

The ORB permits both kernel- and user-level communication. The ORB is implemented as a loadable kernel module to be used by the code residing in the kernel, and as a library, to be used by code executing in a user-level process. Most of the code used in both cases is identical, differing mostly in the xdoor and transport layers. Calls which target objects served by a user-level process are routed through the kernel xdoors.

6.3 C++ in the kernel

All Solaris MC extensions are implemented in C++, and are incorporated into a Solaris kernel as loadable modules. In order to mix the C++ code with the existing kernel, it was necessary to create a special loadable module containing the C++ runtime support. The basic task of the loadable module is to provide some new relocation types to the kernel dynamic linker. It also supports the “new” and “delete” operators, as well as the C++ exception handling mechanism, which in turn is used to implement our ORB’s exceptions.

So far we have had no problems with code size or speed, finding no major difference with C compiled code. We use C++ not only for writing the components defined with the IDL interfaces, but also any other code performing auxiliary or complementary functions. We make conservative use of C++ features, only using those with sufficient advantages for their cost. Thus, we do not use virtual base classes (the current compiler implementation makes them very space-inefficient), or return objects by value. On the other hand, we find C++ exceptions extremely useful. Exceptions are extensively used throughout our code to signal abnormal conditions and errors.

7. Support for High Availability

Solaris MC integrates the support for high availability into the operating system. Solaris MC divides the responsibility for high availability into several layers: failure detection and membership service, object and communication framework, reconfiguration of system services, and reconfiguration of user level programs. A more complete description of the high availability support in Solaris MC will be described in a forthcoming technical report. Here we provide a brief description of the architecture and some examples.

At the lowest level, a cluster membership monitor (CMM) detects a communication or node failure. The CMM informs the ORB that a cluster reconfiguration is in progress. The CMM then uses a distributed membership protocol to reach a global agreement on the current cluster configuration. Once an agreement is reached, the ORB is contacted. In turn, the ORB invalidates all client xdoors for objects residing in the failed node. Furthermore, the ORB runs a reference recovery protocol which removes all object references held by processes on the failed node.

Distributed system services can learn about a failure in two ways. First, they may get an exception indicating the failure of the node servicing a currently invoked object reference. Second, they can use special handlers which are notified when the xdoor they selected to perform an invocation has been invalidated as a result of a failure. For example, when a node caching file pages crashes, the pager object receives an *unreference* upcall. The pager then cleans up any locks held by the cache, allowing other nodes to access the file.

Figure 6 illustrates one system configuration to make Solaris MC system services highly available. The figure shows a four node system configured as an NFS server connected to an Ethernet. A file system is stored in a disk volume mirrored on two disks. The disks are dual ported and connected to a pair of nodes. One node is designed as the primary server for the file system (Node 3); the other is a backup (Node 4). A journaling file system is used to minimize file system recovery time. The file system is exported via NFS to remote clients (C1). The file is also accessed by processes P1 and P2 running locally on the cluster. The network interface *if1* is the primary interface for the cluster IP address; *if2* is a backup.

If Node1 crashes, Solaris MC relocates the IP address to if2 on Node 2. The crash is transparent to P2 and the remote NFS client. If Node 3 crashes, the backup volume manager on Node 4 takes over and recovers the disk volume. The backup IDL objects on Node 4 continue to serve the file system. The crash and takeover are transparent to P1, P2, and any remote client.

A special object handler is used to implement the file objects on Node 3 to facilitate transparent failover to the backup object. The special handler stores enough information in the object reference to perform a switchover from a primary object on Node 3 to a backup object on Node 4. The switchover is transparent to the holder of the object reference (e.g. a pxfs proxy vnode). In the example we assume that a journaling file system makes changes to the file non-vola-

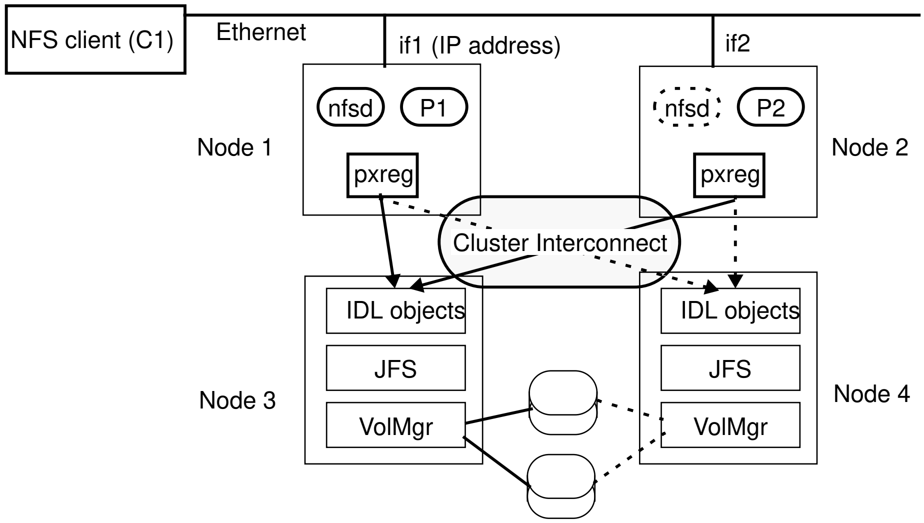

Figure 6. An example of high availability in Solaris MC. The system is configured as a NFS server with redundancy.

tile state recoverable. As for the volatile state (i.e., locks and tokens), there are basically three strategies to preserve it across a takeover by the backup node: backup solicits the volatile state from clients during takeover (long takeover), backup has an up-to-date volatile state (high runtime overhead), or the volatile state is in stable storage (high runtime overhead if no hardware support).

The focus on high availability differentiates Solaris MC from layered cluster software products, such as AT&T LifeKeeper, IBM HACMP 6000, HP MC/Service Guard, and Sun SPARCcluster PDB. Solaris MC is much easier to configure for high availability than the layered products, making the administration model scalable to a high number of nodes. Tight integration of key high availability protocols (such as the node membership protocol) with the low level communication framework results in faster failure detection and recovery than is achievable in layered systems.

8. Status

Most of the architecture described in this paper has been implemented: communication and object support, including the ORB and object reference counting; C++ support in the kernel; the global file system; most of process management, including remote exec, wait, signals, and the /proc file system; global networking support for TCP/IP, including extensions to the API to allow more than one server to service incoming connections on the same port; access to remote I/O devices, with no modifications to device drivers; a group membership and status monitor; and a set of extensible performance evaluation tools.

The system was initially developed on Solaris 2.4 and has since been moved to Solaris 2.5 with little effort. The prototype is fairly complete—we regularly exercise the system by running parallel makes and multiple copies of a commercial database server.

All implementation work is done in C++, and all new interfaces are defined using IDL. With the exception of a few minor changes to the kernel proper, the bulk of Solaris MC extensions consist of a set of loadable modules or user-land servers.

The hardware prototype currently consists of sixteen dual-processor SparcStation 10 and 20 machines, partitioned into two or more clusters. The developers' workstations act as front-end machines to the Solaris MC cluster. We use a variety of networks as the system interconnect, including 100baseT ethernet and Myrinet [4].

9. Related Work

Solaris MC uses ideas from earlier distributed operating systems such as Chorus Mix, LOCUS, MOS, OSF/1 AD TNC, Spring, Sprite, and VAXclusters. There are, however, significant differences in our approach, compared to previous systems:

- Solaris MC shows how a commercial operating system can be extended to a cluster while keeping the existing application base.
- Solaris MC emphasizes high availability.

- Solaris MC uses an object-oriented design. The system was built with the Corba object model.
- Solaris MC uses new ideas from Spring, especially filesystem and virtual memory ideas.

For example, unlike systems such as Spring, and Sprite, Solaris MC builds on a commercial operating system while maintaining binary compatibility with a large existing application base. Also, most of the other systems, with the notable exception of VAXclusters, do not emphasize high availability. Finally, Solaris MC introduces object-oriented techniques based on the Corba object model, and builds on the experience of the Spring system.

10. Conclusions & Future Work

We have built a prototype operating system that provides a single-system image for distributed tightly-coupled hardware systems. The prototype consists of a set of extensions to a commercial operating system. The resultant system thus leverages all existing applications of the OS and enables the OS to extend to a new class of computers. By extending an existing operating system, rather than writing an entirely new one, we were able to leverage off the existing OS code base and device drivers, dramatically reducing the development effort.

We made the early decision to build our system using the Solaris system in order to leverage the large investment in application and system software. Several Solaris features made our job easier, including loadable modules and dynamic linking, both for the kernel and user processes; the basic system VFS and DDI interfaces; the multi-threaded architecture of the system; and the new door IPC mechanism. On the other hand, there are portions of the system that are difficult to extend to a clustered system, including the STREAMS framework because many existing modules do not adhere to the framework and assume that they are running on a shared memory system. Some UNIX semantics are also difficult (but not impossible) or are inefficient to extend to a clustered system, including POSIX controlling terminal and sessions semantics, file descriptor sharing, and fork semantics.

Our experience so far with the use of IDL and CORBA to design and implement Solaris MC is very positive. The use of IDL gives us a collection of clearly defined interfaces, and the IDL to C++ translator conveniently creates the glue we need to perform arbitrary service requests from our components. An additional advantage of using IDL is the little effort it took us to adapt Spring technology to Solaris MC.

The structure of Solaris MC ORB allows us to create special handlers to efficiently transmit bulk data and other user-defined structures between nodes. These handlers make use of the transport layer to avoid extra copying of large amounts of data.

On the other hand, we find the CORBA model lacking in two areas. First, there is no straightforward way to perform asynchronous object invocations. Secondly, there is no support for performing group invocations to a set of objects supporting a common interface, a feature that would be useful for the cluster membership monitor. We had to go outside the CORBA model for group multicast.

More work remains for the future. High availability support needs more work. We plan to port a volume manager and provide more support for system administration. We also plan to move to faster interconnect hardware, and to measure and analyze the system performance. Finally, we plan to experiment with heterogenous clusters.

Acknowledgments

We would like to acknowledge Madhu Talluri, Remzi Arpaci, Francesc Munoz Escoi, and Aman Singla for their contributions to the project.

References

[1] T.E. Anderson, D.E. Culler, D.A. Patterson, “A Case for NOW (Networks of Workstations),” *IEEE Micro*, February, 1995.

[2] A. Barak and A. Litman, “MOS: A Multicomputer Distributed Operating System,” *Software—Practice & Experience*, vol. 15(8), August 1985.

[3] N. Batlivala, *et al.*, “Experience with SVR4 over CHORUS,” *Proceedings of USENIX Workshop on Microkernels & Other Kernel Architectures*, April 1992.

[4] N. J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, W. Su, “Myrinet: A Gigabit-per-Second Local Area Network,” *IEEE Micro*, February 1995.

[5] F. Douglis and J. Ousterhout, “Transparent Process Migration: Design Alternatives and the Sprite Implementation,” *Software--Practice & Experience*, vol. 21(8), August 1991.

[6] Intel Corporation, *Intel Paragon XP/S Supercomputer Spec Sheet*, 1992.

[7] Yousef A. Khalidi and Michael N. Nelson, "Extensible File Systems in Spring," *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP)*, December 1993.

[8] Yousef A. Khalidi and Michael N. Nelson, "An Implementation of UNIX on an Object-oriented Operating System," *Proceedings of Winter '93 USENIX Conference*, January 1993.

[9] Yousef A. Khalidi and Michael N. Nelson, "The Spring Virtual Memory System," Sun Microsystems Laboratories Technical Report SMLI-TR-93-09, February 1993.

[10] Yousef A. Khalidi and Michael N. Nelson, "A Flexible External Paging Interface," P*roceedings of the Usenix conference on Microkernels and Other Architectures*, September 1993.

[11] Steven R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", *Proceedings of '86 Summer Usenix Conference*, pp. 238-247, June 1986.

[12] N. Kronenberg, H. Levy, and W. Strecker, "VAX-clusters: A Closely-Coupled Distributed Systems," *ACM Transactions on Computer Systems*, vol. 4(2), May 1986.

[13] Mark Linton and Douglas Pan, "Interface Translation and Implementation Filtering," *Proceedings of the USENIX C++ Conference*, 1994.

[14] C. Maeda, B.N. Bershad, "Protocol Service Decomposition for High-performance Networking," *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP)*, December 1993.

[15] James G. Mitchell, *et al.*, "An Overview of the Spring System," *Proceedings of Compcon Spring 1994*, February 1994.

[16] Michael N. Nelson, Graham Hamilton, and Yousef A. Khalidi, "A Framework for Caching in an Object-Oriented System," *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems*, December 1993.

[17] Michael N. Nelson, Yousef A. Khalidi, and Peter W. Madany, "Experience Building a File System on a Highly Modular Operating System," *Proceedings of the 4th Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, September 1993.

[18] Object Management Group, *The Common Object Request Broker: Architecture and Specification,* Revision 1.2, December 1993.

[19] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch, "The Sprite Network Operating System," *IEEE Computer*, February 1988.

[20] G. Popek and B. Walker, *The LOCUS Distributed System Architecture*, MIT Press, 1985.

[21] G. Hamilton, M. L. Powell, and J. G. Mitchell, "Subcontract: A flexible Base for Distributed Programming," *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP)*, December 1993.

[22] Glenn C. Skinner and Thomas K. Wong, "Stacking Vnodes: A Progress Report." *Proceedings of the Summer 1993 Usenix Conference*, 1993.

[23] Sun Microsystems, Inc. *IDL Programmer's Guide*, 1992.

[24] Thinking Machines Corp., *The Connection Machine System: CM-5*, 1993.

[25] M. Yuhara, C. Maeda, B.N. Bershad, J.E.B. Moss, "Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages," *Proceedings of Winter '94 USENIX Conference*, January 1994.

[26] Roman Zajcew, *et al.*, "An OSF/1 UNIX for Massively Parallel Multicomputers," *Proceedings of Winter '93 USENIX Conference*, January 1993.

Yousef A. Khalidi (yak@eng.sun.com) is currently a Senior Staff Engineer and Principal Investigator at Sun Microsystems Laboratories. His interests include operating systems, distributed object-oriented software, computer architecture, and high speed networking. He is one of the principal designers of the Spring operating system, and a co-winner of Sun's President Award in 1993. He has M.S. and Ph.D. degrees in Information and Computer Science from Georgia Institute of Technology, where he was one of the principal designers of the Ra and Clouds operating systems.

José M. Bernabéu (josep@iti.upv.es) is currently a Senior Staff Engineer at Sun and a Visiting Professor from the Universidad Politécnica de Valencia, Spain, where he heads the Distributed Systems Group, and directs the "Instituto Tecnológico de Informática." His interests include operating systems, distributed algorithms, distributed object-oriented software, shared memory models, and reliability. He received an M.S. in Physics from the University of Valencia in 1982, and M.S. and Ph.D. degrees in Computer Science from the Georgia Insti-

tute of Technology, where he was one of the principal designers of the Ra and Clouds Operating Systems.

Vlada Matena (vlada@eng.sun.com) is a Senior Staff Engineer at Sun Microsystems Laboratories and PI for High Availability. His interests include OS, reliable distributed systems, and database technology. He has been a key contributor to Sun's database server technology as the principal designer of the SPARCcluster Parallel Database, Sun Database Excellerator, SunDBM, and NetISAM products. He received an SMCC Engineering Excellence Award in 1994. He received an M.S. in Computer Science from the Prague Institute of Technology in 1984.

Ken Shirriff (shirriff@eng.sun.com) is a Staff Engineer at Sun Microsystems Laboratories. His interests include operating systems, cryptography, and fractals. He obtained a B.Math degree from the University of Waterloo in 1987 and M.S. and Ph.D. degrees in computer science from U. C. Berkeley where he worked on the Sprite operating system.

Moti N. Thadani (thadani@eng.sun.com) is a Staff Engineer at Sun Microsystems Laboratories. His interests include computer networking and distributed and object systems. Before joining Sun, he worked at Unisys, IBM, and Data General. He received an M.S. in Computer Science from Tulane University and a B.Tech. in Electrical Engineering from the Indian Institute of Technology, New Delhi.