Failure Detectors as First Class Objects*

Pascal Felber†, Xavier Défago, Rachid Guerraoui‡
Swiss Federal Institute of Technology
Operating Systems Lab.
1015 Lausanne, Switzerland

Philipp Oser
Ubilab
UBS AG
8098 Zurich, Switzerland

Abstract

One of the fundamental differences between a centralized system and a distributed one is the notion of partial failures*. The ability to efficiently and accurately detect failures is a key element underlying reliable distributed computing. In current distributed systems however, failure detection is either left to the application developer or hidden from the programmer and provided in an ad hoc manner behind the scene. We plead for an intermediate approach where failure detectors are* first class objects*. We view failure detection as an abstraction, the complexity of which is encapsulated behind well defined interfaces. The various roles of a failure detection service are all represented as first class objects. Following our approach, one can reuse existing failure detection protocols as they are or, through composition or refinement, define new protocols that match the application requirements. We describe an interesting result of a composition that mixes push and pull failure monitoring and we show how scalability issues may be addressed by using a hierarchical failure detection configuration. We also discuss the implementation of our failure service both in CORBA and in Java.*

1 Introduction

The notion of *partial failures* is a fundamental characteristic of a distributed system: at a given time, some components of the system might have failed whereas others might be operational. The ability to hide partial failures from applications is usually considered a crucial way to measure the reliability of the system. All reliability schemes that we know about rely, to some extend, on *failure detection* mechanisms. Such mechanisms are particularly valuable for transaction management, replication, load balancing, distributed garbage collection, as well as for specific monitoring applications like supervision and control systems.

1.1 Current Practices

In most distributed systems, failure detection is left to the application developer (e.g., in DCE and PVM). Failures are handled through mechanisms like exceptions and it is up to the programmer to distinguish a physical failure from a logical failure specific to the application's semantics. Some reliable distributed programming toolkits however (e.g., group oriented systems [3] or transaction monitors [2]) provide support for failure detection through the use of timeouts. Nevertheless, the specific code that handles timeouts is usually mixed with the code of the distributed protocols (e.g., group membership and atomic commitment). It is very difficult, if not impossible, to adapt the failure detection mechanism to the network topology without modifying the application or the underlying distributed protocols. The only parameters that are usually left to the developer are timeout values. These are indeed fundamental parameters that enable the developer/user to trade latency (short timeouts) with accuracy (long timeouts). She/he cannot however parametrize the failure detection protocol itself. This can be viewed as a serious drawback of existing distributed systems and can seriously reduce their scalability and more generally their applicability in various contexts. For example, according to the network topology and the communication pattern of the application, the choice between a *push* (*heartbeat*) or a *pull* (*are-you-alive*) monitoring model can have an important impact on the performance of the system. Furthermore, in a large scale system, one might use either of those models in a hierarchical or a randomized gossiping style to reduce the number of messages exchanged in the network.

*Research supported by OFES, as part of the ESPRIT project OpenDREAMS (projects 20843 and 25262).

†Current affiliation is Oracle Corp., 1000 SW Broadway, Suite 1200, Portland, OR 97205.

‡Current address is HP Labs, 1501 Page Mill Road, Palo Alto, CA 94304-1126.

1.2 Failure Detectors as First Class Objects

It is until very recently that the idea of considering failure detection as a first class distributed service has emerged. Axiomatic properties of failure detectors have been exhibited and it was shown that even unreliable failure detectors can help circumventing known impossibility results in distributed computing [5, 4]. Roughly speaking, a failure detector is viewed as a distributed oracle that provides hints about failures in the system. One can prove the correctness of distributed agreement protocols (e.g., consensus, atomic broadcast, or non-blocking atomic commitment) simply by relying on abstract axiomatic failure detector properties. Following that theoretical work, researchers at Cornell University suggested to consider the failure detection as an operating system service that sits among established services such as naming, authentication, and file management [16, 14]. In this paper, we go a step further by considering failure detectors as *first class citizens*. Roughly speaking, failure detection is not transparent to the developer but rather hidden behind abstract, yet accessible, first class *object interfaces*.

On the one hand, we decouple failure detection mechanisms from other mechanisms in the system, thus enhancing modularity and extensibility. In fact, we even decouple the various roles of failure detection components: namely, *monitor*, *monitorable*, and *notifiable* objects. The failure detection service is viewed as a hierarchy of well defined interfaces and one can reuse existing mechanisms or build new ones through composition or refinement. We present a simple example of composition where *push* and *pull* failure monitoring models are mixed inside a *dual monitoring scheme*.

On the other hand, we consider the entities being monitored as abstract objects in the system and we eliminate the mismatch between (1) the need for failure detection at the level of application objects and (2) the support provided by some operating systems to detect host failures. One can configure the failure detection service in such a way that the monitored units can range from specific application objects, to threads, processes, machines, or even subnets. We give an example of a scalable hierarchical configuration and we discuss how the randomized gossiping scheme of [14] can be developed with our infrastructure.

1.3 Current Status

We developed our failure detection service in the context of the European ESPRIT project OpenDREAMS,[1] which aims at providing a CORBA compliant reliable framework for supervision and control systems. The failure detection service is a part of a family of services that address the dependability requirements of distributed applications [7, 8]: these also include an *Object Group Service* and a *Consensus Service* [9]. We experimented the portability of our failure detection service on three ORBs: ORBIX [10], VisiBroker [15], and ORBacus [6]. As we point out in the paper, our experiments revealed some fundamental variations in the way these ORBs handle timeouts on remote invocations. Our work in the context of CORBA, and particularly our failure detector model, has significantly influenced many of the proposals that have recently been made to the OMG in the context of its undergoing standardization effort towards fault-tolerant CORBA [13].

More recently, we also implemented our failure monitoring architecture in the form of a Java component. We packaged our service as a programmatic general purpose API useful for distributed enterprise computing. We used for that several interfaces defined in the Enterprise Java platform [1].

1.4 Roadmap

The rest of this paper is organized as follows. Section 2 presents our generic failure detection architecture, recalls the two well known failure monitoring models, namely *push* and *pull*, and then introduces our generic dual monitoring scheme that combines the properties of those two models. Section 3 discusses the interactions between the components of the failure detection service. Section 4 describes how our architecture helps the configuration of failure detection to scale according to the underlying network. Section 5 and Section 6 discuss some issues we faced when implementing our architecture as a CORBA service and as a Java component, respectively. Finally, Section 7 presents some concluding remarks.

2 A Generic Failure Detection Architecture

Roughly speaking, a failure detection service is a distributed oracle (a *monitor*) aiming at providing some distributed objects (*notifiable* objects) with hints about the crash of other objects (*monitorable* objects).

This section presents the architecture of our object monitoring service. The service is generic in the sense that it supports several interaction styles and may be configured in various ways. The interfaces of the monitoring service are arranged in a hierarchy that provides different views of the service and different interaction paradigms for failure detection (Figure 1). In particular, the hierarchy includes specialized interfaces for the *push* and the *pull* execution styles. A dual monitoring model is defined in a clean way by simply inheriting from the push and pull models. For simplicity of presentation, the interfaces have been intentionally kept

[1] Projects 20843 and 25262.

minimal. In particular, management operations used to configure the failure detectors have been partially omitted.

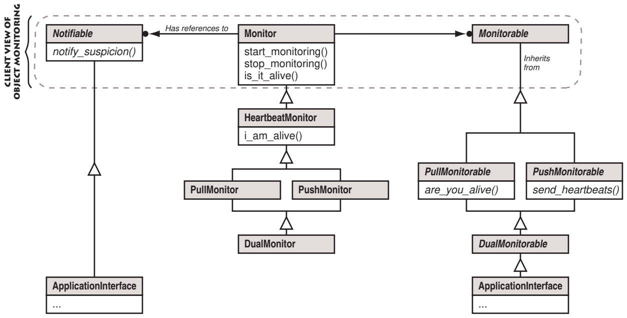

Figure 1. Class Diagram of the Object Monitoring Service Interfaces

2.1 Application-Oriented Interfaces

Client applications that use the service for monitoring remote objects have *a limited view* of the service, restricted to the three topmost interfaces of Figure 1. These interfaces abstract the flow model used for object monitoring. As a consequence, applications that use the service do *not* need to care about the interaction paradigms used for monitoring objects. In particular, this makes it possible to mix several monitoring models in the same distributed application with no impact on the clients.

These three interfaces abstract the roles of objects involved in a monitoring system:

- *Monitors (or failure detectors)* are the objects that collect information about component failures. In this paper, we focus on *failure monitoring*, and we consider the terms "monitor" and "failure detector" as equivalent.
- *Monitorable objects* are objects that may be monitored, i.e., the failure of which may be detected by the failure detection system.
- *Notifiable objects* are objects that can be registered by the monitoring service, and that are asynchronously notified about object failures.

Monitorable and notifiable objects are generally application-specific. In other words, the interfaces deriving from `Monitorable` and `Notifiable` are interfaces that the application must support for the service to call back to the application. Default implementations of monitorable objects are provided by our service. However, these objects must be instantiated by the application.

2.2 Service-Oriented Interfaces

In contrast to the monitorable and notifiable interfaces, monitors are implemented by the service and do not need to be instantiated by the application. More precisely, the interfaces deriving from `Monitor` are service objects (Figure 1), the implementation of which is provided by the service. These interfaces abstract the behavior of monitoring protocols and the way the information about component failures is propagated in the system, i.e., the *flow policy*. There are two basic forms of unidirectional flow, *push* and *pull*, plus several variants [11]. These flow policies correspond to simple monitoring protocols. We outline below these protocols and we describe a new one that results from a combination of *push* and *pull* monitoring schemes.

The Push Model

In the *push* model, the direction of control flow matches the direction of information flow. With this model, monitorable objects are active. They periodically send *heartbeat* messages to inform other objects that they are still alive. If a monitor does not receive the heartbeat from a monitorable object within specific time bounds, it starts suspecting the object. This method is efficient since only *one-way* messages are sent in the system, and it may be implemented with hardware multicast facilities if several monitors are monitoring the same objects.

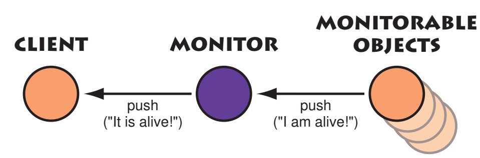

Figure 2. The Push Model for Object Monitoring

Figure 2 illustrates how the push model is used for monitoring objects. Note that the messages exchanged between the monitor and the client are different from the heartbeat messages sent by monitorable objects. The monitor generally notifies the client *only* when a monitorable object changes its status (i.e., becomes suspected or is no longer suspected), while heartbeat messages are sent continuously.

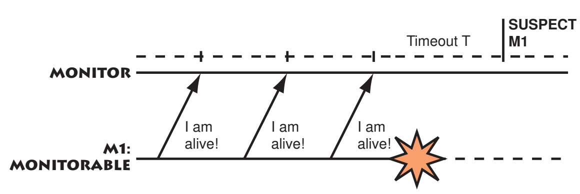

Figure 3. Monitoring Messages in the Push Model

The messages exchanged between the monitor and the monitorable object with a push-style protocol are shown in Figure 3. The monitorable object periodically sends heartbeat messages to the monitor. Upon message reception, the monitor sets a timer that triggers a suspicion if it expires before the reception of a new heartbeat message from the same object.

The Pull Model

In the *pull* model, information flows in the opposite direction of control flow, i.e., only when requested by consumers. With this model, monitored objects are passive. The monitors periodically send *liveness requests* to monitored objects. If a monitored object replies, it means that it is alive. This model may be less efficient than the push model since *two-way* messages are sent to monitored objects, but it is easier to use for the application developer since the monitorable objects are passive, and do not need to have any time knowledge (i.e., they do not have to know the frequency at which the monitor expects to receive messages). Figure 4 illustrates how the pull model is used for monitoring objects.

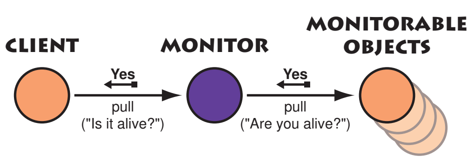

Figure 4. The Pull Model for Object Monitoring

The messages exchanged between the monitor and the monitorable object with a pull-style protocol are shown in Figure 5. The monitor sends periodically a liveness request to the monitorable objects, and waits for a reply. If it does not get the reply, a timeout triggers a suspicion.

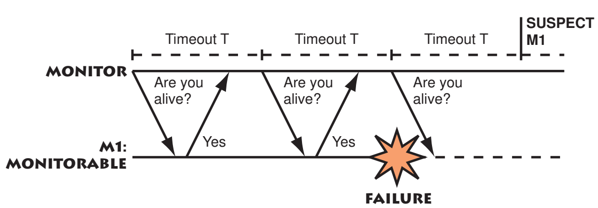

Figure 5. Monitoring Messages in the Pull Model

2.3 The Dual Scheme

In the pull model, the monitor parameters (e.g., timeouts, which may need dynamical adjustment) need only reside in the monitor and are not distributed in all the monitorable objects. However, using push style communication between monitor and monitorable objects is more efficient and may reduce the number of messages generated when using hardware multicast facilities (such as IP multicast) if several monitors are listening to the heartbeats.[2] Both models are thus complementary, and the type of interaction to use depends on the nature of the application.

Therefore, we introduce a model resulting from the combination of the two models, called the *dual* model, in which the push and pull models can be used at the same time with the same set of objects. Informally, the dual monitoring protocol works as follows. The protocol is split in two distinct phases. During the first phase, all the monitored objects are assumed to use the push model, and hence to send *liveness messages* (heartbeats). After some delay, the monitors switch to the second phase, in which they assume that all monitored objects that did not send a heartbeat during the first phase use the pull model. In this phase, the monitors send a *liveness request* to each monitored object, and expect a *liveness message* (similar to the push model) from the latter. If the monitored object does not send this message within some specific time bounds, it gets suspected by the monitor.

Our dual model is not a new failure detection protocol per se. It should rather be viewed as a way to mix different styles of monitoring *without requiring the monitor to know which model is supported by every single monitorable object*. It hence provides more flexibility by letting monitorable objects use the best suited interaction style.

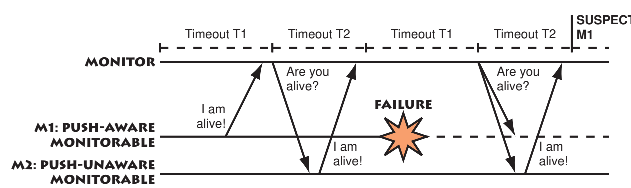

Figure 6. Monitoring Messages in the Dual Model

Figure 6 illustrates object monitoring with a dual-style protocol. In this example, two objects are being monitored. The first object, $M1$, is push-aware, i.e., it is active and periodically sends liveness messages (heartbeats). The second one, $M2$, is not push-aware, i.e., it only sends liveness messages when it is asked to. The monitor uses two timeout periods $T1$ and $T2$ for phases 1 and 2. It expects liveness messages of push-aware monitorable objects during phase 1. After $T1$, the monitor switches to phase 2 and sends a liveness request to each monitorable object from which it did not receive a liveness message, expecting a reply during $T2$. After $T2$, the monitor suspects every process from which it did not receive a message. In this example, $M1$ sends a liveness message during $T1$ in the first phase, and crashes soon after. In the second phase, the monitor sends a liveness request to $M1$, but does not get a liveness

[2]Note that heartbeat messages generated by a large number of monitorable objects may also inadvertently flood the network, while this situation is easier to detect and avoid if the messages are sent from less sources.

message before the end of $T2$. Thus, it starts suspecting $M1$ to have crashed.

3 Basic Interactions of Failure Detection Components

This section describes the interface and semantics of the failure detection components by presenting the interactions between these components. There are two types of interactions between application clients, monitors, notifiables, and monitorable objects:

1. *Monitor* $\leftrightarrow$ *client* and *monitor* $\leftrightarrow$ *notifiable:* this interaction allows the application to obtain information about object failures.

2. *Monitor* $\leftrightarrow$ *monitorable:* this interaction is performed by the monitoring service to keep track of the status of monitorable objects.

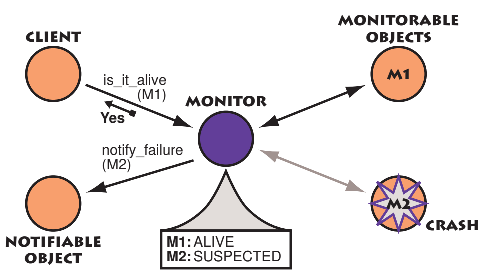

Figure 7. Components and Interactions of an Object Monitoring System

Figure 7 illustrates the components and interactions of an object monitoring system. This sample configuration comprises a client, a notifiable object, a monitor, and two monitorable objects $M1$ and $M2$. The monitor keeps track of component failures. The client explicitly asks the monitor about the status of monitorable objects. Upon the crash of a monitored object, the monitor asynchronously informs the notifiable object of the failure.

3.1 Monitor $\leftrightarrow$ Monitorable

The basic interaction paradigm of the monitoring service consists in having monitors and monitorable objects communicate with each other using remote method invocations. When using the push execution style, monitorable objects periodically invoke the `i_am_alive()` operation of the monitors they are registered with, in order to advertise the fact that they are alive. When using the pull execution style, monitors periodically invoke the `are_you_alive()` operation of monitorable objects; this operation is one-way, and the monitorable objects should react by invoking the `i_am_alive()` operation of the monitor that originally issued the liveness request. When using the dual execution style, these interfaces allow to marry the push and the pull models. During the first phase of the dual protocol, the monitor assumes that all monitorable objects are using the push execution style, and expects heartbeat messages. During the second phase, the monitor assumes that all monitorable objects from which it did not receive a heartbeat are using the pull execution style. Thus, it sends liveness requests to these objects.

The default way for a monitor to keep track of the status of the components in the system is to periodically check whether they are alive or not. This information is stored in a local table, and given to clients when they ask about the status of a particular object. Liveness information is typically associated with a *time-to-live* value (which may change on a per-object basis) telling when to invalidate and re-evaluate the suspicion information. Another way to obtain information about the status of monitored objects is to do it on client's demand (lazy evaluation). With this scheme, the monitorable object is checked on client demand (i.e., when the client asks the monitor for the status of an object). This makes the system less reactive since the client has to wait for the liveness request to return before it knows the object's status. However, monitoring objects solely on client's demand may significantly reduce the number of monitoring messages exchanged in the system.

3.2 Monitor $\leftrightarrow$ Client and Monitor $\leftrightarrow$ Notifiable

A client can ask the monitor to start and stop monitoring an object by invoking the `start_monitoring()` and `stop_monitoring()` operations, and obtain the status of an object by invoking the `is_it_alive()` operation. From a monitor's point of view, a monitored object can have one of three states:

- *SUSPECTED* means that the object is suspected by the monitor.

- *ALIVE* means that the object is considered as alive by the monitor.

- *DONT_KNOW* means that the object is not being monitored.

Although most applications need to invoke the monitor synchronously at specific points during protocol execution, it may sometimes be useful to receive asynchronous notifications when the state of an object changes. In particular, when protocols are implemented using a state machine approach, a suspicion can be seen as an event that triggers some specific action. In this situation, asynchronous suspicion notifications greatly reduce the complexity of the protocol's implementation.

A parameter of the `start_monitoring()` operation allows us to register an object with the `Notifiable` interface. The monitor invokes the `notify_suspicion()` operation of each registered notifiable object when the status of a monitored object changes (if an object becomes suspected, or if an object that was previously suspected is discovered to be alive). The client may still pass a null reference as notifiable object if it is not interested in asynchronous notifications.

4 Putting Failure Detectors to Work

This section describes how our generic failure detection architecture can be applied to various system configurations. We consider the sample network topology of Figure 8, where several clients, monitors, and monitorable objects are split over three different *Local Area Networks* (LANs). We present how these monitors can be configured in a hierarchy, and how they can use gossip-style protocols to reduce the number of messages exchanged in the network.

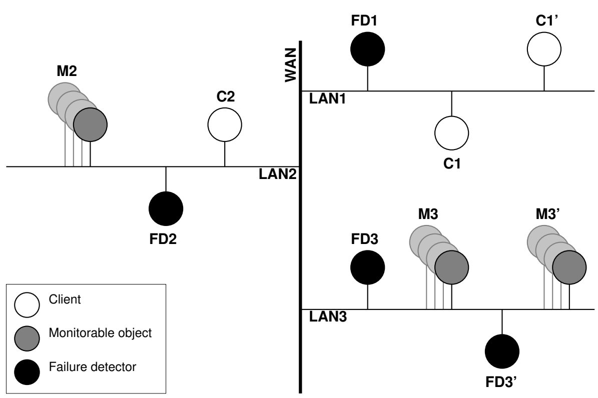

Figure 8. A Sample Network Topology

The problem of scalability is a major concern for a monitoring service that has do deal with large systems. A traditional approach to failure detection is to augment each entity participating in a distributed protocol with a local monitor that provides it with suspicion information. However, this architecture raises efficiency and scalability problems with complex distributed applications, in which a large number of participants are involved. In fact, if each participant monitors the others using point-to-point communication, the complexity of the number of messages is $O(n^2)$ for n participants. Wide area communication is especially costly and increases the latency of the whole system. It is thus very important to reduce the amount of data exchanged between distant hosts.

4.1 A Hierarchical Configuration

The interfaces of our monitoring service make it easy to configure the monitoring system in a hierarchy, as shown in Figure 9. In a LAN, one or several failure detectors can keep track of the state of all local monitorable objects, and transmit status information to remote monitors in other LANs, thus reducing the number of costly inter-LAN requests. Similarly, the developer may choose to install one monitorable object per host, per process, or per thread, depending on the kinds of failures that she/he considers. These configuration choices may be taken at runtime, and do not require modifications in the interfaces of the service.

A monitor may receive liveness information about a specific monitorable object from another monitor rather than directly from the monitorable object. This second-hand information may be obtained in two ways:

1. By asking the other monitors about the status of each individual object;

2. By transmitting complete tables of suspicion informations, thus reducing the communication overhead. This solution requires an extension to the service's interfaces of Section 2 in order to transmit these tables.[3]

The hierarchical configuration is independent of the model used for monitoring objects (push, pull, or dual model). It permits a better adaptation of monitor parameters (such as timeouts) to the topology of the network or to the location of monitored objects, and reduces the number of messages exchanged in the system between distant hosts. A monitor located in a LAN can adapt to the network characteristics and provide a specific quality of service. The reduction of network traffic, especially when a lot of monitorable objects and clients are involved, is the main reason for the good scalability of this hierarchical approach.

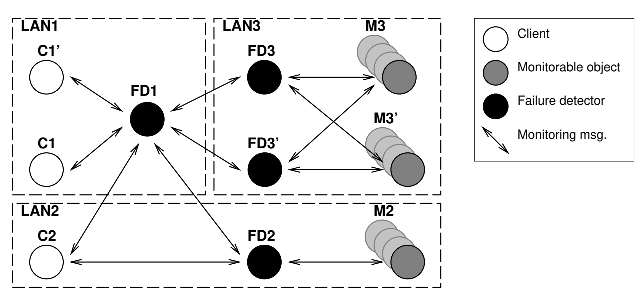

Figure 9. A Typical Hierarchical Configuration

In the hierarchical configuration of Figure 9, two groups of objects ($M3$ and $M3'$) are both being monitored by two

[3] A simple extension consists in subclassing the interfaces to add an operation to get the status of several objects from a list (e.g., `are_they_alive()`), and an operation to inform about the status of multiple objects (e.g., `they_are_alive()`).

distinct monitors (or failure detectors) ($FD3$ and $FD3'$) in a LAN ($LAN3$). Two clients ($C1$ and $C1'$) are located in another LAN ($LAN1$), and monitor the former objects indirectly through a local failure detector ($FD1$). There are two distinct paths between $FD1$ and the monitorable objects, making the failure of $FD3$ or $FD3'$ transparent to the clients. However, there is no redundancy in $LAN1$, and the failure of $FD1$ would prevent clients from getting liveness information about $M3$ and $M3'$. This configuration example reduces inter-LAN communication, when compared to a traditional approach with one local monitor per client, and messages exchanged between each failure detector and monitorable object.

An interesting extension of the hierarchical configuration consists in viewing a monitorable object as *a monitor object that only monitors itself, and that never suspects itself.* A call to `mon->are_you_alive()` would be replaced by `mon->is_it_alive(mon)`. The new interfaces would be simpler, since there would be no monitorable object, and would provide a clean and orthogonal design of hierarchical object monitoring.

Since a link may break anywhere in the hierarchy, a set of simple rules for hierarchical invocations helps determining if a particular object is suspected to have crashed or not:

- If a failure detector says that a monitorable object is alive, this object was actually alive some time before.
- If an invocation to a failure detector fails when asking for the status of a monitorable object, the invoker must assume that the object is suspected by the failure detector.
- If there is more than one path leading to a monitorable object, and this object is not suspected by the failure detectors of at least one path, it must be considered as alive.

4.2 A Gossip-Style Protocol

In gossip protocols, unlike in traditional protocols, a member forwards new information to *randomly chosen* members. Gossip protocols tend to combine the flexibility of hierarchical dissemination with the robustness of flooding protocols (in which a member diffuses the information to all its neighbors or to all other members). In [14], a simple, yet powerful, gossip style protocol is proposed for detecting remote component failures in a distributed environment. In this protocol, each member maintains a list of values that indicates for each member a strictly increasing heartbeat counter. Members occasionally send their list to randomly chosen members, or broadcast it to all members. Upon reception of a list, a member merges the old and the new lists by keeping the maximum heartbeat counter of each member. This protocol has been extended to scale well in the Internet, by using the network topology and handling gossiping in a different ways in LANs and across LANs.

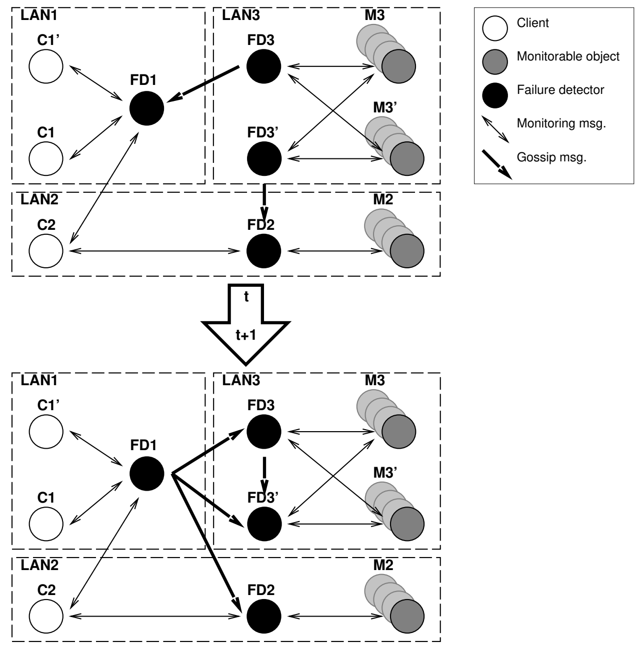

Figure 10. Two Typical Dynamic Configurations

One can easily build this protocol with our generic architecture, by having specific implementations of failure detectors that occasionally send their suspicion information to other failure detectors. The interaction between failure detectors and clients/monitorable objects is not affected. Unlike the hierarchical configuration presented in Section 4.1, sending information to randomly chooses failure detectors creates an invocations graph that evolves in time as a *dynamic hierarchy*.

Figure 10 presents two dynamic configurations of failure detectors that use a gossip-style protocol. Thick arrows denote gossip messages exchanged between failure detectors. One can see in the bottom invocation graph that $FD1$ broadcasts its list to all members instead of sending it to a randomly chosen member.

5 Failure Detection as a CORBA Service

Our CORBA implementation of failure detection has been written in C++ and tested with three different ORBs: Orbix [10], VisiBroker [15], and ORBacus [6]. In this section, we first point out some issues related to timeout-based failure detection in these ORBs. Then, we describe the IDL interfaces of our failure detection service and we discuss some of its configuration characteristics.

5.1 On the Use of Timeouts in CORBA

Ultimately, any failure detection implementation is based on timeouts or time events. Associating a timeout value to a remote invocation specifies how long one has to wait for a reply from a potentially failed object. The CORBA 2.x specification [12] leaves open how request timeouts should be handled. This is an important concern when using two-way invocations for object monitoring and leads to proprietary and incompatible ways to handle timeouts. In particular, the three ORBs that we have used differ by the following properties:

- *Timeout resolution:* millisecond or second.
- *Exception raised upon timeout:* `CORBA::NO_RESPONSE` or `CORBA::COMM_FAILURE`.
- *Lifetime of timeout settings:* One request or lifetime of the connection.
- *Semantics of timeouts:* Timeout occurs if *any* blocking period (request or reply) exceeds specified timeout, timeout is the maximal time the caller has to wait for the reply, or two timeouts must be specified independently for the request and for the reply.

These differences make it difficult to develop a *portable* implementation of the monitoring service. Therefore, we have chosen to handle timeouts on top of CORBA using one-way invocations (without timeout) for interactions between failure detectors and monitorable objects.[4] The request invoking party uses its own notion of time to provide the timeout semantics it needs.

5.2 Monitoring Service IDL Interfaces

Figure 11 presents excerpts[5] of the IDL interfaces of our CORBA object monitoring service. These interfaces correspond to the class diagram of Figure 1 in Section 2.

An alternative approach to object monitoring would consist in reusing portions of the CORBA event and notification services, which provide the two basic push and pull interaction models and asynchronous notification facilities. Failure detectors would be specific implementations of event channels, and monitorable objects would be consumers or suppliers depending on the monitoring model. Although some mechanisms of these services may be reused, this approach requires important modifications to their interface to match the failure detection problem (e.g., to start monitoring an object, or to inquire about the status of monitored object).

[4]Note that CORBA one-way invocations provide only best effort semantics. A better solution would be to use OMG's forthcoming messaging service, which provides various qualities of service such as asynchronous reliable communication.

[5]Management operations have been omitted.

```
// IDL
module mMonitoring {
  // Client interfaces for all flow models
  enum Status { SUSPECTED, ALIVE, DONT_KNOW };

  interface Monitorable {
  };

  interface Notifiable {
    void notify_suspicion(in Monitorable mon,
                          in boolean suspected);
  };

  interface Monitor {
    void start_monitoring(in Monitorable mon,
                          in Notifiable not);
    void stop_monitoring(in Monitorable mon,
                         in Notifiable not);
    Status is_it_alive(in Monitorable mon);
  };

  // Interfaces for all models
  interface HeartbeatMonitor : Monitor {
    oneway void i_am_alive(in Monitorable mon);
  };

  // Pull model
  interface PullMonitor : HeartbeatMonitor {};

  interface PullMonitorable : Monitorable {
    oneway void are_you_alive(in PullMonitor mon);
  };

  // Push model
  interface PushMonitor : HeartbeatMonitor {};

  interface PushMonitorable : Monitorable {
    void send_heartbeats(in PushMonitor mon,
                         in long frequency);
  };

  // Dual model
  interface DualMonitor : PullMonitor,
                          PushMonitor {};

  interface DualMonitorable : PullMonitorable,
                              PushMonitorable {};
};
```

Figure 11. IDL Interfaces for the Object Monitoring Service

5.3 System Management and Configuration

Besides the object monitoring interfaces previously described, the monitoring service defines interfaces for managing the configuration of the hierarchical system. These interfaces define administrative operations for tasks such as linking failure detectors together, exchanging suspicion tables, or finding invocation paths to monitored objects. Describing these interfaces is not in the scope of this paper since they are not explicitely used by the clients of the service. However, some aspects of system configuration are of direct interest for clients. In particular, clients need to discover the failure detectors currently running in the system, and get hints about their relative proximity. In our implementation, this information is given to clients through the CORBA *Naming Service*.

The naming service maintains name-to-object mappings in a federated architecture. These name-to-object associations are called *name bindings*. A name binding is defined relative to a *naming context*, which is a CORBA object responsible for maintaining a set of bindings with unique names. Different names can be bound to an object in the same or in different contexts at the same time. Because a context is like any other object, it can also be bound to a name in a naming context. Binding contexts in other contexts creates a naming graph. We use this naming graph to describe the hierarchical architecture of our monitoring service.

Although CORBA aims at providing location transparency, we need to have some knowledge of the network topology in order to take advantage of it in the monitoring service. Therefore, we map topological domains (e.g., LANs) to naming contexts in a hierarchical fashion. Each failure detector has an entry in its domain, and is tagged with an attribute that identifies it as a failure detector. Monitorable objects can (but are not obliged to) register themselves in the naming service, in the context corresponding to their local domain. The clients can thus know which failure detectors are local to their domain, and can take advantage of this information. When invoking a failure detector located in a different naming graph than the current domain, it is likely to involve wide area communication. Although this approach based on the naming service has some obvious limitations, it has the advantage of being simple to use and maintain, and of being based on standard interfaces.

The CORBA *Trading Object Service* provides an alternate way for a client to locate a nearby failure detector. The trading object service is similar to the naming service, but maintains (typed) properties-to-object mappings rather than name-to-object mappings. The trading object service is built as a federation of traders. In short, the trading object service provides the functionality of *yellow pages* where the naming service acts as *white pages*.

With the trading object service, a client contacts a trader requesting a monitor that corresponds to some properties, e.g., physical proximity. The trader then hands back a reference to the monitor that best satisfies the requirements. We did not however implement this solution, since there still exist only few implementations of the trading object service when compared to the naming service.

6 Failure Detection as a Java Component

The second implementation of our monitoring architecture comes in the form of a Java component. Our goal with this Java component is to provide a general purpose API useful for distributed enterprise computing. We make use of several interfaces defined in the Java Enterprise platform [1]. This technology defines a model for the development and deployment of reusable Java server components, i.e., pre-developed pieces of application code that can be assembled into working application systems. The Java monitoring services uses the following Java enterprise APIs:

- The *Java Remote Method Invocation* (RMI) API for communications bewteen remote objects. This API provides native Java support for invocations between remote components. RMI is used for synchronous invocations in the monitoring service, such as querying a failure detector.
- The *Java Messaging Service* (JMS) API for optional efficient support of the push model through publish/subscribe technology. This API supports asynchronous communication in the Java platform through mechanisms such as reliable queues and publish/subscribe services. Several implementations of the JMS specifications already provide publish/subscribe communication through lightweight and efficient mechanisms based on hardware multicast facilities. Such facilities greatly improve the scalability of the system when using a push monitoring model.
- The *Java Naming and Directory Interface* (JNDI) API to access the naming services that maintain the information about the failure detector and the objects they are monitoring in the system. This API provides unified access to several types of naming and directory services, such as DNS, NDS, NIS+, LDAP, and COS Naming.
- Although not supported in the current implementation, the service will use the *Java IDL* API for reusing portions of the current CORBA implementation, and interoperating with it through IIOP. This API creates remotes interfaces to support CORBA communication in the Java platform.

While our Java implementation is less mature than our CORBA implementation, it has several advantages on its counterpart. Being implemented completely in Java, it benefits from Java's "write once run everywhere" property. Since Java objects can be transmitted by value, it is possible to migrate objects (such as failure detectors) from one location to another one. This can be also useful for uploading specialized implementations of monitorable objects to sites that must be monitored. Finally, by using the Java Enterprise APIs, it relies on standard interfaces widely accepted by the industry, while benefiting from advanced facilities like efficient asynchronous communication.

7 Concluding Remarks

Middleware systems usually provide a set of services for naming, trading and management, or higher level business domain specific objects, e.g., financial applications. Very few services however provide support for managing failure situations and this can be viewed as a serious drawback for critical applications where high availability is a major concern. The OMG is very aware of this shortcoming and is in the process of standardizing a set of interfaces for *Fault-Tolerant CORBA* [13].

In this paper, we advocated an approach where failure detection is considered as a service among well established services like naming and file management. Our approach follows the CORBA tradition of service design and has influenced many of the proposals that have recently been made to the OMG standardization effort in the area of fault-tolerance.

We do not claim that a failure detection service should be used by all developers. There are indeed many applications where failure detection would just be hidden behind other services that address reliability issues such as group membership or transaction management. On the one hand however, the modularity of services (like transaction or group membership) would be increased if the failure detection is encapsulated inside a separate component. On the other hand, applications such as supervision and control or network management systems directly need to handle failures. It is thus also important to encapsulate the complexity of failure detection inside *first class (application level) components* with well defined interfaces.

We did not describe, in this paper, a specific failure detection protocol, but we rather presented a modular architecture to compose and customize failure detection protocols according to the topology of the system and the communication pattern of the application. We described a dual monitoring scheme which results from the composition of the well known push and pull monitoring models.

We have implemented our monitoring architecture as a reusable component in two distributed environments: (1) as a monitoring service for CORBA, and (2) as a class library for Java. Our CORBA implementation of failure detection has been written in C++ and tested with three different ORBs: Orbix [10], VisiBroker [15], and ORBacus [6]. The Java version makes use of the Java Enterprise APIs [1].

Although we mainly targeted *failure* monitoring, we believe that one can extend our architecture to object monitoring in general, e.g., monitoring specific object values or properties.

References

[1] S. Asbury and S. Weiner. *Developing Java Enterprise Applications*. John Wiley & Sons, 1999.

[2] P. Bernstein. Transaction processing monitors. *Communications of the ACM*, 33(11):75–86, Nov. 1990.

[3] K. Birman. *Building Secure and Reliable Network Applications*. Manning Publications Co., 1996.

[4] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

[5] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[6] O.-O. Concepts. *ORBacus, Version 3.0 Documentation*. Object-Oriented Concepts, Inc., 1998. http://www.ooc.com/.

[7] P. Felber, B. Garbinato, and R. Guerraoui. The design of a CORBA group communication service. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, pages 150–159, October 1996.

[8] P. Felber, R. Guerraoui, and A. Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[9] R. Guerraoui and A. Schiper. Consensus service: a modular approach for building agreement protocols in distributed systems. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, pages 168–177, June 1996.

[10] IONA. *Orbix 2.3 Programming Guide*. IONA Technologies Ltd., Oct 1997.

[11] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 1997.

[12] OMG. *The Common Object Request Broker: Architecture and Specification*. OMG, February 1998.

[13] OMG. *Fault tolerant CORBA Using Entity Redundancy, Request For Proposal*. OMG, Apr 1998. http://www.omg.org/techprocess/meetings/schedule/Fault_Tolerance_RFP.html.

[14] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, Sept. 1998.

[15] Visigenic. *VisiBroker for C++ 3.2 Programmer's Guide*. Visigenic Software, Inc., Mar 1998.

[16] W. Vogels. World wide failures. In *Proceedings of the ACM SIGOPS 1996 European Workshop*, Sept. 1996.