Engineering a Failure Detection Service for Widely Distributed Systems

Bruno G. Catão , Francisco V. Brasileiro , Ana Cristina A. Oliveira

[1]Universidade Federal de Campina Grande
Coordenação de Pós-graduação em Informática
Laboratório de Sistemas Distribuídos
Av. Aprigio Veloso, 882, Bodocongó
58.109-970, Campina Grande, Paraíba, Brasil

{catao,fubica,cristina}@dsc.ufcg.edu.br

Abstract. *Unreliable failure detectors are recognized as important building blocks for implementing fault-tolerant distributed systems. Further, there has been a lot of discussion on how to provide them with sophisticated features that allow for adaptation, flexible use, scalability and quality of service enforcement. Despite that, we are not aware of any real distributed system that uses a sophisticated failure detection service. In fact, most systems deployed use the trivial failure detection scheme provided by the underlying communication technologies (e.g. TCP/IP timeouts). We believe that this state of affairs is due to two main reasons: i) there is no widely supported failure detection service API that incorporates these advanced features in a suitable way; and ii) the benefits of using a sophisticated failure detection service are not clearly understood. This paper targets the first issue by proposing a failure detection service that addresses the main necessities of widely distributed systems and implements the state-of-the-art in failure detection mechanisms. Moreover, to improve the usability of the service we took special care in the design of its programming interface.*

1. Introduction

In widely distributed systems it is normally not possible to establish an a priori upper bound on the end-to-end communication delays between processes. Therefore, failure detection in these systems is unreliable, since it is impossible to differentiate a crashed process from one that is simply running very slowly.

Nevertheless, in a seminal work, Chandra and Toueg proposed a theoretical framework to define the properties of useful unreliable failure detectors [Chandra and Toueg, 1996]. Following this theoretical work, various implementations of unreliable failure detectors were proposed (e.g. [Felber et al., 1999, Défago et al., 2003, Stelling et al., 1999, Hayashibara et al., 2004, Bertier et al., 2002]).

In addition, a lot of work has been devoted to extend the simple implementation sketched by Chandra and Toueg [Chandra and Toueg, 1996] and add more sophisticated features to a failure detection service. For instance, the first failure detection services proposed did not scale well. They generated too much control traffic when a large number of processes were monitored at the same time, degrading both the service as well as the network. Another problem is that these services were based on static timeouts. This leads to poor quality of service (QoS) in face of fluctuations in the execution environment. Further, earlier implementations did not provide

any guarantees concerning the QoS provided by the failure detection service. To address these problems, many approaches have been proposed in the literature. They provide ways to allow for scalability [Gemmell, 1997, Chu et al., 2000, Birman et al., 1999], adaptability [Chen et al., 2000, Bertier et al., 2002, Hayashibara et al., 2004], flexibility [Hayashibara et al., 2004] and QoS enforcement [Chen et al., 2000] to failure detection services.

Despite the many advances in the area, few (if any) real widely distributed systems use the sophisticated technologies developed so far. Instead, most systems seem to rely on very simple implementations based on the failure detection features provided by the underlying communication technologies (e.g. the default timeouts of TCP/IP sockets). We argue that two main reasons are responsible for this state of affairs. Firstly, since there is no widely supported ready-to-use failure detection service providing these functionalities, developers must cope with the cost of implementing the service from scratch. Secondly, the benefits of using a sophisticated failure detection service are not clearly understood. Therefore, it is not surprising that simpler solutions already available have been preferred.

As an attempt to reduce the gap between the state-of-the-practice and the state-of-the-art in failure detection for widely distributed systems, we propose the architecture of a failure detection service that encompasses sophisticated features in an easy to use way. Differently from previous works, our focus is both on the software engineering issues to develop a sophisticated and efficient failure detection service, as well as on the provision of a suitable programming interface for the service.

The remaining of the paper is structured in the following way. Section 2 specifies the requirements of a failure detection service for a widely distributed system. Section 3 summarizes the most suitable mechanisms proposed in the literature to implement the specified requirements. Section 4 discusses the service programming interface. Then, in section 5 we present the architecture of a failure detection service that efficiently accommodates the mechanisms discussed in Section 3. Section 6 shows how an application uses the service. Finally, Section 7 concludes the paper and presents some perspectives for future work.

2. Service Requirements

As discussed before, a failure detection service targeted for widely distributed systems must provide QoS guarantees at the same time that is scalable, adaptable and flexible. In this section we will explain in more details each one of these features.

Providing QoS guarantees is the ability of the service to be dynamically configured by applications to maintain a certain QoS level at runtime. The primary QoS metrics proposed by Chen et al. [Chen et al., 2000] are: detection latency, mistake recurrence time and mistake duration. The first metric is a random variable representing the total time that elapses from the time a process fails until its failure is permanently indicated by the failure detection service. The second metric is a random variable that represents the time between two consecutive mistakes (wrong suspicions of the failure detection service). Finally, the third metric is a random variable that represents the interval of time during which the failure detection service remains wrongly suspecting a process. Ideally, a client of the failure detection service should be able to define different upper and lower bounds[1] on the expected values of these metrics on a per-process basis.

A scalable failure detection service is able to monitor a large number of entities simultaneously without suffering a significant impact on its performance. The basic prob-

[1] Upper bounds for the first and third metrics, and lower bound for the second metric.

lem concerning scalability in widely distributed systems is how to transfer control messages from many sources to different distributed destinations without saturating neither the network nor the processing nodes, yet sustaining acceptable QoS levels.

Similarly, adaptability is the ability of the service to maintain an acceptable QoS even in face of changes in the execution environment. The environment over which widely distributed systems execute are very susceptive to these fluctuations. This is due to various factors, such as momentary increase on network traffic or CPU loads. A service designed to run in a widely distributed environment should be affected as less as possible by these fluctuations.

Finally, flexibility is the ability of the service to be usable by different applications. Conceptually, any service should be usable by the most different kinds of applications. In the context of a failure detection service this means that applications should be able to interfere in the task of deciding whether a process should be suspected or not. It is worth to point out that different decision requirements do not necessarily mean different QoS guarantees. A failure detection service for widely distributed systems must provide this flexibility level.

3. Providing the desired properties

Although the necessity of a failure detection service is evident, there are no ready-to-use failure detection service that provide all the desired properties described before. In this section we summarize the main contributions in the literature regarding the provision of these properties.

3.1. Providing scalability

To address this issue many works were developed. We can classify these works in two groups, the hierarchic and the epidemic solutions. The hierarchic solutions organize the entities in tree-based arrangements, in order to optimize the communication among the entities of the system [Ballardie et al., 1995, Chu et al., 2000, Jannotti et al., 2000]. On the other hand, the epidemic solutions work by diffusing its data using a random pattern that mimics the way infectious deceases spread themselves [Birman et al., 1999, Ganesh et al., 2001, Gupta et al., 2002]. Both solutions reduce the impact of the communication from an order of O(n^2) to an order or O($n \log n$), where n is the number of processes that monitor each other.

The main proposals to address scalability that relies on hierarchic approaches are based on multicast provided by the underlying network technologies (e.g. IP multicast) or data diffusion using overlay networks. The main advantage of the proposals based on native multicast technologies is that they are the most efficient way to transmit data from multiple sources to multiple destinations [Gemmell, 1997]. Other advantage is that all multicast functionalities are already implemented by the transport layer of the communication protocol. However, although there is more than a decade that such technologies have been released, the majority of routers still do not support them. Finally, another great disadvantage of these proposals is that some applications need QoS on the data diffusion, and native multicast technologies do not ensure this.

The main advantage of multicast trough overlay networks is that there is no necessity of a specific infrastructure. There is not even the necessity of a specific network protocol such TCP/IP. Other advantages are: it is possible to create specific solutions for each problem domain; the diffusion is transparent to the routers involved; the gain on scale due to the reduction of the amount of information stored and exchanged among routers;

and the simplified management. The main disadvantages of multicast through overlay networks are: bigger latency than in native multicast technologies, and the replication of messages in some links of the network.

Finally, a very interesting approach, that does not rely on explicit hierarchies, is the epidemic diffusion. Epidemic, or gossip, diffusion simulates the transmission of a infectious disease; an infected individual transmits its illness to other individuals in a randomized way. Besides being very simple, this method is also very effective. The first attempt to implement an epidemic protocol was done in the early 1980's. This protocol was developed for the USENET News Protocol. This protocol constructs a communication graph of a set of processes that should communicate. When a process wants to communicate with other processes it gossips its data to a subset of processes by choosing randomly some processes (at least $logn$ processes) and sending them the desired data. The processes that receive this data repeat this process several times, until reaching some probabilistic condition.

3.2. Providing adaptability

The main proposals to address adaptability in failure detection services are based on the use of dynamic timeouts. Following we will discuss the most important proposals.

One very important adaptable failure detector was proposed by Chen et al. [Chen et al., 2000]. Roughly speaking, this failure detector adapts itself by dynamically adjusting the timeout that each module sets when waiting for a heartbeat message (the lack of such message causes the service to start suspecting the process). This dynamic adjustment is done in a statistical way, sampling the heartbeat arrival times in order to estimate the future arrival times. A safety margin defined at deployment time is added to this estimated arrival time.

Another important adaptable failure detector was presented by Bertier et al. [Bertier et al., 2002]. Its adaptation mechanism is similar to the one shown in the previous solution with the difference that its safety margin is dynamically calculated using a mechanism based on the Jacobson's algorithm [RFC, 2000].

3.3. Providing flexibility

The first approaches intended to provide flexibility to failure detection services were based on the use of two timeouts [Defago, 2000, Defago et al., 1999, Defago et al., 1998]. By using these timeouts it is possible to configure the service to work at the same time with both applications that have conservative requirements (e.g. few wrong suspicions), as well as applications that have aggressive requirements (e.g. low detection latency).

The approaches based on two timeouts are useful, but they are not flexible enough for the wide range of requirements of widely distributed applications. To circumvent this problem, a new failure detector paradigm was proposed - the accrual failure detector [Hayashibara et al., 2003]. The main innovation introduced by the accrual failure detectors was the information provided by them. Traditional failure detectors export to applications binary information about the state of the monitored processes (suspected or correct). Accrual failure detectors, on the other hand, provide a probabilistic value associated with each monitored process, and this value indicates the confidence level about the liveness of the process.

3.4. Ensuring the quality of service

The proposals to ensure the QoS of a failure detection service are based on monitoring the QoS metrics, and, if required, adjusting the value of some control variables of the failure

detection service (e.g. the rate of heartbeat messages emission) so that it maintains its QoS as near as possible from an established set point [Chen et al., 2000].

In a heartbeat based implementation the heartbeat arrival times are stored in a sliding window of a pre-defined length, and the QoS metrics are calculated as a function of the values contained in this window. The length of the window is directly proportional to the accuracy of the QoS metrics and inversely proportional to the reactivity of these metrics. Also, the frequency of heartbeat sending, for each monitored process, is adjusted dynamically to meet the QoS requirements. For instance, if the detection latency is higher than the desired, the frequency of heartbeat sending is increased as a function of the difference between the desired and the measured detection latencies.

4. Service Programming Interface

Most of the papers that propose failure detection services pay little attention to the programming interface that applications use to have access to the service. They normally assume only a synchronous interface that allows clients to query the failure detection service for the status of a particular process. However, our experience developing real distributed systems [Our, 2005] shows that most of the time asynchronous interactions are preferred. This is because most systems are designed to act upon the occurrence of a failure. In other words, it is easier to program mechanisms for fault-tolerance by having them notified of the occurrence of a failure on a process, instead of having them constantly querying the failure detection service for the status of this process. This is also evident in the pseudo-code of many fault-tolerant algorithms presented in the literature (e.g. [Chandra and Toueg, 1996, Guerraoui and Raynal, 2005]). Of course, in some less frequent scenarios, synchronous interactions are also useful. For instance, before performing a costly task, it is worth querying the failure detection service to avoid the waste of resources (even though a query that returns that a process is correct gives no guarantees that it will remain correct in the near future).

In this section we propose an extended interface for a failure detection service that we believe is more useful for developers of fault-tolerant distributed systems. It provides both query-style and callback-style interactions. Different from previous proposals, it incorporates an asynchronous notification scheme that allows applications to register callback functions that are invoked whenever important events happen.

The architecture of the service proposed, shown in Figure 1, is based on the classic model that has three basic types of entities: monitors, monitorables and notifiables [Felber et al., 1999]. Monitor entities are responsible for probing the monitorable entities and notifying the notifiable entities whenever a failure happens. We extend this architecture by adding new relevant events, as described later in this section. Moreover, the monitor functionality is extended to allow for the monitoring of the QoS that is being provided, as well as the extra resource consumption due to the execution of the failure detection service.

There are four classes of events that may trigger a notification. The first class is related with the failure detection capabilities of the service. There are failure notification and wrong suspicion notification events. The second class comprises the QoS related notification events. They are used to notify applications when the current QoS levels attained by the service are below the QoS requirements defined by the application. There are three events in this class, one for each primary QoS metric (see Section 2). The third class comprises the events that are related with the impact that the service has on its execution environment. It has two events that are used to indicate, respectively, when the bandwidth consumption is above or below a given threshold. Finally, the fourth class is

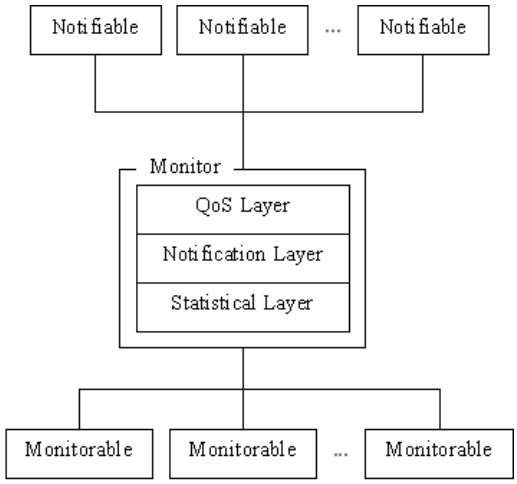

Figure 1: The main entities of the failure detection service.

related with configuration events. The single event of this class is used to notify notifiable entities of monitorable entities that are registered with some monitor entity. These events are useful for allowing notifiable entities to discover monitorable entities dynamically.

5. Implementing the service

In Section 2 we discussed the desirable properties that a failure detection service must possess. Then, in Section 3 we presented a summarized description of the most suitable mechanisms proposed in the literature to implement these properties. In this section we show how we packed together some of these mechanisms to implement a failure detection service that provides the programming interface described in the previous section.

The core of the failure detection service is implemented by the monitor entities. Monitorable and notifiable entities just implement simple interfaces to allow information to flow from monitorables to notifiables through a collection of monitors that cooperatively implement the service. The monitorable interface allows a monitor to probe a monitorable entity, while the notifiable interface allows a monitor to inform a notifiable entity that a relevant event has happened. This provides a lot of flexibility, since relevant events are specified on a per-entity basis. For instance, a given application may implement a notifiable entity that registers with a monitor to receive notifications of failures of a particular monitorable entity, whenever the probability of failure of the monitorable entity is above a particular confidence level.

To allow for adaptability and QoS enforcement it is important to have control on the rate with which control messages are exchanged between monitor and monitorable entities. Moreover, since the monitors are the entities responsible for controlling the behavior of the failure detection service, the communication among monitorable and monitor entities follows a pull style, i.e., the monitor entities send *"are you alive"* messages to monitorable entities, and monitorable entities respond to these messages by sending back *"I am alive"* messages. The pull communication model simplifies the control of the rate with which heartbeat messages are exchanged. If a push style were used, whenever a change in this rate was required, monitors would have to inform this to all monitorable entities.

The functionality of the monitor entities is implemented by three modules: the statistical module, the notification module, and the QoS module.

The statistical module receives control messages from monitorable entities and

computes two statistics about them: i) the probability that a process has failed, and; ii) the bandwidth generated by the failure detection service. It maintains a sliding window containing the size and the arrival time of each control message received. The probability that a process has failed is computed by calculating the average and the standard deviation of arrival times, using them to build a probability distribution and then, using its cumulative distribution function to obtain the probability value. By default the service uses a normal distribution, but other distributions can be used. The bandwidth generated by the failure detection service is calculated by adding the total of bytes received and dividing this by the time elapsed between the reception of the first and the last message in the window.

The notification module provides one of the interfaces that the notifiable entities have with the failure detection service. It allows the notifiable entities to specify the events whose occurrence will trigger the execution of a callback function. The other interface available to the notifiable entities is provided by the QoS module. It allows for both the definition of the desired levels of QoS, as well as the registration of callback functions that will be invoked whenever the current level of QoS attained is below the QoS requested.

Scalability was addressed in our implementation in an hierarchical way. Monitor entities were developed to behave also as notifiable entities. In this way, monitor entities could register with other monitor entities their interest in the state of some monitorable entities. Therefore, applications can build hierarchies of any number of levels following this pattern, as shown in Figure 2.

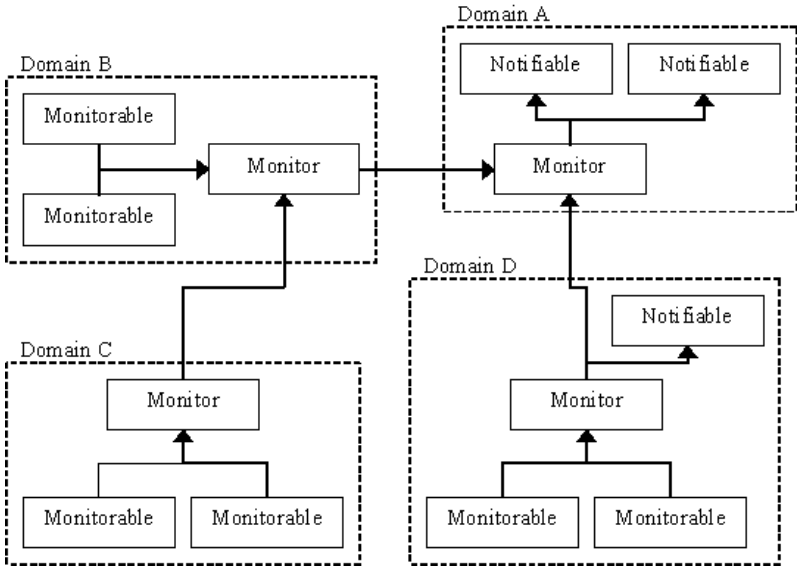

Figure 2: The monitors of the domain A, B, C and D compose a tree-based hierarchy; the notifiable entities of domain A could be registered as interested in monitorable entities of domain C.

To provide an asynchronous, transparent and simple way by which the entities of the system communicate, we have devised an event-based messaging service. This service is based on the concepts presented in [Starovic et al., 1995] and [Brasileiro et al., 2002].

The main entities of the messaging service are: Event Channel, Event Source, Event Listener, and Notification Context, as shown in Figure 3. The Event Channel hides from developers issues related to distributed communication. The Event Source entities are responsible for generating events and sending them to the event channel. The job of the Event Listener entities is to consume the events produced by the Event Source entities. Finally, the Notification Context is an abstraction of a domain within which the events exchanged by the entities will transit; this abstraction is very useful to reduce the network traffic delivering events only to the entities interested on them.

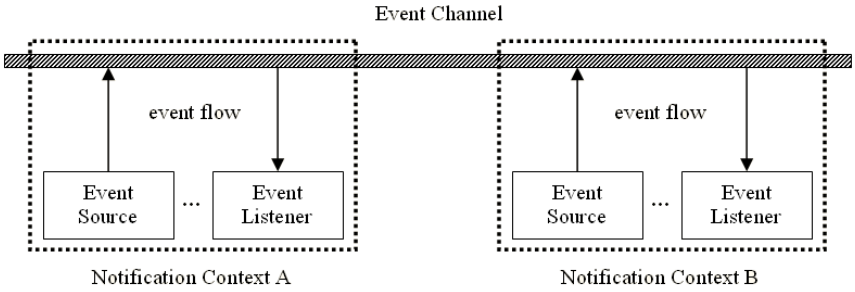

Figure 3: The entities of the messaging service.

It is worth to mention that the messaging service can be implemented in several ways. Our implementation uses Jini [JIN, 2005] as the communication technology. Another possibility is to use a gossip-style communication technology. We note that such an implementation would provide an alternative way to address the scalability issue in a more transparent way.

6. Using the Service

Figure 4 shows the interfaces of the main components of the failure detection service implemented.

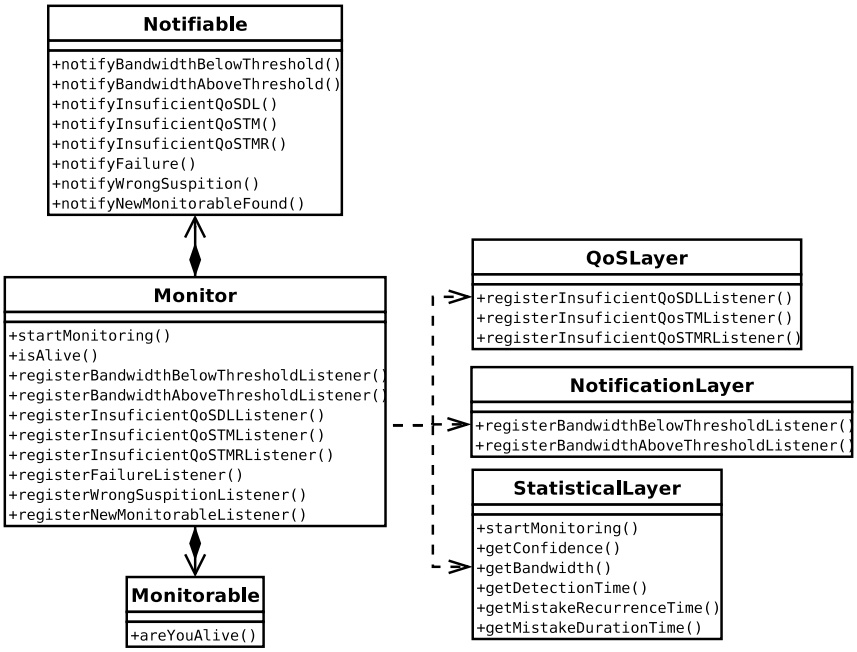

Figure 4: Simplified class diagram of the failure detection service displaying its main entities.

To use the service, applications must firstly obtain a monitor instance and then register with it the required monitorable and notifiable instances. A monitorable instance must be associated with every entity of the system that should be monitored. A notifiable instance must be associated with every entity of the system that is interested on the occurrence of relevant events. The actual monitoring is started by invoking the *startMonitoring* method on the appropriate monitor. For each pair of notifiable and monitorable entities a call to *startMonitoring* must be issued. By doing that, the invoker can define distinct QoS levels for each pair of notifiable and monitorable entities.

The process of locating and publishing entities in the failure detection service is accomplished through the messaging service. It defines primitives to manage the mem-

bership of the entities of the service. For every new instance of the service, the developers must call the *publish* method of the messaging service, passing to this method a name that uniquely identifies this instance in the system. The instances of the failure detection service can be searched by either their unique names or by their interfaces. In both cases the developers must call the *lookup* method, passing as a parameter the desired name or interface. The functionality of searching entities based on the interface is in fact a dynamic resource discovery mechanism. By using this feature, a monitor can, for instance, discover all the monitorable entities that are registered in the messaging service and start monitoring them, with no a priori knowledge about them. This is an alternative to using the configuration events described in Section 4. The sequence diagram in Figure 5 shows the steps that an application should take to use the failure detection service.

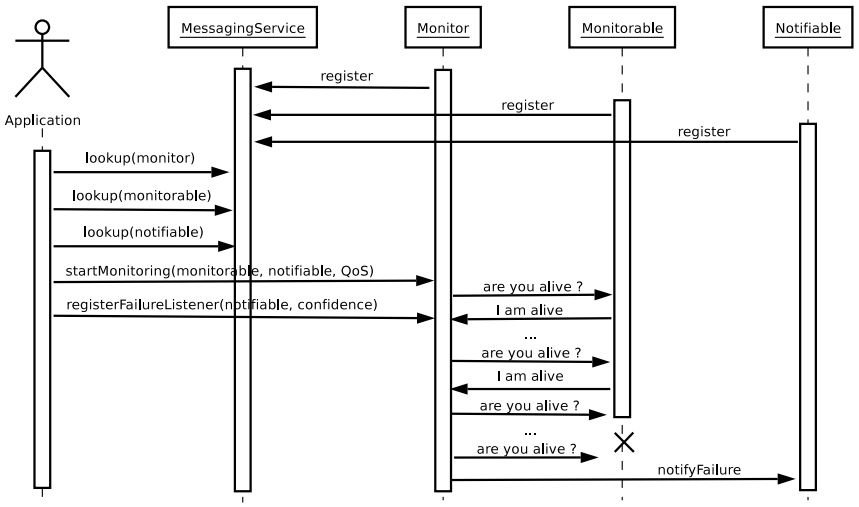

Figure 5: Sequence diagram showing how an application could use the failure detection service.

The monitorable interface is implemented by defining its *areYouAlive* method. This method is called by the monitor entities with which the monitorable is associated. A trivial implementation for this interface is to have the *areYouAlive* method always returning the *Monitorable.I_AM_ALIVE* value.

To implement the notifiable interface, developers must code the method that will be triggered by the monitor. The notifiable interface defines several methods that applications should implement according to their needs. See the notifiable interface in Figure 4.

The process of registering monitorable entities within a monitor entity is done in the following way. Firstly, it is necessary to obtain an instance of the monitor interface. Then, the application can call the *registerMonitorable* method of the monitor instance, passing to this method the required level of QoS and an instance for the corresponding monitorable entity.

The process of registering notifiable entities with a monitor entity is done in the following way. Again, the first step is to obtain an instance of a monitor entity. Then, it is possible to call one of the monitor's register methods. Applications must pass to these methods the parameters that specify when a notification should be triggered (e.g. the level of confidence that triggers a suspicion) and an instance for the correspondent notifiable entity.

The notification mechanisms are normally used as follows. First an appropriate QoS level is defined for each pair of notifiable and monitorable entities. This is indicated by invoking the *startMonitoring* method on the appropriate monitor passing as parameters the references for the notifiable and monitorable entities, as well as the QoS level

required. Since the QoS required may not be enforced by the failure detection service (for instance, because of the current load conditions of the environment), it is advisable that notifiable entities register to be notified whenever one of the QoS metrics is violated. When such a notification is received, the application may configure new QoS levels. For instance, sustain the level of mistake duration by augmenting the minimal detection latency required. Also, to balance acceptable QoS levels and resource usage, one can define lower and upper bounds on bandwidth consumption and register notification actions that are triggered whenever one of these thresholds are surpassed. When the upper limit on bandwidth consumption is signalled, the application may want to define less restrictive QoS levels in order to reduce control message traffic. On the other hand, when the resource consumption falls below the lower limit defined, application may want to increase its QoS level requirements.

Now we discuss the use of the failure detection service in the context of a particular example. Let us suppose that one wants to implement a service for supporting the installation, configuration and management of distributed applications. Such a system may be implemented by a centralized coordinator that executes at the management node and a number of distributed agents that run at the deployment nodes. The coordinator is responsible for instructing the appropriate agents to install a particular distributed application in all nodes that must run part of the application. Each agent is responsible for the installation of the part of the application that must execute at the agent's node. Moreover, a failure detection service is used for detecting the failures of both agents and applications, as well as crashes at the remote nodes. For simplicity, we assume that the coordinator does not fail. Whenever the part of the application or the agent that is running at a remote node are suspected, the coordinator is notified and takes the necessary actions to restart the application or the agent. Likewise, when a node fails, the coordinator chooses a new node to restart a new agent and the part of the application that was running at the faulty node. The failure detection service is used to notify the coordinator about failures on applications, agents and nodes.

To set the failure detection service up, first the coordinator instantiates a local monitor entity and registers itself to be notified of any monitorable entity that registers with this monitor. Whenever the coordinator is notified about a new monitorable, it registers itself with the monitor to be notified of the failures suspicions associated with this monitorable entity (with a confidence level that the coordinator indicates). Further, the coordinator starts the monitoring of the monitorable entity by invoking the *startMonitoring* method on the local monitor, passing as a parameter the required QoS level. Whenever a failure suspicion occurs, if the monitorable entity is associated with an application, then the coordinator contacts the corresponding agent, instructing it to reinstall the appropriate part of the application. On the other hand, if the monitorable entity is associated with an agent, then the coordinator may take one of two actions: i) if the application that runs on the same node of the agent is not suspected of having failed, then the coordinator will try to reinstall the agent at that node; ii) if both agent and application are suspected, then the coordinator assuming that the node has crashed, chooses a new node to start the agent and then tries to reinstall the application on this node.

At the distributed nodes, the failure detection service is set up in the following way. Each agent instantiates a local *Sensor*[2] that implements the monitorable interface on behalf of the agent. This *Sensor* locates the appropriate coordinator's monitor and registers itself with it. When an agent installs an application at its node, it instantiates a new *Sensor* to implement the monitorable interface on behalf of the application. This

[2] An auxiliary class *Sensor* that already implements the monitorable interface for generic Java objects is provided with the failure detection service API.

monitorable entity is also registered with the coordinator's monitor.

In this example the monitor that runs at the coordinator's node is responsible for probing all monitorable entities. If the number of monitorable entities is high, this may lead to a bottleneck. A trivial solution for this problem is to implement a hierarchy of monitors as discussed in Section 5.

7. Conclusion

Despite the many advances in the area of failure detection, few (if any) real distributed systems use failure detection services that implement the sophisticated mechanisms proposed in the literature. We believe that one of the causes of this state of affairs is the unavailability of a ready-to-use service that provides the appropriate API for most distributed applications. This paper tries to fill in this gap by presenting the architecture and the implementation of such a service[3].

As future work, we intend to provide alternative implementations of the messaging service based on gossip-style protocols. Further, we are integrating the service into the OurGrid[4] system [Our, 2005], a distributed grid middleware to support the execution of *bag-of-task* parallel applications. Our intent is to evaluate the benefits that a sophisticated failure detection service can bring to real distributed systems. We believe that the lack of perception of these benefits is another impairment to the broad utilization of services such as the one presented in this paper.

Acknowledgements

This work was partially developed in collaboration with HP Brazil R&D. Francisco Brasileiro would like to thank the financial support from CNPq/Brazil (grant 300.646/96).

References

(2000). Computing tcp's retransmission. N. W. Group. Rfc 2988. http://www.rfc-editor.org/rfc/rfc2988.txt.

(2005). The Jini Community. Sun Microsystems. http://www.jini.org.

(2005). The OurGrid Project. http://www.ourgrid.org.

Ballardie, T., Francis, P., and Crowcroft, J. (1995). Core based trees (cbt): An architecture for scalable multicast routing. In *ACM Sigcomm*, pages 88–95, San Francisco, USA.

Bertier, M., Marin, O., and Sens, P. (2002). Implementation and performance evaluation of an adaptable failure detector. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 354–363. IEEE Computer Society.

Birman, K. P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., and Minsky, Y. (1999). Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88.

Brasileiro, F. V., Greve, F., Hurfin, M., Narzul, J. P. L., and Tronel, F. (2002). Eva: an event-based framework for developing specialised communication protocols. In *IEEE International Symposium on Network Computing and Applications*, pages 108–119.

[3]The software is available for download at http://www.spotter.lsd.ufcg.br/.

[4]Available at http://www.ourgrid.org/.

Chandra, T. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267.

Chen, W., Toueg, S., and Aguilera, M. K. (2000). On the quality of sevice of failure detectors. In *International Conference on Dependable Systems and Networks (DSN'2000)*, pages 191–200, New York, USA.

Chu, Y.-H., Rao, S. G., and Zhang, H. (2000). A case for end system multicast. In *Measurement and Modeling of Computer Systems*, pages 1–12.

Defago, X. (2000). *Agreement-Related Problems: From SemiPassive Replication to Totally Ordered Broadcast*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland. Number 2229.

Defago, X., Felber, P., and Schiper, A. (1999). Optimization techniques for replicating corba objects. In *4th Int'l Workshop on Object-oriented Real-time Dependable Systems (WORDS'99)*, pages 1–8, Santa Barbara, CA, USA.

Défago, X., Hayashibara, N., and Katayama, T. (2003). On the design of a failure detection service for large scale distributed systems. In *Proc. Int'l Symp. Towards Peta-Bit Ultra-Networks (PBit 2003)*, pages 88–95, Ishikawa, Japan.

Defago, X., Schiper, A., and Sergent, N. (1998). Semi-passive replication. In *Symposium on Reliable Distributed Systems*, pages 43–50.

Felber, P., Guerraoui, R., Défago, X., and Oser, P. (1999). Failure detector as first class objects. In *International Symposium on Distributed Objects and Applications (DOA)*, pages 132–141, Edinburgh, Scotland,.

Ganesh, A. J., Kermarrec, A.-M., and Massoulie, L. (2001). SCAMP: Peer-to-peer lightweight membership service for large-scale group communication. In *Networked Group Communication*, pages 44–55.

Gemmell, J. (1997). Scalable reliable multicast using erasure-correcting re-sends. Technical report msr-tr-97-20, Microsoft Research Center.

Guerraoui, R. and Raynal, M. (2005). The information structure of indulgent consensus. *IEEE Transactions on Software Enginnering*, 54(4):453–466.

Gupta, I., Kermarrec, A., and Ganesh, A. (2002). Efficient epidemic-style protocols for reliable and scalable multicast. In *IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 180–189.

Hayashibara, N., Défago, X., and Katayama, T. (2004). The φ accrual failure detector. In *Symposium on Reliable Distributed Systems (SRDS'2004)*, pages 66–78, Florianópolis, Brazil.

Hayashibara, N., Défago, X., and Katayama, T. (2003). Two-ways adaptive failure detection with the φ-failure detector. In *Workshop on Adaptive Distributed Systems (WADiS03)*, pages 22–27.

Jannotti, J., Gifford, D. K., Johnson, K. L., Kaashoek, M. F., and O'Toole, Jr., J. W. (2000). Overcast: Reliable multicasting with an overlay network. pages 197–212.

Starovic, G., Cahill, V., and Tangney, B. (1995). An event based object model for distributed programming. In *OOIS (Object-Oriented Information Systems) '95*, pages 72–86, London. Springer-Verlag.

Stelling, P., DeMatteis, C., Foster, I. T., Kesselman, C., Lee, C. A., and von Laszewski, G. (1999). A fault detection service for wide area distributed computations. *Cluster Computing*, 2(2):117–128.