Extending a Traditional OS Using Object-Oriented Techniques

Jose Bernabeu, Vlada Matena, and Yousef Khalidi
Sun Microsystems Laboratories

Extending a Traditional OS Using Object-Oriented Techniques

Jose M. Bernabeu-Auban Vlada Matena Yousef A. Khalidi

Sun Microsystems Laboratories

Abstract

This paper describes a new object infrastructure designed for tightly-coupled distributed systems. The infrastructure includes an object model, interface-to-C++ translator, an object request broker (ORB) based on the CORBA architecture model, and a transport layer. The infrastructure allows the use of multiple data marshalling handlers, includes object recovery features, and provides remote communication through a mechanism called xdoors.

The object infrastructure has been used to extend the Solaris™ operating system into a prototype clustered operating system called Solaris MC[1]. This illustrates how the CORBA object model can be used to extend an existing UNIX implementation into a distributed operating system. It also shows the advantages of defining strong kernel component interfaces in the IDL interface definition language. Finally, Solaris MC illustrates how C++ can be used for kernel development, coexisting with previous code.

1. Introduction

Trends in the computer industry indicate that large server systems will tend to be built as distributed memory servers with nodes that are small shared memory multiprocessors. The fundamental reason to build these distributed memory clusters is to obtain scalable throughput and high availability. In addition, by building such systems out of standard commodity nodes, cost-effective systems can be built to serve a particular need.

The model of organization for such distributed systems is still a subject of research. One approach is to view the system as a collection of independent computing nodes that use standard networking protocols for communication. Unfortunately, the operating system in this approach provides little support for building clustered applications and administering the system.

At the other extreme, it is possible to take an existing shared memory OS, devised for a shared memory system, and port it to a distributed memory environment. However, it is very difficult to build a highly-available system with such an approach. Also, special support in the networking hardware would be needed to implement cache coherent memory. In addition, performance would be limited unless the operating system was extended to include knowledge of the non-uniform memory access times between nodes.

In between this two extremes there are several possibilities. One of the most promising is to extend an existing OS to work as a distributed OS that doesn't require shared memory. This allows one to build a scalable cost effective multicomputer out of off-the-shelf component nodes, by connecting them through standard high speed interconnects, without the need to have special memory caching hardware. The cluster can then be programmed and managed as a single computer, rather than making the distributed nature of the system available to application programmers.

The Solaris MC project extends the Solaris operating system to multi-computers (clusters) with distributed memory. The main goal is to provide a single-system image of the cluster, while maintaining compatibility with previous application programs. Another requirement is to make the main services of the OS, including the file system and networking, highly available in the presence of failures of one or several nodes.

In such a distributed environment, every service provided by the system uses the communications infrastructure. The adoption of a suitable interprocess communication model, and its associated communications framework was a key design issue in Solaris MC. On the one hand we needed a model allowing us to encapsulate existing kernel components within clean interfaces, and to be able to access them remotely. On the other hand we wanted an efficient

1. Solaris MC is the 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.

communication mechanism that can be tailored to different cluster interconnects. Early in the design phase we made the decision to adopt the CORBA architecture [11], and its associated object model. This led to the need to design an *Object Request Broker* tailored to the needs of a multicomputer OS such as Solaris MC.

Solaris MC borrows some of the distributed techniques developed by the Spring object-oriented OS [10] and migrates them into the Solaris UNIX system. Solaris MC imports from Spring the idea of using a CORBA-like object model [11] as the communication mechanism, the Spring virtual memory and file system architecture [5, 6], 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 object-oriented OS like Spring.

The rest of the paper is organized as follows. In Section 2 we give an overview of Solaris MC. Section 3 discusses the requirements which gave rise to the particular design of our ORB. In Section 4 we present the main components of the framework. Section 5 describes the handlers, Section 6 explains how extended doors (xdoors) provide communication, and Section 7 gives a sketch of the reference recovery mechanism used by the ORB. Section 8 shows how the framework is used to implement a distributed file system. In Section 9 we describe the status of the implementation. Finally, we conclude the paper in Section 10.

2. Overview of Solaris MC

Solaris MC [7] is a prototype distributed operating system for multi-computers 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 binary and programming interfaces as Solaris, running unmodified applications.

Solaris MC has been designed to run on multi-computer systems formed by independent nodes linked through a high performance interconnect. Thus a Solaris MC system can be seen as a “tightly coupled” distributed system.

The components of Solaris MC are implemented in C++ through a CORBA-compliant object oriented framework with all new services defined by the IDL interface definition language. Objects communicate through a runtime system that borrows from Solaris doors and Spring subcontracts.

Solaris MC builds upon the Solaris kernel by accessing its components through newly defined interfaces. There are, thus, two sets of interface boundaries in Solaris MC:

- New interfaces defined using IDL and built using an object-oriented model. We use a version control scheme based on interface inheritance, and have clearly defined failure modes.
- Existing Solaris interfaces where the implementation of the new IDL-based interfaces plugs into the system. The *vnode* [8] interface is an example of such an interface.

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 distributed process management and a global /proc file system. The external networks are 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.

2.1 Existing Solaris interfaces

The implementation of the new IDL-based interfaces plugs into Solaris using existing UNIX interfaces. For example, the implementation of the distributed file system interface communicates with the existing system through the vnode interface.

The use of existing Solaris interfaces adheres to the following guidelines:

- The Solaris interfaces are used by the implementation of the IDL interfaces, but new extensions are done at the IDL level.
- Clients of the new extensions only see the IDL interfaces, not the underlying implementations.
- Only (relatively) well-defined Solaris interfaces are used. For example, we use the vnode interface, but we do not touch the internals of the Unix file systems.

The above guidelines are formulated to strike a balance between using the existing system code and achieving our goal of migrating the system towards object-oriented IDL-based interfaces. Figure 1

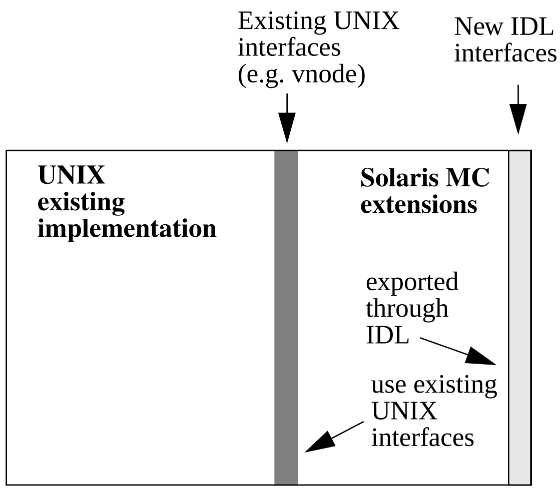

FIGURE 1. New Solaris MC extensions use IDL but build on the existing UNIX system

summarizes the relationship of new Solaris MC extensions to the existing UNIX implementation.

Our intention is to slowly migrate more and more of the existing system interfaces toward IDL.

3. Object Framework Requirements

The object framework underlies the Solaris MC system. In order to build our distributed system, we identified the following framework requirements:

- Support for component encapsulation through strongly-typed interfaces, with well defined invocation semantics and clear failure modes.
- Efficient local and remote object communication.
- Support for building highly available services.
- Ease of integration into UNIX kernel.
- Adaptability to different networking technologies.

Well defined interfaces are essential for long-term system maintenance and evolution. We built well defined interfaces around many parts of the existing kernel in order to take advantage of existing kernel implementations that lacked good interfaces.

The CORBA object model, together with the IDL language, provides sufficient support for defining new components in the system. The CORBA standard also provides a mechanism for remote invocation of the operations in an interface. This is used to communicate between different Solaris MC nodes.

The implementation language must both be able to invoke existing Solaris kernel code, and be suitable for implementing the new code. We use C++ for the implementation language since C++ code can easily call existing kernel code written in C, and C++ code can be easily generated by translation from the IDL specification.

The performance of the communication mechanism selected is an important factor in the performance of the a distributed OS. There are several CORBA-based products on the market right now. However, such products have been designed with user level applications in mind, without considering multicomputer requirements. Thus, it became necessary to design and build an ORB tuned to the needs of Solaris MC.

The communication infrastructure interface is extremely important for good performance, as it critically affects the performance of any inter-node communication mechanism implemented on top of it. In addition, since networking technologies are rapidly evolving it would be an error to design our communication infrastructure around a particular transport.

Finally, in a multicomputer OS, it is not feasible to allow the failure of one component to cause the failure of the whole system. Thus, it is necessary to provide a robust communication infrastructure to allow OS services to withstand system failures.

The above requirements led us to the following decisions:

- Adopt the CORBA object model and the IDL language for interface specification.
- Define an architecture to support the development of highly available services and applications.
- Use C++ as the implementation language.
- Implement our own ORB to provide good performance and integration with UNIX.
- Define a layered structure for the ORB architecture, with well defined interfaces between levels.
- Define a transport facility with both reliable and datagram message delivery facilities. The transport encapsulates all interconnect-specific implementation details.
- Provide support for global reference counting, and automatic notification of the absence of references within the ORB.
- Use a *cluster membership monitor* (CMM) to reconfigure the cluster after a fault is detected. The CMM is used to recover the references possibly lost due to the fault, allowing the ORB to provide strong RPC semantics with clear failure modes.

The result of the above decisions is the Solaris MC object framework.

4. The Solaris MC Object framework

Solaris MC is built using a set of components that extend the base Solaris kernel. The components include most OS services, from file system support to global process management and networking management. The components are implemented as dynamically loadable kernel modules and as user-level servers.

The object infrastructure provides support for implementing the components and the communication between components. The framework includes

- an object model,
- an IDL to C++ compiler,
- an object request broker (ORB), and
- a low-level communication transports.

The rest of this section describes these four components.

4.1 Object Model

Solaris MC components require a mechanism to access each other both locally and remotely, as well as a mechanism to determine when a component is no longer used by the rest of the system. At the same time, each new component must have a clearly specified interface that permits its maintenance and evolution.

Interfaces are defined by using CORBA's *Interface Definition Language* (IDL). 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 the ORB, or runtime. When the invocation is local (within the same address space), it proceeds as 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 CORBA standard also defines some elements outside the basic object model (specifically, the so-called basic object adaptor, a dynamic invocation interface, and a set of object services). We do not provide these features, as we felt that their potential usefulness within Solaris MC did not justify their implementation cost.

4.2 IDL to C++ compiler

The stubs used by the client code and the skeletons used by the server code are generated automatically from the IDL interface definition by a modified version of the Fresco CORBA IDL to C++ compiler [9].

4.3 Object Request Broker

The components of the system communicate with each other 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 failure recovery. The two main goals of our ORB architecture are to provide an efficient object invocation mechanism, and support for high availability.

Solaris MC's ORB is composed of three layers: the handler, the xdoor, and the transport (Figure 2). Each object reference is associated with a handler, which 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 part of the subcontract paradigm [3], providing flexible means of introducing multiple object manipulation mechanisms, such as zero-copy and serverless objects. Section 5 describes object handlers in more detail.

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.[2] With xdoors, it is possible to perform arbitrary object invocations (intra- or inter-node) following an RPC scheme. References to xdoors are carried out with invocations and replies, permitting the ORB to locate particular instances of implementation objects. Together, the handler and xdoor layers support efficient parameter passing for inter- and intra-node invocations. Xdoors are explained in more detail in Section 6.

The xdoor layer implements a fault-tolerant distributed reference counting mechanism for ORB and application structures. It employs a cluster member-

2. The doors mechanism is a fast inter-process communication mechanism in Solaris 2.5 that is based on the Spring doors IPC [10].

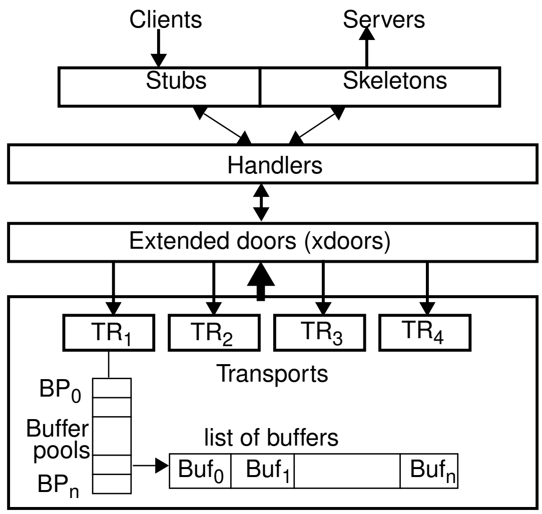

FIGURE 2. Organization of the ORB

ship monitor (CMM) to detect faults and reconfigure the cluster. The same services are also employed to provide clear RPC semantics. In Section 7 we present the general scheme used to recover from faults.

4.4 Communication Transports

The transport layer defines the interface that a communication channel has to satisfy to be used by the xdoor layer. The communication channel would typically be an inter-node high-speed network such as Myrinet [1].

The ORB assumes that the transport implements reliable *sends* of arbitrary length messages. It also assumes the presence of a datagram service, which is used to deliver the CMM messages. The marshaling functions of the ORB request the needed buffers from the particular transport that the request will use. This enables the system to select the best memory to store the marshaled parameters.

To implement efficient parameter passing and zero copy schemes, a transport is capable of gathering information from different buffers to compose an outgoing message. The Solaris MC ORB also assumes that the transport can scatter different pieces of a message into different destination buffers. To do so, the transport offers a set of buffer pools with lists of buffers, waiting to be filled by incoming messages. Arriving messages are stored in the buffers from the buffer pools specified in the message's headers. Since the sender can specify the buffers into which the data should be received, the receiver does not have to copy the received data.

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.

Together, the handler, xdoor, and transport layers provide support for efficient parameter marshaling and passing. They also make it possible to implement zero-copy schemes for inter-node communications if the communications hardware meets certain minimal requirements.

5. Object Handlers

Each object reference in our framework (whether server or proxy) is associated with an object handler. The functionality of our object handlers is similar to that of *Subcontract* in Spring [2], and *Exchanges* in Fresco [9], providing flexible reference counting and marshaling.

An object handler is involved in all object reference manipulations that are independent of the type (interface) of the object. Thus, the handlers are responsible for keeping accurate reference counting for the objects and delivering an *unreferenced* call to the object when no more references exist. The calls for narrowing, duplicating, and releasing an object reference are all routed through the object reference's handler, which uses them to keep the reference count.

A handler keeps count of the existing references only within the domain (address space) of the handler. The extended doors layer is responsible for keeping count of the number of domains in the distributed system holding a reference to a given extended door. Together, both reference counts keep track of the existing references to a given object (Section 6).

The handler is also involved when marshaling and unmarshaling object references. This makes it possible to create specific handlers for special objects. The handler type to be associated with a particular object implementation is specified when the particular implementation class is defined.

Each handler type comes in two versions, one for the client side and another for the server side. It is straightforward to add new handlers to the system by deriving from the abstract handler class, implementing each of the methods in the interface. The system maintains a small list of system defined handler types. When an object reference is marshaled, the handler type code is included in the marshaling

```
class handler : .... {
        ...
    void unreferenced(ULong count);
    void accept_unreferenced();
    void marshal(service&, CORBA::Object_ptr);
    void invoke(CORBA::Object_ptr,
                ArgInfo&, ArgValue*);
    void* narrow(CORBA::Object_ptr,
                CORBA::TypeId,
                ORB::ProxyCreator);
    void* duplicate(...);
    void  release(CORBA::Object_ptr);
}
```

FIGURE 3. Object Handler interface.

information, making it possible for the receiving node to associate the right type of handler with the object reference it receives. Figure 3 shows part of the object handler interface.

When an inter-domain request is being made on an object, its client-domain handler takes control of the parameter marshaling process through the *invoke* method. Marshaling is accomplished by using helper functions implemented by the ORB, as well as type information generated by the IDL compiler.

A handler marshals a parameter list by writing the marshaled image of each parameter in the *main marshal stream*. The main marshal stream is a logically sequential array of bytes where a domain-independent representation of the parameters is written in the order the parameters are specified in the operation's signature. It is possible to avoid copying large pieces of data by concatenating several buffers at the end of the main marshal stream, forming what we call the *offline buffers*.

To marshal object references, a *special* buffer, separate from the main marshal stream, is allocated. An object reference is usually marshaled in two parts: The first part contains the handler type associated with the object, which is written in its corresponding place within the main marshal stream. The second part contains the internal xdoor number associated with the handler, and is left in the special buffer. The handler for the reference being marshaled determines which additional information is sent within the main marshal stream, the special buffer or as special offline buffers. The only restriction is that the special buffer must contain internal xdoor references only. The above scheme gives the handler writer ample room to implement different marshaling schemes.

In order to release the xdoor associated with a handler when there are no more references to the object within the handler's domain, the handler notifies the associated xdoor that there are no more references. To avoid races, the handler does not go away until the xdoor notifies it with the *accept_unreferenced* method.

On the server side, a similar protocol is carried out from the xdoor to the handler. When the handler decides there are no more references for the object, it invokes the implementation's *unreferenced* method, which is a method on every implementation object. The implementation object can choose what to do in this case.

When the xdoor detects that there are no more remote references, it notifies the handler by calling the unreferenced method. However, the xdoor does not go away until it is also notified through its *accept_unreferenced* method.

Currently we have defined four different types of handlers and are working on a few more types. The types currently defined are the *simple* handler, the *counter* handler, the *standard* handler, and the *bulkio* handler.

The simple handler is used for objects that represent permanent well-known services (e.g., the primary system name service). It does not make sense to keep a reference count for this type of object, so the reference count functionality is left out of the simple handler.

The counter handler is used only on objects which are not going to be exported beyond the limits of a given domain. Usage of this type of handler permits taking advantage of the garbage collection facilities of the Solaris MC ORB even for objects local to an address space.

The standard handler is used for most objects in the system. The handler associates an object with one xdoor, and keeps track of local reference counting, delivering an unreferenced call after the total reference count drops to zero. This handler marshals a reference to its object by simply writing the xdoor number in the xdoor special buffer, followed by the handler type id.

Finally, the bulkio handler implements a zero-copy scheme for page frame lists. The bulkio handler associates a list of page frames with a (pseudo) object reference. When the object is being marshaled in a call to some other interface, the handler marshals the object by attaching an offline buffer containing the list of page frames associated with the object's reference.

The above examples show how the structure of the ORB can be used to implement a variety of schemes for handling object references and parameter passing. Currently we are working on a *high availability* handler that permits a client to transparently switch over to a secondary copy of an object when the primary fails.

6. Extended Doors

6.1 Overview

Extended doors (xdoors) implement the basic RPC mechanism of our framework, relying on transports that provide reliable message delivery. Xdoors also maintain reference counts in the distributed system.

An object can receive an invocation from another domain only if there is an xdoor that refers to it. Each xdoor is associated with just one object (with its handler, to be precise), which may be in the same domain as the door (server xdoor) or may be in another domain (client xdoor).

Xdoors are identified by an integer number within a domain. Client xdoors refer to the location of their server by the server xdoor's local identifier, and the server node identifier. The xdoor layer guarantees the uniqueness of xdoors. That is, any given domain will have at most one client xdoor referring to the same server xdoor.

The xdoor layer maintains reference counts while ensuring the following properties:

- ***Liveness***: Once no node other than the server node has a reference to a server xdoor, an unreferenced call is sent to the handler of the server xdoor. The handler itself may decide to send an *unreferenced* call to the object implementation if the total number of local references is also zero.
- ***Safety***: No unreferenced call is sent to the handler associated with the server xdoor while there is still a valid client xdoor associated with the server xdoor anywhere in the cluster.

It is important to note that the above properties are maintained even after cluster re-configuration (see Section 7).

6.2 Xdoor reference counting

We implement an optimistic reference counting protocol that fulfills the properties mentioned above without imposing a significant overhead on the system during normal operation. In essence the protocol proceeds as follows:

- When the xdoor layer performs the translation from internal xdoor names to global xdoor names, it increases the external xdoor count of each xdoor whose reference is being included in the message.
- When a domain receives an xdoor reference, if it needs to create a new client xdoor (one that does not yet exist in this domain), it registers the xdoor with its corresponding server xdoor. If the xdoor reference is not new, a de-registration notification is sent right away.
- When a registration is completed, the client xdoor knows that the external xdoor counter of the server has been increased on its account, and can now proceed to send a de-registration notification to the domain that gave it the reference in the first place.
- The sending of a registration or de-registration notification is performed asynchronously.
- When a client xdoor is notified by its handler that there are no more valid references to its object, and its external counter is zero, it sends a de-registration notification to its server xdoor.
- When the external xdoor counter of a server xdoor drops to zero, it calls the unreferenced method of its associated handler. As discussed in the case of the handlers, a special two-phase protocol has to be maintained between the xdoor and its handler to avoid race conditions. Thus the xdoor space will not be reclaimed unless its handler calls it back with an accept_unreferenced method.
- When an xdoor receives a registration notification it increases its external count.
- When an xdoor receives a de-registration notification, it decreases its external xdoor count.

It is straightforward to see that the above protocol satisfies the properties given above for the reference counting protocol.

6.3 Object Invocation Flow

In Figure 5 we show the flow of execution of a standard invocation. The stub's job is to prepare the arguments in a predefined structure which is passed to the handler by calling its *invoke* method. The handler performs parameter marshaling, after which the xdoor is given a buffer list with the invocation data through the xdoor's *request* method. The xdoor function at this point is to get an endpoint from the transport for the node where the server xdoor resides. At the same time it inserts the xdoor reference in the invocation

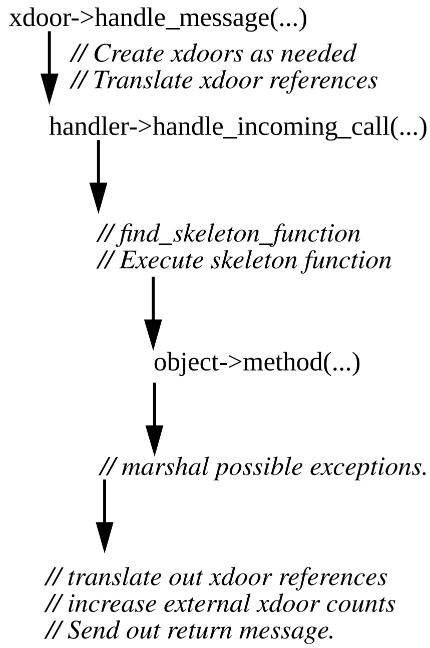

FIGURE 4. Flow of control on the server.

message, and translates the internal xdoor references being passed to cluster-wide xdoor names. When doing so, the external xdoor reference count for the xdoors being passed is increased. After xdoor id translation, the invocation message is ready to be sent. The endpoint's *send* method is used to send the message to the server node.

On return, xdoor references are first extracted from the return message, and client xdoors are created if needed. Then control is returned to the handler, which unmarshals the return parameters. Finally, the end of the stub takes care of preparing the return arguments, and the return value of the object's method call.

On the server side, control flows as shown in Figure 4. First, the invocation message is processed by the xdoor, which inspect the xdoor references being imported, creating client xdoors as needed. At the same time, xdoor references are translated to internal xdoor names.

Afterwards, the handler associated with the server xdoor is called using its *handle_incoming_call* method. This method just finds the skeleton function associated with the type under which the object is being invoked. The skeleton is invoked to perform unmarshaling of the parameters as well as invoking the implementation of the object.

Once the implementation returns, the return parameters are marshaled if no exception was produced. Otherwise, the exception itself is marshaled. At this point, the marshalled return message is ready within the xdoor code. The xdoor translates out the xdoor references being sent back, increasing their external xdoor counts. Finally, the resulting return message is sent back to the client node.

6.4 User-level ORB

The ORB as described so far can be used unchanged either from within the kernel, or as part of ordinary processes. However, this last possibility would force the ORB to consider each process as a "node" in the cluster, with a failure mode independent of every other "cluster" node. In addition, the xdoor layer

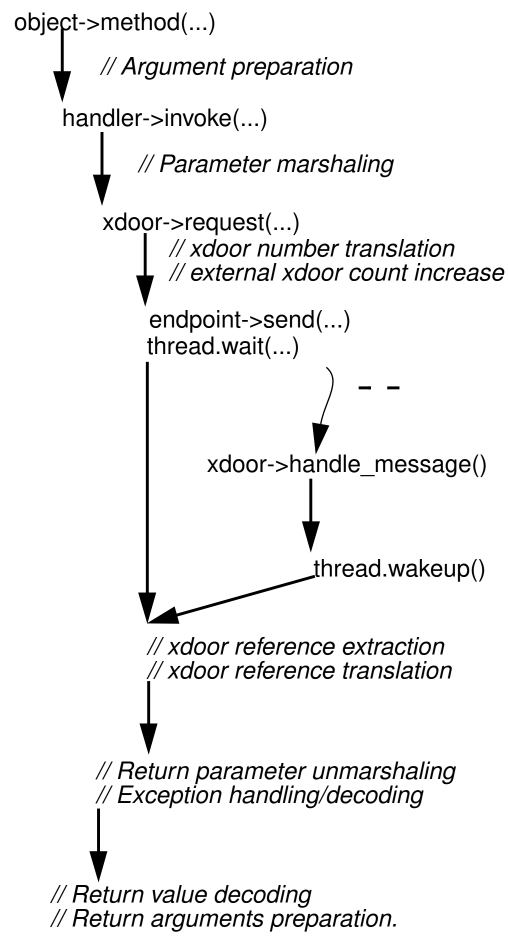

FIGURE 5. Execution flow for a standard invocation

cannot reside in untrusted user processes since it can generate a reference to any xdoor in the system.

Therefore, in user-level processes, we use the same ORB down to (and including) the handler level. We substitute the xdoor layer previously described, by a Solaris *door* mediated xdoor layer. A Solaris door is a secure capability-like end point that is used for fast interprocess communication. Thus, a user-level ORB relies on the Solaris door mechanism to perform invocation requests to arbitrary implementation objects, either within the local node, or to remote cluster nodes. Essentially, Solaris doors become a class of xdoors specialized in providing access to user-level implementation objects.

Note that the code is the same for clients, servers, stubs, and handlers when used in the kernel or in user-level processes. This was a conscious design decision, as we wanted to maintain the flexibility of placing a given server or client in the kernel or in user-level processes.

7. Object Framework Recovery

To support high availability, the xdoor layer never aborts an outstanding invocation unless a node is declared to be down. In that case, outstanding invocations to the failed node is aborted, and an exception is raised which can be caught by the handler layer or by the client code itself. To be more precise, an ongoing RPC is only aborted by the system in one of the following cases:

- *Insufficient resources*. The call is known not to have completed.
- *Invalid xdoor*. Such an xdoor may result after cluster reconfiguration, if its target domain is found to be down. The xdoor is tagged invalid, although not immediately deleted. As before, the invocation has not been executed.
- *Node Failure*. The invocation message of the RPC was sent out, and afterwards the cluster got reconfigured, determining that the target node is down. In such a case, the invocation may have been totally or partially executed on the failed node.

An RPC which has been started is never timed out. There are only two ways of returning from an RPC: receiving a reply, or receiving a notification that the other node is faulty. Thus our RPC mechanism implements exactly once semantics if the domain of the invoked object does not fail. In addition, only cluster node failures introduce the possibility of an undetermined execution status for an invocation. Services needing high availability make use of the information in the exception either directly or by means of special handlers (e.g. the high availability handler mentioned earlier on).

The ORB contains a *cluster membership monitor* (CMM) that is capable of detecting faults and reconfiguring the cluster. When the CMM detects a possible failure in the cluster, it runs a membership protocol during the cluster reconfiguration phase. The outcome of such protocol is a group of "surviving" cluster nodes, which form the new cluster.

After each cluster reconfiguration there are two basic elements whose state has to be recovered: the transport and the xdoor layer. Recovery proceeds in phases executed in lock-step by all surviving nodes and controlled by the CMM:

- Initially, the transports are notified of the new group membership. After such notification, transports can stop the send of reliable messages to the failed nodes. Send requests to failed nodes will raise an exception. Reliable messages arriving from failed nodes can be safely discarded.
- Afterwards, ORBs are also notified of the new membership. All ongoing invocations to failed nodes are aborted raising a *node_down* exception on each invocation. At the same time, all client xdoors referring to server side xdoors on failed nodes are flagged as stale. Further invocations through or making use of those xdoors will raise an exception. The ORB can function normally during this phase.
- The third phase recovers the reference counts. Notice that the server xdoor does not keep track of the location of its client xdoors. Thus, after a cluster reconfiguration, the ORB has to proceed through a reference regeneration phase before it can proceed passing references around again. The algorithm followed is optimistic in nature, performing most of its work whenever a fault is detected. Essentially, each node sends every other node its list of client xdoors, such that, node n only receives the list of xdoors whose server is in node n. All external xdoor counters for client xdoors are set to zero. During this phase, no ORB reference passing activity is allowed.
- Finally, normal operation is reestablished. Server xdoors check their external xdoor count at this moment. Those xdoors having a zero value for that count invoke the unreferenced call of their handler.

If at any point in the above protocol the ORB is notified by the CMM of another reconfiguration, the whole process is started again. At the end, the ORB and transport find themselves in a consistent state with the reference counting system functioning properly. Stale xdoor storage is reclaimed when the handler detects no more references in the domain.

8. Example Subsystem: The Proxy File System

One of the most important components of Solaris MC is its global file system, which makes 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 interface. PXFS interposes on file operations at the vnode interface and forwards them to the vnode layer where the file resides.

The proxy file system makes heavy use of the object infrastructure, exercising most of its features. The files system handles a large number of objects (each file is an object). Because the space overhead introduced by the ORB is small, the impact of the file system on system resources is minimized.

It is very common for different elements in the file system to invoke objects which reside in the same address space. The efficient intra-address-space invocations of the Solaris MC ORB are essential for the performance of the Proxy file system.

Finally, the file system has to move large amounts of data as memory pages. The ability to define new ways to marshal parameters through the definition of special handlers (in particular the bulkio handler), permitted us to implement paging-related operations that avoided superfluous copying of large amounts of data, thus increasing the performance of the file system as a whole.

9. Implementation Status

All functionality described in this paper has been implemented and tested in Solaris 2.5. All Solaris MC extensions are implemented in C++, and are incorporated into the Solaris kernel as loadable modules. In order to mix the C++ code with the existing kernel, it was necessary to create a 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.

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 pass 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.

Preliminary performance numbers are very encouraging. For example, an intra-address space object invocation consists of 9 SPARC instructions vs. 4 instructions for a local procedure call. We plan to report on the performance of the system in a future paper.

10. Conclusions

This paper has illustrated how an object system based on CORBA can be used to extend a traditional monolithic operating system into a distributed cluster operating system. Solaris MC is built as a set of extensions to the base Solaris UNIX® system and provides the same binary and programming interfaces as Solaris, running unmodified applications.

The Solaris MC object system provides several important features. It provides a transport-independent communications layer. It allows multiple object handlers with flexible marshaling and reference counting policies. It uses the xdoors mechanism for reliable RPC. Finally, it includes recovery mechanisms to handle node failures.

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.

Unlike "pure" object-oriented systems such as Choices [2] or Spring [10], Solaris MC is a hybrid system that builds on an existing commercial OS thus leveraging current investment in the OS, device drivers, and most importantly, applications.

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 reduced effort it took us to adapt Spring technology to Solaris MC. The structure of Solaris MC ORB allows us to create special handlers to transmit bulk data and other user-defined structures efficiently between nodes without extra copying.

Perhaps most importantly, the use of the ORB enabled us to provide the system programmers with well-defined and efficient reference counting and failure handling support, that would otherwise have to be programmed, tested, and maintained separately in each subsystem. Building, maintaining, and evolving distributed system is very difficult. By providing our object infrastructure support, we believe we are making the process of building distributed systems easier, less error prone, and in the long run, less expensive.

References

[1] 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.

[2] R. Campbell and Nayeem Islam, "A Parallel Object-Oriented Operating System." In *Research Directions in Concurrent Object-Oriented Programming,* MIT Press, 1993.

[3] 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.

[4] 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.

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

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

[7] Yousef A. Khalidi, Jose M. Bernabeu-Auban, Vlada Matena, Ken Shirriff, and Moti Thadani, "Solaris MC: A Multi Computer OS," *Proceedings of 1996 USENIX Conference*, January 1996.

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

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

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

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