On the Design of a Failure Detection Service for Large-Scale Distributed Systems

Xavier Défago[*,†] Naohiro Hayashibara[*] Takuya Katayama[*]

[*]School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai, Tatsunokuchi, Ishikawa 923-1292, Japan

[†]"Information and Systems," PRESTO,
Japan Science and Technology Corporation (JST)

E-mail: {defago,nao-haya,katayama}@jaist.ac.jp

Abstract

It is widely recognized that distributed systems would greatly benefit from the availability of a generic failure detection service. There are however several issues that must be addressed before such a service can actually be implemented.

In this paper, we highlight the main issues related to ensuring failure detection in large-scale systems, and overview the main solutions proposed in the literature so far. Then, we outline a pragmatic architecture for a failure detector service based on the φ-failure detector, and a combination of techniques proposed in related work.

1 Introduction

The ability for a distributed system to detect the failure of its processes is widely recognized as an essential issue for fault-tolerance. In fact, virtually any practical fault-tolerant distributed application relies on a form of failure detection mechanism or another to react appropriately in the face of failures. In such applications, failure detection can be invoked either directly, or indirectly through the use of a group membership service or other group communication primitives (e.g., consensus, total order broadcast).

Our objective is to implement and provide a generic failure detection service for large-scale distributed systems. The idea of providing failure detection as an independent service is not itself particularly new (e.g., [3, 8, 21, 22]). However, several important points remain to be addressed before a truly generic service can be proposed. Unfortunately, although several problems have been studied in isolation, no existing implementation of failure detectors is effectively able to address all of them simultaneously.

In this paper, we discuss some critical problems that must be addressed to ensure failure detection in large-scale distributed systems, and overview the main solutions proposed so far. We then discuss a novel approach called φ-failure detection, and describe the architecture of our failure detection service, based on φ-failure detection and a combination of techniques proposed in related work. The last point constitutes the original contribution of this paper.

The remainder of the paper is structured as follows. Section 2 presents the basic assumptions and gives some important definitions. Section 3 overviews some important known results about failure detection. Section 4 surveys some of the most important solutions to address the scalability of failure detection. Section 5 describes the principle of φ-failure detection. Section 6 describes the architecture of the failure detection service, and the role played by φ-failure detection. Finally, Section 7 concludes the paper and discusses open questions.

2 System Model and Definitions

We represent a distributed system as a set of processes $\{p_1, p_2, \ldots, p_n\}$ which communicate only by sending and receiving messages. We assume that every pair of processes is connected by two unidirectional quasi-reliable communication channels [1]. A quasi-reliable channel is defined as a communication channel which guarantees (1) no message loss, (2) no message corruption, and (3) no creation of spurious messages. We consider that processes may only fail by crashing, and that crashed processes never recover.

We assume the system to be asynchronous in the sense that there exist bounds neither on communication delays nor on process speed. For each communication channel, we assume message delays to be determined by some random variable whose parameters are unknown, independent of other communication channels, and whose distribution is positively unbounded. We assume that the parameters of the random variable can change over time, but that they eventually become stable.

Formally, this system model is a little stronger than the asynchronous model described by Fischer et al. [10] because we make some assumptions on the probabilistic behavior of the system. However, our model remains weaker than any of the partially synchronous models defined by Dwork et al. [7], because the fact that the distribution is positively unbounded implies that no bound (known or unknown) can ever exist on communication delays.

The system model described in this section provides an adequate basis to study fault-tolerant group communication protocols, in particular when the timing behavior of the system is not guaranteed (e.g., unlike end-to-end real-time communication systems). Protocols developed on this basis tend to be quite robust as they do not rely on any strong timing guarantees. Large-scale communication over the Internet (including Grid systems) is for instance particularly prone to changing network conditions. Beside, heterogeneity, as well as unpredictable system loads, imply that the speed of process is not homogeneous and cannot be predicted accurately.

3 Failure Detection

In this section, we briefly introduce some of the most important concepts related to failure detection in distributed systems. Firstly, we introduce the theoretical foundation of failure detection (§3.1). Secondly, we present one of the most common implementations of failure detectors, namely the heartbeat approach (§3.2). Thirdly, we discuss a second aspect of failure detection: the notification of failures (§3.3). Distinguishing the detection of failures from the notification of their occurrence might be nearly useless in local networks, but it is essential in large-scale systems.

3.1 Unreliable failure detectors

Chandra and Toueg [3] define failure detectors as a distributed oracle with well-defined properties. A failure detector is a distributed entity which consists of a set of failure detector modules, one attached to each process. A failure detector module FD_p, attached to a process p, maintains a set of suspected processes. Process p can query its failure detector module at any time. Whenever some process q appears in the set maintained by FD_p, we say that p *suspects* q (that is, p suspects that q has crashed). The failure detector is however unreliable in the sense that its modules are allowed to make mistakes (1) by erroneously suspecting some correct process (wrong suspicion), or (2) by failing to suspect a process that has actually crashed. A module can also change its mind, for instance, by stopping to suspect at time $t + 1$ some process that it suspected at time t.

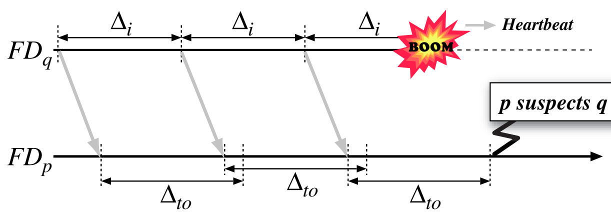

Figure 1. Heartbeat messages

Several classes of failure detectors are defined according to two properties which restrict the mistakes that the failure detector can make. For instance, a failure detector of class $\Diamond\mathcal{P}$ must meet the following properties of completeness and accuracy.

Property 1 (Strong completeness) *Eventually every process that crashes is permanently suspected by every correct process.*

Property 2 (Eventual strong accuracy) *There is a time after which correct processes are not suspected by any correct process.*

3.2 Heartbeat strategy

The heartbeat strategy to implementing failure detectors is quite common. In this strategy, every failure detector module periodically sends a heartbeat message to the other modules, to inform them that it is still alive (see Fig. 1). The period is determined by the heartbeat interval Δ_i. A process p suspects a process q if FD_p, the module attached to process p, fails to receive any message from FD_p for a period of time determined by a timeout Δ_{to}.

There is the following tradeoff. If the timeout Δ_{to} is short, crashes are detected quickly, but the likeliness of wrong suspicions is high. Conversely, if the timeout is long, the chance of wrong suspicions is low, but this comes at the expense of the detection time. Beside, the fact that the timeout is fixed means that the failure detection mechanism is unable to adapt to changing conditions. This is because a long timeout in some system setting can turn out to be very short in a different environment. Beside, in practical systems, network conditions can greatly vary over time (e.g., depending on external load).

3.3 Information propagation

In practice, failure detectors play two fundamental roles: detecting when monitored processes fail, and conveying this information to the monitoring processes. In local networks, these two roles are combined. This is not the case in large-scale distributed systems, where the two aspects should be

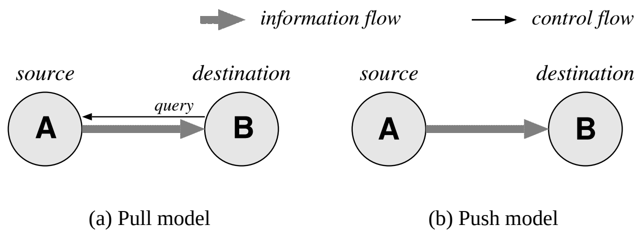

Figure 2. Interaction models for failure notification

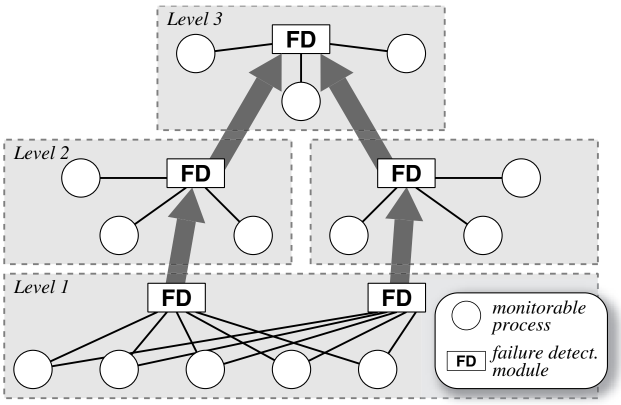

Figure 3. Hierarchical protocols

distinguished. Doing so allows to ensure that the detection of failures remains a local mechanisms, whereas the distribution of failure suspicions is done by some notification mechanism.

We now focus on the notification aspect of failure detectors. As with any other notification services, the information can be conveyed using two basic interaction models, namely the *push model* and the *pull model*. Figure 2 illustrates these two interaction models with two entities A and B. Although only the two endpoints are depicted here but, in the general case, there could be any number of intermediates on the path between A and B.

In the pull model, control and information flows follow opposite directions. As shown on Figure 2(a), entity B (the destination) queries entity A (the source) for information, and the source answers by sending back the available information. In other words, entity B must ask for information in order to obtain it.

In contrast, with the push model, control and information flows follow the same direction. This is depicted on Figure 2(b). When information is available at entity A (the source), this information is directly sent to entity B (the destination).

4 Scalable Failure Detection

There has been many proposals to address some of the problems of ensuring scalable failure detection. In this section, we give a brief overview of these approaches and discuss their respective strengths and weaknesses. A more detailed study can be found in earlier work [14].

4.1 Hierarchical protocols

When the number of processes becomes very high, it is largely impractical to allow all of them to monitor each other. In an attempt to mitigate this issue, hierarchical protocols arrange processes into some form of hierarchy (e.g., tree, forest) along which traffic is channeled. A three-levels hierarchy is illustrated on Figure 3. Edges represent local heartbeat traffic and the arrows show the flow of information. A failure detector module is responsible for monitoring all processes (or objects) which reside at the same level. The modules form a hierarchy along which the information gets propagated. In general, the hierarchy is assumed to closely match the physical topology of the network for increased benefit, but this is not absolutely necessary.

Using the hierarchical approach, failure detector modules monitor processes directly or indirectly through other failure detectors. This makes it possible to reduce traffic by combining information about several processes in a single message, and by temporarily caching some information at several places in the system.

There has been several propositions of hierarchical failure detection schemes. For instance, Stelling et al. [21] propose a failure detection service for the Globus Grid toolkit, a middleware platform to support Grid applications [11]. Their proposition is based on a simple two-level hierarchy. However, there are several shortcomings with this proposal. The main issue is that the hierarchy is limited to only two levels, which means that it actually fails to take full advantage of the hierarchical approach. Felber et al. [8] propose an architecture for a CORBA failure detection service. The paper presents a generic interface for failure detection whereby failure detectors are addressed as first-class CORBA objects. The structure could be adapted to various approaches, although the paper focuses on a hierarchical architecture. The author argue that the same interface could easily be adapted to a gossip-like architecture (see Sect. 4.2).

4.2 Gossip-style protocols

The technique of gossiping is a popular approach to disseminating information rapidly in large distributed systems [16]. The basic idea is that processes randomly pick partners with whom they exchange their information. Doing so ensures, with high probability, that any all processes eventually obtain any piece of information. Protocols based on this approach are also sometimes called epidemic protocols.

This approach has been successfully applied to failure detection, as a way to efficiently propagate failure suspicions. One of the very strong advantage of gossip-style protocols for failure detection is that they are completely oblivious of the underlying topology, and hence of possible topology changes. In other words, this class of failure detectors is completely resilient to topology changes, without requiring any additional mechanism.

The application of gossiping to failure detection was pioneered by van Renesse et al. [22]. The authors distinguish between two variations of gossip-style failure detectors: *basic gossiping* and *multilevel gossiping*. In the basic gossiping protocol, a failure detector module is resident at each host in the network. It keeps track of the every other modules it knows about and the last time it heard from them. Regularly, failure detector modules randomly picks some other module and send its list to it, regardless of the physical topology. The multilevel gossiping is a variant aimed at large-scale networks. The protocol defines a multilevel hierarchy using the structure of Internet domains and subdomains as defined by comparing their respective IP addresses. Given two hosts, the longer the common prefix of their IP addresses, the closer they are in the hierarchy. In the failure detection protocol, most gossip messages are sent using the basic protocol within a subnet. Then, fewer gossip messages are sent across different subnets, and even fewer across different domains.

The total number of messages can be kept low regardless of the actual network topology. The number of messages at a given domain only depends on the number of subnets in that domain. According to van Renesse et al. [22], this protocol tolerates lossy communication, although the detection time is affected by the probability of message loss. There is however a price to pay for this. First, this protocol does not work well when a large percentage of components crash or become partitioned away. Second, detecting the occurrence of a specific failure can potentially take a fairly long time (it is in fact unbounded).

Gupta et al. [13] also proposed a failure detection based on gossiping, developed for the SWIM group membership protocol [5]. The main difference with the gossip-style failure detector of van Renesse et al. [22] is that SWIM failure detector modules do not communicate randomly. Instead, they inquire failure detector modules located in their neighborhood. The SWIM failure detector can be tuned to ensure a given detection time, in terms of protocol periods, and with an accuracy that depends on several parameters such as network conditions. But, because the join/leave of processes is handled by the membership protocol, it can only handle a fixed set of monitored processes. This discussion is however beyond the scope of this paper.

4.3 Adaptive protocols

Adaptive protocols are designed to adapt dynamically to their environment and, in particular, to adapt their behavior to changing network conditions. Failure detectors can also be made to adapt to changing application behavior.

Adapting to network conditions

There exist several propositions of adaptive failure detection mechanisms (e.g., [2, 4, 9, 20]). The proposed solutions are based on a heartbeat strategy, although nothing seems to preclude the use of other strategies such as interrogation. The principal difference with the heartbeat strategy described in Section 3.2 is that the timeout is modified dynamically according to network conditions.

Fetzer et al. [9] proposed a protocol with a simple adaptation mechanism. The protocol adjusts the timeout by using the maximum arrival interval of heartbeat messages. The protocol assumes a partially synchronous system model [7], wherein an unknown bound on message delays eventually exists. The authors show that their algorithm belongs to the class $\Diamond\mathcal{P}$ in this model.

Chen et al. [4] propose a different approach based on a probabilistic analysis of network traffic. The protocol uses arrival times sampled in the recent past to compute an estimation of the arrival time of the next heartbeat. The timeout is set according to this estimation and a safety margin, and recomputed for each interval. The safety margin is determined by application QoS requirements (e.g., upper bound on detection time) and network characteristics (e.g., network load).

Bertier et al. [2] propose a different estimation function, which combines Chen's estimation with another estimation of arrival times developed by Jacobson [17] for a different context. Bertier's estimation provides a shorter detection time than Chen's, but generates more wrong suspicions. The resulting failure detector is shown to belong to class $\Diamond\mathcal{P}$ when executed in a partially synchronous system model.

Sotoma et al. [20] propose the implementation of an adaptive failure detector with CORBA. Their algorithm computes a timeout, based on the average time intervals of heartbeat messages, plus a ratio between arrival intervals.

Adapting to application requirements

Let us illustrate with a simple example what we describe as the adaptation to application requirements. Consider for instance two applications A_{in} and A_{db}, where A_{in} is an interactive application and A_{db} is a heavyweight database application. Consider also than both applications run simultaneously and rely on the same system-wide failure detection service. With A_{in}, the actual crash of a process must be detected quickly to prevent the system from blocking. In contrast, A_{db} launches a multi-terabytes file transfer whenever a process is suspected, and hence requires accurate suspicions. While A_{in} favors the reactivity of the failure detector, A_{db} requires high accuracy.

Some of the adaptive failure detectors mentioned above [4, 2] can be tailored to match diverse applications requirements. This is done by using QoS requirements to compute the parameters of the failure detector. Then, the failure detectors adapt to changing network conditions in such a way that the QoS requirements are met with high probability.

The drawback with these studies is that the parameters of the failure detectors are determined statically (i.e., at deployment time), and cannot easily be changed dynamically (i.e., at runtime). This a problem for applications with requirements that can change over time. For instance, an application might have very stringent QoS requirements for a certain period of time, and more relaxed one the rest of the time. Unless the cost of enforcing the stringent requirements is negligible, it is desirable to adapt the parameters of the failure detector when requirements are more relaxed.

A second (and more important) drawback of the failure detectors mentioned above is that they are designed with one single application in mind. This means that, even if parameters can be adjusted to match QoS requirements, they can only meet those of one single application at a time. Arguably, QoS requirements could be set as a least common factor of all concurrent applications. However, this is unfortunately not that simple in practice, as doing only results in tradeoffs that are impossible to address.

Défago et al. [6] outlined a different approach, which consists in managing two different timeouts to generate distinct levels of suspicions. An aggressive yet inaccurate timeout ensures a fast response in case of failures, while a conservative but accurate one supports long-term operations.

Adapting to application behavior

In addition to adapting to changing network conditions, failure detectors can also adapt to changing behavioral patterns of the application. For instance, during a certain period, if a process p is monitored by no other process in the system, it would be reasonable for p to temporarily stop sending useless heartbeat messages.

Sergent et al. [19] analyze several failure detector implementations, among which they propose a more specific approach, called "ad hoc heartbeat" failure detector. It requires the application programmer to identify "critical messages" in its protocol. Critical messages indicate implicitly that some sending process p starts monitoring the destination process q until, say, some result is returned back from q to p. In the meantime, and until the result is available, q sends heartbeat to p. When the result is sent back to p (another critical message interpreted by the failure detector), q stops sending heartbeats which have become unnecessary. The mechanism can be compared to the progress bar found in most graphical user interfaces. Sergent et al. [19] illustrate and analyze the ad hoc failure detector with a consensus protocol, but the principle can be easily applied to other protocols. In the case of consensus, they show that their approach can significantly reduce the overhead of the failure detector.

5 Adaptive φ-Failure Detection

As mentioned earlier, adaptation can occur in several different ways. To be truly generic, a failure detection service must be equally adaptive to (1) changing network conditions, (2) applications requirements, and (3) application behavior.

The conflicting requirements mentioned above cannot possibly be reconciled by traditional timeout-based implementations of failure detectors. This is regardless of their ability to adapt to changing network conditions, as shown by adaptive failure detectors described in the literature. Roughly speaking, these failure detectors are based on heartbeat messages and some timeout Δ_{to} determined dynamically. The value of Δ_{to} is computed with the recent history of message arrivals and an estimation function to predict the arrival time of the next heartbeat. With this approach, the failure detector can adapt to changing network conditions, but its accuracy is determined by the estimation function. Adapting to the requirements of several applications would require to manage as many timeout values, which is of course not acceptable. An alternate approach would allow applications to set their own timeouts, with the obvious drawback that failure detection can no longer adapt to changing network conditions.

To address the problem mentioned above, we have recently developed a novel approach to failure detectors, called the φ-failure detector [15]. φ-failure detectors use no timeout and reconcile all three types of adaptation. The key idea is that a φ-failure detector provides information on the degree of confidence that a given process has actually crashed. More specifically, the failure detector associates a value φ_p to every known process p. The value φ_p increases dynamically according to a normalized scale and represents the degree of confidence, at the time of query, that process p has crashed.

The interactions between the applications and the failure detector are hence different than in the traditional case. Indeed, distributed applications use the value φ_p associated

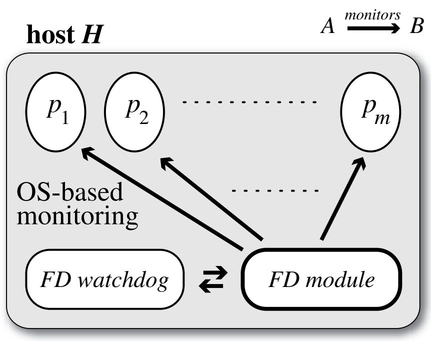

Figure 4. Failure detection within a single host.

with a process p to decide on a course of action. For instance, applications can set some finite threshold for φ_p and decide to suspect p if φ_p crosses that threshold. Different applications can then set different thresholds for the same process. For instance, some applications would set a low threshold to obtain prompt yet inaccurate failure detection (i.e., with many wrong suspicions), while applications with stronger requirements would set a higher threshold and obtain more accurate suspicions. Consequently, this approach can effectively adapt to application requirements because the threshold can be set on an per-application basis (and also on a per-communication channel basis within each application). Beside, the scale ensures that the value set as a threshold is meaningful for the application (it represents the degree of confidence). In practice, we compute the value φ_p based on the history of arrival intervals between heartbeat messages (see [15] for details).

6 Architecture

In this section, we outline the architecture of our failure detection service. More specifically, we look at three different levels of the system. Firstly, we describe the mechanisms involved within the scope of a single host (§6.1). Secondly, we show the interactions of the failure detection modules of two direct neighbor hosts (§6.2). Thirdly, we consider the failure detection service in large-scale distributed systems, and explain how φ-failure detection is used in this context (§6.3).

6.1 Intra-host failure detection

The detection of process failures within the scope of a single machine is illustrated on Figure 4. In the figure, $p_1, p_2, \ldots, p_m$ denote the application processes[1] running on the machine. The failure detection mechanism is then composed of three parts: a daemon process, a watchdog, and a library linked with each process.

- The daemon process, or *failure detector module* (FD module in Fig 4) is the main part. The role of the failure detector module is to monitor local processes and interact with the modules of other machines to exchange informations on failure suspicions. As much as possible, the module relies on system calls to monitor the status of the processes. Given adequate support from the operating system, the local detection of a process crash can be completely certain in most cases. In other words, there are cases when it is in fact possible to distinguish a crashed process from a very slow one. For instance, if a process p is found in zombie state or entirely disappears from the list of processes, there is no possible doubt that p has crashed. This is clearly unlike the situation faced when trying to detect failures over a network.

- The *failure detector library* (not represented on Fig. 4) is used by the monitored/monitoring processes to interact with the failure detector. Among other things, the library linked with process p_i registers and unregisters p_i to the failure detector module, it informs the module of the identity of processes that p_i wants to monitor, and it notifies p_i when some of the processes it monitors must be suspected. In addition, the library can implement a substitute mechanism for systems where the operating system does not provide sufficient reliable information on the status of running processes.

- The *failure detector watchdog* (FD watchdog in Fig 4) is used to cope with a possible crash of the FD module. The watchdog monitors the status of the FD module and restarts it if it is found to have crashed. The watchdog can be implemented either as a daemon or as a periodic system task (i.e., a cron job in POSIX lingo). When the watchdog is implemented as a process, the failure detector module also monitors the watchdog in addition to the application processes. Then, the FD module acts itself as a watchdog for the FD watchdog.[2]

6.2 Inter-hosts failure detection

Failure detector modules of neighboring hosts interact to exchange information on suspicions. Beside, each pair of FD module uses the φ-failure detector to detect the possible

[1]A process is understood with the same meaning as in the POSIX specification.

[2]Given the notorious unreliability of some popular commodity operating systems, the introduction of a second FD watchdog or even a second FD module can be considered. For earlier versions of these operating systems, the simple fact of using them denotes a total lack of concern for reliability, and hence failure detection is probably no longer an issue.

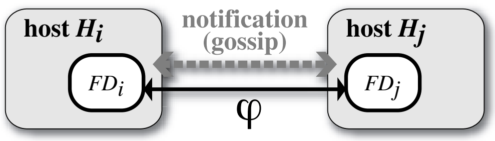

Figure 5. Direct interactions across hosts (between FD modules FD_i and FD_j).

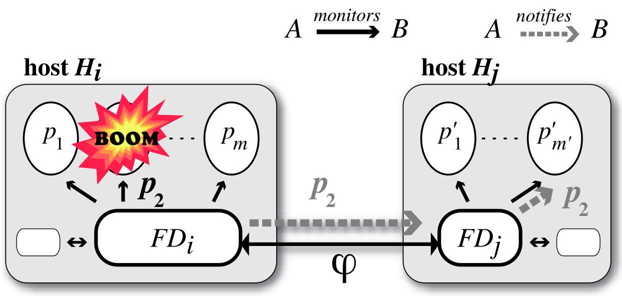

Figure 6. The crash of process p_2 on host H_i is notified to process $p'_{m'}$ on host H_j.

failure of its peer. This is illustrated on Figure 5, where two neighboring hosts H_i and H_j are represented. Their respective failure detector modules FD_i and FD_j interact in two different ways. Firstly, they report the crash of an application process when it occurs. Secondly, the modules monitor each other as a means to detect the crash of hosts.

Of course, if a host crashes, all its processes (including the failure detection module) also necessarily crash at the same time. The opposite is however not true and, in particular, the failure detector module can crash while the hosts remains operational. However, the watchdog ensures that the crash of the module is only transient.

Figure 6 illustrates what happens as the result of the crash of an application process. In the example, process p_2 of host H_i is being monitored by process $p'_{m'}$ of host H_j. In particular, this means that p_2 has previously registered to module FD_i, and that $p'_{m'}$ has declared itself as interested in the status of p_2. When p_2 crashes, the operating system on host H_i notifies FD_i of the crash. In turn, module FD_j informs its neighbors, including module FD_j. Finally, module FD_j knows that process $p'_{m'}$ monitors the status of p_2, and hence notifies it of the crash. In other words, the failure detector service includes a form of publish-subscribe communication mechanism.

6.3 Global failure notification

At a larger level, failure detector modules organize themselves according to a hierarchical structure. The hierarchy is used to convey information about process suspicions in a scalable manner. The hierarchy is determined dynamically, and depends partly on the physical topology of the networking infrastructure (static part), and partly on the use monitoring patterns of processes (dynamic part).

More concretely, this occurs in several phases, as described below.

1. Each failure detection module FD_i maintains a list of other modules as its direct neighbors. The size of the list is determined heuristically to be a little over the binary logarithm of the total number of failure detector modules present in the system. This ensures that the graph formed by these neighboring relations is connected with high probability (e.g., see [12]). Links are established as end processes begin monitoring each others. Links are discarded using an LRU replacement policy.[3] In addition to this, FD_i keeps a direct contact with all other modules located on the same subnet and sends them heartbeat using IP-multicast for low overhead.

2. For each link between two failure detector modules, the modules monitor each other using a φ-failure detector. The links between neighbor modules form a (directed) graph. To each edge, a cost is associated, with the weight defined as the current value of φ for this particular link. The whole graph is not supposedly known by any single module in the system, but results from the aggregation of the partial knowledge of all modules.

3. A minimum cost spanning tree is determined heuristically over the weighted graph mentioned above. This uses a self-stabilizing distributed algorithm developed by Kamei and Kakugawa [18].

4. For any pair of application processes, the φ value between them is determined as the maximal φ value that occurs on the path between the two correspond failure detection modules, along the spanning tree described above.

7 Conclusion

In this paper, we have outlined the design of a failure detection service for large-scale distributed systems. The service makes use of the adaptive φ-failure detector that we developed recently, as well as a hierarchical failure detection scheme. The large-scale part relies on self-stabilizing algorithms, thus ensuring the robustness of the system.

This is still in active development and several points remain to be solved before such a service can actually be implemented. In particular, we intend to experiment with the

[3]LRU = *least recently used*.

hierarchical mechanism described in Section 6.3. We will assess its actual scalability in a real environment, by running extensive performance analyses. We will also compare this approach with a mechanism based on gossiping. We believe that the ideal solution will combine gossiping and the hierarchical approaches.

In our work on the φ-failure detector, we must assume that communication channels are quasi-reliable. This is not absolutely essential in practice, but it is done to facilitate the proofs of correctness. Nevertheless, we are currently working on an extension of the φ-failure detectors for fair-lossy channels,[4] and whose correctness can be established more rigorously.

References

[1] A. Basu, B. Charron-Bost, and S. Toueg. Solving problems in the presence of process crashes and lossy links. Technical Report TR96-1609, Cornell University, USA, Sept. 1996.

[2] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proc. Intl. Conf. on Dependable Systems and Networks (DSN'02)*, pages 354–363, Washington, DC, USA, June 2002.

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

[4] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(2):13–32, 2002.

[5] A. Das, I. Gupta, and A. Motivala. Swim: Scalable weakly-consistent infection-style process group membership protocol. In *Proc. Intl. Conf. on Dependable Systems and Networks (DSN'02)*, pages 303–312, Washington DC, USA, June 2002.

[6] X. Défago, P. Felber, and A. Schiper. Optimization techniques for replicating CORBA objects. In *Proc. 4th IEEE Intl. Workshop on Object-oriented Real-time Dependable Systems (WORDS'99)*, pages 2–8, Santa Barbara, CA, USA, Jan. 1999.

[7] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, Apr. 1988.

[8] P. Felber, X. Défago, R. Guerraoui, and P. Oser. Failure detectors as first class objects. In *Proc. 9th IEEE Intl. Symp. on Distributed Objects and Applications (DOA'99)*, pages 132–141, Edinburgh, Scotland, Sept. 1999.

[9] C. Fetzer, M. Raynal, and F. Tronel. An adaptive failure detection protocol. In *Proc. 8th IEEE Pacific Rim Symp. on Dependable Computing (PRDC-8)*, pages 146–153, Seoul, Korea, Dec. 2001.

[10] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[11] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid. *Intl. Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[12] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Trans. on Computers*, 52(2):139–258, Feb. 2003.

[13] I. Gupta, T. D. Chandra, and G. S. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proc. 20th Annual ACM Symp. on Principles of Distributed Computing (PODC-20)*, pages 170–179, Newport, RI, USA, Aug. 2001. ACM Press.

[14] N. Hayashibara, A. Cherif, and T. Katayama. Failure detectors for large-scale distributed systems. In *Proc. 21st IEEE Symp. on Reliable Distributed Systems (SRDS-21), Intl. Workshop on Self-Repairing and Self-Configurable Distributed Systems (RCDS'2002)*, pages 404–409, Osaka, Japan, Oct. 2002.

[15] N. Hayashibara, X. Défago, and T. Katayama. Two-ways adaptive failure detection with the φ-failure detector. In *Proc. Intl. Workshop on Adaptive Distributed Systems*, Sorrento, Italy, Oct. 2003. In conjunction with 17th Intl. Symp. on Distributed Computing (DISC-17). *to appear*.

[16] S. Hedetniemi, S. Hedetniemi, and A. Liestman. A survey of gossiping and broadcasting in communication networks. *Networks*, 18:319–349, 1988.

[17] V. Jacobson. Congestion avoidance and control. In *Symp. Proc. on Communications Architectures and Protocols (SIGCOMM'88)*, pages 314–329, Stanford, CA, USA, Aug. 1988.

[18] S. Kamei and H. Kakugawa. A self-stabilizing algorithm for the Steiner tree problem. In *Proc. 21st IEEE Symp. on Reliable Distributed Systems (SRDS-21), Intl. Workshop on Self-Repairing and Self-Configurable Distributed Systems (RCDS'2002)*, pages 396–401, Osaka, Japan, Oct. 2002.

[19] N. Sergent, X. Défago, and A. Schiper. Impact of a failure detection mechanism on the performance of consensus. In *Proc. 8th IEEE Pacific Rim Symp. on Dependable Computing (PRDC-8)*, pages 137–145, Seoul, Korea, Dec. 2001.

[20] I. Sotoma and E. R. M. Madeira. Adaptation - algorithms to adaptive fault monitoring and their implementation on corba. In *Proc. 3rd Intl. Symp. on Distributed-Objects and Applications (DOA'01)*, pages 219–228, Rome, Italy, Sept. 2001.

[21] P. Stelling, I. Foster, C. Kesselman, C. Lee, and G. von Laszewski. A fault detection service for wide area distributed computations. In *Proc. 7th IEEE Symp. on High Performance Distributed Computing*, pages 268–278, July 1998.

[22] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In N. Davies, K. Raymond, and J. Seitz, editors, *Middleware'98*, pages 55–70, The Lake District, UK, Sept. 1998.

[4] A fair-lossy communication channel can lose an unbounded number of messages, and only guarantees that, if a message is sent infinitely often, it will eventually be received by its destination. Fair-lossy channels are widely recognized as a good model for communication based on UDP.