2005－DPS－123（5）
2005／6／2

Information Propagation on the φ Failure Detector

NAOHIRO HAYASHIBARA,[†] XAVIER DÉFAGO,[††,†††] MAKOTO TAKIZAWA[†]
and TAKUYA KATAYAMA[††]

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 issue related to propagating information on failures in the φ failure detector for large-scale systems. Traditionally, failure detection systems provide information on suspects to every processes. However, it is not the efficient way in the large-scale system. We consider the notification system that propagates information on suspicions with content-based filtering.

1. Introduction

The ability for a distributed system to detect the failure of its processes is widely recognized as an essential issue for fault-tolerant systems. 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.,[8),12),21),22)]). However, several important points remain to be addressed before a truly generic service can be proposed.

While, to address the problems on adapting network condition and application requirements, the φ failure detector has been proposed by Hayashibara et. al[17]. It allows lots of quality-of-service (QoS) requirements on failure detection from unlimited number of processes without any additional performance cost.

In this paper, we discuss on the way for propagating information of failures in the φ failure detector. In this context, we have to consider which information is really needed. It means that useless information shouldn't be sent by the propagation protocol. We discuss on this topic using publish/subscribe scheme. Information of a certain process's failure is needed by some specific processes. It means that some others do not need this information. Thus, notification of failure information is looks like some form of publish/subscribe system.

The remainder of the paper is constructed as follows. In the section 2, we describe the target system of the paper and some related works. In the section 3, we explain traditional failure detectors and its advanced topics.

2. System Model and Related Works

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[2)]. 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.[14)] 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.[11)], because the fact that the distribution is positively unbounded implies that no bound (known or un-

† Tokyo Denki University
†† Japan Advanced Institute of Science and Technology
††† Japan Science and Technology Agency

known) 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 some of extended solutions for various environments and large number of processes that should be monitored (§3.2). Thirdly, we discuss a second aspect of failure detection: the notification of failures (§4). 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[8] 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.

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 Adaptive Failure Detection

Towards a generic failure detection service, we need to build a scalable and precise failure detection mechanism because failure detector modules are parts of the service. However, lots of problems lay on this goal[16] and no solution does not succeeded completely so far. Now we focus on the ability of adapting to network condition and requirements of processes for failure detection.

Adaptive failure detection mechanisms 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., [3,9,13,20]). 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 is that the timeout is modified dynamically according to network conditions.

Fetzer et al.[13] 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[11], 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.[9] 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.[3] propose a different estimation function, which combines Chen's estimation with another estimation of arrival times developed by Jacobson[18] 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[3),9)] 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.

3.3 The φ Accrual Failure Detector

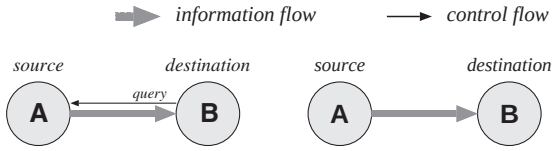
To address the problems mentioned above, we have recently developed a novel approach to failure detectors, called the φ-failure detector[17)]. φ-failure detectors are based on the notion of *Accrual failure detectors*[10)], which use no timeout and reconcile all three types of adaptation. The key idea is that a φ-failure detector provides information on the degree of confidence, called *suspicion level*, 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 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[17)] for details).

4. Propagation of Failure Information

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 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 focus on the notification aspect of failure detectors. As with most of notification services, the information can be conveyed using two basic interaction models, namely the *push model* and the *pull model*. Figure 1 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 interme-



(a) Pull model (b) Push model

Fig. 1 Interaction models for failure notification

diates on the path between A and B.

We consider the separation of failure detection and its notification in large-scale systems. Typically, failure detector modules output information on failures and exchange it by interaction with other modules. For adding scalability, some approaches were proposed such as hierarchical failure detectors[4),12),21)], gossip-based failure detectors[15),22)] and so on. They attempt to distribute information on failures to all failure detector modules. However, no one consider to propagate such information based on its contents. This issue is one of the most important things to build the failure detection service for large-scale systems. In such a system, it has huge amount of nodes and processes. Existing failure detectors and its service will propagate information, which a certain process has been crashed, to all processes. It could make the bandwidth narrow and prevent some messages issued by applications.

Specially, this problem is a serious for the φ-failure detector. It can realize the adaptability for network condition and application requirements by many processes. Each failure detector module outputs suspicion level on a certain process. Suspicion level means how much chance to get a correct suspicion if the failure detector module suspects the process at certain time. It is also represented as a positive real value. It means that this information should be delivered as soon as possible by application processes that are interested in it.

Thus, we now discuss on the content-based propagation of information given by failure detector modules. Propagating information based on requirements were investigated in the context of multicasting and group communication. While it is also studied as *publish/subscribe* interaction model.

In the viewpoint from application processes, normally they don't need information about every process. They are interested only in some specific groups of processes. In this case, the notification service has to provide information matched to their interests. This kind of interaction looks like publish/subscribe model.

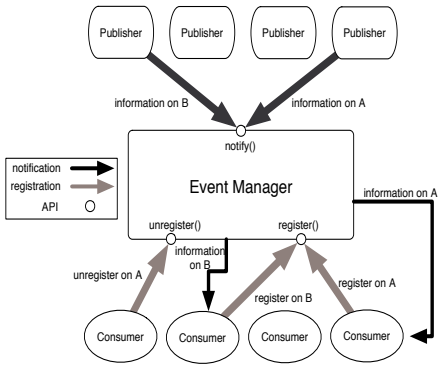


Fig. 2 The publish/subscribe system

4.1 Publish/Subscribe Interaction Model

The publish/subscribe interaction model provides subscribers with the ability to express their interest in events. There exist lots of implementations of publish/subscribe systems[1),5)~7),19)]. Information providers generate information and send it to an event manager (or a broker) and consumers subscribe to the information they want to receive from that manager. This information is called *event* and the act of delivering it is defined as the term *notification*. An event manager has mainly an API `notify()` for publishers and two APIs, `register()` and `unregister()`, for consumers (see Fig. 2). Consumers use `register()` or `unregister()` to have or terminate information according to their interest. To propagate information, publishers can use `notify()` on the event manager.

This model has been applied for mail magazine, net news and so on. It can do that each consumer can get only information that matches its registered interest. In our case, publishers are replaced to failure detector modules and consumers are replaced to application processes.

4.2 Notification Service based on Publish/Subscribe Model

As we said, the propagation of information should be separated from failure detection in large-scale systems. The notification service plays a role that it propagates information on suspicions as events. Failure detector modules ask the notification service to propagate information on a certain process's failure if it has suspected the process. Therefore, the service should lay among failure detectors.

The notification service also has the same APIs as ones in the publish/subscribe system described in the section 4.1. Failure detector modules propagate information on suspicions using the API

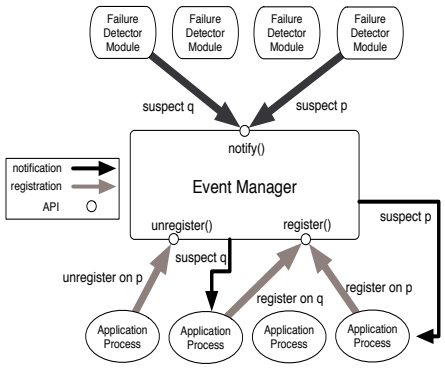


Fig. 3 The notification service with failure detector modules

`notify()` on the notification system. While, application processes register process(es), which they want to know, using `register()`. When application processes want to terminate for delivering events, they can use `unregister()`. The notification system has a facility of filtering events. Thus, it can provide required events to application processes.

Fig. 3 shows the interaction among the notification system, failure detector modules and application processes.

4.3 Propagation in the φ Failure Detector

In the previous section, we described about the notification system. In this section, we explain the propagation of information on monitored processes in the φ failure detector using the notification system. As we said above, this failure detector has a problem on propagating information of suspicions. This is because suspicion level is changed according to time. If the failure detector does not have any interaction with a certain monitored process p, its suspicion level φ_p would increase by elapsing time. Therefore, suspicion levels of processes that are required should be delivered to application processes as soon as possible. Otherwise this information will be outdated.

Each failure detector module in the φ failure detector is located in each node and monitors processes in its local host. The module also monitors other failure detector modules in the same subnet or some range of network. They provide suspicion levels of processes, that they are monitoring, to the notification system periodically using `notify()` (Fig. 3).

Application processes have a certain threshold Φ_p for a process p. It means the requirement for failure detection on p and it is used to decide the suspicion on p compared with φ_p. They can use `register()` to register the process IDs and their thresholds. Then, the notification system delivers event(s) if the suspicion level φ_p in the registered process p exceeded its threshold Φ_p.

Thus, the notification service sends information only to registered application processes with content-based filtering. It means that useless information is not sent. Moreover, the notification system can send the suspicion level, which are desired by some process, it notifies this information to the process immediately.

5. Conclusion

In this paper, we described the relationship between failure detectors and the publish/subscribe system. Specially, we proposed the way of propagating information in the φ failure detector with the publish/subscribe system. This approach can deliver desired suspicion levels to appropriate processes immediately. Moreover, it can reduce the number of information sent by failure detector modules and the notification service. Because the notification service can provide information only to the processes that require it.

In the future direction of our work is to customize the existing publish/subscribe system for building the notification service and define APIs more precisely.

Traditionally, failure detection systems provide information on suspicions to every processes. In this paper, we have shown a new approach to propagate such information efficiently. This topic can be discussed formally and practically as our future work, as well.

References

1) G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proc. of ICDCS'99*, 1999.
2) 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, September 1996.
3) M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proc. of the 15th Int'l Conf. on Dependable Systems and Networks (DSN'02)*, pages 354–363, Washington, D.C., USA, June 2002.
4) M. Bertier, O. Marin, and P. Sens. Performance analysis of a hierarchical failure detector. In *Proc. Intl. Conf. on Dependable Systems and Networks (DNS'03)*, pages 635–644, San Francisco, CA, USA, June 2003.
5) A. Carzaniga, D. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notification

service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

6) Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):100–110, 2002.
7) Raphaël Chand and Pascal Felber. Xnet: A reliable content-based publish/subscribe system. In *Proc. 23nd IEEE Int'l Symp. on Reliable Distributed Systems (SRDS'04)*, pages 264–273, Florianópolis, Brazil, October 2004.
8) T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
9) W.Chen, S.Toueg, and M.K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(5):561–580, May 2002.
10) Xavier Défago, Péter Urbán, Naohiro Hayashibara, and Takuya Katayama. Definition and specification of accrual failure detectors. In *Proc. Int'l Conf. on Dependable Systems and Networks (DSN)*, Yokohama, Japan, June 2005. To appear.
11) C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.
12) P. Felber, X. Défago, R. Guerraoui, and P. Oser. Failure detectors as first class objects. In *Proc. 1st IEEE Intl. Symp. on Distributed Objects and Applications (DOA'99)*, pages 132–141, Edinburgh, Scotland, September 1999.
13) 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, December 2001.
14) M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
15) 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, August 2001. ACM Press.
16) 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, October 2002.
17) Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama. The φ accrual failure detector. In *Proc. 23nd IEEE Int'l Symp. on Reliable Distributed Systems (SRDS'04)*, pages 66–78, Florianópolis, Brazil, October 2004.
18) V. Jacobson. Congestion avoidance and control. In *Proc. of ACM SIGCOMM'88*, Stanford, CA, USA, August 1988.
19) Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.
20) I. Sotoma and E. Roberto M. Madeira. Adaptation - algorithms to adaptive fault monitoring and their implementation on CORBA. In *Proc. of the Third Int'l Symp. on Distributed-Objects and Applications (DOA'01)*, pages 219–228, Rome, Italy, September 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, September 1998.