A Fault-Tolerant CORBA Name Server

Silvano Maffeis*
Cornell University, Ithaca, NY and
Olsen & Associates, Zurich
maffeis@acm.org

February 27, 1996

Abstract

OMG CORBA applications require a distributed naming service in order to install and to retrieve object references. High availability of the naming service is important as most CORBA applications need to access it at least once during their lifetime. Unfortunately, the OMG standards do not deal with availability issues, and the naming services of many of the commercially available CORBA object request brokers introduce single points of failure. In this paper we describe the design and implementation of a replicated, highly-available CORBA name server that adheres to the OMG Common Object Services Specification. Our naming service can be replicated at runtime, while many applications are installing and retrieving object references. We compare our approach with the approaches taken by the ILU, NEO, Orbix, and DOME object request brokers. The performance of our name server is measured for various replication degrees.

Keywords: CORBA, Replication, Fault-Tolerance, Naming

1 Introduction

Many business-oriented, distributed applications need to accommodate several programming languages and to access legacy information systems. Heterogeneity, interoperability, and extensibility must thus be addressed by the system software used to develop next generation business applications. The Object Management Architecture (OMA) [12], proposed by the Object Management Group (OMG), aims at reducing complexity, lowering development costs, and at hasten the introduction of new applications. OMG plans to accomplish this through the introduction of the CORBA architectural framework [10], of a standard interface declaration language, of ORB gateways [9], and of Common Object Services [7]. These are the main components of OMA. Endorsed by more than 550 enterprises world wide, CORBA has become one of the most important standards for open distributed systems.

OMA allows developers to model and to implement distributed applications in an object oriented way. OMA objects are free from the ties imposed by operating systems and programming languages: Two CORBA objects residing on the ORBs provided by two different manufacturers can interoperate through ORB gateways, even when implemented in different programming languages. Applications can be structured in a highly modular way and become easier to maintain and to extend.

This sounds very enticing. However, *partial failures* — a fundamental problem of distributed systems — have not adequately been addressed by the OMG yet. No matter how carefully a CORBA application has been specified, implemented, and tested, its network objects will crash unexpectedly due to power outages, human lapses, hardware faults, and software bugs. CORBA applications should thus be prepared to deal with failures and to treat them as a *normal* occurrence. The OMG standards do not specify the behavior of an application in confront of network partitions and object failures. Fault-tolerance techniques such as object replication are not part of the OMA model. This is unfortunate as the OMG standards are becoming very popular.

To make things worse, many of the commercially available ORBs introduce single points of failure: The name server, an important component of the OMA, is often provided in the form of a singleton process. If that process fails, many applications cannot make progress any more.

*Supported by the Swiss National Science Foundation (Schweizer Nationalfonds), while on leave at Cornell University, Dept. of Computer Science.

In this paper we will argue that the provision of reliable OMA object services requires system support not available in today's ORBs [4]. The goal of our work has been to design and to implement Electra — a CORBA object request broker for reliable, highly available applications. Group communication, runtime replication of CORBA objects, consistent failure detection, and fault-tolerant object services are the key features of Electra. Electra runs atop of group communication subsystems like Horus [13] and Isis [1]. Electra can be ported to other communication subsystems without much effort.

An overview of the Electra ORB is given in [5]. For a description of the underlying replication and reliability protocols refer to [4, 6]. In [11] we describe a graphical availability manager for Electra that monitors, replicates, and restarts CORBA objects. This paper focuses on the design and implementation of a fault-tolerant OMG naming service for Electra.

2 The OMG Naming Service

The OMG Common Object Services Specification (COSS) [7] defines a federated naming service to maintain name-to-object mappings. The specification mainly consists of a textual description of how the service works and of its CORBA-IDL interfaces.

2.1 Terms and Definitions

In the OMA model, a server object that is able to handle requests is called an *object implementation*. We will also use the term *CORBA object*, or simply *object*, to refer to an object implementation. A proxy object used by client applications to access an object implementation is called *object reference*. The object reference is a local representative of a remote object implementation. Both object reference and -implementation obey to the same CORBA-IDL interface.

In the COSS specification, a name-to-object association is called a *name binding*. A name binding is defined relative to a *naming context*. A naming context is a CORBA object that maintains a set of name bindings. We shall use the terms naming context, naming service, and name server interchangeably.

To *resolve* a name is to pass a name to a naming context and to obtain the associated object reference. To *bind* a name is to install an object reference in a naming context under a certain name.

Because a context is like any other CORBA object, it can also be bound to a name in some context. This is how naming graphs (i.e., federated name spaces) are constructed. Figure 1 depicts such a naming graph.

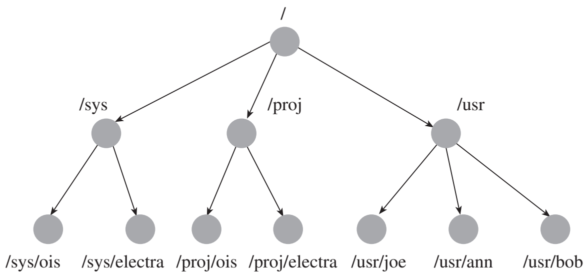

Figure 1: Sample Naming Graph. Circles denote context objects, arrows point from father-contexts to subcontexts.

A *compound name* defines a path in some naming graph to navigate the resolution process. Many of the operations defined on naming contexts take a `Name` object as parameter. A `Name` is a sequence of *components*:

```
<c1 ; c2; ... ; cN>
```

Each component, except the last, is used to access a context object; the last component denotes the bound object. Thus, name resolution is defined through the recursive relation:

```
ctx->resolve(<c1 ; c2; ... ; cN>) =
  ctx->resolve(<c1 ; c2; ... ; cN-1>)
    ->resolve(<cN>)
```

2.2 Naming Context IDL

The code fragment below contains a simplified version of the COSS naming context IDL specification:

```
// CORBA-IDL:
module CosNaming {
    typedef string Istring;
    struct NameComponent { Istring id; Istring kind; };
    typedef sequence <NameComponent> Name;
    enum BindingType { nobject, ncontext };
    struct Binding {
        Name binding_name; BindingType binding_type;
    };

    typedef sequence <Binding> BindingList;
    interface BindingIterator;

    interface NamingContext {
        exception NotFound { ...};
        exception CannotProceed { ...};
        exception InvalidName{ ...};
        exception AlreadyBound{ ...};
        exception NotEmpty{ ...};

        void bind(in Name n, in Object obj)
            raises(NotFound, CannotProceed, InvalidName,
            AlreadyBound);
        void rebind(in Name n, in Object obj)
            raises(NotFound, CannotProceed, InvalidName);
        void bind_context(in Name n,
        in NamingContext nc)
            raises(NotFound, CannotProceed, InvalidName,
            AlreadyBound);
        void rebind_context(in Name n,
            in NamingContext nc)
            raises(NotFound, CannotProceed, InvalidName);
        Object resolve(in Name n)
            raises(NotFound, CannotProceed, InvalidName);
        void unbind(in Name n)
            raises(NotFound, CannotProceed, InvalidName);
        NamingContext new_context();
        NamingContext bind_new_context(in Name n)
            raises(NotFound, CannotProceed, InvalidName,
            AlreadyBound);
        void destroy() raises(NotEmpty);

        void list(in unsigned long how_many,
            out BindingList bl, out BindingIterator bi);
    };
};
```

The `bind` operation installs a name-to-object binding, `rebind` overwrites an existing binding. To resolve a name binding, the `resolve` operation is invoked. To delete a name binding, the `unbind` operation is used.

`bind_context` names an object that is a naming context. Naming contexts that are bound this way participate in the resolution of compound names. `rebind_context` overwrites an existing binding. The `new_context` operation returns a naming context implemented by the context object the operation was invoked on. The `bind_new_context` operation works like `new_context`, except that the created context is bound to the name supplied as an argument. Finally, the `destroy` operation deletes a naming context.

COSS also specifies operations that can be used by applications to browse the entries in a naming context: The `list` operation returns the specified amount of names in the context the operation is applied to. `how_many` specifies the number of names to be returned. If more than `how_many` bindings are contained in the context, a `BindingIterator` object reference is returned. The `BindingIterator` allows the application to retrieve the remaining names through the following interface:

```
// CORBA-IDL:
interface BindingIterator {
    boolean next_one(out Binding b);
    boolean next_n(in unsigned long n,
        out BindingList bl);
    void destroy();
};
```

2.3 An Example

Applications use the `resolve_initial_references` operation defined in the OMG ORB Initialization Specification [8] to initially obtain a reference for the default naming context. The code fragment below illustrates how a server application initially obtains a reference to the default naming context, creates a fictive `FileServer` object implementation, and binds a reference to it under the name `"/usr/tom/filesrv"`:

```
// C++
main(int argc, char **argv) {
    CORBA::Environment_ptr env;
    CORBA::Object_var obj; // plain object reference
    CORBA::ORB_ptr orb =
        CORBA::ORB_init(argc, argv, "Electra_ORB");
    CORBA::BOA_ptr boa = orb->
        CORBA::BOA_init(argc, argv, "Electra_BOA");
    CosNaming::Name name; name.length(1);

    // obtain a reference to the default name server:
    NamingContext_var nc =
        NamingContext::_narrow(
            orb->resolve_initial_references("NameService"));

    // create and bind a FileServer object:
    _im_FileServer fs;
    obj = fs.create(...); // get an object reference
    name[0].id = "/usr/tom/filesrv";
    nc->bind(name, obj, env);
    fs.impl_is_ready();
}
```

The next code fragment illustrates a client application that binds to the `FileServer` object implementation:

```
// C++
main(int argc, char **argv) {
    int fd;
    CORBA::Environment_ptr env;
    CORBA::Object_var obj;
    CORBA::ORB_ptr orb =
        CORBA::ORB_init(argc, argv, "Electra_ORB");
    CORBA::BOA_ptr boa = orb->
        BOA_init(argc, argv, "Electra_BOA");
    CosNaming::Name name; name.length(1);

    NamingContext_var nc =
        NamingContext::_narrow(
            orb->resolve_initial_references("NameService"));

    FileServer_var fsp; // object reference to a FileServer
    name[0].id = "/usr/tom/filesrv";
    obj = nc->resolve(name);
    // handle exceptions ...
    fsp = FileServer::_narrow(obj); // downcast
    // access the FileServer
    fd = fsp->open("/tmp/test", "r", env);
    // ...
}
```

3 Adding Fault-Tolerance to Naming Contexts

This section starts with a description of the name serving mechanisms implemented in the ILU, NEO, Orbix, and DOME object request brokers. In particular, we shall focus on their fault-tolerance techniques. NEO, Orbix, and DOME are commercial CORBA request brokers, ILU is available in source code form at no fee. Afterwards, we shall elaborate on a set of requirements for a fault-tolerant, scalable CORBA name server.

3.1 Naming Context Implementations

3.1.1 ILU

ILU[1] from Xerox PARC provides a simple binding mechanism that employs multicast to find an object implementation registered under a certain name. There is no name server object *per se*; an object implementation simply listens for binding-requests that are multicast by the clients. A binding request contains the name of the sought object implementation. The implementation that is listening to that particular name will return its object reference to the client. A client application that wishes to bind to a certain object implementation submits a multicast and waits for the first matching object implementation to reply.

This scheme is elegant and fault-tolerant as no single point of failure is introduced. However, object implementations must be reachable by multicast, thus limiting the distribution of object implementations to a single subnet, unless a protocol like IP-Multicast is employed. Another drawback is that when an object implementation fails, its name-to-object mapping also disappears. This is unwanted in certain situations. Furthermore, with this scheme it is difficult to avoid that two incompatible objects start listening to the same name, since there is no naming service in the sense of a Naming Context object or object-group. Last but not least, the ILU naming service does not allow clients to browse the name-to-object mappings.

The ILU naming service does not comply with OMG COSS. However, we note that ILU allows programmers to replace this simple naming mechanisms with more sophisticated ones.

[1] ftp://ftp.parc.xerox.com/pub/ilu/ilu.html

3.1.2 NEO

Sun Microsystem's NEO[2] features a COSS-compliant naming context service. The naming service is provided by a set of cooperating server processes that can be distributed across machines. If one of the processes goes down, the part of the naming graph it manages becomes unavailable, but the other parts of the naming graph remain accessible. Thus, access to a particular node in the naming graph is vulnerable to a single point of failure. The run-time replication of NEO name servers is not supported yet.

3.1.3 Orbix

Orbix 2.0[3] from Iona Technologies does not provide a COSS compliant naming facility yet. Orbix presently implements a simple multicast-based binding mechanism which is similar to the one implemented in ILU. Iona is working on a fault-tolerant, COSS compliant name server that will be part of a future release of Orbix. The Iona engineers are experimenting with a primary/backup replication scheme, the naming service is in an early stage of development.

3.1.4 DOME

DOME[4] from Object Oriented Technologies, Ltd. provides a naming service, called Location Broker (LB), that does not comply with the COSS. DOME's name serving philosophy differs from the approaches taken by the other well-known ORBs: In DOME, there is no link between individual objects and a DOME LB.

[2] http://www.sun.com/sunsoft/neo/
[3] http://www.iona.ie
[4] http://goole.octacon.co.uk/onyx/external/oot.co.uk

A DOME system may contain multiple LBs, some will serve different parts of the naming space, and others will serve the same part for fault-tolerance and performance. When the system starts up, each ORB registers its details with each LB that it "knows about". Thus, LBs need have predefined, globally known addresses. The details of an ORB include information like its ORB identifier, its network address, and the C++ classes it supports. An ORB can support many classes, and a class can be supported by many ORBs.

An application contacts a LB to either obtain location details for a specific *ORB* (and not for a specific *object implementation*), or by repeated inquiries can get the location of the ORBs that support a specific C++ *class*.

The information maintained by a LB is retained in persistent store, so a crashed LB can be restarted and re-initialized with that information. Multiple LBs can be initialized with the same data, leading to a simple replication scheme: An ORB needs register with all of the replica. However, there are no mechanisms to ensure that such multiple registrations can be carried out atomically. If a client is looking for an ORB, it queries all LB it "knows about" until it finds the location details it is looking for.

In the author's opinion, the DOME fault-tolerance mechanisms are optimistic and work well only if the name-to-binding mappings are static, which is the case in DOME: An ORB registers its details when it comes up, this information is usually not modified. DOME provides no mechanisms for consistent failure detection or for the handling of partitioned networks.

3.2 Requirements

Following principles should guide the implementation of a COSS compliant `NamingContext` service:

- **Object Service**: The naming context is to be implemented in the form of an object service, and not by using a simple multicast mechanism as in ILU. This is to allow the browsing of name-to-object mapping, to avoid that incompatible objects start listening to the same name, and to separate the lifetime of an object-reference from the lifetime of the applications.

- **Active Replication**: `NamingContexts` can be replicated across machines for fault-tolerance and performance. To that purpose, the *object group* [6] abstraction is employed.

- **Dynamic Replication**: The replication degree can be increased and decreased at run-time, while applications are interacting with the naming service. When a new `NamingContext` is created, the ORB requests a snapshot of the bindings from an existing replica and copies this state information to the newcomer.

- **Group Communication**: Operations that alter the internal state of a `NamingContext` (e.g., the `bind` operation) are transmitted by *totally ordered, reliable multicast* [3] to ensure a consistent replicated state.

- **Efficient Lookup**: Operations that do not alter the replicated state (e.g., the `resolve` operation) are transmitted by point-to-point communication to a nearby `NamingContext` replica.

- **Failure Detection**: All applications have a consistent view of which `NamingContext` objects have failed or are alive. This is made possible by a *failure suspector service* [2].

- **Membership Management**: A client's ORB is informed when a `NamingContext` object joins or leaves the replication group. Thus, an ORB notices a complete failure of the naming context. In such situation, it emits a warning notification and suspends client applications that are trying to interact with the naming service, until a new `NamingContext` is created and joined to the group.

3.3 System Support

The implementation of a naming service adhering to above requirements demands system support not available in the CORBA ORBs in widespread use today. Fortunately, a considerable amount of research on fault-tolerant distributed systems has been performed in the context of the *virtual synchrony* model [1], leading to toolkits like Consul, Delta-4, Horus, Isis, Phoenix, Totem, and Transis. However, most of these systems offer a rather low-level, non-portable `C` API.

Active and passive replication are the main fault-tolerance mechanisms provided by the virtual synchrony model. In this model, processes can be replicated by joining them to a *process group*. Communication with a process group is by reliable, order-preserving multicast. Processes can be replicated at run-time. Failures are detected and reported through a distributed failure detection service [2].

The aforementioned Electra ORB is conceived to run atop of subsystems that implement the virtual synchrony model. It offers the system support necessary to implement highly available object services, and presents a CORBA interface to group communication subsystems.

At first sight, Electra appears to the programmer like any conventional CORBA ORB: Distributed objects are specified in CORBA-IDL. The Electra IDL compiler generates C++ communication stubs as well as skeletons of the object implementations. A programmer implements the empty member functions of the skeletons to obtain fully functional network objects.

The Electra IDL compiler generates three special member functions in the object skeleton, needed to implement a fault-tolerant object [5]. The `get_state` member function is invoked by Electra when a new object joins a replication group. The purpose of this function is to obtain a snapshot of the application-specific, internal object-state from an existing group member. The `_im_NamingContext::get_state` operation, for instance, returns all the name-to-object bindings maintained by a `NamingContext` object[5].

The `set_state` member is invoked on the newcomer object to pass it the state snapshot. Finally, there is a `view_change` member function which is invoked on all group members after an object has joined or left[6] the group.

The Electra Basic Object Adapter (BOA) [10] exports the operations `create_group`, `join_group`, `leave_group`, and `destroy_group` which allow applications to maintain object groups.

Besides replication, object groups can be used to implement efficient distribution of data by protocols like IP-Multicast, object migration, load balancing, and caching [5].

4 Implementing a Highly Available Naming Context

This section describes how a fault-tolerant, scalable COSS name server was implemented with the system support described in Section 3.3. The naming context is part of the Electra run-time system and is in everyday use.

4.1 Service Creation

In order to make a workstation part of the Electra configuration, a daemon process containing an instance of the `NamingContext` object implementation is launched. `NamingContext` objects join a predefined object group, whose stringified object reference is retrieved from a configuration file. Thus, per default, there will be one replica of the naming context per machine. Applications bind to their default name server by reading the aforementioned file. This is transparent to the Electra programmer.

4.2 Binding Object References

Since binding requests install a new name-to-object mapping, they must be multicast to all `NamingContext` replica in a reliable and totally ordered manner. *Reliable* in the sense that every surviving `NamingContext` will eventually dispatch the request (lost requests are retransmitted). *Totally ordered* means that every group member dispatches all requests in exactly the same order [3].

The scalability and efficiency of the binding operation depends on how the total ordering protocol is implemented and on whether it exploits IP or UDP multicast, where available. Electra is conceived to be portable across various group communication subsystems, the present version supports both Horus and Isis. The performance of the naming context thus mainly depends on the configuration of Electra. Performance data for the Horus and the Isis configurations will be presented in Section 5.

4.3 Resolving Object References

No group communication is required to resolve a name. Per default, Electra will point-to-point deliver a `resolve` operation to the naming context running on the same machine as the querying application. A remote object is chosen in case there is no local one. If the object fails, a new one is chosen and any aborted `resolve` operation is restarted.

4.4 State Transfer

This section describes the extra work to be performed by the Electra programmer in order to obtain a fault-tolerant, dynamically replicable implementation of the `NamingContext` interface. The extra work mainly consists in implementing the `get_state` and `set_state` member functions we mentioned in Section 3.3.

```
// C++
void
_im_NamingContext::get_state(CORBA::AnySeq& state){
    state.length(bindings.length() * 2);

    // store mappings in the state sequence:
    for(int index = 0; i < 2 * bindings.length(); i++){
        state[index] <<= bindings[index].name;
        state[index +1] <<= bindings[index].reference;
    };
};
```

[5] the state is encapsulated in a `sequence<any>` datatype.

[6] either explicitly or by crashing.

The `_im_NamingContext::get_state` member function determines the number of name-to-object bindings which are stored in the local `bindings` sequence, and sets the length of the `state` sequence accordingly. Subsequently, the bindings are read out of the `bindings` sequence and assigned to the elements of the `state` sequence.

When a new `NamingContext` replica joins an object group, Electra will call the `get_state` operation of an existing group member, marshal the `state` object, transfer it to the newcomer, unmarshal it and pass it to the newcomer's `set_state` member function:

```
// C++
void
_im_NamingContext::set_state(
    const CORBA::AnySeq& state){
    bindings.length(state.length() / 2);

    // construct bindings out of the received state:
    for(int index = 0; i < state.length(); i+ = 2){
        state[index] >>= bindings[index].name;
        state[index +1] >>= bindings[index].reference;
    };
};
```

Note that state transfers are automatically synchronized with the name server operations issued by the client applications, to ensure a consistent replicated state. This is a key feature of the virtual synchrony model.

4.5 Creating Subcontexts

Because a context is like any other CORBA object, it can also be bound to a name in some naming context to create a naming graph. The `new_context` and `bind_new_context` operations are used to attach subcontexts to an existing context. However, special care regarding fault-tolerance must be taken when implementing these operations: When a `new_context` (or `bind_new_context`) request is sent to a replicated `NamingContext`, each group member will create a `NamingContext` and join it to a new object group. The result is depicted in Figure 2. This allows the creation of fault-tolerant naming graphs. The replication degree of a subcontext equals the replication degree of its father-context.

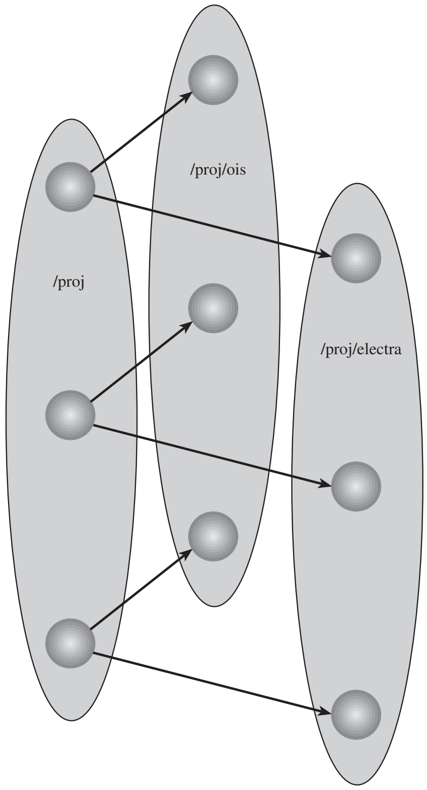

Figure 2: Creation of a Fault-Tolerant Subcontext. Circles represent `NamingContext` objects, ovals denote object groups, arrows point to subcontexts.

4.6 Fault-Tolerant Binding Iterators

As was mentioned in Section 2.2, the `NamingContext::list` operation returns a specific number of bindings. If the naming context contains additional bindings, a `BindingIterator` object reference is returned allowing the application to browse the remaining entries, by issuing the `next_one` and `next_n` operations.

For the sake auf fault-tolerance, binding iterators need to be instantiated in the form of an object group (Figure 3). The replication degree of a binding iterator equals the replication degree of the associated naming context.

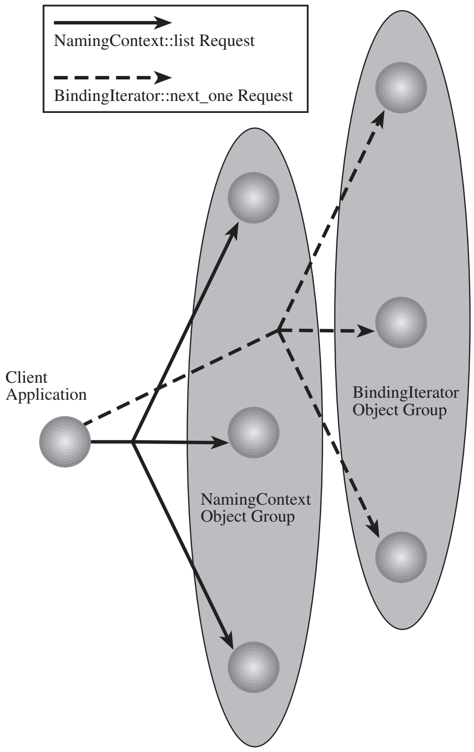

Figure 3: Creation of a Fault-Tolerant Binding Iterator

5 Performance

We have implemented the described fault-tolerant naming context service and incorporated it into the Electra ORB. The performance of the Electra naming context service was measured in relation to various replication degrees. To that purpose, we distributed up to eight replica of the service across a cluster of five SPARCstation 10 and four SPARCstation 20 machines. The cluster was interconnected by a moderately loaded 10 Mbps Ethernet. Each replica was running on its own workstation, and a benchmark application was running on the remaining workstation.

Two different benchmarks were carried out: The first one measured the time needed to perform 300 `bind` operations, the second one the time to perform 300 `resolve` operations. Each experiment was carried out three times for each replication degree, we will report the averages of the three results. Moreover, we tested two configurations of the Electra ORB, one running on the Isis, and one on the Horus toolkit.

A `bind` operation is sent to each group member by totally ordered multicast, since the internal state of the name server is altered. A `resolve` operation, on the other hand, is sent to only one group member, since just a query is performed and no state is altered. This is transparent to the programmer.

For `resolve` operations, Electra tries to select a group member that runs on the same machine as the querying application. However, in our experiment the server objects and the benchmark application were running on different workstations. In this case, programmers can instruct Electra to choose a remote member either at random, in a round-robin fashion, or using an algorithm provided by the programmer. In our experiment we provided a simple algorithm that selected the same group member for all `resolve` requests. If this group member fails, a new member is chosen automatically and any aborted request is restarted.

Figure 4 summarizes the results of the performance measurements. The Y axis reports the average time needed to perform *one* operation with the name server, i.e., the measured elapsed times were divided by 300. The X axis spots the replication degree of the name server.

With the Isis toolkit as communication subsystem, it took an average of 33.7 milliseconds to perform a `bind` operation on eight replica. With Isis, increasing the group size by a factor of two decreased performance of the `bind` operation by approximately a factor of two. However, a replication degree of two or three three will be sufficient in most situations, especially since failed `NamingContext` objects can be restarted at run-time without corrupting the internal state of the service. With three replica, it took an average of 14.2 milliseconds to perform a `bind`. With two replica, it took an average of 11.4 milliseconds.

The Isis `abcast`[7] protocol was used to transmit `bind` operations to the group, the Isis `cbcast`[8] protocol for point-to-point delivery of `resolve` operations to one group member. A `resolve` operation took an average of 4.9 milliseconds in Isis. Note that the performance of query operations does *not* depend on the replication degree. We used Isis Version 3.2.1 on SUN

[7] totally ordered multicast.
[8] causally ordered multicast.

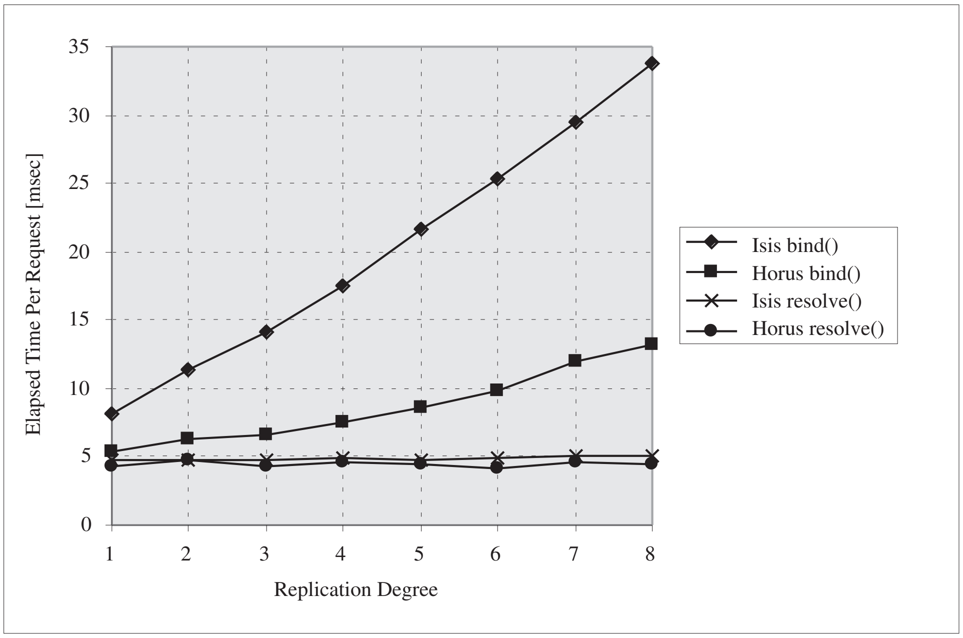

Figure 4: Performance of the Electra Naming Service

OS 4.1.3. The applications were compiled with GNU g++ 2.6.3.

The two graphs in the middle depict the performance of the `NamingContext` running atop of the Horus toolkit (Figure 4). With Horus, the `bind` operation scaled considerably better than with Isis, it took an average of 6.6 milliseconds to transmit a `bind` operation to three replica, and 13.3 milliseconds to transmit it to eight replica. We employed the Horus protocol stack `"BUFFER:ORDER:CAUSAL:UNPACK:TOTAL:MERGE:VIEWS"`. A Horus `resolve` operation took an average of 4.46 milliseconds to complete, regardless of the group size. For `resolve` requests we employed the same protocol stack as for `bind` requests, although no totally ordered multicast is required for `resolve` requests. We are planning to extend Electra to support two different protocol stacks, one for multicast and another for point-to-point requests. Experiments have shown that with such extension, a `resolve` requests takes less than 3.5 milliseconds on the average.

The considerable difference in `bind` performance is due mostly to the fact that Horus employs IP Multicast, and a rotating-token protocol to implement total ordering. IP Multicast leads to much lower communication overhead during multicasts, the rotating token protocol is very effective when just a few applications are sending multicasts at the same time. Note that Isis can be instructed to use IP Multicast on the Solaris operating system. We have not assessed the effect of IP Multicast on our Isis experiment yet.

Finally, the Horus system is geared toward asynchronous and deferred synchronous communication. Although Electra supports these communication styles, we used only blocking communication in our experiments, as this is the predominant interaction style in CORBA. We also note that we have not been working on optimizing the performance of the naming context or of the Electra ORB yet.

6 Conclusions

In this paper we described the design and implementation of a fault-tolerant OMG COSS Naming Context, a service which is aimed at maintaining the name-to-object mappings of a CORBA object request broker. Fault-tolerance is achieved through the active replication of `NamingContext` objects. The replicas are distributed across machines. The Electra ORB directs lookup requests to a nearby replica, binding operations, on the other hand, need be multicast to all of the replica as the internal state of the service is modified.

The system support necessary to implement our design includes object groups, reliable multicast, total ordering, failure detection, and virtual synchrony. Our naming service was implemented atop of Electra – a CORBA Object Request Broker providing this kind of system support. Electra is conceived to run atop of group communication subsystems like Horus and Isis.

Our naming service advances the current state-of-the-art in that the service can be replicated while being accessed by many clients, in that a federated, highly available naming space can be provided, in that it offers good performance, and in that the service adheres to the OMG COSS specification. Object request brokers like ILU, DOME, Orbix, and NEO do not support dynamic replication of CORBA objects.

The performance of our naming service was assessed for several replication degrees. The measurements were carried out on a cluster of SUN workstations interconnected by a moderately loaded 10 Mbps Ethernet network. When running atop of the Horus communication subsystem, it takes an average of 13.3 milliseconds to install a name-binding into eight replica of the naming context, and 6.6 milliseconds for three replica. It takes an average of 4.46 milliseconds to resolve a name. The performance of resolve operations does not depend on the group size, as these need be transmitted to only one group member.

Acknowledgements

The author would like to thank Salil Deshpande (CustomWare, Inc.), Roger Barnett (Object Oriented Technologies, Ltd.), Bill Janssen (Xerox PARC), Chris Horn (Iona Technologies), John Moreau (Iona Technologies), Roy Friedman (Cornell University), Robbert van Renesse (Cornell University), and Alexey Vaysburd (Cornell University & Isis, Inc.) for their support.

Availability

The naming service described in this paper is part of the Electra distribution. Electra is publicly available through the Web at
http://www.cs.cornell.edu/Info/People/maffeis/-electra.html

References

[1] Birman, K. P., and van Renesse, R., Eds. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[2] Chandra, T. D., and Toueg, S. Unreliable Failure Detectors for Reliable Distributed Systems. Tech. Rep. 93-1374, Department of Computer Science, Cornell University, Aug. 1993.

[3] Hadzilacos, V., and Toueg, S. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, S. Mullender, Ed., second ed. Addison Wesley, 1993.

[4] Landis, S., and Maffeis, S. Building Reliable Distributed Systems with CORBA. *submitted to Theory and Practice of Object Systems, John Wiley Publisher, NY* (1995).

[5] Maffeis, S. Adding Group Communication and Fault-Tolerance to CORBA. In *Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies* (Monterey, CA, June 1995), USENIX.

[6] Maffeis, S. *Run-Time Support for Object-Oriented Distributed Programming.* PhD thesis, University of Zurich, Department of Computer Science, 1995.

[7] Object Management Group. *Common Object Services Specification Volume I.* OMG Document 94-1-1.

[8] Object Management Group. *ORB Initialization Specification.* OMG Document 94.10.24.

[9] Object Management Group. *Universal Newtorked Objects*, Sept. 1994. OMG Document 94.9.32.

[10] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 1995. Revision 2.0.

[11] Silvano Maffeis. Piranha – A Hunter of Crashed CORBA Objects. Tech. rep., Cornell University, Dept. of Computer Science, Feb. 1996. Submitted for Publication.

[12] Soley, R. M. *Object Management Architecture Guide.* Object Management Group. OMG Document 92-11-1.

[13] van Renesse, R., Birman, K. P., and Maffeis, S. Horus: A Flexible Group Communications System. *Communications of the ACM* (1996). (to appear).