PIRANHA – A Hunter of Crashed CORBA Objects

Silvano Maffeis*
Department of Computer Science
Cornell University
maffeis@acm.org

January 1996

Abstract

No matter how carefully a distributed application has been specified, implemented, and tested, its network objects will crash unexpectedly due to power outages, human lapses, hardware faults, and software bugs. The OMG CORBA standard is becoming popular for interoperability and extensibility. However, fault-tolerance and reliability have not adequately been addressed yet. This paper describes the design and implementation of Piranha — a CORBA-based, graphical availability management and application monitoring tool. Piranha is unique in that it can be used to increase the availability of many CORBA applications, without complicating their development.

Keywords: CORBA, Availability Management, Network Management, Communication Services Creation, Replication, Object Groups, Piranha, Electra

1 Introduction

Heterogeneity, interoperability, extensibility, and availability are key requirements of business-oriented distributed systems. The OMG CORBA standard [9] addresses just the first three of these:

Objects are specified in CORBA-IDL, a purely declarative language which is independent of the target programming-language and operating system. Thus, CORBA objects become an effective means for hiding the peculiarities of the underlying hardware and system-software behind a portable interface. CORBA objects can invoke operations on each other even when implemented in different programming languages, and even when running on incompatible operating systems.

Wrapper objects and ORB gateways permit programmers to interface new technology to legacy information systems. For example, a Visual C++ object running on Windows 95 can access a COBOL module running on a MVS mainframe, through a high-level CORBA-IDL interface [2]. Finally, a highly modular application is easier to extend and to maintain than a monolithic system.

However, reliability and availability are not adequately addressed neither by CORBA nor by the OMG Common Object Services Specification (COSS) [8]. This despite the fact that many real-world distributed systems need to be highly available. For the sake of availability, CORBA should support dynamic replication of object implementations and consistent detection of partial failures in a distributed application. Furthermore, system services such as the COSS Naming Context should be resilient to failures.

This paper describes the design and implementation of *Piranha* — a CORBA-based availability-manager and –monitor. Piranha is unique in that it can be used to increase the availability of many CORBA applications, without complicating their development.

On one hand, Piranha acts as a network monitor that reports failures through a graphical user interface. On the other hand, Piranha automatically restarts failed CORBA objects, replicates stateful objects on-the-fly, migrates objects, and enforces predefined replication degrees on groups of objects. Piranha is useful for a wide range of CORBA applications that must be highly available.

The potential users of Piranha are mostly software engineers in charge of developing new CORBA applications, and system administrators in charge of running these applications.

The paper is organized as follows. Section 2 describes six requirements a CORBA ORB should fulfill in order to support highly available applications. In Section 3 an overview of Piranha is given. Section 4

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

deals with remote object activation. An example of a real-world application that benefits from Piranha is presented in Section 5. Section 6 describes the IDL interfaces and interaction protocols that lie at the core of Piranha, it also gives some performance figures. Finally, Section 7 summarizes and concludes the paper.

2 Achieving High Availability with CORBA

In order to support highly available object services, the CORBA model needs to be extended in several ways: First, the ORB must allow for the *dynamic replication* of important object implementations. Second, *object references* must remain valid after the associated object implementation has crashed and subsequently restarted. Third, a distributed *Notification Service* allowing objects to report exceptional conditions, such as I/O errors, must be provided. Fourth, a *Failure Detection Service* must be provided to detect and report failed objects consistently. Fifth, failed objects must be restarted automatically, which is accomplished by an *Availability Service*. Sixth, it should be possible to install new versions of an object service on-the-fly, without decreasing the availability of the service, and without needing to restart client applications.

During the following six subsections we will elaborate on these requirements.

2.1 Dynamic Object Replication

Active and passive replication allow the implementation of highly available CORBA objects [5]. In active replication, a CORBA object is replicated on several machines, the replicas are subsequently joined to an *object group* [6]. Object groups allow programmers to view and to name a set of objects in a uniform way. A CORBA object reference is then created for the whole group and installed in a name server.

Applications can retrieve that object reference and bind to the whole group as if it was a singleton CORBA object. Requests issued through the object reference will be transmitted to *all* group members by *reliable, totally ordered multicast*. This permits the group members to maintain a consistent internal state. A request will succeed as long as at least one group member survives. The ORB provides operations such as `create_group`, `join_group`, `leave_group`, and `destroy_group` to allow applications to maintain object groups.

The replication degree of an object can be increased at run-time, by having a new CORBA object joining the group. A CORBA object joins an existing group by retrieving the object reference of the group from a name server, and by subsequently issuing the `join_group` operation with the group reference as a parameter. During the join operation, the ORB requests the current internal state from some group member and transfer it to the newcomer. Finally, the group members are notified of the admission.

In order to guarantee a consistent replicated state, requests that are sent to the group by applications must be synchronized with the state transfer and membership management messages: Every group member dispatches requests and membership notifications in the same order. This execution model is called *Virtual Synchrony* [1]. In the Virtual Synchrony model, group members know about their rank within the group and are notified whenever an object joins or leaves[1] their group.

Object groups can be used to implement passive replication as well: The oldest group member (rank 0) acts as the *primary*, the rest of the group members (ranks 1 to n-1) as the *backups*. Only the primary performs the computations associated with the requests, and multicasts update messages to the backups from time to time.

When the primary fails, the backups will obtain a membership change notification from the ORB. During that membership change, the backup object with rank 1 will be assigned rank 0, i.e., it becomes the new primary.

This simple primary/backup scheme can be optimized in various ways. For instance, the primary role can be rotated among the group members for load balancing. Owing to Virtual Synchrony, CORBA programmers need not be concerned about consensus and fail-over protocols to a large extent.

Thus, dynamic object replication can be supported by extending the ORB with object groups, reliable multicast, and with protocols that implement the Virtual Synchrony model.

2.2 Long-Lived Object References

In most of the CORBA ORBs available today, object references become invalid after the associated object implementation has failed and subsequently restarted on another machine. After a failure, a client application will raise an exception that terminates the application, if the programmer did not implement the algorithms necessary to deal with restarted objects.

[1] either explicitly or by crashing.

For example, when an object reference becomes invalid due to a failure, the application could keep on contacting the naming service until a valid reference is obtained, i.e., until the object has restarted and installed its new reference in the name server. However, this simple policy will lead to a storm of name server requests when a widely used object fails, and programmers need to write rebinding code. Robustness thus becomes harder to achieve. For the sake of robustness, we request the ORB to handle session reconnections and retry of a failed operations transparently.

We suggest to view object references as multicast addresses: When the referenced object (or object group) fails, the reference remains valid and the client's ORB keeps on retrying any pending operation periodically. As soon as an instance of the object is created and bound to the object reference, applications can make progress, without having to contact the naming service. The object reference remains valid even when the object is restarted on another machine.

Another advantage of long-lived object references is that these remain valid even after the ORB has been shut down. Thus, the object reference of an important system service can be made publicly known. For instance, the object reference of the naming service could be stringified and written to a file. Upon start up, applications can bind to the naming service simply by reading its reference from the file. The file needs to be created and distributed only once, when the ORB software is installed.

2.3 Notification Service

The interface of the Notification Service provides operations like `info`, `warning`, and `panic` that let objects report problems. For instance, an object might submit a `panic` notification after having encountered an unrecoverable I/O error, or a `warning` notification when the amount of free memory has fallen below a certain threshold.

```
// CORBA IDL
interface Notify {
    oneway void info(in string message);
    void warning(in string message);
    void panic(in string message);
};
```

There can be several implementations of the `Notify` IDL interface: A graphical console that highlights an icon to represent the machine on which the problematic object is running, or a background process that informs a system administrator of the problem by forwarding the notification to his or her pager.

The notification service consists of a group of `Notify` objects. Notifications are delivered to all group members by reliable multicast. Notification objects can be joined and removed from the group dynamically, without affecting the applications that are using the service. Different implementations of the `Notify` interface can be joined to the same group, which means that notifications can be both brought to graphical consoles and to several pagers, for example.

2.4 Failure Detection Service

Fischer, Lynch, and Paterson proved the impossibility of consensus in asynchronous distributed systems that are prone to crash failures [3]. More precisely, their result asserts that there is no deterministic algorithm that solves consensus in an asynchronous system and tolerates even a single crash failure.

In asynchronous systems, an object that has crashed can not be distinguished from one that is very slow. However, by introducing a *Failure Detection Service*, one can get around the consensus problem [4]. The Failure Detection Service can be seen as a distributed "oracle" that makes suggestions about failed objects using timeout mechanisms and hints from the operating system. The suggestions need not be correct; a failure detector can make mistakes by erroneously assuming a slow object as faulty. However, it is important that failure beliefs be propagated consistently to the application objects; every object has the same view on which objects are healthy and on which are believed faulty. Furthermore, a suspicious object is excluded from the system and prevented from submitting requests. A Failure Detection Service is a necessary component of any ORB that is aimed towards reliable applications.

A reliable ORB should thus provide a function `monitor_set` which takes an object reference and an upcall procedure as the parameters. When the object implementation associated with the reference is believed faulty, the upcall procedure is invoked to trigger an application specific behavior.

2.5 Automatic Restart

Object Request Brokers should tolerate both the failure of individual objects and the failure of machines. Thus, in addition to the replication mechanisms outlined in Section 2.1, automatic restart of failed objects needs to be supported through an Availability Service.

In case of an object failure, the object is simply restarted on the same machine. In case of a machine failure, all of the objects that were running on that machine need to be restarted on the other machines. The Availability Service relies on the Failure Detection Service to determine objects that need to be restarted. It submits a `warning` message through the Notification Service when an object had to be restarted.

The Availability Manager is implemented in the form of an object group. There is one group member per machine. Each member knows about all of the application objects that are running on the other machines, such that it can restart objects when a machine crashes. The group members communicate by reliable multicast with each other to maintain a consistent replicated view of which objects are running on the network.

2.6 On-The-Fly Upgrades

Owing to the object group abstraction, new releases of an object implementation can be installed at run-time, without decreasing availability and without having to rebuild or to restart client applications[2].

To that purpose, a new version of the object is joined to a group that contains the object(s) to be upgraded. However, if the object is stateful, the new version must be able to handle the "old" state information it obtains when it joins the group. This is usually straight forward to implement: A version number is made part of the state information. The newcomer object checks the version number which it has got as part of the state data. If the number denotes the previous version, a state-unpacking member function to handle the old version is called by the newcomer. If the number denotes the new version, the normal unpacking function is called.

After at least one new object has joined the group, the executable belonging to the old version is simply replaced by the new one. The remaining old group members will either crash, since the executable was overwritten, or they will be restarted by the user. In case of a crash, Piranha will restart the object(s) automatically if the Keep Alive criteria was specified.

The advantage of this approach is that upgrades can be performed on-the-fly, while applications are interacting with the object which is being upgraded. The downside is that stateful objects need to be able to cope with two versions of the state information.

[2] as long as the interface of the object is not modified.

3 Piranha

Piranha is a novel CORBA object service that implements the functionality of the aforementioned notification, failure detection, and availability services. Piranha requires system support not yet available in widely used CORBA ORBs, namely object groups, multicast, multi-threading, failure detectors, and Virtual Synchrony. Electra [6, 7] is a CORBA ORB providing this kind of system support, which was used to implement Piranha.

The main goal of Piranha is to allow the monitoring and management of CORBA applications that require the presence of many CORBA objects in order to make progress. The Piranha user interface is graphical and interactive. Users can glance through a network configuration by clicking on graphical icons that represent machines and network objects.

3.1 Graphical User Interface

The top window of the Piranha user interface (Figure 1) contains the activity log where events such as *"machine x has failed"* or *"object x was migrated from machine y to machine z"* are reported.

The canvas displays an icon per configured machine. A "healthy" machine is represented by a terminal icon. A crossed out terminal represents a presumably failed machine. An icon with exclamation mark stands for a machine on which an object has failed or has reported a `warning` notification. Each icon is labeled with the hostname and the number of objects running on the host.

After a certain icon has been selected by the user, the three list boxes in the lower area display a list of objects running on that host, the parameters associated with an object (number of failures, start time, etc.), and the notifications that were produced by an object (e.g., *"running out of memory"*, *"I/O error reading device X"*, *"segmentation fault, core dumped"*, etc.).

The entries in the pull-down menus allow users to start, restart, kill, migrate, and replicate objects on-the-fly.

3.2 Example

A typical situation consists of the user detecting an exclamation mark on an icon. (Piranha emits acoustic warning signals when important events occur). By clicking on that icon, the user will be presented with a list of objects running on the machine. The list entry pertaining to the suspicious object is marked. By selecting that entry, the user will gain access to the

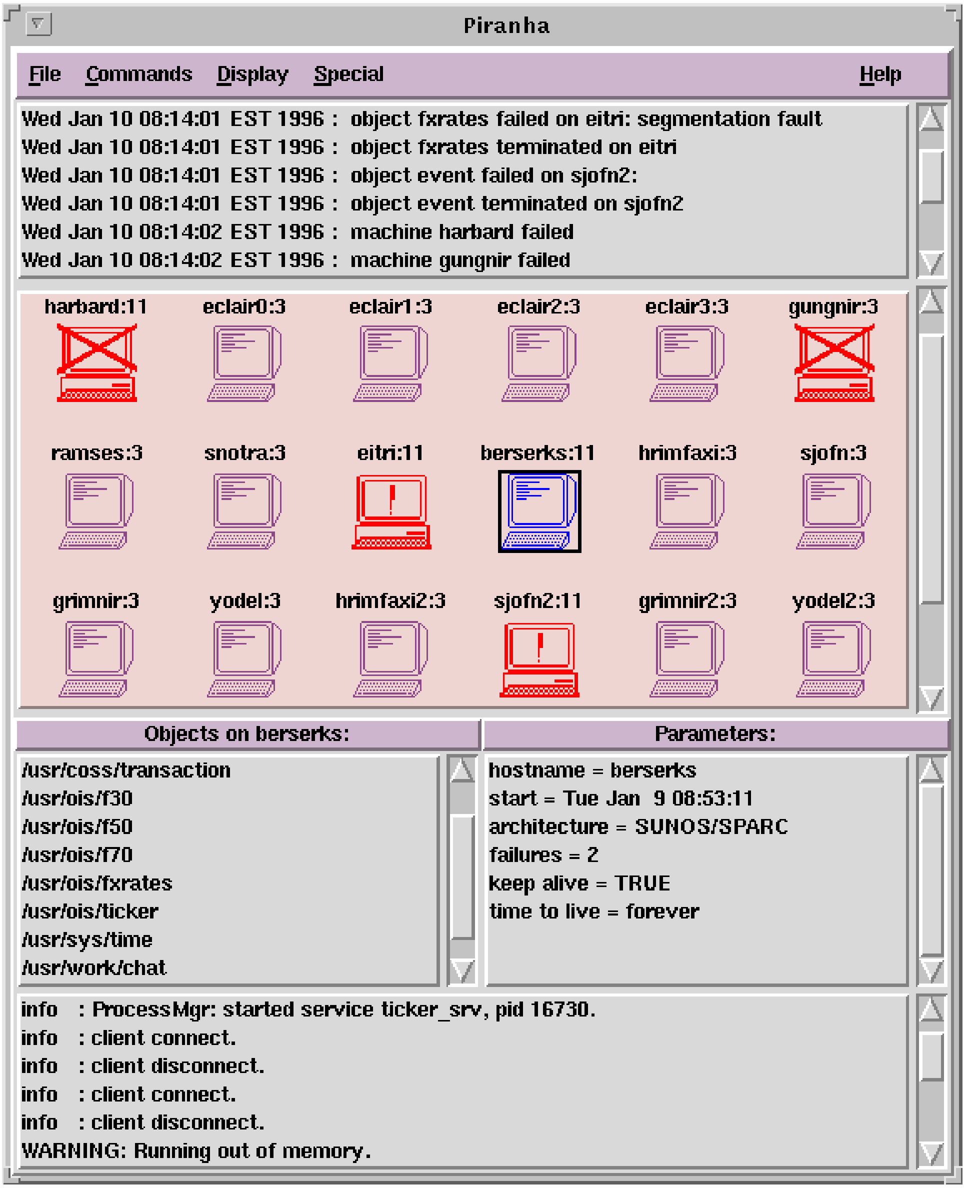

Figure 1: Piranha User Interface

notifications produced by the object. If the object has failed, the user can determine whether the object has been restarted automatically, whether there are operational replicas of the object, and so forth.

4 Launching Object Implementations

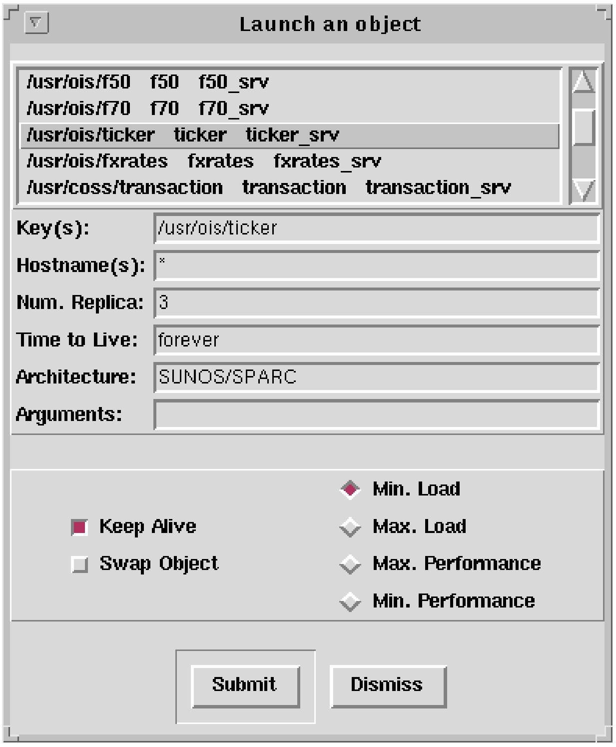

Figure 2: Piranha Launch Tool

Object implementations can be remote-activated either interactively through the Piranha launch tool (Figure 2) or by the application itself, through an API that follows the OMG LifeCycle service specification [8]. Several *activation criteria* can be specified to guide the creation of an object.

4.1 Activation Criteria

Key Criteria: In order to make an object implementation manageable by Piranha, the path to its executable file must be specified in a dialog box, along with a human-readable key under which the object will be registered with the name server. The Key Criteria determines the object to be created.

Hosts Criteria: The target hosts can be specified using simple regular expressions for the host names. For instance, one might request 3 replica of an object and specify the host criteria `"harbard gungnir snotra"` to have one replica placed on each of the hosts. If no Hosts Criteria is specified, Piranha will select the target hosts at random.

Num Replica Criteria: Objects can be instantiated in the form of object groups for fault-tolerance, cooperative services, or for the efficient distribution of data. The Num Replica Criteria specifies the number of replica. Per default, only one object is created.

Time To Live Criteria: If this criteria is specified, the object will be deactivated after the time (in seconds) specified along with the critaria has expired. Per default, the life time of an object is not restricted.

Architecture Criteria: The architecture of the target hosts can be specified with that criteria. For instance, one might request Piranha to create an object on a node running the Windows NT operating system. Per default, no particular architecture is chosen.

Keep Alive Criteria: When this criteria is specified, the object (or any replica of it) will automatically be restarted on failure. This is a default criteria.

Swap Out Criteria: An object that has not serviced requests for a certain amount of time (typically one minute), will be shut down if the Swap Out Criteria is set. Piranha will make the internal state of the object persistent before bringing it down. A request to a deactivated object leads Piranha to reactivate it. The main purpose of this criteria is to preserve system resources. Per default, objects are not swapped out.

Performance and Load Criteria: These allow users to create objects on the least/most loaded hosts or on the hosts with the lowest/highest performance. The default is to create objects on the least loaded hosts.

4.2 Applications Programmer Interface

As was mentioned before, there is an alternative to the graphical launch tool in Figure 2. Applications can activate objects through the operations defined in the `GenericFactory` interface. `GenericFactory` follows the OMG COSS [8] specification:

```
// CORBA IDL
interface GenericFactory {
    Object create_object(in Key k, in Criteria crit)
        raises(NoFactory, InvalidCriteria,
        CannotMeetCriteria);
    void restart_object(in Key k, in Criteria crit)
        raises(NoFactory, InvalidCriteria,
        CannotMeetCriteria);
    void destroy_object(in Key k, in Criteria crit)
        raises(NoFactory, InvalidCriteria,
        CannotMeetCriteria);
    boolean supports(in Key k);
};
```

The `GenericFactory` operations can be used from within an application. Operation `create_object` is a generic interface for creating objects. The `Key` parameter identifies the desired object to be created. On a `create_object` request, the `GenericFactory` searches an entry in its *object configuration table* that matches the key. The entry specifies the type (interface) of the object and the location of an executable file that implements the interface. The second parameter (`Criteria`) allows for the activation criteria in Section 4.1 to be passed. If the generic factory cannot create an object specified by the key, then the `NoFactory` exception is raised. An invalid criteria raises the `InvalidCriteria` exception. Finally, the `CannotMeetCriteria` is raised when a criteria cannot be met, for example, when a replication degree larger than the number of available hosts was specified.

Operation `restart_object` restarts an object or object group, `destroy_object` kills an object or a group. Finally, the `supports` operation checks whether a generic factory can create a certain object.

4.3 Example

```
// C++
extern GenericFactory_ptr gf;
CORBA::Environment env;
CORBA::Object_var ov;
Criteria crit; crit.length(5);
CosNaming::Name n; n.length(1);

crit[0].name = ::crit_hosts;
crit[0].value <<= "*";
crit[1].name = ::crit_max_replica;
crit[1].value <<= (CORBA::UShort)3;
crit[2].name = ::crit_arch;
crit[2].value <<= "SUNOS/SPARC";
crit[3].name = ::crit_keep_alive;
crit[4].name = ::crit_min_load;

n[0].id = "/usr/ois/ticker";
ov = gf->create_object(n, crit, env);
...
```

The above example creates three replica (`crit_max_replica`) of the object which is registered under `"/usr/ois/ticker"` in the object configuration table. Failed instances will be restarted by Piranha (`crit_keep_alive`). The three SPARCstations (`crit_arch`) with the least load (`crit_min_load`) are chosen.

5 Dealing with Failures

In this section we give an example of a real-world CORBA application that can benefit from Piranha to achieve high availability. Economists at Olsen & Associates have developed algorithms to perform forecasts of foreign exchange rates, for most of the popular currencies[3]. Consider a simple forecaster object with the following CORBA-IDL interface:

```
// CORBA IDL
interface FX_forecast {
    enum currency { US_dollar, D_mark, CH_franc,
        F_franc, JP_yen };
    typedef string date;
    struct { currency c; sequence<double> rates; } history;

    void new_fx_rate(in currency from, in currency to,
        in double fx_rate, in date when);
    double fx_forecast(in currency from, in currency to,
        in date when);

    attribute sequence<history> past_fx_rates;
};
```

The `new_fx_rate` operation is used to feed real-time exchange rates into the forecaster. The forecaster relies on that data to perform the forecasts. Exchange rates are stored in the `past_fx_rates` data structure. Thus, `FX_forecast` objects are stateful. In order to perform a forecast, the `fx_forecast` operation is issued. The `date` parameter defines a date in the future for which the prospective exchange rate of currency `from` to currency `to` is to be returned.

Since Olsen's customers depend on the `FX_forecast` service, high availability is very important. In the following subsections we describe how Piranha helps in increasing the availability of the `FX_forecast` service.

5.1 GUI Notifications

The Piranha graphical user interface visualizes the situation of applications that are running on a cluster of networked servers and workstations. The GUI is updated in real-time, as objects are being started,

[3] the forecasting service is accessible at no fee through the World Wide Web at `http://www.olsen.ch`.

replicated, and destroyed. Several Piranha sessions can be run concurrently by several users. Every notification is multicast to all of the sessions, and each user can monitor the actions performed by the other users.

Thus, problems that occur with the forecaster can be noticed and tracked down more quickly and more easily with Piranha. This increases the availability of the forecaster.

5.2 Replication

By specifying a Num Replica Criteria larger than one, the `FX_forecast` service will be activated in the form of a replication group. Client applications can issue requests on the service as long as at least one `FX_forecast` object is operational. Piranha allows one to increase (or decrease) the replication degree at run-time. Owing to Virtual Synchrony and to state transfer, new `FX_forecast` replicas can be created while several applications are issuing `new_fx_rate` operations that alter the forecaster's internal state.

5.3 Automatic Restart

By specifying the Keep Alive Criteria, Piranha will attempt to automatically restart failed instances of the `FX_forecast` interface. If a host holding a forecaster object goes down, the object will automatically be restarted on another host.

5.4 Migration

Objects can be migrated from one host to another, by selecting the *Migrate Object* entry from the *Commands* pulldown menu. This feature is useful for preventive maintenance: If a host is to be shut down or rebooted, its objects can be migrated on-the-fly.

Internally, object-migration in Piranha works as follows: If an `FX_forecast` object is to be migrated from host `A` to host `B`, Piranha will first create an `FX_forecast` replica on `B`. The internal state is automatically copyed from the object on `A` to the newly created one. The two objects run in sync. Finally, Piranha simply kills the object on host `A`. Owing to Virtual Synchrony, the behavior of the forecaster remains predictable and no state information is corrupted or lost.

5.5 Retrying Requests

If everything goes wrong, i.e., if all `FX_forecast` objects fail and cannot be restarted due to a bug in the `FX_forecast` code, client applications are not forced to quit. Since Electra object references are long-lived, client applications will automatically retry any pending `new_fx_rate` or `fx_forecast` operation periodically, and submit warning notifications through Piranha, to inform the system administration of the problem. This is the default behavior of client applications. As soon as the problem has been solved and an improved `FX_forecast` object installed, the client applications will make progress again.

6 Piranha Implementation

This section focuses on the Piranha IDL interfaces and on coordination protocols used among the Piranha processes. Piranha runs on Electra, a CORBA ORB supporting the object group abstraction, reliable multicast, state transfer, and Virtual Synchrony [6]. Electra builds atop of group communication subsystems like Isis [1] and Horus [10]. Electra makes the implementation of a service like Piranha a straightforward task; Piranha consists of 1800 lines of Tcl/Tk code and of 3000 lines of C++. It took less than one man month to implement and to test the service.

There are two kinds of Piranha processes, one that is running in the background monitoring and restarting objects, and another providing the graphical user interface (Figure 3). Both implement the `PiranhaGenericFactory` interface below. We will refer to the first kind of process as the *monitor* and to the second as the *GUI*. When we speak of *the group* we refer to the object group holding all of the Piranha monitor and GUI processes. There is only one monitor per host, but several GUI processes can be created even on the same machine.

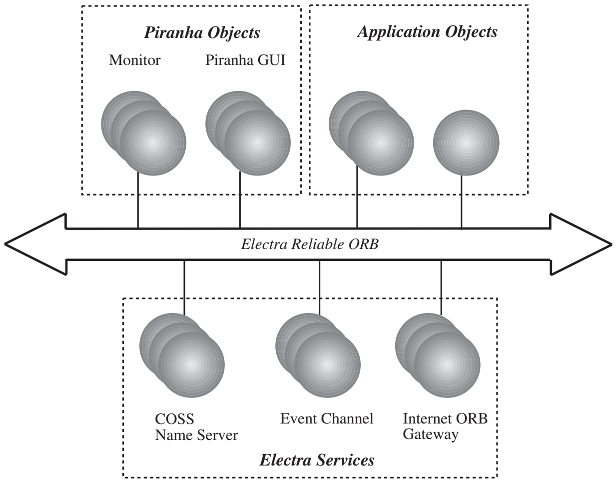

Figure 3: Electra Architecture

6.1 IDL Interface

```
// CORBA IDL
interface PiranhaGenericFactory: GenericFactory {
    struct object_info { double load; ...};

    void get_info(in Key key, out object_info oi);
    void i_am_here(in short rank, in string host,
        in string arch);
    void i_am_leaving(in string host);

    void i_have(in Key k, in string itf, in Criteria crit,
        in string host, in string time);
    void i_nomore_have(in Key k, in string itf,
        in Criteria crit, in string host, in string reason);
    void i_restarted(in Key k, in string itf, in string host,
        in short num_failures, in string reason,
        in string time);

    void list_configured_objects(
        out sequence<string> keys,
        out sequence<string> itf,
        out sequence<string> paths);
    void check_in(in Key k, in string itf, in string path)
        raises(ObjectExists);
    void check_out(in Key k, in string itf)
        raises(NoSuchObject);
};
```

6.2 Protocols

When a monitor is started, it first introduces itself to the group by multicasting the `i_am_here` request. Along with the request, it passes its rank within the group, the name of the host on which it is running, and the architecture of the host. When a monitor terminates, it submits the `i_am_leaving` request. If a monitor terminates without sending the `i_am_leaving` request, the associated host is assumed to have crashed. The Failure Detection Service (Section 2.4) is responsible for reporting failures to the group members. The group members have a consistent view on which objects are running on the hosts. This allows any monitor to restart the objects that were running on a host that has crashed.

After a host has created an object in response to a `GenericFactory::create_object` request, it sends the `i_have` notification to inform the group of the creation. `i_nomore_have` is submitted in response to a `GenericFactory::destroy_object` request, to notify the group that an object is not running any more. `i_restarted` informs the group that a certain object has been restarted on a certain host.

The remaining operations deal with maintaining a replicated configuration table of the objects that are to be managed by Piranha. The operations are issued mostly by the GUI processes. `list_configured_objects` returns the configuration table. Each entry in the table specifies an object through its key, interface name, and executable-path. New objects can be configured through the Piranha Object Configuration Tool (Figure 4). The `check_in` request informs the group that a new object was configured, the `check_out` request is multicast after a user has requested to remove an object from the configuration table.

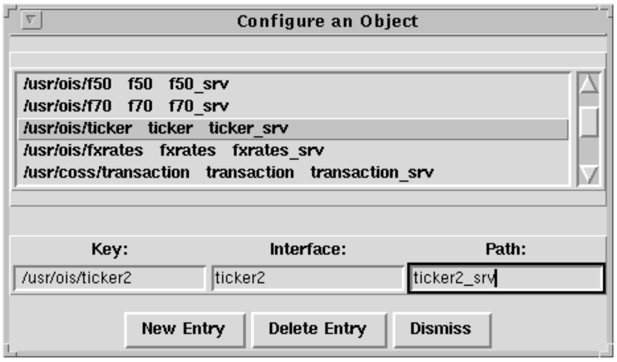

Figure 4: Piranha Object Configuration Tool

6.3 Performance

The time it takes Piranha to create an object-group was measured for groups with one to six members. The experiment was carried out on a cluster of four SPARCstation 10 and three SPARCstation 20 workstations interconnected by a moderately loaded 10Mbps Ethernet. A C++ benchmark application was run to measure the time needed to perform a `GenericFactory::create_object` request. For this experiment, the Electra ORB was configured to run atop of the Isis Toolkit.

A `create_object` operation works as follows. First, the `get_info` request is multicast to the monitors in order to determine their local load and other information (Figure 5). Then, according to that information and to the activation criteria, a list of the hosts that shall create an instance of the object is composed. Subsequently, the list is multicast to the monitor processes. If a monitor process finds its hostname on the list, it will create an object by using the key that was passed along with the `create_object` request. If it does not find itself on the list, the request is discarded.

Subsequently, the `create_object` operation waits until the group has come up and registered itself with the name server. Finally, the object reference of the group is retrieved from the name server and returned to the application. This protocol is implemented within a client-side *Smart Proxy* object.

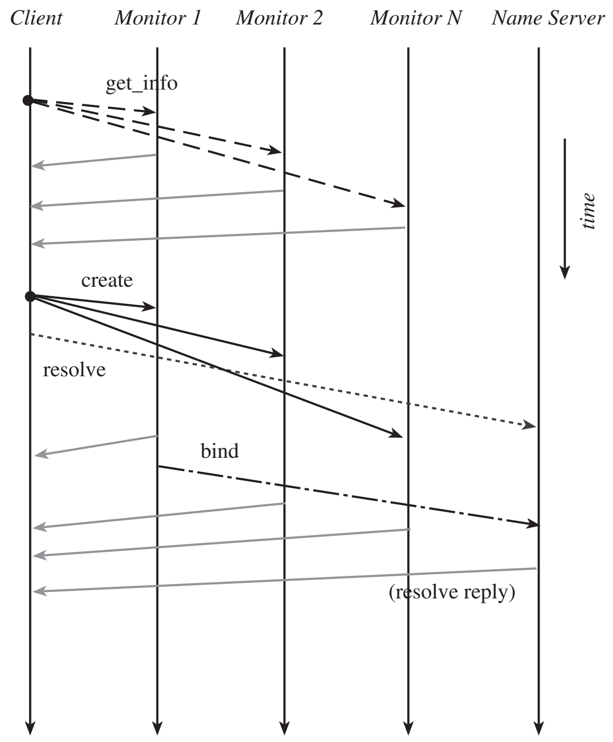

Figure 5: Object Creation procedure. Black arrows denote request messages, gray arrows denote reply messages.

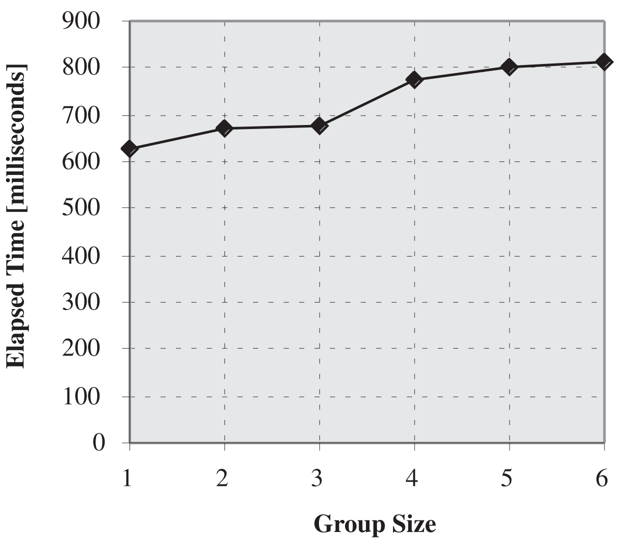

Figure 6: Performance of the `create_object` request.

In our experiment, creating an object meant creating a new process. Thus, up to six processes were launched during a `create_object` request. The size of the executable file for the processes was 1 megabyte. The benchmark application and the processes were running on different workstations.

In Figure 6, the Y axis depicts the average time needed for a `create_object` request, in relation to the group size (X axis). It took an average of 627 milliseconds to launch an object, and 815 milliseconds to bring up an object group with six members. Note that the time values comprise the whole creation procedure outlined in Figure 5.

We also measured the time it takes Piranha do detect a crash and to start reacting on it. The results we obtained are even more dependent on the configuration of the Electra ORB, with Isis being only one option. It typically takes less than 1 second for Piranha to detect a crashed object. For detecting the failure of a machine, Piranha relies on the failure detector of the underlying communication substrate, i.e., of Isis in our case. Thus, it takes Piranha an average of approximately 30 seconds to detect the failure of a machine: Isis periodically exchanges heart-beat messages to check whether a machine has failed. The frequency with which these messages are exchanged can be set when the Isis run-time is started. A frequency of 30 seconds is typical, 10 seconds is the lowest recommended value. The lower the value, the higher the probability that a healthy but slow machine will erroneously be suspected as faulty.

Detecting the failure of an object requires much less time since the process that holds the object was forked off by a Piranha monitor. When an object crashes, the monitor that created it will receive a signal from the operating system.

7 Conclusions and Future Work

Reliable real-world distributed applications require system support that goes well beyond replication and state checkpointing. In this paper we have described Piranha — a novel CORBA object service to manage and monitor the availability of distributed applications. Piranha basically fulfills two different tasks, namely application monitoring and availability management.

For application monitoring, Piranha provides a graphical representation of network objects that run on a distributed system. System administrators can watch computers going up and down, objects being created on the machines, and so forth. Crashes of machines or objects are visualized through a graphical user interface.

For availability management, Piranha can be employed for the dynamic restart, replication, and migration of stateful CORBA objects. This is accomplished through the same graphical user interface. Piranha can be instructed to enforce a certain replication degree on an object by automatically restarting any failed instance of it. If a host goes down, all of its objects will be restarted on another host, granted that the Keep Alive activation policy was specified for the object.

This paper also identified six major requirements CORBA ORBs should fulfill in order to support highly available and reliable applications: (1) dynamic replication of object implementations, (2) long-lived object references, (3) a distributed Notification Service, (4) a Failure Detection Service, (5) automatic restart of crashed objects, and (6) support for on-the-fly upgrades to software components.

Piranha requires system support as provided by the Electra CORBA ORB, namely object groups, reliable multicast, state transfer, and Virtual Synchrony. Electra is conceived to run on top of group communication systems such as Horus and Isis.

In the future, we intend to extend Piranha in several ways. Bringing up an application that consists of many network objects can be cumbersome, since objects often need to be started one by one through the Piranha launch tool or through the `GenericFactory` API. To allow easy initialization and management of complex applications, we envision the development of a *System Definition Language (SDL)*. Whereas a CORBA-IDL file only captures requirements for individual network objects, an SDL file specifies key assumptions the objects of an application make about each other. There will be one SDL file per application which is to be managed by Piranha. An SDL file mainly defines which objects need to be present for the application to be able to make progress, stipulates replication degrees and other activation policies, and also deals with security aspects.

Piranha is part of Electra release 2.0. Electra is publicly available through the Web at http://-www.cs.cornell.edu/Info/People/maffeis/electra.html.

Acknowledgements

The author would like to thank Roy Friedman, Robbert van Renesse, and Alexey Vaysburd for their support.

References

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

[2] Cummins, F. A., and Ibrahim, M. H. Wrapping Legacy Applications. *First Class — The Object Management Group Newsletter 4*, 2 (Apr. 1994).

[3] Fischer, M. J., Lynch, N. A., and Paterson, M. S. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM 32*, 2 (Apr. 1985).

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

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

[6] 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.

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

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

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

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