Accrual Failure Detectors

by

Naohiro HAYASHIBARA

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

Supervisor: Professor Takuya Katayama

School of Information Science
Japan Advanced Institute of Science and Technology

June 30, 2004

Abstract

Failure detection is a fundamental building block for ensuring fault tolerance in distributed systems. Failure detector is a distributed entity and is consist of the set of failure detector modules. Each module outputs the set of suspected processes. Failure detectors are basic components to solve agreement problems (e.g., consensus, atomic broadcast, atomic commitment), especially in asynchronous distributed systems. There are lots of approaches and implementations in failure detectors. However, providing flexible failure detection in off-the-shelf distributed systems is difficult.

Practical solutions to failure detection rely on some adaptive mechanism to cope with the unpredictability of networking conditions. However, they lack the necessary flexibility to provide failure detection as a system-wide service. In particular, traditional solutions take a "one size fits all" approach, which prevents them from simultaneously supporting several distributed applications with very diverse QoS requirements.

In this dissertation, we present a novel approach, called accrual failure detectors, that addresses the flexibility issue. Accrual failure detectors are fundamental approaches for detecting failures in point-to-point communication with a pair of nodes or processes. Conventional failure detectors provide information of a boolean nature (i.e., suspect or not suspect). In contrast, accrual failure detectors provide a value to express the confidence that a given process has crashed. If the process actually crashes, the value accrues until eventually reaching any given threshold set by applications, hence we named our failure detectors "Accrual failure detectors". On the other hand, each application has own threshold respect to its requirement. It suspects some process itself according to the threshold and the value given by the failure detector module.

In the dissertation, we describe the concept and the definition of accrual failure detectors. Then we present mechanisms and implementations of two instances of accrual failure detectors, the φ-failure detector and the κ-failure detector. We also show our measurement of our failure detectors and comparison of these failure detectors and other adaptive failure detectors.

Acknowledgments

First of all, I wish to express my gratitude to my supervisor, Prof. Takuya Katayama for guiding me to the Ph.D. and his support in everything I need for my work for five years.

I am also very grateful to Associate Prof. Xavier Défago for his creative collaboration. Discussions with Xavier are always interesting for me and I have learned a lot of scientific skills from him.

I wish to thank to Associate Prof. Adel Cherif for teaching me fundamentals on distributed systems. He led me to the work mainly when I was a master student.

I am also grateful Prof. André Schiper and Péter Urbán for their collaboration at EPFL. Prof. Schiper also served us resources in his laboratory for our experiments. Péter helped me and gave me insightful comments on the performance comparison of consensus algorithms with Neko. The research work at EPFL was really good experience for me to study abroad and to start the work on failure detectors.

I am also thankful to Prof. Michel Raynal and Rami Yared for their collaboration and comments on accrual failure detectors.

I wish to thank to the members of jury, Dr. Richard D. Schlichting, Prof. Yoichi Shinoda, Prof. Makoto Takizawa and Associate Prof. Adel Cherif for their careful reading and insightful comments that helped me improve the quality of the dissertation.

My thanks go to all colleagues in the Foundations of Software Laboratory at JAIST and the Distributed Systems Laboratory at EPFL. I am especially thank to Associate Prof. Katsuhiko Gondow, Dr. Kei Ito, Dr. Toshiaki Aoki, Dr. Masumi Toyoshima, Mitsutaka Okazaki, Hayato Kawashima and Kenro Yatake for their comments, discussion, support and friendship in all the life in JAIST. I am also thank to Dr. Pawel Wojciechowski, Dr. Mattias Wiesmann, Arnas Kupsys and Sergio Mena for their friendship. Staying in Switzerland was very enjoyable.

Last but not least, I'm very much thankful to my family and friends whom I met in Ishikawa. Specially, my parents have been taking care of my health and always encouraging me. It is one of the great contributions for the dissertation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and Motivation

Over recent years, distributed systems have gradually evolved to take a prominent position in our society, playing an essential role in many activities within such areas as commerce, communication, science, or even entertainment. For instance, Grid systems, web services, or e-Business are nothing but specific instances of distributed systems on a very large-scale, with many participants and long distances.

Fault-tolerance and reliability are particularly important to distributed systems in general, especially in large-scale settings. Normally, users of these systems expect them to remain operational in spite of technical mishaps, and so the systems must be able to continue working even if some of its participants have crashed. The crash of a host can be caused by a variety of reasons, such as hardware or software failures, human mishandling, natural catastrophes, or simply routine maintenance operations. With a large number of participants and long running times, the probability that at least one of the hosts crashing during the execution is incredibly high, regardless of the physical reliability of each individual host. Consequently, a global system must be designed and engineered in such a way that it can tolerate seamlessly the occurrence of a reasonable number of host failures.

Failure detection and process monitoring are the cornerstone of most techniques for tolerating or masking failures in distributed systems. Implementing failure detectors over local networks is, by now, a rather well-known issue, but it is still far from being a solved problem with large-scale systems. Indeed, there are many reasons why traditional mechanisms, developed for local networks, fail in large-scale systems. For instance, traditional solutions fail to address important factors such as the potentially very large number of monitored processes, the higher probability of message loss, the ever-changing topology of the system, and the high unpredictability of message delays. To rationalize communication in large-scale distributed systems, it is highly desirable for failure detectors to be factored in as a common generic service (e.g., [vRMH98, FDGO99, SFK+98]) rather than as redundant ad hoc implementations (e.g., [SDS01]).

In distributed systems, failure detectors are traditionally based on a simple interaction model wherein processes can only either trust or suspect the processes that they are monitoring. Adaptive failure detectors, proposed in several papers [CTA02, BMS02, SM01, CRV95, SDS01, FRT01], can adjust an appropriate timeout to network conditions and application requirements. One of the major obstacles to building such a service is that applications with completely different requirements and running simultaneously must be effectively tuned by the service to meet their needs. Moreover, many distributed applications can benefit greatly from setting different levels of failure detection to trigger different reactions (e.g., [CBDS02, DSS98, USS03]). For instance, an application can take precautionary measures when confidence in a suspicion reaches a given level, and then take more drastic action once the confidence rises above a second (much higher) level.

In contrast, what is advocated here is an approach whereby a generic failure detection service outputs a value on a continuous normalized scale. Roughly speaking, this value captures the degree of confidence in the judgment that the corresponding process has crashed. It is then left to each application process to set a suspicion threshold according to its own quality-of-service requirements. In addition, even within the scope of a single distributed application, it is often desirable to trigger different reactions to increasing degrees of suspicion. The main advantage of this approach is that it decouples the failure detection service from running applications. Its very design allows it to scale well with respect to the number of simultaneously running applications and/or triggered actions within each application.

Large-scale distributed systems (e.g., grid), often have many users and running applications. Each has diverse requirements such as a quick reaction by the failure detector module even at the expense of low accuracy, or the reverse.

This can be illustrated with a simple example what can be described as an adaptation to application requirements. Consider for instance two applications App_{in} and App_{db}, where App_{in} is an interactive application and App_{db} is a heavyweight database application. Consider also the situation where both applications are running simultaneously and relying on the same system-wide failure detection service. With App_{in}, the actual crash of a process must be detected quickly to prevent the system from blocking. In contrast, App_{db} launches a multi-terabyte file transfer whenever a process is suspected, and hence requires accurate suspicions. While App_{in} favors the reactivity of the failure detector, App_{db} requires high accuracy.

A failure detection service has to address these requirements flexibly. However no one has so far been able to address the problem completely, even with existing adaptive failure detectors.

The long-term goal is to define and implement a generic failure detection service for large-scale distributed systems, and to provide it as a generic network-service (e.g., Domain Name Service (DNS), Network Information System (NIS), Network File System (NFS), Sendmail, etc.). The service can be considered as consisting of two parts, *failure detection* and *information propagation*. The former monitors processes, nodes etc., and detects their failures. This part corresponds closely to traditional failure detectors. The latter propagates information about failures of processes that spread over the system and need such information. In this dissertation, the focus is mainly on improving the failure detection part.

The idea of providing failure detection as an independent service is not particularly new (e.g., [CT96, FDGO99, SFK$^+$98, vRMH98]). Nevertheless, several important points remain to be addressed before a truly generic service can be effectively realized. In particular, a failure detection service must adapt to changing network conditions, as well as to application requirements. Several solutions proposed recently address the first issue specifically [BMS02, CTA02, FRT01, SM01]. However, to the best of the present author 's knowledge, none of the solutions proposed so far, effectively address the second problem, namely, the adaptation to diverse application requirements. A few propositions (e.g., [CTA02]) have been made to adapt the parameters of a failure detector service to match the requirements, but unfortunately they are designed to support a *single* class of requirements. Hayashibara et al. [HCK02] have pointed this out recently in a short survey. So far, it seems that only Cosquer et al. [CRV95] have identified the problem. Their proposition is interesting, but remains somewhat inflexible as they do not question the Boolean nature of failure detection.

As mentioned above, lots of problems remain in implementing a generic failure detection service. Where should we start from? In this dissertation the question is answered and the steps towards realizing the goal are shown.

1.2 Contribution of the Dissertation

In this dissertation, a novel concept of failure detectors, is shown as a way to address the problems mentioned above. It is completely different from other approaches. Rather than a failure detector for each process, the concept should be seen as a way of extending an existing failure detection scheme in order to address the diverse requirements of application processes and network conditions.

Accrual Failure Detectors The main contribution of the dissertation is the proposition and definition of a new concept of failure detectors, called *accrual failure detectors*. An accrual failure detector is an abstract entity which defines an interaction model and its properties. In fact, it outputs an accrual value, which monotonically increases with elapsed time if the corresponding process has crashed. Therefore, the value is eventually initialized if the process is alive. Applications query the failure detector module to get the accrual value of the corresponding process. Each application has its own threshold, which reflects its requirement, and which it uses to interpret the accrual value using its own threshold.

The advantage of the failure detector is that it can adapt in two ways according to the network condition and application requirements. The failure detector however, uses no timeout and reconciles adaptations with the network condition and application requirements.

The φ failure detector is an instance of accrual failure detectors. More specifically, the failure detector associates a value φ_p with every known process p. The value φ_p increases according to a normalized scale φ_p is initialized if the failure detector module receives a heartbeat from p. Each φ failure detector module has a window buffer to capture received heartbeats for a certain period and to compute φ_p. The failure detector adapts to application requirements because each application can trigger suspicions according to its own threshold. Meanwhile, the failure detector can adapt to changing network conditions using the window buffer, because the scale is defined accordingly.

The κ failuredetector is an extension of the φ failure detector. It outputs a value which is calculated as the sum of contributions from expected heartbeats. Hence it is not perplexed by message losses. The main point of the κ failure detector is that it can implement a very conservative failure detector. Actions triggered by high thresholds will be less sensitive to long bursts of message losses and/or temporary network partitions. In this dissertation, the κ failure detector is described and some important properties proven.

These instances of accrual failure detectors are basic building blocks for implementing a failure detection service.

Performance Evaluation of Accrual Failure Detectors. Pragmatic implementations of the φ failure detector and the κ failure detector have been implemented. Their behavior over a transcontinental network connection, between Japan and Europe over the period of three weeks has been evaluated.

Briefly speaking, the proposed implementation works as follows. The protocol samples the arrival time of heartbeats and maintains a sliding window of the most recent samples. This window is used to estimate the arrival time of the next heartbeat, similar to other adaptive failure detectors [BMS02, CTA02]. In addition, the distribution of past samples is used to approximate the probabilistic distribution of future heartbeat messages. With this information, it is possible to compute the value of degree of confidence with a scale that changes dynamically to match recent network conditions.

This failure detection scheme was evaluated under normal transcontinental conditions (between Japan and Switzerland). Heartbeat messages were sent at a rate of one every 30 s using the user datagram protocol (UDP), and the experiment ran uninterruptedly for a period of three weeks, gathering a total of more than $60,000$ samples. Using these samples, the behavior of the failure detector was analyzed and compared with traditional adaptive failure detectors [BMS02,CTA02]. By providing exactly the same input to every failure detector, the fairness of the comparisons could be assured. The results show that the failure detector implementation performed well when compared with traditional implementations, with the additional advantage that its design provides virtually limitless flexibility. In particular, the experiments showed that the κ failure detector can be tuned in the conservative range to avoid wrong suspicions.

Taxonomy and Survey of Failure Detectors. An identification of the basic problems, a survey of existing failure detection implementations, and a qualitative comparison of the different approaches with respect to the problems are provided. More specifically, the following six fundamental problems that must be addressed efficiently an ideal implementation are: *message explosion*, *scalability*, *message loss*, *flexibility*, *dynamism*, and *security*.

A careful study of the literature on failure detection yields several protocols that address parts of the identified problems. These implementations can be grouped into three distinct categories, namely, *hierarchical protocols*, *gossip-style protocols*, and *adaptive protocols*. Among all the mechanisms surveyed, it can be shown none of them addresses all six problems properly, not even from a strictly qualitative standpoint. On top of that, it will be shown that none of the currently known implementations of failure detectors addresses the the identified problem of flexibility.

1.3 Roadmap for the Dissertation

In Chapter 2, system models are presented, together with the definitions and notations used in this dissertation. In Chapter 3, existing approaches to failure detectors are surveyed and classified in terms of large-scale distributed systems. In Chapter 4 definitions and an overview of accrual failure detectors are given. In Chapter 5, the φ failure detector is developed as an adaptive failure detector for applications running on a large-scale distributed systems. In Chapter 6, the κ failure detector for implementing flexibility in failure detection is shown. Finally, the conclusions to the dissertation are given in Chapter 7.

Chapter 2

System Models and Definitions

Various assumptions about the system ’s behavior appear in this dissertation. Defining them is important for discussing some of the protocols or algorithms on which they run. In this chapter, we describe variations of system model and their definitions.

2.1 System Models

Consider a distributed system as a set of processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ which communicate only by sending and receiving messages. Assume that every pair of processes is connected by communication channels. Processes correspond to the processors and channels, and are used in the construction of a distributed system.

In this dissertation, several types of models which have different assumptions are used. Variations of these models and their assumptions are presented here and individual assumptions appear in each chapter.

2.1.1 Synchronism

A number of system models that restrict the behavior of the system have been developed. Specifically, synchronism is an important factor required for solving problems in distributed systems. Models need to deal with synchronism including timing assumptions. Mainly, these models are defined by (i) the relative speeds of the processors and (ii) message transmission delays.

Synchronous System With a synchronous system, the relative speeds of the processors and message transmission delays are assumed to be bounded.

Partially Synchronous System Several types of model ranging from synchronous systems to asynchronous systems have been developed: (i) processors are completely synchronous and communication is partially synchronous, (ii) both processes and communication are partially synchronous, (iii) processes are partially synchronous and communication is synchronous [DLS88]. Let us first consider the case in which the processes are completely synchronous and communication is partially synchronous. In this situation, the system has an upper bound Δ^U on message transmission delay but it is unknown, or it holds a known upper bound Δ^U after a *global stabilization time* (GST), unknown to the processors.

Then, an extension of this model in which both processors and communication, are partially synchronous can be considered. That is, the upper bound on the relative processor speeds Φ^U can exist but be unknown, or Φ^U can be known but actually hold only from some time GST onward.

It is easy to define models where processors are partially synchronous and communication is synchronous (Δ^U exists and is known a priori).

Asynchronous System There is no assumption about the relative processor speeds and message transmission delay.

2.1.2 Failure Models

In a distributed system, the components of the system, both processes and channels, can fail. Defining types of possible failure is important to the discussion about problems described in this dissertation. Now, some failure models are introduced and the types of failure assumed in this dissertation discussed.

Process Failures

Processes can fail for various reasons and they behave differently after failure. Process failures are classified three ways with respect to the behavior of a process after failure.

(FAIL-STOP FAILURE) A faulty process stops permanently and does nothing from that point on but behaves correctly before stopping. All other processes, which do not crash, eventually detect the state with no erroneous detection. This failure is called *fail-stop*.

(CRASH FAILURE) A faulty process stops permanently and does nothing from that point on but behaves correctly before stopping. Some other processes, which do not crash, may not detect the state. This failure is called *crash*.

(OMISSION FAILURE) A faulty process intermittently omits to send or receive messages.

(BYZANTINE FAILURE) A faulty process can exhibit any behavior whatsoever, such as changing state arbitrarily.

All the algorithms and the systems in this dissertation assume only the crash-failure model in processes. Thus, we call processes that never crash *correct* and processes that have crashed *faulty* or *incorrect*. Note that correct/faulty are predicates over a whole execution: a process that crashes is faulty even before the crash occurs. Of course, a process cannot determine if it is faulty and some other components (i.e., failure detector modules) cannot make processes faulty.

Channel Failures

Channels are also unreliable in real systems. Consider types of failures in channels and their definitions.

(CRASH FAILURE) A faulty channel stops transporting any message but behaves correctly before stopping.

2.1.3 Channel Properties

There exist several definitions for communication channels that provide different guarantees.

Quasi-reliable channels. Quasi-reliable channels [ACT99] are defined by P1, P2 and:

P1: (*No Creation*) If a process q receives a message m from process p, then p sends m to q.

P2: (*No Duplication*) q receives no more than one message m from p.

P3: (*No Loss*) If p and q are correct and p sends m to q, q eventually receiving m.

Quasi-reliable channels can be implemented over unreliable channels, using error detecting or correcting codes, sequence numbers and retransmission in cases of message loss. The TCP protocol is a good approximation to quasi-reliable channels. Thus, quasi-reliable channels ensure that messages are not lost in transit. An alternative specification in reliable channels is *reliable channels* which requires that even messages sent by faulty processes are eventually received.

Fair-lossy channels. Fair-lossy channels [BCBT96] can lose messages in transit. The channel from p to q is *fair lossy* if it satisfies P1, P2 and:

P4: (*Fair Loss*) If p sends an infinite number of messages to q, and q often executes receive actions infinitely, then q receives an infinite number of messages from p.

A fair-lossy channel can lose an unbounded number of messages, and only guarantees that, if a message is sent infinitely often, it will eventually be received at its destination. Fair-lossy channels are widely recognized as a good model for communication based on UDP. An alternative specification in lossy channels is *eventually reliable channels* which allows a finite number of messages to be lost.

2.2 Concept of Failure Detectors

A *Failure detector* can be viewed as a distributed oracle for giving a hint on the state of a process. In fact, a failure detector consists of *failure detector modules* that communicate with each other by exchanging messages. A process, called a *monitoring process*, can query its failure detector module about the status of some process, called a *monitored process*. The monitoring process thus obtains information about whether or not the monitored process is suspected to have crashed.

This section briefly presents the principal definitions relevant to failure detectors. More specifically, the formal definition of failure detectors, the main interaction models, and the traditional implementations are rapidly overviewed.

2.2.1 Unreliable Failure Detectors

Being able to detect the crash of other processes is a fundamental requirement in distributed systems. In particular, several distributed agreement problems (e.g., Consensus) cannot be solved deterministically in asynchronous systems if even a single process might crash [FLP85]. The impossibility is based on the fact that, in such a system, a crashed process cannot be distinguished from a very slow one.

Chandra and Toueg [CT96] define the notion of unreliable failure detectors, based on the following model. For every process p_i in the system, there is a module FD_i attached that provides p_i with potentially unreliable information on the status of other processes. At any time, p_i can query FD_i and obtain a set of processes containing those that are suspected of having crashed.

Table 2.1: classes of failure detectors defined by accuracy and completeness properties

	Accuracy			
Completeness	Strong	Weak	Eventual Strong	Eventual Weak
Strong	$\mathcal{P}$	$\mathcal{S}$	$\diamond\mathcal{P}$	$\diamond\mathcal{S}$
Weak	$\mathcal{L}$	$\mathcal{W}$	$\diamond\mathcal{L}$	$\diamond\mathcal{W}$

The impossibility result mentioned above, no longer holds if the system is augmented with some unreliable failure detector oracle [CT96]. An unreliable failure detector is one that can, to a certain degree, make mistakes.

The failure detector is a distributed entity that consists of all modules and whose behavior must exhibit some well-defined properties. Depending on the properties that are satisfied, a failure detector can belong to one of several classes. Now, these properties are as follows:

[*Completeness Properties*]

(Strong Completeness) Eventually every faulty process is permanently suspected by all correct processes.

(Weak Completeness) Eventually every faulty process is permanently suspected by some correct process.

[*Accuracy Properties*]

(Strong Accuracy) No process is suspected before it crashes.

(Weak Accuracy) Some correct process is never suspected.

(Eventual Strong Accuracy) There is a time after which every correct process is never suspected by any correct process.

(Eventual Weak Accuracy) There is a time after which some correct process is never suspected by any correct process.

Perfect failure detector $\mathcal{P}$ satisfies strong completeness and strong accuracy. There are eight such pairs, obtained by selecting one of the two completeness properties and one of the four accuracy properties. The definitions and corresponding notations are given in Fig. 2.1.

As an example, the class $\diamond\mathcal{S}$ of failure detectors, is one of the weakest Failure detectors to solve Consensus, are defined by Strong Completeness and Eventual Weak Accuracy. Interestingly, any given failure detector that satisfies weak completeness can be transformed into a failure detector that satisfies strong completeness. There also exist transformation algorithms for failure detectors from strong completeness to weak completeness. This means that a failure detector with strong completeness and a failure detector with weak completeness are equivalent, thus, $\diamond\mathcal{W}$ is also the weakest failure detector for solving Consensus.

The problem is, that in asynchronous distributed systems,[1] it is impossible to implement a failure detector of class $\diamond\mathcal{S}$ in a literal sense. The definition of $\diamond\mathcal{S}$ failure detector is nevertheless highly relevant in practice. Algorithms which assume the properties of a $\diamond\mathcal{S}$ are incredibly robust because they can tolerate an unbounded number of timing failures. In other words, and in a more pragmatic way, an application is guaranteed to make progress as long as the failure detector "behaves well" for "long enough" periods. Conversely, the application might stagnate during "bad periods" and resume only after the next period of stability.

Fetzer [Fet01] explains how the ability to force the crash of processes using watchdogs can be used to transform a failure detector of class $\diamond\mathcal{S}$ into a failure detector that never makes mistakes (i.e., class $\mathcal{P}$). In a slightly different way, this transformation is also provided by group membership services (see [CKV01] for a survey). In either case, the idea is, that whenever the $\diamond\mathcal{S}$ failure detector suspects some process p_i, the system forces the crash of p_i (or ejects p_i from the group) and announces the suspicion. This ensures that the failure detector is never wrong when announcing suspicions, albeit sometimes *a posteriori*.

In practice, the behavior of a failure detector largely depends on how well the failure detector is tuned to the behavior of the underlying network.

2.3 Quality-of-Service of Failure Detectors

Chen et al. [CTA02] proposed a set of metrics to evaluate the Quality-of-service (QoS) of failure detectors. Given the assumption that there are two processes p and q where p monitors q, the metrics that are used in this dissertation are defined below. Notice that the first definition relates to completeness, whereas the other metrics relate to the accuracy of the failure detector.

Definition 1 (Detection time T_D). *The detection time is the time that elapses from the crash of q until p begins to suspect q permanently.*

Definition 2 (Mistake recurrence time T_{MR}). *The mistake recurrence time measures the time between two consecutive wrong suspicions. T_{MR} is a random variable representing the time that elapses from the beginning of a wrong suspicion to the next one. T_{MR}^U and T_{MR}^L also denote upper and a lower bounds respectively on the mistake recurrence time.*

[1] An asynchronous distributed system is a system wherein no assumptions can be made about message delays and process speeds.

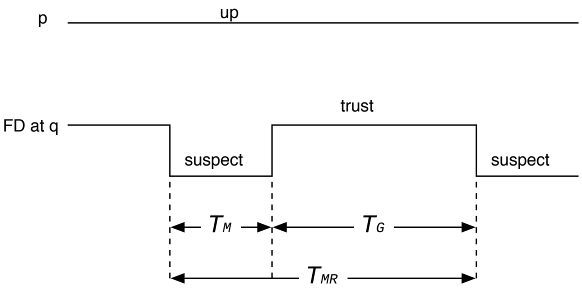

Figure 2.1: Mistake duration T_M, good period duration T_G, and mistake recurrence time T_{MR}

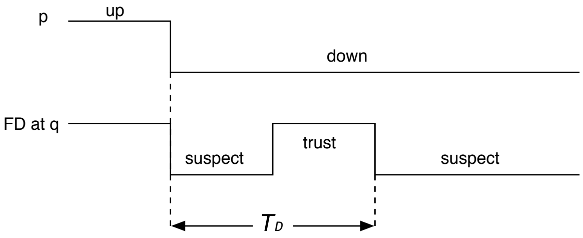

Figure 2.2: Detection time T_D

Definition 3 (Mistake duration T_M). *The mistake duration measures the time that elapses from the beginning of a wrong suspicion until its end (i.e., until the mistake is corrected). This is represented by the random variable T_M, the upper and lower bounds of which are denoted by T_M^U and T_M^L respectively.*

Definition 4 (Good period duration T_G). *This is a random variable T_G representing the time at which p stops trusting q and q. It can be expressed as $T_G = T_{MR} - T_M$.*

Definition 5 (Average mistake rate λ_M). *This measures the rate at which a failure detector generates wrong suspicions. It can be expressed by $\lambda_M = \frac{1}{E(T_{MR})}$.*

Definition 6 (Freshness point τ_i). *The failure detector module computes a freshness point τ_i for the i-th heartbeat message. If the module does not receive a message from a process p until τ_i, it suspects p, otherwise it trusts p (see Fig. 2.3).*

2.4 Agreement Problems

Agreement problems form a fundamental class of problems in distributed systems. They all follow a common pattern: all participating processes must agree on some common decision, the nature of which depends on the specific problem: e.g., the decision might be the delivery order of messages or the outcome (commit or abort) of a distributed transaction.

Agreement problems are numerous in the context of *group communication*. Group communication is a means of providing point-to-multipoint and multipoint-to-multipoint communication, by organizing processes in groups and defining communication primitives that work with groups. These primitives have richer semantics than the usual point-to-point primitives, in terms of flexibility, ordering guarantees, and guarantees of fault tolerance. Ultimately, they

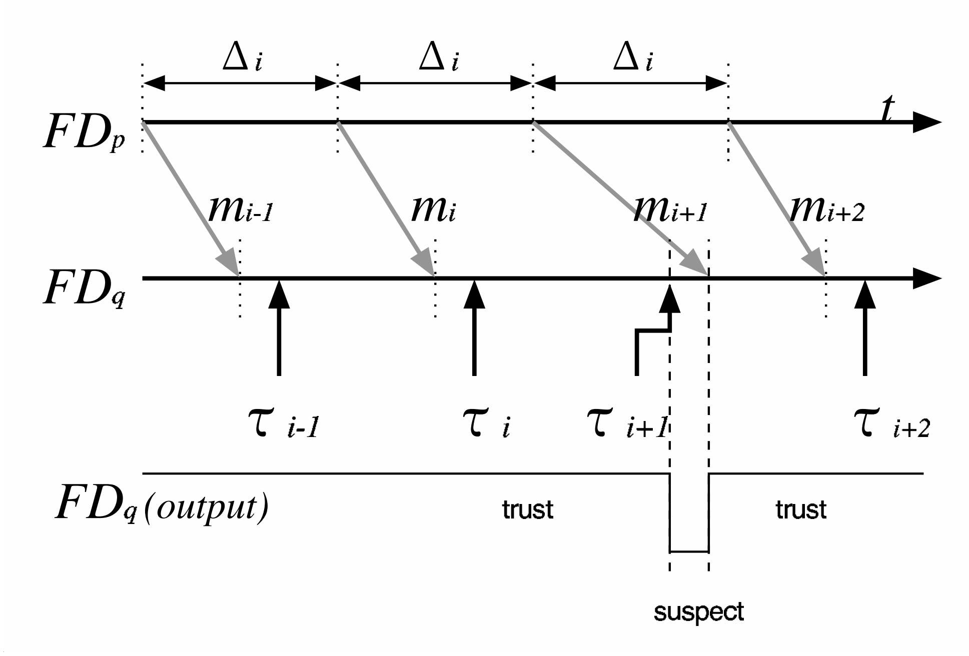

Figure 2.3: Freshness point τ_i

ease the construction of certain types of distributed applications, e.g., fault-tolerant distributed applications.

In this section, informal and formal definitions for four agreement problems are given, Consensus, Reliable Broadcast, Atomic Broadcast, as well as group membership and view synchronous communication. Solutions to these problems often use failure detectors. In order to simplify the explanation, the assumption is made that there is only one group that includes all processes in the system (except when defining group membership where the group changes over time).

2.4.1 Consensus

Consensus is a fundamental problem within agreement problems: a lot of agreement problems can be reduced to consensus [GS01], i.e., algorithms that solve these problems and can be built using a consensus algorithm. Formally, Consensus was defined in terms of two primitives, *propose(v)* and *decide(v)*, where v is some value. Roughly speaking, each participant of a consensus proposes some value, and each of them receives the same decision value, which is the value proposed by someone.

The consensus problem is defined over a set of processes. Each process executes two primitives: Propose(v_i) by which a process proposes its initial value, and Decide(v) by which a process decides a value. The decision must satisfy the following conditions:

(TERMINATION) Every correct process eventually decides.

(VALIDITY) If a process decides v, v is the initial value of some process.

(AGREEMENT) Two correct processes cannot decide differently.

Fischer *et al.* [FLP85] proved that there is no deterministic algorithm for solving Consensus in asynchronous systems where at least one process can crash. They assume asynchronous systems without any timing assumption and any oracles (e.g., failure detectors, process-controlled crash and randomization), so that we can not distinguish faulty processes from very slow processes. On the other hand, algorithms at least as strong as the failure detector of class $\diamond\mathcal{S}$ or $\diamond\mathcal{W}$, can be used to solve Consensus in asynchronous systems [CT96].

Note that Consensus is possible in asynchronous systems with randomization [BO83]. It is also possible in asynchronous systems with process-controlled crash. This is because (1) a perfect failure detector $\mathcal{P}$ can be emulated in such a system [Fet01], (2) $\mathcal{P}$ failure detector is strictly stronger than both $\mathcal{S}$ and $\diamond\mathcal{S}$ failure detectors [CT96], and (3) Consensus can be solved with either $\mathcal{S}$ or $\diamond\mathcal{S}$ failure detectors [CT96].

2.4.2 Reliable Broadcast

Failure detectors can be used by reliable broadcast algorithms. Reliable Broadcast ensures that all processes deliver a message broadcast previously, even if process crashes occur. Note that this is only difficult to ensure when the sender of the message crashes while sending the message: in this case, it is possible that only a subset of all processes receives the message. More formally, Reliable Broadcast is defined by two primitives R-broadcast(m) and R-deliver(m) where m is some message.

Reliable Broadcast can be solved even in an asynchronous system model.

2.4.3 Total Order Broadcast

Total Order Broadcast, also called Atomic Broadcast (ABCAST), can be extended from Reliable Broadcast: in that besides ensuring that all processes receive the messages, it also ensures that processes see the messages *in the same order*. More formally, Total Order Broadcast is defined by two primitives TO-broadcast(m) and TO-deliver(m) where m is some message.

Consensus with $\diamond\mathcal{W}$ failure detector can be used to solve Total Order Broadcast [CT96]. On the contrary, Total Order Broadcast primitives can be used to solve Consensus [DDS87] because all participants obtain the same sequence of messages and events. Total Order Broadcast is hence equivalent to problems like Consensus. Moreover, if there is an algorithm that can solve Consensus, then it can be transformed to solve Total Order Broadcast.

2.4.4 Group Membership

The task of a group membership service is to maintain a list of currently active processes. This list can change with new members joining and old members leaving or crashing. Each process has a *view* of this list, and when this list changes, the membership service reports the change to the process by installing a new view. The membership service strives to install the same view of all correct processes.

A view V consists of a unique identifier V.id and a list of processes V members. For simplicity, we take view identifiers from the set of natural members. Processes are notified of view changes by the primitives view_change(V): we say that a process p *installs* view V. All actions at p after installing V, up to and including the installation of another view V', are said to occur in V.

Group membership can be solved with $\mathcal{P}$ failure detectors [CKV01] if a majority of processes are correct.

Group membership is a popular approach to ensuring fault-tolerance in distributed applications. In short, a group membership keeps track of what process belongs to the distributed computation and what process does not. In particular, a group membership usually needs to exclude processes that have crashed or have been partitioned away. For more information on the subject, refer to the excellent survey by Chockler et al. [CKV01]. A group membership can also be seen as a high-level failure detection mechanism that provides consistent information about suspicions and failures [USS03].

2.4.5 Leader Election

The leader-election problem describes the situation where at any time, at most one process considers itself the leader, and at any time, if there is no leader, a leader is eventually elected. Algorithms to solve leader election requires a failure detector of class $\mathcal{P}$ [SM95]. Strictly speaking, Leader election is harder than Consensus.

Chapter 3

Taxonomy and Survey of Failure Detectors

The main contribution of this chapter is to provide an identification of the basic problems, a survey of existing failure detection implementations, and a qualitative comparison of the different approaches with respect to the problems. More specifically, the following six fundamental problems that an ideal implementation must address efficiently are identified: *message saving*, *scalability*, *message loss*, *flexibility*, *dynamism*, and *security*. A careful study of the literature on failure detection yields several protocols that address parts of the identified problems. These implementations can be grouped into three distinct categories, namely, *hierarchical protocols*, *gossip-style protocols*, and *adaptive protocols*. Among all the mechanisms surveyed, it will be shown that none of them properly addresses all six problems, not even from a strictly qualitative standpoint. On top of that, it will also be shown that one of the identified problems (i.e., flexibility) is not addressed by any of the currently known implementations of failure detectors.

It turns out that the failure detection protocols surveyed usually address different problems, and hence it is useful to obtain an accurate view of their respective strengths and weaknesses. As a secondary contribution, the dissertation outlines an evaluation framework, consisting of a set of measures and benchmarks, to quantify the ability of a given failure detector implementation to address each specific problem. Given specific system settings, these metrics can be used to compare the effectiveness of the various failure detector implementations with respect to each problem. In this dissertation, no actual quantitative results are provided. Indeed, it turns out that the actual measures depend greatly on a large number of parameters, most of which are actually independent of the failure detector implementation. Any quantitative comparison necessarily introduces an arbitrary bias, possibly leading to meaningless results and erroneous conclusions.

3.1 Problems in Large-Scale Distributed Systems

The traditional failure detection protocols of Sect. 3.2.1 completely fail to address the needs of large-scale distributed settings because they simplify choices made for local area networks (LAN) and limited-scale systems. In particular, traditional implementations are not designed to cope with a very large number of processes, a high degree of message loss, a dynamic network topology, and the unpredictability of wide area networks (WANs). In this section, six basic problems that failure detection protocols for very large-scale distributed systems, such as Grids, must necessarily address, are identified.

Problem 1 (message saving). The failure detector generates unnecessary messages.

The failure detector must be thrifty in terms of generating control messages. For instance, a process does not need to send heartbeat messages when it is not monitored. Also, if the number of processes is very large, it might be impossible for a single process to keep track of more than part of the whole system.

Problem 2 (scalability). The message complexity of the failure detector is too big.

Large-scale distributed systems often have a large number of resources distributed over a wide area network. A failure detection service must be able to monitor such large numbers of resources efficiently. To do so, a failure detector module must diffuse information about suspected processes to all other failure detector modules.

Problem 3 (message loss). Message loss generates many wrong suspicions.

The failure detector must be adaptive to network conditions. In WANs, message losses, sometimes leading to network partitions, occur significantly more frequently than in LANs.

Problem 4 (flexibility). Different applications have different monitoring patterns.

In large-scale systems, especially in Grid, applications must coexist in spite of having different goals, requirements, and policies. As these applications should ideally share a common failure detection service, this service must be tunable or adaptable to the applications.

Problem 5 (dynamism). The failure detector behaves properly only when the system has been stable over a long period.

The composition and topology of large-scale distributed systems are highly dynamic, with processes joining and leaving all the time, usage patterns varying, etc. Failure detectors must be aware of reconfigurations and adapt accordingly.

Problem 6 (security). Failure detectors can be compromised and leveraged against applications.

failure detector modules that reside on compromised hosts could be reprogrammed by a malicious intruder to prevent legitimate applications from making progress. In this case, the compromised failure detectors could collude and act like an optimal adversary that generates false rumors about the status of processes. This problem is raised but not addressed further in this dissertation.

3.2 Taxonomy and Survey

In the literature on failure detectors, several attempts at addressing some of the problems mentioned in the previous section have been reported. Here, a simple classification for those techniques is surveyed and presented. Their respective strengths and weaknesses with respect to the problems identified in Sect. 3.1 are also discussed.

3.2.1 Traditional Implementations

Traditionally, there exist two basic failure detection protocols in local networks, both of which are based on the use of timeouts. These two protocols are known as *heartbeat* and *ping-style* failure detector respectively. The former follows the push model described above, whereas the latter follows the pull model.

Heartbeat Strategy

The *heartbeat* strategy for 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. 3.1). 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_q 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. Besides, 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 settings can turn out to be very short in a different environment. Besides, in practical systems, network conditions can vary greatly over time (e.g., depending on external load).

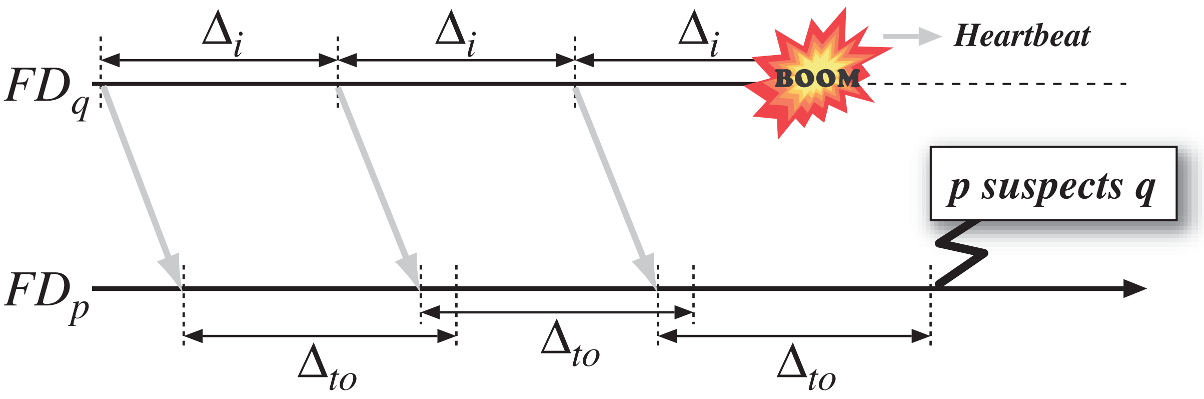

Figure 3.1: Heartbeat messages

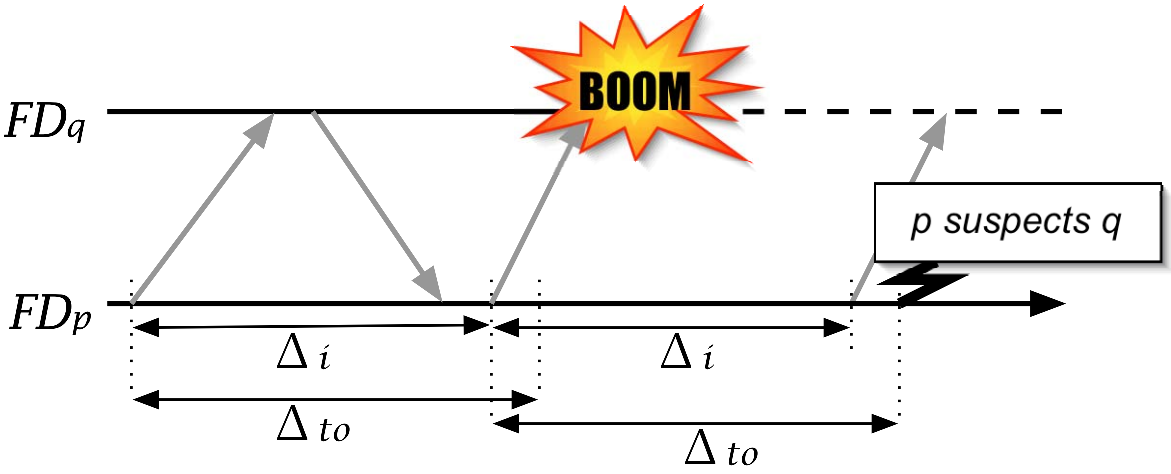

Figure 3.2: Ping-style failure detector

Ping-style Strategy

With the *ping-style* failure detector, the monitored process takes a purely responsive role. As shown in Fig. 3.2, the idea is simple. A process p monitors a process q by sending regular “Are you alive?” query messages to q. Upon receiving of such a message, the monitored process replies with a message “I am alive!” If p fails to receive a reply from q, it begins to suspect q after some timeout.

3.2.2 Hierarchical protocols

Hierarchical protocols are based on a multilevel hierarchy in order to keep local a large part of the traffic. This is illustrated in Fig. 3.3 for three levels. The 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.

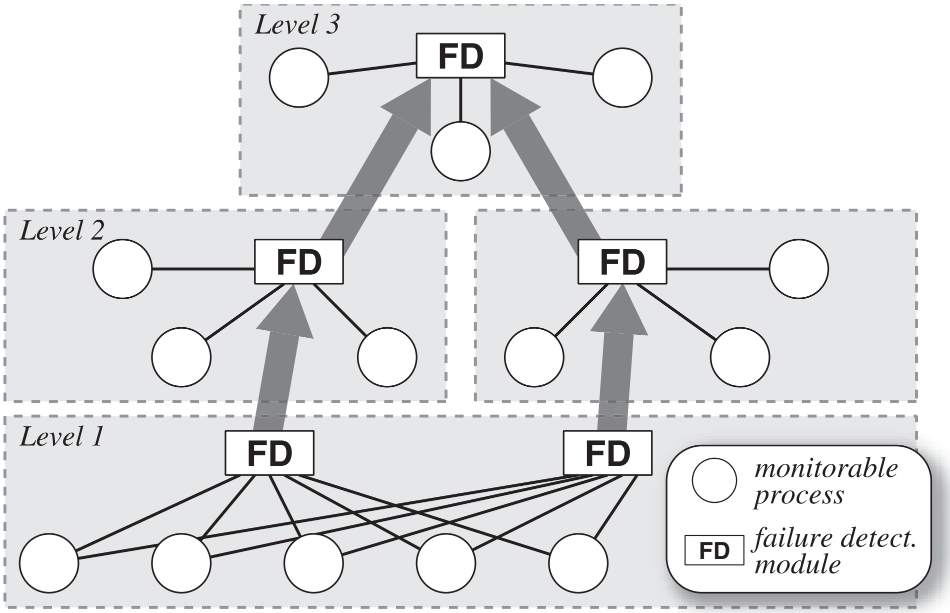

Figure 3.3: Hierarchical protocols

Compared with traditional implementations, hierarchical protocols generate fewer control messages as the information is propagated along tree-like structures rather than along a fully connected graph. In addition, the approach can potentially take advantage of the physical topology of the system.

CORBA failure detector

Felber *et al.* [FDGO99] proposed an architecture for a CORBA failure detection service. The authors presented a generic interface that could be adapted to various approaches. In their paper, the focus was put on hierarchical architecture (see Fig.3.4), although they also argued that the same interface could easily be adapted to gossip-style architecture (see Sect. 3.2.3).

The concept is better explained through an example. Figure 3.4 shows a configuration with three subnets (LAN1, LAN2, LAN3) and four failure detector modules (FD1, FD2 and FD3). The figure also shows monitored objects as circles and monitoring objects (or clients). Monitoring objects can be either objects or processes that use the failure detection service to monitor the status of some of the monitored objects.

Direct interactions between processes- and failure detector modules are usually kept within the same subnet (e.g., FD1↔monitored objects in LAN1). In contrast, interactions across subnets usually involve two failure detector modules (e.g., FD1↔FD2). There can be several

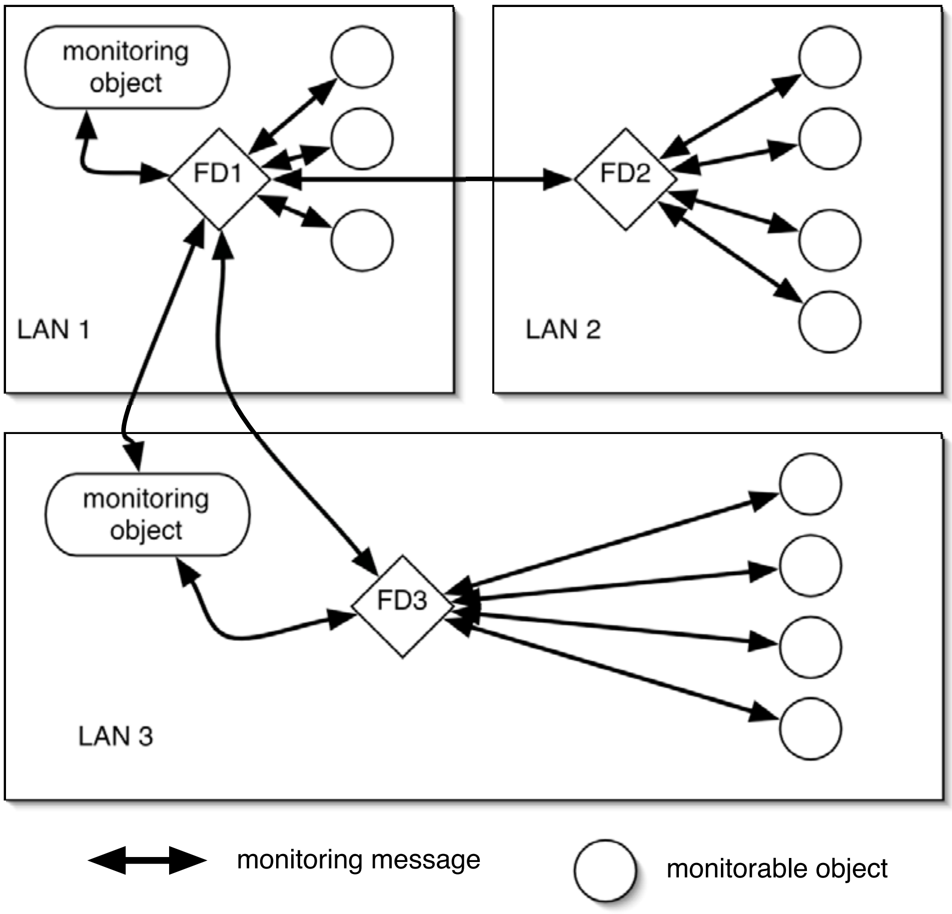

Figure 3.4: The hierarchical configuration

interactions between a client and the object it monitors. For instance, if a monitoring object in LAN3 wants to monitor the status of a monitored object in LAN1 or LAN2, the interaction path will go through FD1 and FD3. In addition, failure detector modules can themselves be monitored as they may also be subject to failures.

Using this approach, failure detector modules monitor objects 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. Thus, the total number of messages is significantly less than with traditional approaches and hence the approach successfully addresses the problem of scalability 2. This approach is, however, based on a rather static topology and makes it difficult to address the problem of dynamism. 5

Globus Failure Detector

Stelling *et al.* [SFK$^+$98] proposed a failure detection service for the Globus Grid toolkit. Globus is a middleware platform to support Grid applications [FKT01].

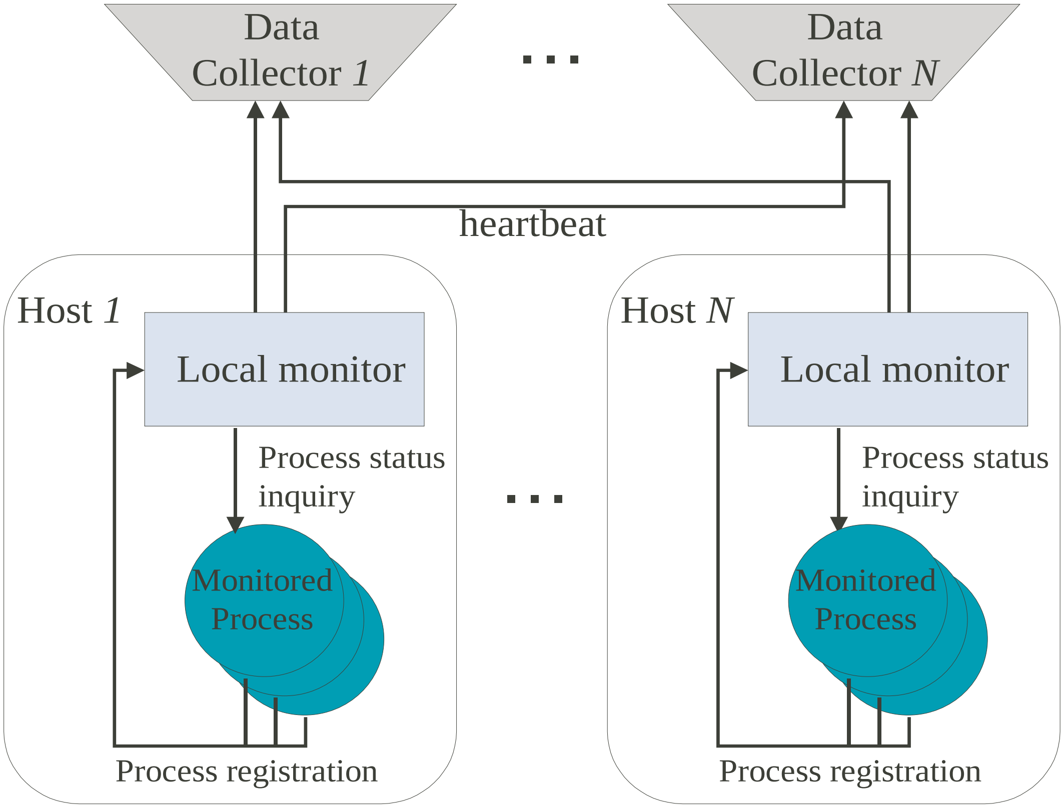

Figure 3.5: The Globus failure detection service

The architecture of the proposed failure detector is based on two layers. The lower layer consists of *local monitors*, while the upper layer consists of *data collectors* (see Fig. 3.5). A local monitor is responsible for monitoring all processes that run on the same host. The local monitor periodically sends heartbeat messages to data collectors, including information about the monitored processes. Data collectors gather heartbeats from local monitors, identify failed components, and notify the applications about their suspicions.

As it is based on a hierarchical approach, the Globus failure detector better addresses the problem of scalability 2 than traditional approaches. However, there are several shortcomings. First, the hierarchy is limited to only two levels, which means that the method does not take full advantage of the hierarchical approach. Second, the local monitors are expected to broadcast heartbeat messages Periodically to all data collectors, thus failing to address the problem of 1 message saving. Third, although aimed at Grid systems, the failure detector is based on a partly static architecture which makes it not very suitable for addressing the problem 5 of dynamism.

Bertier's hierarchical protocol

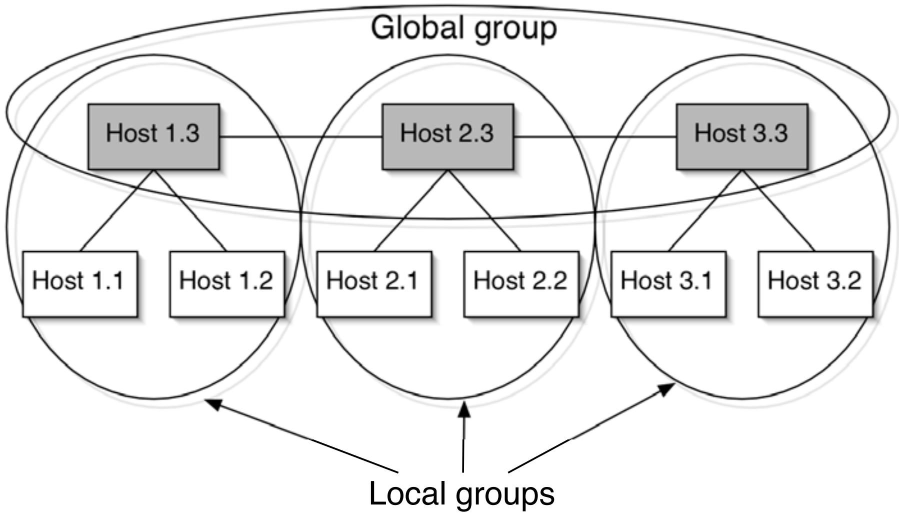

Figure 3.6: The structure of Bertier's failure detector

Bertier *et al.* [BMS03] studied the performance of the failure detection service with an hierarchical structure (see Fig.3.2.2). Implementation of the service has the hierarchical organization mapped upon the network topology. In this case, hosts in the global group are leaders. Each leader monitors nodes in its local group and diffuses information about failures to other leaders when it notifies failures. Other leaders report the information received from other leaders. The advantage of this implementation is that it reduces message complexity while continuing to recover failures, especially on leader nodes.

The failure detector module is split into two layers; a basic layer and an adaptation layer. The basic layer provided a short detection time and adapts the emission interval to the network conditions. The adaptation layer customizes the quality of service provided by the basic layer in accordance with application needs. Each adaptation layer, is attached to an application, and processes information from the basic layer. In other words, information provided by the basic layer needs some processing to adapt to each application requirement. It seems that there are redundant mechanisms for failure detection.

3.2.3 Gossip-style protocols

Gossip-style failure detectors [vRMH98, GCG01], also sometimes called epidemic-style failure detectors, are based on the observation that rumors (or diseases) can propagate very efficiently within a system, even a very large one. More concretely, failure detector modules pick random partners with whom they exchange information about suspected processes. Doing so ensures that suspicions are eventually propagated over the whole system.

One of the very strong advantages of gossip-style protocols is that they are completely oblivious to the underlying topology, and hence are completely oblivious to topology changes. In other words, topology changes do not need to affect the performance of this class of failure detectors.

Gossip-style failure detector

Although the idea of using epidemic protocols to propagate information efficiently was proposed a long time ago, it was first applied to failure detectors by van Renesse *et al.* [vRMH98]. For their failure detector, the authors distinguished 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 every other module it knows about and the last time it heard from them. Regularly, failure detector modules randomly pick some other module and send its list to it, regardless of the physical topology. Upon reception of such a list, the module merges the received list with its own. If a module fails to receive heartbeats from one of its neighbors, it begins to suspect that neighbor, after some timeout.

Multilevel gossiping is a variant aimed at large-scale networks. The protocol defines a multilevel hierarchy using the structure of Internet domains and sub-domains as perceived by looking at the respective IP addresses of processes. For instance, two hosts with respective IP addresses “192.168.0.1” and “192.168.0.2” share the same subnet, while host “10.1.1.1” belongs to a completely different domain. In the failure detection protocol, most gossip messages are sent using the basic protocol within a subnet. Then, fewer gossip messages are sent across subnets, and even fewer between domains.

Gossip-style protocols address the problem 2 of scalability quite efficiently. The total number of messages for propagating information about suspected processes can be kept low, regardless of the actual network topology. The number of messages in a given domain depends only on the number of subnets in that domain. According to [vRMH98], this protocol tolerates message loss (Problem 3), 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 (in fact it is unbounded). In this approach, the information propagation route is not static, and thus, it addresses the problem of dynamism 5.

SWIM failure detector

Gupta *et al.* [GCG01, DGM02] developed the *SWIM* process Group-membership protocol. This is a gossip-style group-membership protocol which also includes a gossip-style failure detector similar to the one mentioned in Sect. 3.2.3 [vRMH98]. Each failure detector module has a set of modules as a view. It selects members at random and sends them a ping message. If it does not receive an ack message corresponding to the ping message, it sends ping-req messages to randomly selected members with a view to asking them to send ping messages to its target (in this case, the target is FD_3, see Fig. 3.7). FD_1 and FD_2 in the figure send ping messages to FD_3 after receiving the ping-req messages from FD_4. They send ack messages to FD_4 if they receive ack messages from FD_3.

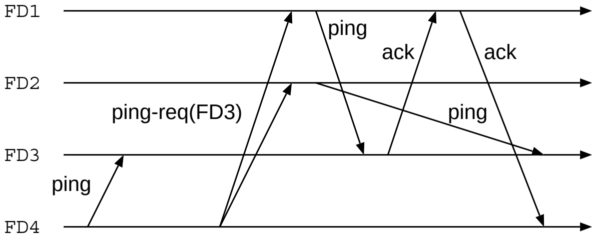

Figure 3.7: A scenario for a protocol period of SWIM failure detector

The main difference is that, in the case of SWIM, failure detector modules have a redundant mechanism for failure detection in order to avoid false suspicions. It also 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. Thus, the protocol can address the problems 2 of scalability, 3 message loss and 5 dynamism.

However, it can only handle a fixed set of monitored processes. This is because the joining/leaving of processes is handled by the membership protocol, a discussion of which is beyond the scope of this dissertation.

3.2.4 Adaptive protocols

Adaptive protocols [CTA02, SDS01, FRT01] can be configured to adapt to the behavior or requirements of an application, network conditions, etc. [GCG01] and can also be configured to application requirements. They have a strong relationship with applications.

Chen's failure detectors

Chen et al. [CTA02] studied a set of quantitative metrics to specify the *quality of service (QoS)* of failure detectors (e.g., speed of the reaction against failures). They also proposed several algorithms with synchronized and unsynchronized clocks based on a probabilistic analysis of network traffic and QoS requirements. For instance, NFD-U and NFD-E are algorithms with unsynchronized, drift-free clocks. These failure detectors consider the n most recent heartbeats. It needs the sequence number $s_i, \ldots, s_n$, the arrival date $A_1, \ldots, A_n$ and the interval of emission of heartbeats η to compute an estimated arrival time (EA_{l+1}) for the next heartbeat using sampled arrival times from the recent past. For instance, NFD-U computes the estimated arrival time as follows.

$$EA_{l+1} \approx \frac{1}{n}\left(\sum_{i=1}^{n} A_i - \eta s_i\right) + (l+1)\eta. \tag{3.1}$$

The next freshness point τ_{l+1} is set by $EA_{l+1} + \alpha$, and is computed on receipt of each heartbeat. Let p_L be the message loss rate and $V(D)$ be the variance of message delay. Both are computed by the Estimator of the probabilistic behavior of heartbeats. Let T_D^U be the upper bound on the detection time and T_D^u derived by $T_D^U - E(D)$, where $E(D)$ is the expected value (or mean). The emission interval η and safety margin α are computed by the Configurator as in the following steps at the beginning of monitoring.

1. Compute γ' = $(1-p_L)(T_D^u)^2/(V(D)) + (T_D^u)^2$ and let $\eta_{max} = min(\gamma', T_D^U, T_D^u)$.
   If $\eta_{max} = 0$, then stop computing; otherwise continue.

2. Let $f(\eta) = \eta \cdot \Pi_{j=1}^{\lceil T_D^u/\eta \rceil - 1} \frac{V(D)+(T_D^u - j\eta)^2}{V(D)+p_L(T_D^u - j\eta)^2}$
   Find the largest $\eta \leq \eta_{max}$ such that $f(\eta) \geq T_{MR}^L$, where T_{MR}^L is a lower bound on the mistake -recurrence time.

3. Set the safety margin $\alpha = T_D^u - \eta$.

The algorithm obtains η and α as an output of the Configurator at most once. A failure detector suspects some process if it does not have any heartbeat from the process until τ_{l+1}. Thus, τ_{l+1} has almost same role as a timeout. The safety margin is determined by the Configurator based on QoS requirements (e.g., upper bound of detection time T_D^U) and network conditions (e.g., message loss probability P_L).

Chen's approach adjusts a timeout according to a QoS requirement from an application. Hence, it addresses the problem of 4 flexibility

Bertier's Failure Detectors

Bertier *et al.* [BMS02] proposed a different estimation function, which combines Chen's estimation with another estimation due to Jacobson [Jac88] and developed in a different context. The estimation function is defined as a recursive equation. A monitoring process estimates the next arrival time as follows, until the process receives at least n heartbeat messages from a monitored process:

- $U_{(l+1)} = \frac{A_k}{k+1} \cdot \frac{k \cdot U_{(k)}}{k+1}$ average of arrival time
- $EA_{(l+1)} = U_{(l+1)} + \frac{k+1}{2} \cdot \eta$
- with $U_{(1)} = A_0$

When the monitoring process has received more than n heartbeat messages, it uses:

$$EA_{(l+1)} = EA_{(k)} + \frac{1}{n}(A_k - A_{(k-n-1)})$$

The safety margin α is evaluated using Jacobson's round-trip time estimation algorithm when the monitoring process receives a heartbeat message from the monitored process. The estimation algorithm constantly adapts the margin with respect to the network conditions. However, it can take a long time to converge. The freshness point is computed by:

$$\tau_{(l+1)} = EA_{(l+1)} + \alpha_{(l+1)}$$

Bertier's proposal provides a shorter detection time, but generates more wrong suspicions with Chen's estimation. The estimation method assumes a local area network which has very small variance in arrival time. The resulting failure detector is proved to belong to class $\Diamond\mathcal{P}$ when executed within a specific partially synchronous system model.

ADAPTATION failure detectors

Sotoma *et al.* [SM01] proposed an implementation of an adaptive failure detector with CORBA. The approach extends the Fault Tolerant CORBA OMG specification. Their algorithm computes the timeout based on the average time of arrival intervals between heartbeat messages, and some ratio between the arrival intervals. The implementation provides interfaces with both heartbeat style and ping-style monitoring.

Lazy Failure Detectors

In the lazy failure detection protocol [FRT01] processes monitor each other by using application messages whenever possible to get information on processor failures. This protocol requires that each message be acknowledged. In the absence of application messages between two processes, control messages are instead used.

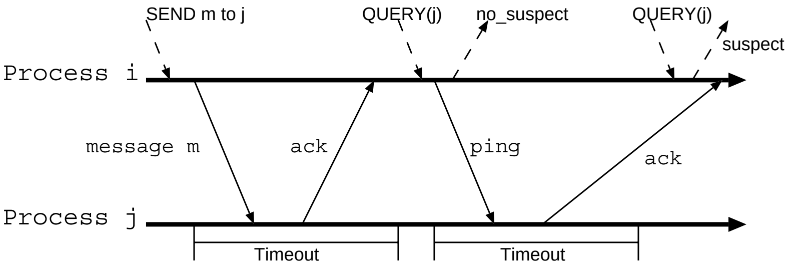

Figure 3.8: Lazy failure detection protocol

As illustrated in Fig. 3.8, an application process can use three primitives to get information about monitored processes. The first one is the SEND primitive which is used by some process p_i to send an application message m to another process p_j. The SEND primitive includes some control information with the application message. The second one is the RECEIVE primitive used by p_i to receive an application message. The third one is the QUERY method which is used to know whether p_j is suspected of having crashed[1]

The lazy failure detection protocol decreases the number of failure detection messages and hence addresses the problem 1 of message saving. Although originally proposed for LANs, the approach is relevant to large-scale distributed systems. The efficiency of this approach depends largely on the actual communication patterns of the application. So, it may perform well for some kinds of applications and poorly for others.

3.2.5 Other Implementations

Two Timeout Failure Detectors. The two-timeout approach [Déf00, DFS99, DSS98, USS03] can also be seen as a first step toward adapting to application requirements, but the solution lacks generality. The two-timeout approach was proposed and discussed in relation to group membership and consensus. In short, it was proposed to implement failure detection based on two different timeout values; an aggressive one and a conservative one. The approach is well suited for building consensus-based group-communication systems. However, the protocol was

[1] The QUERY method returns *suspect* if p_i does not receive a reply from p_j until the timeout, otherwise it returns *no_suspect*.

not rendered adaptive to changing network conditions (although this would be feasible) and, more importantly, still lacks the flexibility required by a generic service. Indeed, this could support only two classes of application.

Ad hoc failure detectors. Sergent *et al.* [SDS01] analyzed several implementations of failure detectors and their impact on the performance of a Consensus algorithm. Although this study was done in the context of a LAN, there are several relevant points. For instance, they propose a manner by which failure detector implementations can be specialized to match the communication behavior of the applications they support. In their case, they show that such a specialization can significantly reduce the overhead of the failure detector. This approach can be seen as a way to ensure that the failure detector generates messages only when necessary, thus effectively addressing the problem of 1 message saving. However application-based failure detectors are difficult to adapt to Grids and other large-scale distributed systems. Furthermore, the specialization goes against the idea of providing a generic service.

Tailorable failure detectors. Cosquer *et al.* [CRV95] proposed configurable failure "suspectors" whose parameters can be fine-tuned by a distributed application. The suspectors can be tuned directly, but they are used only through a group membership service and view synchronous communication. There is a narrow range of parameters (only 4 choices) that can be set. Hence it is difficult to apply it to pragmatic applications and satisfy their requirements. The proposed solution also remains unable to support simultaneously several applications with very different requirements. However, it can address the problem of message saving 1 because it can configure the frequency of sending control messages (high or low) from failure detector modules.

Other approaches for failure detectors. Mostefaoui *et al.* proposed an approach for implementing failure detectors with a *query-response*[2] mechanism and proved that the approach can be used to implement the failure detector belonging the class $\diamond\mathcal{S}$ [MMR03] without any additional synchrony assumptions. They assume that the query-response mechanism-exchange obeys a pattern where the responses from some processes to a query arrive among the $(n - f)$ first ones, where n is the total number of processes, and f is the maximum number that can crash, with $1 \leq f < n$.

As far as we know, only a few failure detector implementations exist which allow non-trivial tailoring on the part of the applications, let alone the requirements of several applications

[2] The SWIM failure detector has a protocol very similar to the query-response protocol

running simultaneously.

3.3 Qualitative analysis

The protocols introduced in the previous section address some of the identified problems, but not all of them. Taking one problem at a time, methods for addressing each has been discussed. The results are summarized in Table 3.1.

Table 3.1: Relationship between existing approaches and problems

	Message saving	Scalability	Message loss	Flexibility		Dynamism
				single	multi	
Globus Failure Detection Service		○				
CORBA Failure Detector		○				
Gossip-style Protocol		○	○			○
SWIM Failure Detector		○	○			○
Chen’s Failure Detector				○		
Bertier’s Failure Detectors		○		○		
ADAPTATION Failure Detector				○		
Lazy Failure Detection protocol	○					
Ad hoc Failure Detectors	○					
Cosquer’s Failure Detector				○		

The problem 1 of message saving is a critical problem for a failure detection service, as it greatly reduces the performance of the service. Ad hoc failure detectors (see §3.2.5) address the problem by implementing the application-specific silent failure detector. It does not send any message if an application does not need a monitoring processes. In a different manner, the lazy failure detection protocol also addresses this problem (see § 3.2.4). Tailorable failure detectors can reduce the number of messages sent from failure detector modules if the parameters are set properly. However, the interface for setting parameters is not elaborate (just needing high or low frequency of emission to be set).

The problem 2 of scalability is addressed by Gossip-based protocols (gossip-style protocol [vRMH98], the SWIM failure detector [GCG01,DGM02]), and the hierarchical approach (Globus failure detector [SFK$^+$98]. The CORBA failure detector [FDGO99] and Bertier's hierarchical protocol [BMS03]) address this problem effectively. To propagate information efficiently, the former uses a probabilistic approach and the latter relies on a scalable hierarchical configuration. Both approaches can significantly decrease the total number of failure detection messages in comparison with traditional approaches.

The problem 3 of message loss has a negative effect on the accuracy of the failure detector, with more wrong suspicions being generated. Experimental results published in [vRMH98, GCG01] show that a gossip-style protocol can address this problem.

The problem 4 of flexibility is partially addressed by Chen's failure detector [CTA02], and both Bertier's failure detectors [BMS02,BMS03] and Sotoma's ADAPTATION algorithm [SM01]. However, these failure detectors can adjust a timeout (or called a freshness point) dynamically to a *single* requirement or process. Whereas numerous processes may need such a failure detector module simultaneously, Cosquer's tailorable failure detector [CRV95] can be tailored to several applications, but only to a limited extent and with a serious jump in the complexity. It means that this failure detector can only allow tailoring of two parameters and can set them High or Low. Thus, it is difficult to satisfy pragmatic application requirements in large-scale distributed systems. Flexibility for multiple processes is actually not addressed by any of the surveyed protocols in the context of large-scale distributed systems (Table. 3.1).

Solving the problem 5 of dynamism is essential when a failure detector is proposed as a service. For instance, a Grid system may change its configuration during its execution. The failure detection service should be aware of these configuration changes, and should be able to adapt accordingly. Gossip-style protocols and the SWIM failure detector can address this issue well, since they do not depend on the system configuration.

3.4 Discussion

The failure detector implementation proposed by Chen *et al.* [CTA02] can also be tuned to application requirements. However, the parameters must be dimensioned *statically*, and can only match the requirements of a *single* application. The implementation proposed by Cosquer *et al.* [CRV95] can be tailored to several applications and it comes closest to addressing the flexibility problem. Yet, this does not come without practical limitations on the number of applications that can be served at any one time. Hence, in spite of their merit, this author thinks that they do not fully solve the problem of adapting to many application requirements. With respect to other failure detection protocols, it can be said that they provide a "hardwired" degree of accuracy which must be shared by all applications.

This section identified problems for designing and implementing a scalable and generic failure detector service in very large-scale distributed systems, such as Grid. Several approaches proposed in the literature that address some of these problems have been surveyed and discussed. It turns out that no known implementation addresses all of the problems, and that not a single one address the problem 4 of flexibility. Several metrics for measuring and comparing failure detector implementations with respect to four of the identified problems have been discussed. Given specific system settings, these metrics provide a fair method for comparison. The proposed metrics extend the metrics of Chen *et al.* [CTA02].

3.5 Summary

This chapter identified problems for designing and implementing a scalable and generic failure detection service in a large-scale system. Several approaches proposed in the literature were studied. Their effectiveness and limitations in addressing the identified problems have been discussed. Each approach successfully addresses one or more of these problems but no approach provides a complete and satisfactory solution. Particularly, they are lacking in flexibility in the context of large-scale systems. Thus it is felt that a combination of some of these approaches and concepts is required to provide an efficient failure detection service for large-scale systems.

Chapter 4

Accrual Failure Detectors

4.1 Motivation

In the previous section, we pointed out some problems for implementing a failure detection service for large-scale distributed systems. Specifically, no approach can address the QoS requirements of several distributed applications simultaneously, in other words, they lack flexibility.

The conflicting requirements (mentioned in §1.1) 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 the adaptive failure detectors described in the literature. Roughly speaking, these adaptive failure detectors are based on heartbeat messages and some timeout Δ_{to} determined dynamically. The value of Δ_{to} is computed using the recent history of message arrivals and an estimation function to predict the arrival of the next heartbeat. Using this approach, failure detectors can adapt to changing network conditions, but their accuracy is determined by the estimation function. Adapting to the requirements of several applications would require management of as many timeout values, which is, of cours, not acceptable. An alternative approach would allow applications to set their own timeouts, with the obvious drawback that failure detection cannot easily adapt to changing network conditions.

4.2 Overview

Practical solutions can nevertheless be developed for systems in which message delays follow some probability distribution (e.g., [CTA02]). In particular, adaptive failure detection mechanisms [BMS02, CTA02, FRT01] consider some system where the parameters of this distribution are unknown, and can change over time, but eventually stabilize for periods that are "long enough" for the whole system to make some progress. [1] The idea of adaptive failure detection is that a monitored process p periodically sends heartbeat messages ("I'm alive!"). A process q begins to suspect p if it fails to receive a heartbeat from p after some timeout. Adaptive failure detection protocols change the value of the timeout dynamically, according to the network conditions measured in the recent past. In doing so, adaptive protocols are able to cope adequately with changing networking conditions, and hence are particularly appropriate for common networking environments, or the Internet. In particular, they are able to maintain a good compromise between how fast they detect actual failures, and how well they avoid wrong suspicions.

The main drawback of adaptive failure detection protocols that we are aware of [BMS02, CTA02, FRT01] is their inability to address the QoS requirements of several distributed applications *simultaneously*, in other words, their lack of flexibility [HCK02].

Let us illustrate this with a simple example. Consider for instance a situation where two applications are running simultaneously, with one an interactive application and the other a heavy-weight database service. The former application must always be highly responsive; it needs fast yet possibly inaccurate failure detection. Meanwhile, the latter application has a high reconfiguration overhead, and needs highly-accurate failure detection, even though it might be slow. Addressing the requirements of both applications is not possible with the usual "one size fits all" approach, adopted by the known adaptive protocols.

The obstacle comes from having a timeout for a pair of a corresponding processes and an application requirement. In this case, it is difficult to realize effectively the adaptation for diverse application requirements.

A novel approach is proposed, called *accrual failure detectors*, that can adapt to the network conditions and application requirements. They do not have timeout inside the failure detector module to reconcile both types of adaptation. The key idea is that accrual failure detectors provide an *accrual value* that a corresponding process is suspected to have crashed. The value increases monotonically by elapsed time if the corresponding process has crashed. Each application has its own threshold. It queries an accrual failure detector module and gets the accrual value of the process. It then decides whether or not to suspect the process according to the threshold and accrual value. In fact, the threshold reflects the requirement of the application. Thus, this approach can provide failure detection with diverse requests.

In a recent position paper, Friedman [Fri02] proposed to investigate the notion of fuzzy group-membership as an interesting research direction. The idea is that each member of the group is associated with a fuzziness level instead of binary information (i.e., member or not a

[1] Exact assumptions can vary slightly between authors.

member). Although Friedman does not actually describe an implementation, it is believed that a fuzzy group membership could be built, based on the κ failure detector.

Similarly, the κ failure detector could also be useful as a low-level building block for implementing a partitionable group membership, such as Moshe [KSMD02]. Such group membership must indeed distinguish between message losses, network partitions, and actual process crashes. For instance, Keidar et al. [KSMD02] decided that a network partition has occurred when more than three consecutive messages have been lost. Typically, this could be done by using the κ failure detector and setting an appropriate threshold.

4.3 System Model

4.3.1 Assumptions of the System

It is assumed that two processes p and q, p are monitoring q. They communicate only by sending and receiving messages. Assume that every pair of processes is connected by two unidirectional fair-lossy channels. In practice, a fair-lossy channel can be implemented by some best-effort lossy communication service, such as UDP/IP.

Assume the system to be asynchronous in the sense that there are no bounds either on communication delays or on process speed. For each communication channel, 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. Assume that the parameters of the random variable can change over time, but that eventually they become stable.

This system model is possibly a little stronger than the asynchronous model described by Fischer et al. [FLP85] because some probabilistic assumptions are made about the behavior of the system. However, the model remains weaker than any of the partially synchronous models defined by Dwork et al. [DLS88], because the positively unbounded distribution implies that no bound (known or unknown) can exist on communication delays.

4.3.2 Probabilistic Network Behavior

Assume that communication channels behave independently with regard to their respective timing behavior. For each communication channel, assume message delays are determined by some random variable whose characteristics are unknown, independent of other communication channels, and can change over time. Assume bursty traffic, which means that two consecutive messages are very likely to have similar probabilistic characteristics. Periods during which all consecutive messages follow the same probabilistic behavior are considered *stable*. During stable periods, we assume that inter-arrival times of periodically sent messages follow a normal distribution whose parameters (mean and variance) are not known a priori.

More intuitively, we can regard the system as moving from one stable period to another with different characteristics, and possibly with some unstable behavior during the transition. For instance, this can model the fact that traffic in a corporate network is significantly different during working hours compared with during the night, when fewer people are using it.

4.4 Definitions

Accrual failure detectors are defined by computing accrual values and the interaction pattern. Each module of the accrual failure detector outputs an accrual value of a corresponding process that it suspects to have crashed. The module somehow obtains information from the process about received messages and computes the value.

Each accrual failure detector module has a function f, called *Accrual function*, for computing an accrual value. The value shows the accuracy if the failure detector suspects the corresponding process at time t. Therefore, the value indicates the degree to which the process is suspected to have crashed.

Definition 7 (Accrual function). *Let t be the time of calling the accrual function f. f outputs an accrual value of the corresponding process as follows.*

$$f(t) \quad : \quad \mathbb{R} \longrightarrow \mathbb{R}^{+} \tag{4.1}$$

Therefore, any accrual value output by the accrual function $f(t)$ failure detector module is a positive real number.

An accrual value increases monotonically by elapsed time if a corresponding process has crashed. If the process does not crash, the value will be initialized at the time that the failure detector knows the process is alive (e.g., receipt of the next message).

Definition 8 (Monotonicity). *There is a time after which an accrual value monotonically increases by t if and only if the corresponding process has crashed.*

- *There is a point after which $f(t)$ is monotonically increasing if a process is faulty.*

- $\lim\limits_{t \to +\infty} f(t) = +\infty$*, iff process is faulty.*

Meanwhile, processes can query the local module to get information about the status of the process. Each process has its own threshold, which means its requirement, and it decides its suspicion of the process according to its own threshold (see Fig. 4.4). In fact, processes could suspect a process if the accrual value corresponding to the process is larger than its own threshold. Thus, this approach is able to address the flexibility problem (see § 3.1).

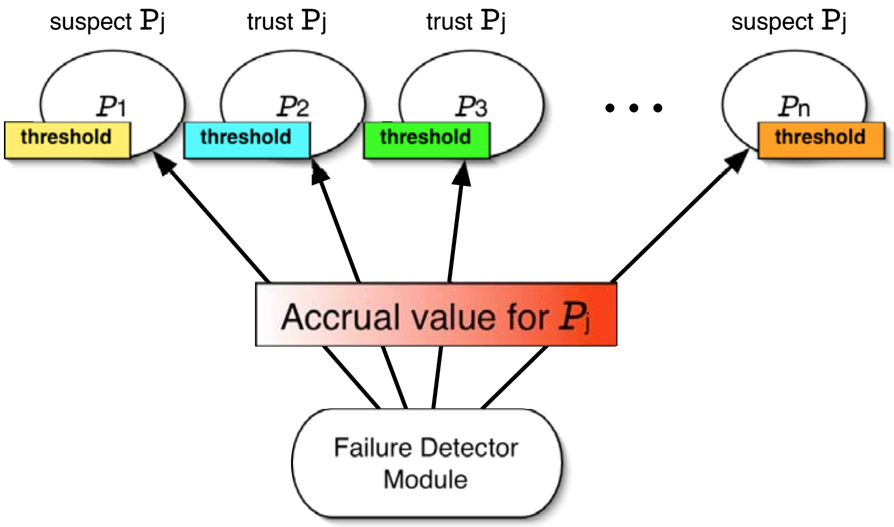

Figure 4.1: The interaction pattern of accrual failure detectors

4.5 Architecture

4.5.1 Interaction Models

From an architectural standpoint, failure detectors can be seen as a form of notification service which conveys failure notification events from one process to another. As with any other notification service, the information can be conveyed using two basic interaction models, namely the *push model* and the *pull model*.

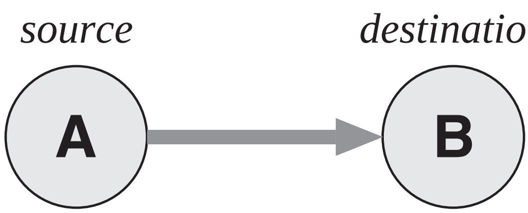

Figure 4.2: Push model

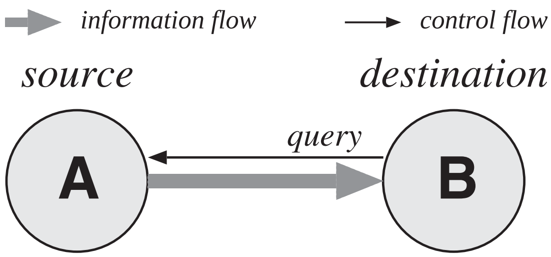

Figure 4.3: Pull model

In the push model, control and information flow in the same direction, as illustrated in Fig. 4.2. When information is available at the source (entity A), this information is sent directly to the destination (entity B). For a more concrete example, this is what happens with event notification based on callbacks (or "spam" with marketing!).

In the pull model, control and information flow in opposite directions. As shown in Fig. 4.3, the destination (entity B) queries the source (entity A) for information, and the source answers by sending back available information. In this case, the protocol is based on the same principle as busy waiting or on-demand event notifications.

4.5.2 Information Propagation

The failure detection service is simply divided into two mechanisms; one for suspecting failures and the other for propagating information on suspects. The technique of accrual failure detectors is for point-to-point communication with a pair of nodes, thus it is included in the former mechanism. In this dissertation, the former technique forms the main focus. On the other hand, the latter technique is also important for implementing a scalable failure detection service.

Two main structures for information propagation are discussed in this section. The role of accrual failure detectors by the argument are also presented.

Tree structured propagation

Tree structure is the traditional generic service for large-scale systems such as Domain Name Service (DNS), etc. Each node constructs a tree structure and some leader nodes are located in branches. Leader nodes maintain nodes that are connected directly and report information to the leader at a higher level. They also relay information which comes from lower layers to high layers and information also flows in the reverse direction. Thus, each leader node only knows about its slaves.

The main advantage of the tree structure is that it reduces the number of messages generated by failure detector modules. Bertier *et al.* presented an hierarchical protocol for failure detectors [BMS03]. They mentioned the fact that the complexity as a function of number of messages is $n^2/g + g^2 - g - n$, where n is the total number of hosts and g is the number of local groups if n hosts are divided equitably into g local groups [2]. It also relaxes the load on processors and the network. Meanwhile, the failure of leader nodes is a vulnerability. It needs a redundancy or some other mechanism for crashing leaders.

[2] In a flat system, the complexity is $n * (n - 1)$.

Epidemic-style gossip propagation

The gossip-style protocol is often used on peer-to-peer networks. The protocol propagates information in a probabilistic manner. Someone selects neighbors uniformly at random and gives them information. The protocol has attractive scalability and reliability properties.

On scalability, the protocol diffuses information with high probability if each node has a list of neighbors, taking the list size of the order of the logarithm of the system size [GKM03]. On reliability, the reachability of information is almost 100% in the system when the system has 100,000 nodes. On the other hand, the protocol requires lots of nodes to guarantee high reliability. In contrast, the arrival time of information at some node cannot be estimated.

The epidemic-style gossip protocol can be organized as a hierarchical structure. Gupta *et al.* [GKG02] and Kermarrec *et al.* [KMG03] analysed the performance of the flat-gossip and hierarchical-gossip protocols. In fact, the latency of flat gossip is logarithmic in group size, whereas, that of hierarchical appears to increase as the square of the logarithm.

4.6 Experimental Setup

My experiments involved two computers, one located in Japan and the other in Switzerland. The two computers communicated through a normal intercontinental Internet connection. One machine ran a program sending heartbeats (thus acting like process p) while the other one was recorded the arrival times of each heartbeat (thus acting like process q). Neither machine failed during the experiment.

4.6.1 Hardware, Software and Network

- *Computer p (monitored; Switzerland)*: The sending host was located in Switzerland, at the Swiss Federal Institute of Technology in Lausanne (EPFL). The machine was equipped with a Pentiun III processor at 766 MHz and 128 MB of memory. The operating system was Red Hat Linux 7.2 (with Linux kernel 2.6.9).

- *Computer q (monitoring; Japan)*: The receiving host was located in Japan, at the Japan Advanced Institute of Science and Technology (JAIST). The machine was equipped with a Pentium III processor at 1 GHz and 512 MB of memory. The operating system was Red Hat Linux 9 (with Linux kernel 2.4.20).

All messages were transmitted using the UDP/IP protocol. Interestingly, using the `traceroute` command has shown that most of the traffic was actually routed through the United States, rather

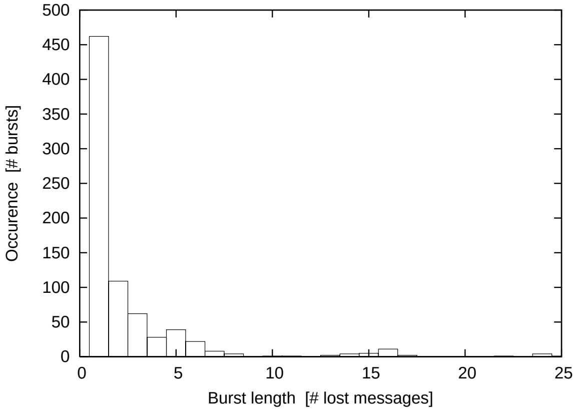

Figure 4.4: Distribution of the length of loss bursts. A burst is defined as the number of consecutive messages that were lost.

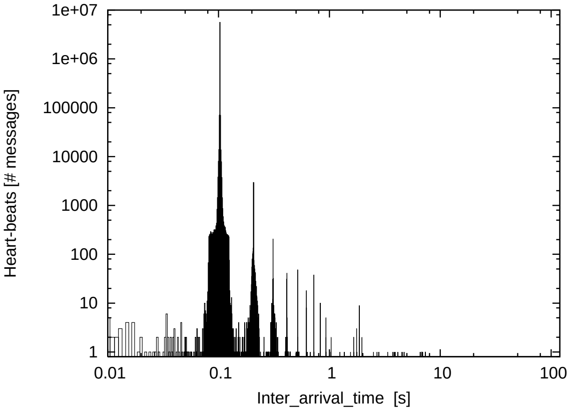

Figure 4.5: Distribution of heartbeat inter-arrival times as measured by the receiving host. Horizontal and vertical scales are both logarithmic.

than directly between Asia and Europe.

In addition, the CPU load average-usage on the two machines was monitored during the whole period of the experiments. It was observed that the load was nearly constant throughout, and well below the full capacity of the machines.

4.6.2 Heartbeat sampling

The experiment was run for exactly one full week. In the course of the week during which the experiment lasted, heartbeat messages were generated at a target rate of one heartbeat every 100 ms. The average sending rate actually measured, was of one heartbeat every 103.5 ms (standard deviation: 0.19 ms; min.: 101.7 ms; max.: 234.3 ms). In total, 5,845,712 heartbeat messages were sent of which 5,822,521 were received (about 0.4 % message loss).

It was observed that message losses tended to occur in bursts, the longest of which was 1094 heartbeats long (i.e., it lasted for about 2 minutes). 814 different bursts of consecutively lost messages were observed. The distribution of burst lengths is represented in Figure 4.4. Beyond 25, there is a flat tail of 48 bursts that are not depicted in the figure. After 25, the next burst is 34 heartbeats long, and the lengths of the five longest bursts were respectively 495, 503, 621, 819, and 1094 heartbeats.

The mean inter-arrival time of received heartbeats was 103.9 ms with a standard deviation of about 104.1 ms. The distribution of the inter-arrival times is represented in Figure 4.5.

A different view of the inter-arrival times is given in Figure 4.6. The figure relates the arrival

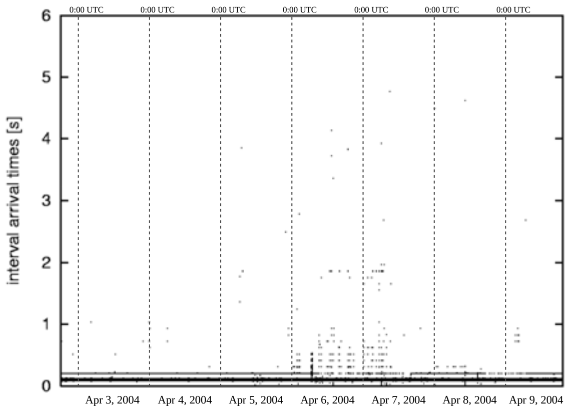

Figure 4.6: Arrival intervals and time of occurrence. Each dot represents a received heartbeat. The horizontal position denotes the time of arrival. The vertical coordinate denotes the time elapsed since the reception of the previous heart beat.

intervals (vertical axis) to the time when the second heartbeat of the interval arrived (horizontal), over the whole duration of the experiment. Very long intervals are not depicted. The first (thick) line of points at the bottom of the graph represents heartbeats that arrived normally, within about 100 ms. The second (thinner) line represents intervals obtained after a single heartbeat was lost, and so on with the other lines above it. At that frequency, losing a single heartbeat seems to be a normal situation. There is a period (April 6 and 7) where more messages were lost.

Round-trip times

During the experiment, the round-trip time was measured (RTT), albeit at a low rate. An average RTT of 283.3 ms with a standard deviation of 27.3 ms, a minimum of 270.2 ms, and a maximum of 717.8 ms was measured.

Average detection time

An *estimation* was computed for the average detection time T_D as follows. Assuming that a crash would occur exactly after successfully sending a heartbeat,[3] The time elapsed until the failure detector reports a Suspicion was measured. With the φ failure detector, the threshold Φ was considered and the computation of φ reversed to obtain the equivalent timeout. This equivalent timeout is computed each time a new heartbeat is received and the mean value $\Delta_{to,\Phi}$ taken.

[3] This is a worst case situation because any crash that occurred later (but before sending the next heartbeat) would be detected at the same time, and any crash that occurred earlier would actually prevent the last heartbeat from being sent. Either case would result in a shorter detection time.

The mean propagation time Δ_{tr} based on measurements of the round-trip time is estimated. Then, the average (worst-case) detection time is estimated simply as follows.

$$T_D \approx \Delta_{tr} + \Delta_{to,\Phi} \tag{4.2}$$

A fair comparison can be made by putting exactly the same average transmission time for all failure detectors.

Experiment

To conduct the experiments, heartbeat sending and arrival times were recorded using the experimental setup described above. The sending times were used to compute the statistics mentioned above. Then, the receiving times recorded for each different failure detector implementation and every different value of the parameters were replayed. As a result, the failure detectors were compared based on *exactly* the same scenarios, thus resulting in a fair comparison.

All three failure detectors considered in these experiments rely on a window of past samples to compute their estimations. Unless stated otherwise, the failure detectors were set using the same window size of $1,000$ samples. As the behavior of the failure detectors is stable only after the window is full, we have excluded from the analysis all data obtained during the warm-up period—i.e., the period before the window is full.

Chapter 5

φ Failure Detector

In this chapter, we show an instance of accrual failure detectors, called φ failure detectors, which use no timeout and reconcile both types of adaptation. Then a pragmatic implementation of the φ failure detector is presented, and its behavior between Japan and Europe over the period of one week is analyzed.

The key idea is that a φ failure detector provides information on the degree of confidence that a given process has actually crashed, instead of a Shakespearean nature (i.e., *suspect* or *not suspect*). More specifically, the failure detector associates a value φ_p with every known process p. This value is expressed on a continuous scale that roughly represents the current level of confidence that process p has crashed. The scale itself is adapted dynamically to match the current network conditions and to ensure adaptive behavior. Simultaneously running applications receive exactly the same information, and can set a threshold for φ_p according to their respective requirements; a small threshold value yields fast and inaccurate failure detection, whereas a large value results in accurate yet slow failure detection.

The φ failure detector can adapt to application requirements because each application can trigger suspicions according to its own threshold. Meanwhile, the failure detector can adapt to changing network conditions because the scale is defined accordingly.

The presented failure detection scheme has been evaluated under normal transcontinental conditions (between Japan and Switzerland). As we described in the previous chapter, heartbeat messages were sent at a rate of 10 times every second using the user datagram protocol (UDP), and the experiment ran uninterruptedly for a period of one week, gathering a total of about $6,000,000$ samples. Then, using these samples, the behavior of the failure detector was analyzed and compared with traditional adaptive failure detectors [BMS02, CTA02]. By providing exactly the same input to every failure detector, the fairness of the comparisons could be ensured. The results show that our failure detector implementation performed well when compared with traditional implementations, with the additional advantage that its design provides virtually limitless flexibility.

5.1 The Concept of the φ Failure Detector

As mentioned above, adaptation can occur in several different ways. To be truly generic, a failure detection service must be equally adaptive to (1) changing network conditions, and (2) application requirements. More concretely, a failure detection service must be able to meet the requirements of a wide range of distributed applications *running simultaneously*. Timeout-based failure detectors are intrinsically limited by the fact that one timeout value is necessary for each set of requirements or, in the worst case, one timeout for each concurrent application. In particular, this is also the case with adaptive failure detectors (see Sect. 3.2.4). Thus, the latter can adapt to changing network conditions, but they are unable to meet the different requirements of several concurrent applications realistically.

A novel approach to failure detection has been developed called φ failure detectors, which can address both adaptation to changing network conditions, and meeting the requirements of multiple distributed applications. The principle of the φ failure detector is as follows. Each failure detector module associates a value $\varphi_p \in \mathbb{R}^+$ to every known process p instead of managing a list of suspected processes. The value φ_p represents the degree of confidence that process p has crashed. This value is expressed according to a normalized scale, where $\varphi_p = 0$ means that there is currently no reason to doubt that p is operational, and conversely, $\varphi_p = \infty$ indicates an absolute confidence that p has crashed. Thus, failure detection modules maintain a list of pairs (p, φ_p) for every monitored process, which can be queried at any time by any application. More precisely, φ_p is defined along the following scale. Let P_{acc} denote the probability that the statement “process p has crashed” will not be contradicted in the future (by the reception of a late heartbeat). Then, φ_p can be determined by Eq. (5.1), which leads to the scale illustrated in Table 5.1.

$$\varphi_p = -log_{10}(1 - P_{acc}) \tag{5.1}$$

The failure detector must guarantee that φ_p increases monotonically, between two periods where it is reset to 0.

The interactions between the applications and the failure detector are hence different from 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

Table 5.1: The relationship between φ_p and P_{acc}

φ_p	0	1	2	3	...	∞
P_{acc}	0	0.9	0.99	0.999	...	1

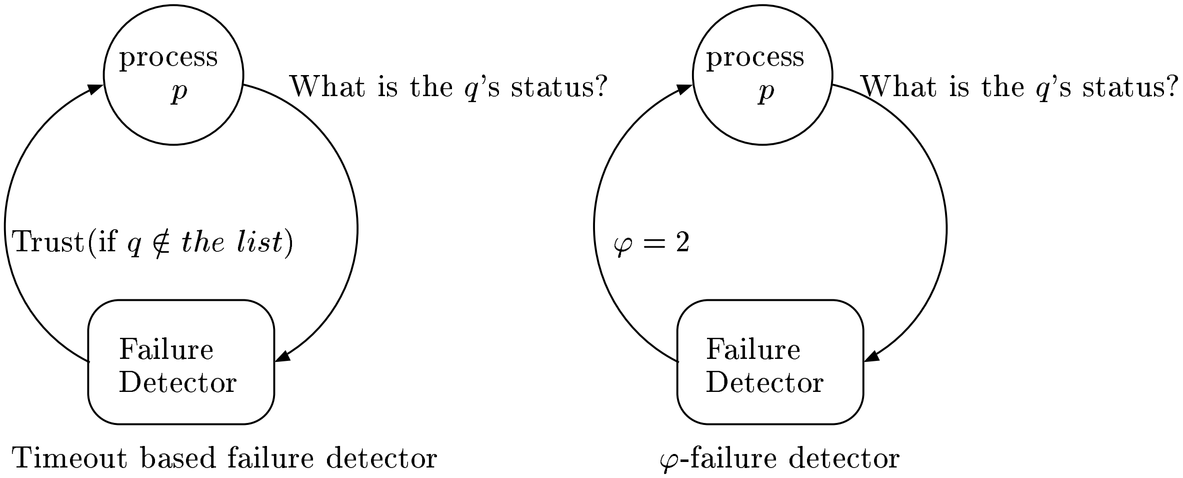
process p
What is the q's status?
Trust(if $q \notin$ *the list*)
Failure Detector
Timeout based failure detector
process p
What is the q's status?
$\varphi = 2$
Failure Detector
φ-failure detector

Figure 5.1: Timeout-based failure detector vs. φ failure detector

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). Besides, the scale ensures that (1) the value set as a threshold retains some meaning for the application (it represents the degree of confidence), and (2) the failure detection adapts to changing network conditions even with a fixed threshold (because the scale adapts).

In practice, the value P_{acc} can be computed based on the history of arrival intervals between heartbeat messages. In a possible implementation, a failure detector module analyzes the tendency of the system condition, network load and so on from the history of heartbeat messages.

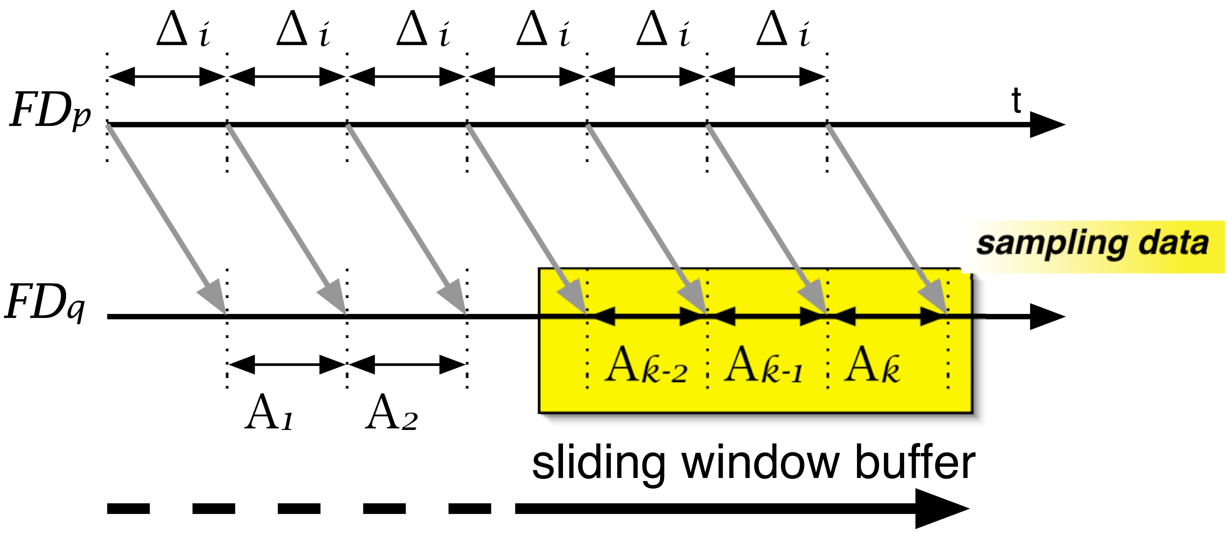

Figure 5.2: Sampling data using the sliding window

5.2 The φ Failure Detector Implementation

We propose to implement φ-failure detectors using a stochastic approach. These failure detectors have a history H of arrival intervals of heartbeat messages per a monitored process.

5.2.1 Implementation Based on a Sliding Window

In this section, the implementation of the φ failure detector, which uses a stochastic approach is described. In short, the approach is simple; a sliding window is maintained and used to compute estimated arrival times, as well as approximate the probabilistic distribution of future arrivals.

Task 1: Sampling. For each monitored process p, the failure detector modules maintain a sliding window SW_p of arrival intervals of heartbeat messages sent by p. The sliding window SW_p is implemented as a simple circular buffer and has a certain length ws_p. Let A_k be the arrival time for the k-th heartbeat message from process p. Then, the history SW_p is a sequence $\{A_1, A_2, A_3, \ldots, A_{|ws_p|}\}$, where A_1 is the arrival time of the most recent heartbeat from process p, and $|ws_p|$ is the length of the sliding window for that process. Assume that arrival time *intervals* follow a normal distribution in the implementation. It may not be a really applicable choice but arrival intervals, except message losses, gather around their mean. So, based on the history SW_p, we compute the mean μ_p and the variance σ_p, and use these parameters to estimate the probabilistic distribution of arrival times.

In fact, computing the mean μ and the variance σ^2 require only little computation. To do this, two additional variables; the sum and the sum of squares are kept. Whenever a new sample is received, $A_{|ws_p|}\}$ (or its square) is subtracted from the sums, the new sample is added and appended to the bounded history SW_p. Consequently, the size of the sliding window has no effect on the amount of computation needed to obtain the parameters of the distribution, and compute φ_p.

The task keeps track of four values that are of particular importance for the estimation of φ: the mean μ and variance σ^2 of inter-arrival times, as well as the rank k, where k has the highest rank among all received heartbeats. For the first two values, this is done by simply keeping track of the sum and the sum of squares of inter-arrival times.

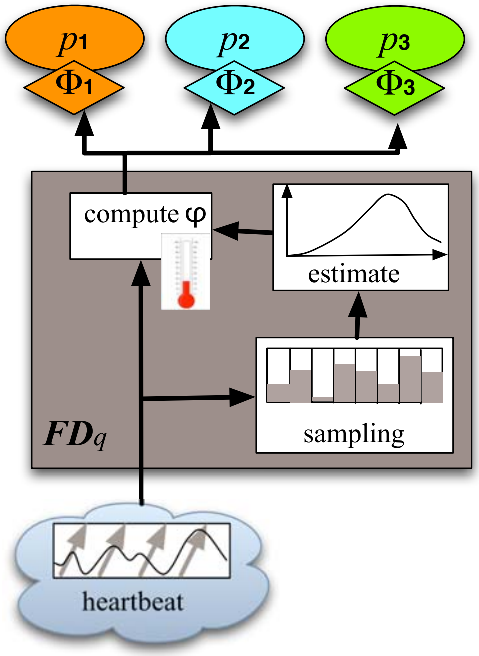

Figure 5.3: The mechanism for the φ failure detector

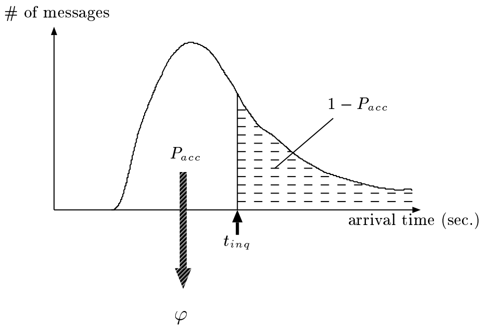

Figure 5.4: The translation between the distribution and φ

Task 2: Computing φ_p. The φ failure detector modules analyze the tendency of the heartbeat arrival time for some process q, based on H_q. H_q can be seen as a discrete distribution (see Fig. 5.3). Then, the failure detector module smoothes the distribution to get a continuous distribution, which allows P_{acc} to be estimated at any given time. The module can use this distribution to compute P_{acc}, based on time t_{inq}, where t_{inq} is the time that a monitoring process p inquires of the local failure detector module (see Fig. 5.4). P_{acc} is then converted into the φ scale defined in the previous section.

In fact, the curve in Figure 5.4 is positively unbounded, because the distribution includes a virtual message in the future. This means that there is always the possibility that the next heartbeat message will arrive at some future instant. Consequently, P_{acc} never reaches 1 in an asynchronous system (see Table 5.1).

This task is for computing the output value φ_p. The task is invoked when the failure detector module receives queries from applications. First, a failure detector module computes an accrual value that some process is suspected to have crashed according to the normal distribution. Given four values computed by the first task (i.e., μ, σ^2, k) and t is the time when some application invokes the task used to compute P_{acc}.

$$P_{acc} = \frac{1}{\sigma\sqrt{2\pi}} \int\limits_{-\infty}^{t} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx \tag{5.2}$$

Then, P_{acc} is transformed into φ_p using Eq. 5.1. The transformation is only for better understanding of φ_p.

5.2.2 Interaction with applications

The φ failure detector provides a simple interface for distributed applications; the failure detector is queried through a function call which returns the time when it was called and the computed value for φ_p (see Fig. 5.1). When an application process queries its failure detector module on the current status of some process p, the module computes the value φ_p at that time, based on the estimated distribution and the time elapsed since the receipt of the last heartbeat (see Fig. 5.3). Then, the value computed for φ_p is returned to the application process. When polling is not acceptable, the application process can set a callback that is triggered when φ_p grows beyond a given value.

5.2.3 Message losses

In this system model, it can be assumed that there is no message loss. In fact, the occurrence of message loss generates a wrong suspicion. This is consistent with the behavior of both Chen's [CTA02] and Bertier's [BMS02] failure detectors. Implicitly, we assume that, taken over long periods, the probability of message loss remains low. This assumption is confirmed by the experimental results between Japan and Switzerland, where an average loss rate below 0.4% was measured (see details in Sect. 5.3). In fact, it was observed that messages were rarely lost individually, but rather that several consecutive messages were lost, probably as the result of some network partition. During the experiments described in this dissertation, a total of

219 messages were lost, but only 117 suspicions resulted from these losses.

5.3 Performance Analysis

In this section, experimental results of the implementation of the φ failure detector are described, as well as a comparison with two other adaptive failure detector implementations, namely Chen et al. [CTA02] and Bertier et al. [BMS02]. The experiments were carried out over transcontinental links, for a consecutive duration of one week.

5.3.1 Objective

The performance of the φ failure detector was evaluated in the following two ways using the sequence of heartbeats (see §4.6). First, the influence of a threshold Φ_p and the size of the sliding window buffer to a value φ_p were measured. Then, the estimated timeout (mentioned in §2.3) of the φ failure detector, Chen's failure detector and Bertier's failure detector with the same mistake rate λ_M (see §2.3) were compared. The comparison shows how quickly a failure detector can suspect faulty process.

5.3.2 Scenarios and parameters

We ran our experiments between two machines, with one located at JAIST in Japan, and the other located in Switzerland, at the Swiss Federal Institute of Technology in Lausanne (EPFL). The experiments consist of two parts. In the first, the performance of the φ failure detector, was measured changing two parameters. First the effects of changing the threshold value for φ on both the mistake rate and the detection time were measured. Then, the impact of window size on the behavior of the failure detector was observed.

In the second part, the presented failure detector was compared with two different adaptive failure detectors [CTA02, BMS02]. In particular, the performance of all three failure detector implementations were measured by injecting each time the same one week sequence of measured heartbeat arrival times. With this approach, all three failure detectors were compared under exactly the same conditions, while the experiment was based on real traffic.

5.3.3 Tuning Parameters of the φ Failure Detector

From the standpoint of an application, it is considered that it sets a threshold Φ_p, and decides to suspect or not a process p based on the value φ_p returned by the failure detector; i.e., suspect p

if and only if $\varphi_p > \Phi_p$. As far as the application is concerned, the threshold Φ_p plays the role of a timeout. A major difference is that thresholds are set on a per-application basis, and, within each application, can also be set on a per-channel basis. Also, the threshold need not remain constant over time.

The impact of Φ_p on the implementation of the φ failure detector has been studies. In addition, the impact of the size of the sliding window has been measured.

Experiment 1: Average mistake rate

In the first experiment, the average mistake rate λ_M obtained with the φ failure detector was measured. In particular, the evolution of the mistake rate was measured when the threshold Φ, used to trigger suspicions, increased.

Figure 5.5 shows the results obtained when plotting the mistake rate on a logarithmic scale. The figure shows a clear improvement in the mistake rate when the threshold increased from $\Phi = 0.5$ to $\Phi = 2$. This improvement is due to the fact that most late heartbeat messages are caught by a threshold of two or more. The second significant improvement comes when $\Phi \in [8; 12]$. This corresponds to the large number of individually lost heartbeat messages (i.e., loss bursts of length 1). As those messages no longer contribute to generating suspicions, the mistake rate drops significantly.

Experiment 2: Average detection time

In the second experiment, the average detection time (see 4.6.2) obtained with the φ failure detector was measured, and how it evolves when changing the threshold Φ was established.

Figure 5.6 depicts the evolution of the detection time as the suspicion threshold Φ increases. The curve shows a sharp increase in the average detection time for threshold values beyond 10 or 11.

Experiment 3: Effect of window size

The third experiment measured the effect of the window size on the mistake rate of the φ failure detector. The window size was set from very small (20 samples) to very large ($10,000$ samples) and the accuracy obtained by the failure detector when run during the full week of the experiment was measured. The experiment was repeated for three different values of the threshold Φ, namely $\Phi = 1$, $\Phi = 3$, and $\Phi = 5$. Figure 5.7 shows the results, with both axes expressed on a logarithmic scale.

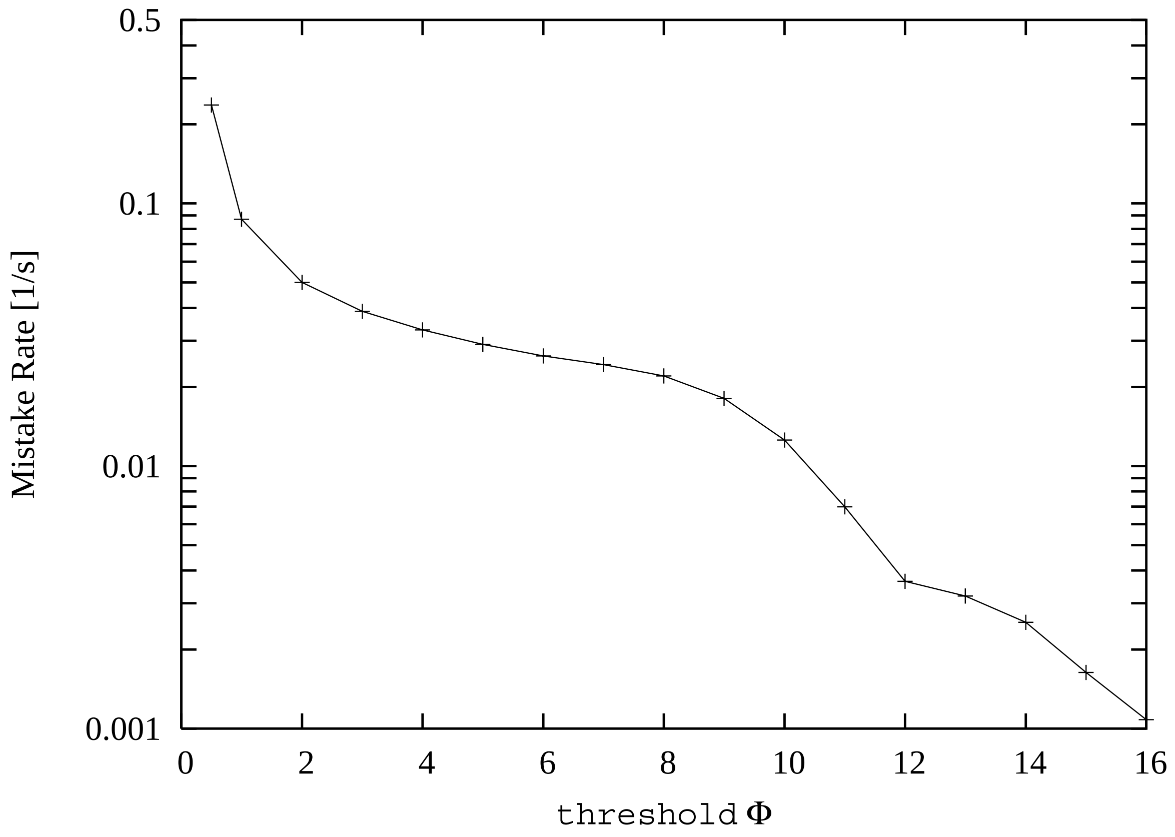

Figure 5.5: Exp. 1: average mistake rate as a function of threshold Φ. Vertical axis is logarithmic.

The experiment confirmed that the mistake rate of the φ failure detector improves as the window size increases (see Fig. 5.7). The curve seems to flatten slightly for large values of the window size, suggesting that increasing it further yields only a little improvement. A second observation is that the φ failure detector seems to be affected equally by the window size, regardless of the threshold.

5.3.4 Comparison with Chen's FD and Bertier's FD

In this section, the φ failure detector is successively compared with two adaptive failure detectors, namely Chen's failure detector [CTA02] and Bertier's failure detector [BMS02]. The goal of the comparison was to show that the additional flexibility offered by the φ failure detector does not incur any significant performance cost.

The three failure detectors do not share any common tuning parameter, which makes comparing them difficult. To overcome this problem, the behavior of each of the three failure detectors was measured using several values of their respective tuning parameters. The combinations of QoS metrics (average mistake rate, average worst-case detection time) obtained with each of

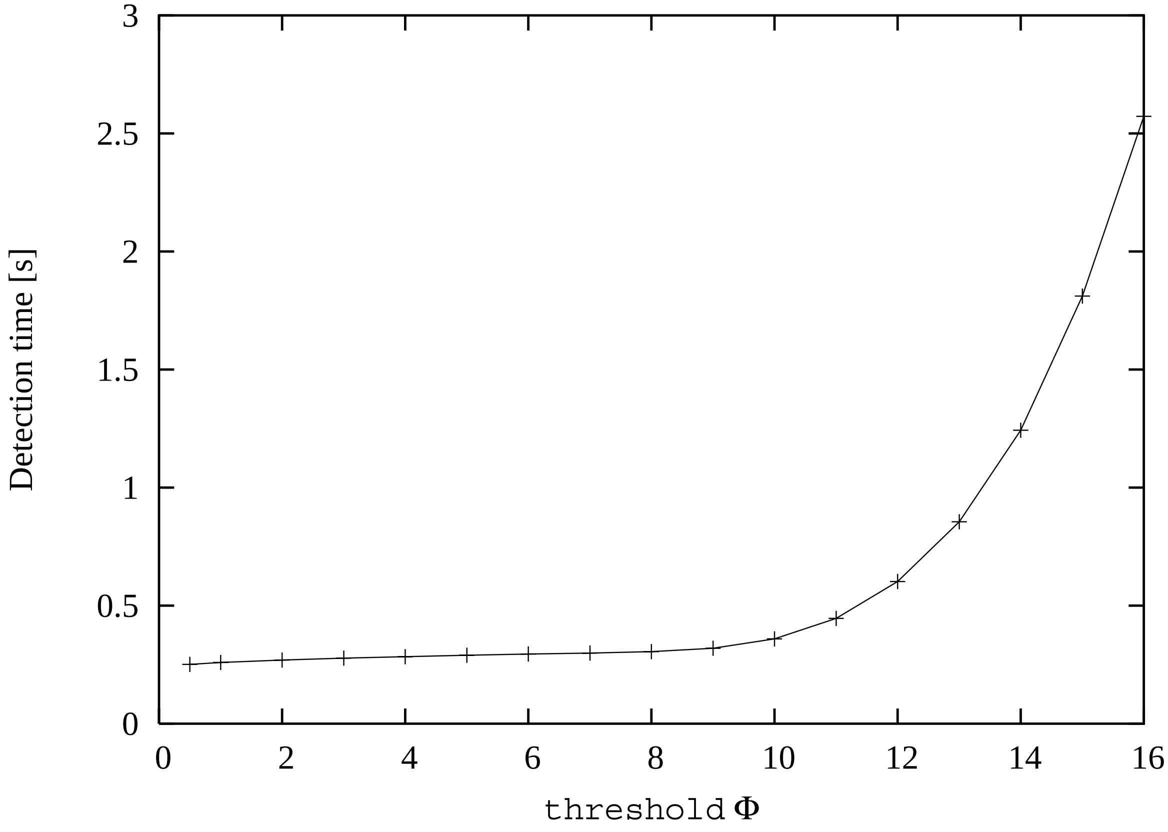

Figure 5.6: Exp. 2: Average detection time as a function of threshold Φ.

the three failure detectors were plotted.

The tuning parameter for the φ failure detector was the threshold Φ (values are also represented in Fig. 5.5 and 5.6). The φ-failure detector was executed with given Φ, where $\Phi \in [0.5; 16.0]$. The tuning parameter for Chen's failure detector was the safety margin α; this is simply an additional period of time that is added to the estimate for the arrival of the next heartbeat. We set α within $[0.0; 25.0]$. Unlike the other two failure detectors. Bertier's itself has no tuning parameter. Parameters $\beta = 1$, $\phi = 4$, were set to values that are typical in Jacobson's Roundtrip-time estimation algorithm [Jac88] and $\gamma = 0.1$ was set to follow the experiments in Bertier's papers [BMS02, BMS03]. The parameters β and ϕ permits to the variance of arrival time to be considered and γ represents the importance of the new measure with respect to the previous arrival time. These parameters influence the computation of the dynamic safety margin. Finally, as already mentioned, the window size for all three failure detectors was set to the same value of $1,000$ samples.

The results of the experiment are depicted in Figure 5.8. The vertical axis, representing the mistake rate, is expressed on a logarithmic scale. The horizontal axis, representing the estimated average detection time, is on a linear scale. Best values are located towards the lower

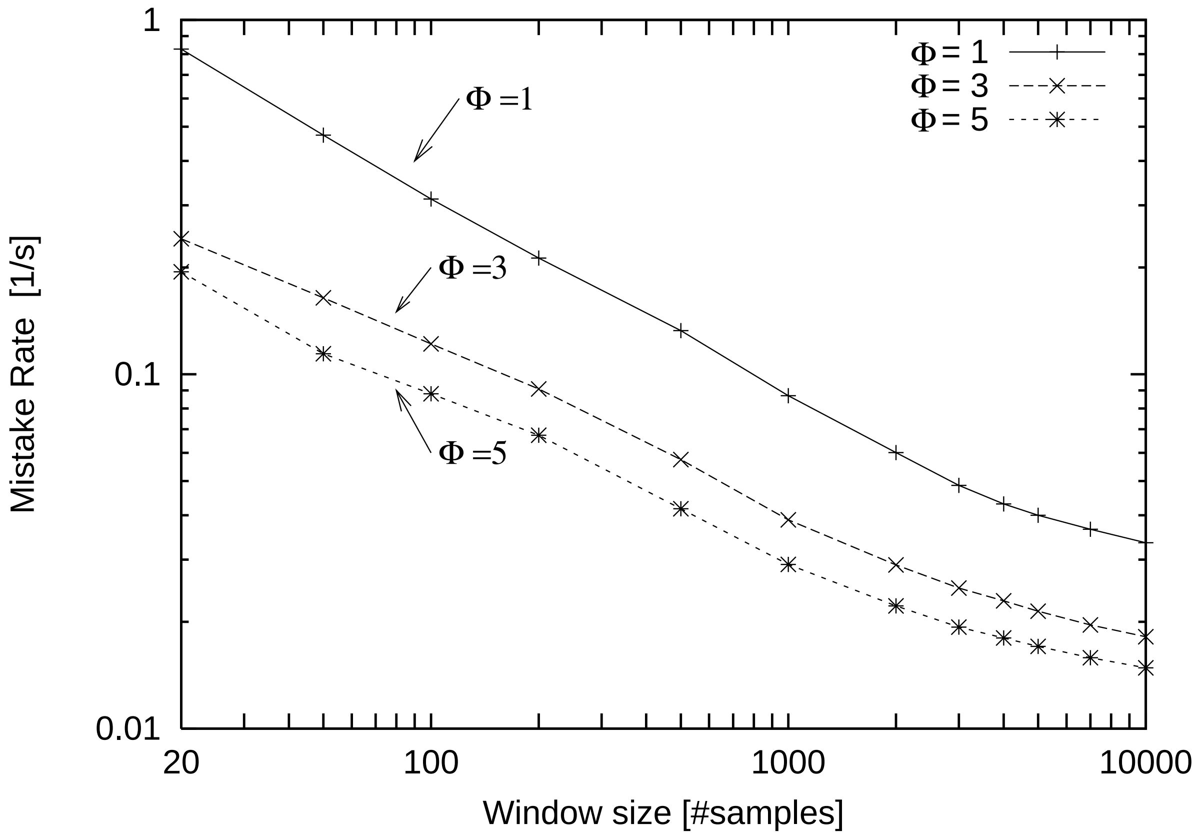

Figure 5.7: Exp. 3: Average mistake rate as a function of the window size, and for different values of the threshold Φ. Horizontal and vertical axes are both logarithmic.

left corner because this means that the failure detector provides a short detection time while keeping mistake rate low.

The results show clearly that the φ failure detector does not incur any significant performance cost. When compared with Chen's failure detector, both failure detectors follow the same general tendency. In this experiment, the φ-failure detector behaved a little better in the aggressive range of failure detection, whereas Chen's failure detector behaved a little better in the conservative range.

Quite interestingly, Bertier's failure detector did not perform very well in the present experiments. By looking at the trace files more closely, this failure detector was observed to be more sensitive than the other two (1) to message losses, and (2) to large fluctuations in the receiving time of heartbeats. It is however important to note that, according to their authors [BMS02], Bertier's failure detector was primarily designed to be used over *local* area networks (LANs), that is, environments wherein messages are seldom lost. In contrast, these experiments were done over a wide-area network.

Putting too much emphasis on the difference between Chen and φ would not be reasonable

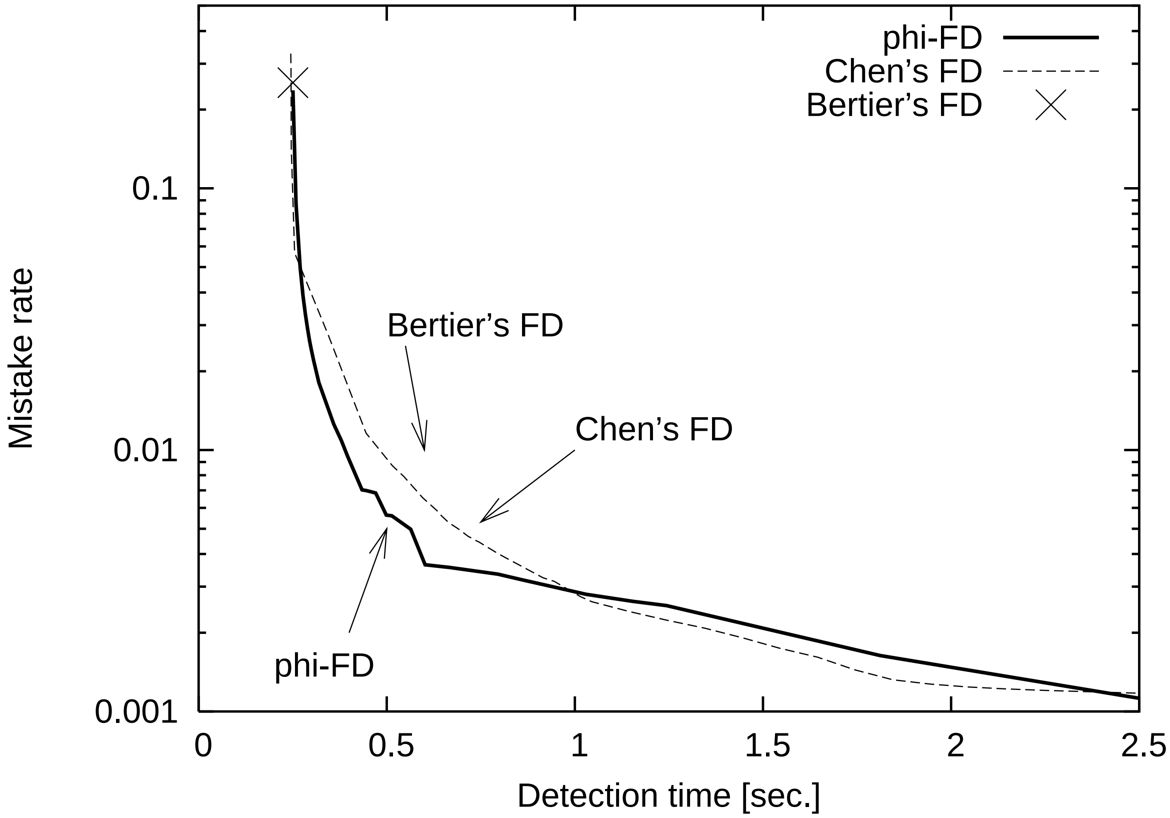

Figure 5.8: Exp. 4: Comparison of failure detectors. Mistake rate and detection time obtained with different values of the respective parameters. Most desirable values are towards the lower left corner. Vertical axis is logarithmic.

as other environments might yield to other conclusions. It is however safe to conclude that the flexibility of φ does not come with any drop in performance, especially when used over wide-area networks.

5.4 Discussion

In this chapter, a variation of accrual failure detectors called the φ-failure detector was presented. The behavior of the φ failure detector over a transcontinental Internet link, based on nearly 6 million heartbeat messages was analyzed. Finally, the behavior of this failure detector was compared with two important adaptive failure detectors, namely, Chen's [CTA02] and Bertier's [BMS02].

By design, φ failure detectors can adapt equally well to changing network conditions, and the requirements of any number of concurrently running applications. Traditional adaptive failure detectors (see Sect. 3.2.4) can also adapt to changing network conditions, based on a stochastic approach. However, they are unable to adapt to disparate application requirements.

As far as is known, this is currently the only failure detector that addresses both problems and provides the flexibility required for implementing a truly generic failure detection service. In particular, the two other adaptive failure detectors mentioned above do not address both problems.[1]

In addition to interesting observations about transcontinental network communication, these experimental results show that the failure detector behaves reasonably well if parameters are well-tuned. In particular, it can be seen that the impact of the window size is significant. Comparisons with the other failure detectors show that the φ failure detector does not induce any significant overhead as performances are similar. Nevertheless, it is believed that there is still room for improvement. In particular, techniques and mechanisms are being investigated that will (1) improve the estimation of the distribution when computing φ, (2) reduce the use of memory resources, and (3) better cope with message losses for highly conservative failure detection.

It is argued that a failure detection service must deal equally well with being sufficiently generic, and hence adaptive in aspects of both network conditions and application requirements.

In fact, this is the first proposition for a failure detection service that can simultaneously adapt to changing network conditions and disparate application requirements, without setting practical limitations on the number of applications. In some sense, the φ failure detector can be seen as a way of quantifying the confidence of the failure detection. The present work is however still in progress and several issues must be solved before the solution can be fully implemented and used in practice. The main issues that we intend to refine and investigate in the near future are as follows.

The first issue, and arguably the most important one, is to refine the estimation of the distribution of message delays. In particular, it is important to model this distribution properly, or else the scale could become completely meaningless. In this implementation, the distribution of arrival intervals is assumed to be associated with the normal distribution as a first attempt because at least arrival intervals, except message losses, gather around their mean (see §4.6). Similarly, it is desirable to compute the confidence intervals associated with the estimated distribution.

5.5 Summary

The concept of the φ failure detectors has been presented and its implementation described. The behavior of this failure detector has been analyzed using transcontinental Internet communication over a period of one week. Finally, the behavior of the failure detector has been compared

[1] These two failure detectors were aimed at different problems, that they both solve admirably well.

with two important adaptive failure detectors; Chen's estimation [CTA02] and Bertier's dynamic estimation [BMS02].

By design, φ failure detectors can adapt equally well to changing network conditions, and the requirements of any number of concurrently running applications. As far as is known, this is currently the only failure detector that addresses this problem, and provides the flexibility required for implementing a truly generic failure detection service. In particular, the two other failure detectors studied in this dissertation completely fail to address that problem.

In addition to interesting observations about transcontinental network communication, the experimental results show that the failure detector behaves reasonably well if parameters are well-tuned. In particular, it can be seen that the impact of the window size is significant. Comparisons with other failure detectors show that the performance of Chen's and Bertier's estimations are slightly better than that of the φ failure detector. It also however provides nearly limitless flexibility. Nevertheless, it is believed that there is still room for improvement. Entries in the sliding window have exactly same weight, but they should have different weights because the importance of data gradually reduces. Putting different weights for each entry in the sliding window is being considered. The question of which scale is better than the normal distribution for fitting the network condition is also being investigated. The performance of the three failure detectors remain comparable, and hence it can be concluded conclude that all three approaches are equally realistic with respect to their quality of service.

As has been observed, message losses account for a very significant number of wrong suspicions. In particular, with a well-tuned failure detector,[2] nearly all wrong suspicions come as the result of message losses and temporary network partitions. This means that (1) there is not much point in fine-tuning the failure detectors beyond a certain point, and (2) the failure detectors cannot meet the requirements of applications with a need for very high-accuracy. The only way to address this problem is to somehow reduce the effect that message losses have on wrong suspicions. It is believed that it is an important issue because it limits the flexibility of the failure detector.

The important thing is to consider message loss on fair-lossy channels. In other words, the failure detector must consider the possibility that heartbeat messages are lost. In fact, the present implementation of the φ failure detector works on fair-lossy channels but any message loss is handled as wrong suspicions.

Also, in this chapter, the case has not bee discussed when several consecutive heartbeats are expected from process p but have not been received. The value φ_p should be computed in

[2] Note that this observation is true for all three failure detector implementations studied in this dissertation, since they are based on similar assumptions.

a way that each missing heartbeat contributes to increasing the confidence that p has crashed. This would effectively ensure that every process that has crashed is eventually suspected, for any finite threshold.

A core approach for implementing an flexible failure detector, which does not take message losses into account has been shown in this chapter. The next chapter presents an approach for a “message loss resilient” failure detector, which can provide very conservative failure detection.

Chapter 6

κ-Failure Detector

In the previous chapter, a failure detection scheme based on a similar approach, called the φ failure detector was outlines. Unfortunately, current adaptive failure detectors either ignore the problem (e.g., [BMS02, HDK03b]) or are based on the assumption that the loss of consecutive messages are uncorrelated [CTA02]. In contrast, experiments have shown that message losses are strongly correlated and tend to occur in bursts of various length, which is consistent with observations made by Keidar et al. [KSMD02], as well as many people in the networking research community.

In this chapter, the concept of the κ failure detector as a way to address the problems mentioned above, is presented. Rather than a failure detector for each process, the concept should instead be seen as a way of extending an existing failure detection scheme in order to address the requirements of conservative failure detection. The κ failure detector outputs a value which is calculated as the sum of contributions from expected heartbeats. Actions triggered by high thresholds will be less sensitive to long bursts of message losses and/or temporary network partitions. From a different standpoint, it can be seen that the κ failure detector is a framework for implementing accrual failure detectors allowing conservative settings because, any adaptive failure detection mechanism (e.g., [CTA02, BMS02, SM01]) could be a contributing function for the κ failure detector. In this chapter, the κ-failure detector is described and some important properties are proved. An implementation based on the φ failure detector and its behavior over a transcontinental network connection evaluated. The experiments show that the κ failure detector can be tuned in the conservative range to avoid wrong suspicions.

6.1 κ Failure Detectors

In this section, describe the κ failure detector is described as a generic concept rather than as a specific implementation (a possible implementation is described in Sect. 6.2). The basic idea is that each missed heartbeat contributes to raising the level of suspicion of the failure detector. First, the contribution of the heartbeat is defined more precisely. Then, the mechanism is described by which the the value of the κ output by the failure detector is determined. Finally, the completeness is proved[1]

6.1.1 Heartbeat Contributions (definition)

The κ failure detector requires the existence of a time function to represent the evolution of the confidence that a given heartbeat will not be received in the future, either because it was lost or because the sending process has crashed. The function returns a value between 0 and 1, where the latter means total confidence and the former means no confidence at all. Initially, the value is zero and remains so until some time when the heartbeat begins to be expected. Then, the value increases and ultimately converges to one. This function is considered here as a black box. A possible implementation is proposed in Section 6.2.

More precisely, the contribution function is defined as follows:

Definition 9 (Contribution function). *The contribution function is a function of time which satisfies the properties below.*

$$c : \mathbb{R} \longrightarrow [0;1]$$

- c *is monotonic.*
- $c(0) = 0$
- $\lim_{t \to +\infty} c(t) = 1$

The function is used for each heartbeat to determine the evolution of the confidence with respect to that heartbeat. Notice that the function can be based on parameters that change dynamically, when new heartbeats are received. It can be considered that there is a time, called the starting time, before which the heartbeat is not expected.

[1] The accuracy of the failure detector is not proved since essentially the model assumed in this dissertation does not allow accuracy in the formal sense (deterministically) to be ensured, although it does so in a more pragmatic way (i.e., stochastically). Nevertheless, the QoS parameters describing the accuracy of the failure detector experimentally are evaluated in Section 6.3.2.

Definition 10 (Starting time). *Let H^i denote the i-th heartbeat (with $i = 1, 2, \cdots$). Its starting time T^i_{st} has the following property.*

- $\forall j \left(i < j \Leftrightarrow T^i_{st} < T^j_{st} \right)$

It follows that the contribution of some heartbeat H^i can be computed simply by (6.1).

$$c^i(t) = c(t - T^i_{st}) \tag{6.1}$$

In practice, the nature of the contribution function is important for aggressive failure detectors but less so for conservative ones. This is because the contribution function defines the meaning of the fractional part of the value output by κ.

In reality, one can think of various possible contribution functions. In this chapter (Sect. 6.2), an implementation based on the φ failure detector as described in a recent technical report [HDK03a], is proposed. Alternatively, the contribution of a heartbeat could be defined as a step function, thus matching single-heartbeat failure detectors based on a conventional *"trust-or-suspect"* scheme, such as Bertier's failure detector [BMS02].

6.1.2 Computing the κ Function

The value output of the failure detector is given by a function of time $\kappa(t)$, obtained by summing the contributions of all expected heartbeats with rank higher than the most recent heartbeat. This is expressed by the function $\kappa(t)$ defined below.

Definition 11 (κ). *Let k be the rank of the most recent heartbeat.*

$$\begin{aligned} \kappa \quad : \quad & \mathbb{R} \longrightarrow \mathbb{R}^+ \\ & \kappa(t) = \sum_{i=k+1}^{\infty} c(t - T^i_{st}) \end{aligned} \tag{6.2}$$

Notice that can also be assumed that, if process p is correct, then p sends an infinite number of heartbeat messages.

6.1.3 Important Properties

All properties mentioned below are based on the assumption that, when process q monitors process p, q suspects p based on a positive constant[2] threshold K.

Lemma 1. *Let p and q be two processes, where q is correct and monitors p. For any finite threshold K, if p crashes, then eventually $\kappa(t) > K$ and this is permanent.*

Proof. If p crashes, there is a time after which q never receives any heartbeat from p. Let H^k be the most recent heartbeat received from process p.

Let $c(t)$ denote the contribution function after that time. Since no more heartbeat messages are received, the function does not change.

By definition, the contribution function is monotonic and converges to one. This means that there is a time after which the contribution is always greater than say $\frac{1}{2}$. Let 蕊 s call this time $T_{\frac{1}{2}}$.

Given a finite threshold K, the lemma is proven by showing that there exists a time $\overline{T}$ for which $\kappa(\overline{T}) > K$. Choose $\overline{T}$ as follows: $\overline{T} = T_{st}^{k+2\lceil K\rceil+1} + T_{\frac{1}{2}}$. We can now start from $\kappa(\overline{T})$ and develop.

$$
\begin{aligned}
\kappa(\overline{T}) &= \kappa(T_{st}^{k+2\lceil K\rceil+1} + T_{\frac{1}{2}}) \\
&\quad \textit{Eq. (6.2):} \\
&= c(\overline{T} - T_{st}^{k+1}) + \cdots + c(\overline{T} - T_{st}^{k+2\lceil K\rceil+1})\{...\} \\
&\quad \textit{Def. 9:} \\
&\geq c(T_{\frac{1}{2}}) + \cdots + c(T_{\frac{1}{2}}) \\
&\geq (2\lceil K\rceil + 1) \cdot \tfrac{1}{2} \\
&> 2\lceil K\rceil \cdot \tfrac{1}{2} = \lceil K\rceil \geq K
\end{aligned}
\tag{6.3}
$$

This proves the first part of the lemma. It is now easy to show the second part. Indeed, being the sum of monotonically increasing functions, $\kappa(t)$ is itself monotonically increasing. It follows that, for any time $t' > \overline{T}$, $\kappa(t') \geq \kappa(\overline{T}) > K$. □

Theorem 1 (Strong completeness). *A crashed process is eventually suspected by all correct processes.*

Proof. This assumes that all processes monitor each other. Let p be some crashed process, and q be some correct process that monitors p. By Lemma 1, q eventually suspects p (i.e., $\kappa(t) > K$). Since both p and q have been chosen arbitrarily, this completes the proof. □

[2] The assumption that K is constant is made in order to keep the proofs simple. This need not be the case in practice.

6.2 Implementation

This section describes a possible implementation of the κ failure detector, based on the φ failure detection strategy. In fact, the κ failure detector is a framework for implementing the accrual failure detector. Thus, the κ failure detector allows any strategy (e.g., [CTA02, BMS02, SM01], etc.) as a contribution function $c(t)$. In the other words, a contribution function is a plug-in component. In this dissertation, a special focus is on using the failure detection strategy of the φ failure detector as a contribution function. This implementation has been used to run the experiments presented in Section 6.3.

6.2.1 Description

In implementing the κ-failure detector, the contribution function of the heartbeats is computed from the arrival intervals between two consecutive heartbeats. Namely, the estimation made for the φ failure detector [HDK03a]is used, and the arrival interval between two consecutive heartbeats is considered to be a random variable that is approximated by a normal distribution. The failure detector module is divided into two main tasks, namely, the sampling of heartbeat arrivals, and the computation of the current value for $\kappa(t)$. The two tasks of the failure detector are now described.

Task 1: Sampling The sampling task is executed whenever a new heartbeat is received, and gathers information about recent heartbeat arrivals. In particular, the task maintains a sliding window of past arrivals with parameter ws as the window size. Upon receiving a new heartbeat, the task reads the process clock and stores the heartbeat rank and arrival time in the sliding window (thus discarding the oldest heartbeat if necessary). In fact, the task is almost the same as the computation of φ described in Sect. 5.2.1. The difference is only that time A_i between the receipt of two heartbeats is processed by $\frac{A_i}{j+1}$ if j $(0 < j)$ messages are lost during A_i.

The task keeps track of four values that are of particular importance for estimating κ: the mean μ and variance σ^2 of inter-arrival times, as well as the rank k and arrival time A_k, where k is the highest rank among all received heartbeats. For the first two values, this is done by simply keeping track of the sum and the sum of squares of inter-arrival times.

Task 2: Computing κ This task is invoked when some application process queries the failure detector. The task reads the process clock and computes the value for the function $\kappa(t)$. This is done by approximating the contribution function of expected heartbeats and summing each of them.

Given the values obtained by the first task (i.e., μ, σ^2, k, A_k), the contribution function $c(t)$ is approximated from the cumulative normal distribution function.

$$c(t) = \begin{cases} \frac{1}{\sigma\sqrt{2\pi}} \int\limits_{-\infty}^{t} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx & \text{if } t > 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.4}$$

It follows that, for some heartbeat H^i, where $i > k$, the contribution is computed by the function $c^i(t)$ shown below. Also, in Eq. (6.4), the contribution of heartbeat H^i starts one heartbeat interval before its estimated arrival. Hence, the starting time T^i_{st} of heartbeat H^i is given by the following equation.

$$\begin{aligned} &T^i_{st} = A_k + (i - k - 1)\mu \\ &c^i(t) = c(t - T^i_{st}) \end{aligned} \tag{6.5}$$

Computing the current value of function $\kappa(t)$ is done by summing the contribution of all heartbeats H^i for which the starting time has past (i.e., where $t > T^i_{st}$).

6.3 Experiments

Experiments have been performed to analyze the behavior of the implementation of the κ failure detector using the failure detection strategy of the φ failure detector over a transcontinental network connection for a total duration of three weeks. The section is begun by describing the experimental setup, then the experimental results are shown, and the section finished with a discussion.

The main goal of the experiments was to observe the ability of the implementation to tolerate message losses in a tunable way. For this reason, the κ failure detector implementation was compared with other adaptive failure detectors, and the effect of certain parameters on a real-world environment observed.

6.3.1 Experiments Overview

The experiment was carried out in two phases. First, heartbeat arrivals were recorded using the experimental setup described above. Then, simulation was used to replay the recorded traces with different failure detector implementations. As a result, the failure detectors were compared based on *exactly* the same scenarios, thus ensuring fairness of the comparisons.

Phase 1: Recording heartbeat arrivals For the first phase, a program was run on the EPFL machine to generate heartbeat messages. Another program was run on the JAIST machine to record the arrival time of each heartbeat and to log the information into a file. Neither machine failed during the experiment. The experiment was exactly the same as that described in §4.6. It lasted for one week, during which heartbeat messages were generated at a constant rate of one every 100 milliseconds. A total of $5,845,712$ heartbeat messages were generated of which $5,822,521$ were received.

As a final note, the CPU load average on the two machines was monitored during the whole period of the experimentation. It was observed that the load was nearly constant throughout, and was well below the capacities of the machines.

Phase 2: Executing failure detectors Using the trace file obtained during the first phase of the experiments, several executions involving the κ failure detector with various parameters were run. To provide a reference for comparison, the failure detectors of Chen et al. [CTA02], Bertier et al. [BMS02], the φ-failure detector and the κ-failure detector were all executed. In all experiments described in this chapter it was assumed that the window size of each failure detector was $1,000$. In particular, experiments were carried out according to the two scenarios described below (the corresponding results are discussed in Section 6.3.2).

Scenario 1 (Mistake rate λ_M and Average detection time vs. Threshold K). *The first scenario measures the mistake rate λ_M and the average detection time, obtained with the κ-failure detector. In particular, the evolution of the mistake rate when the threshold K that triggers suspicions increases, can be observed. The scenario was important for determining how the κ failure detector behaved with respect to conservative failure detection.*

Scenario 2 (Comparison with other failure detectors). *The κ failure detector was compared with the other failure detectors mentioned above. The goal of the comparison was to observe differences in performance among these failure detectors. .*

For a fair test, we compared the mistake rates λ_M and the average detection time T_D. Exactly same trace file as that in Chapter 5 was used. Parameters for each failure detector were set as widely as possible. In Bertier's failure detector, its settings giving in the literature were used [BMS02, BMS03]. The parameters were tuned as in the following Table 6.1.

Note that in the implementation of the φ failure detector, the computation is limited by the upper bound of Threshold Φ is 16.0.

In addition to setting, all failure detectors were set to use the same window size of $1,000$ samples for computing their estimation. In order to compare the failure detectors in their stable

Table 6.1: Parameter settings for each failure detector

	κ-FD	φ-FD	Chen's FD	Bertier's FD
Parameters	Threshold, K	Threshold, Φ	Safety margin, α	Dynamic safety margin
Range	$[0.1; 1200.0]$	$[0.5; 16.0]$	$[0.0; 120.0]$	$\gamma = 0.1, \beta = 1.0, \phi = 4.0$

states, all results obtained during the warmup period—i.e., the period before the window was full—were simply ignored.

6.3.2 Experimental results & discussions

This section presents the results obtained after running the experiments described in Sect. 6.3.1. The first scenario measured the behavior of the κ failure detector, whereas the latter scenario compared the κ and Chen failure detectors [CTA02].

Mistake rate λ_M and Average detection time vs. Threshold K (Scenario 1)

The mistake rate λ_M of the κ-failure detector were measured, when the threshold K_p varied (see Fig. 6.1).

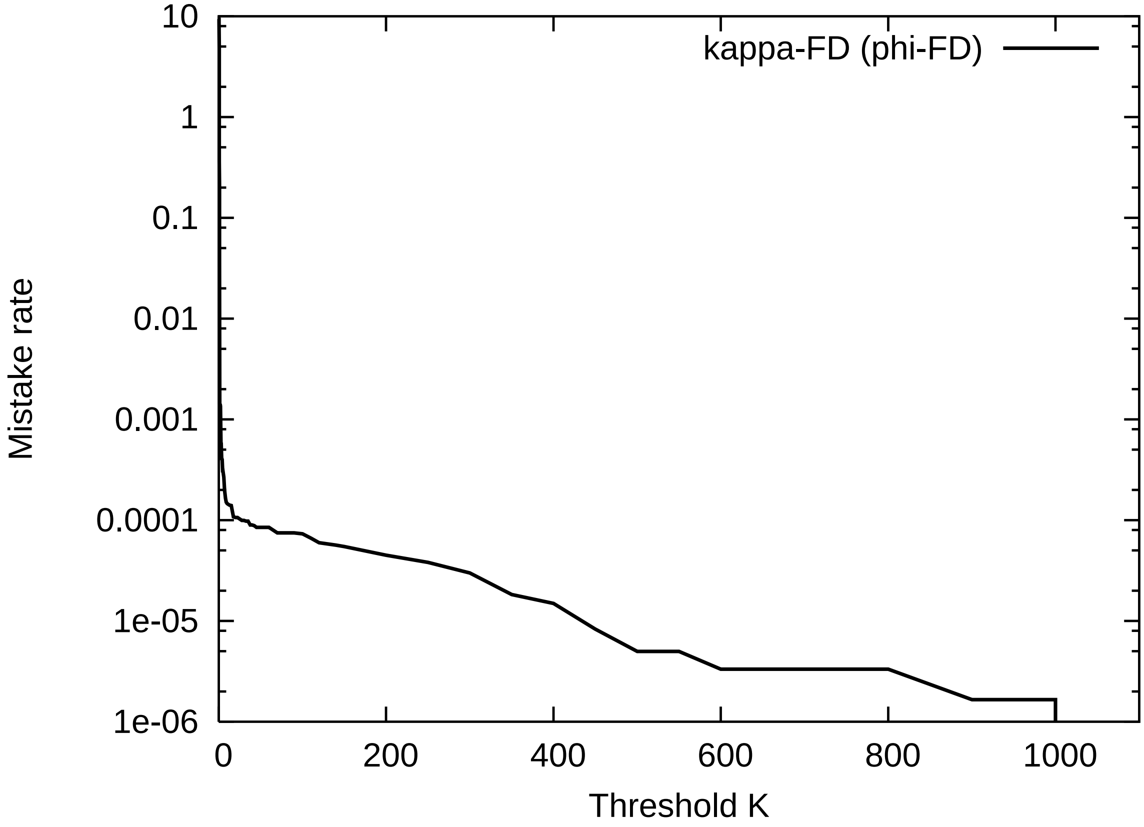

Figure 6.1: K_q vs. Mistake rate λ_M (y-axis is logarithmic scale)

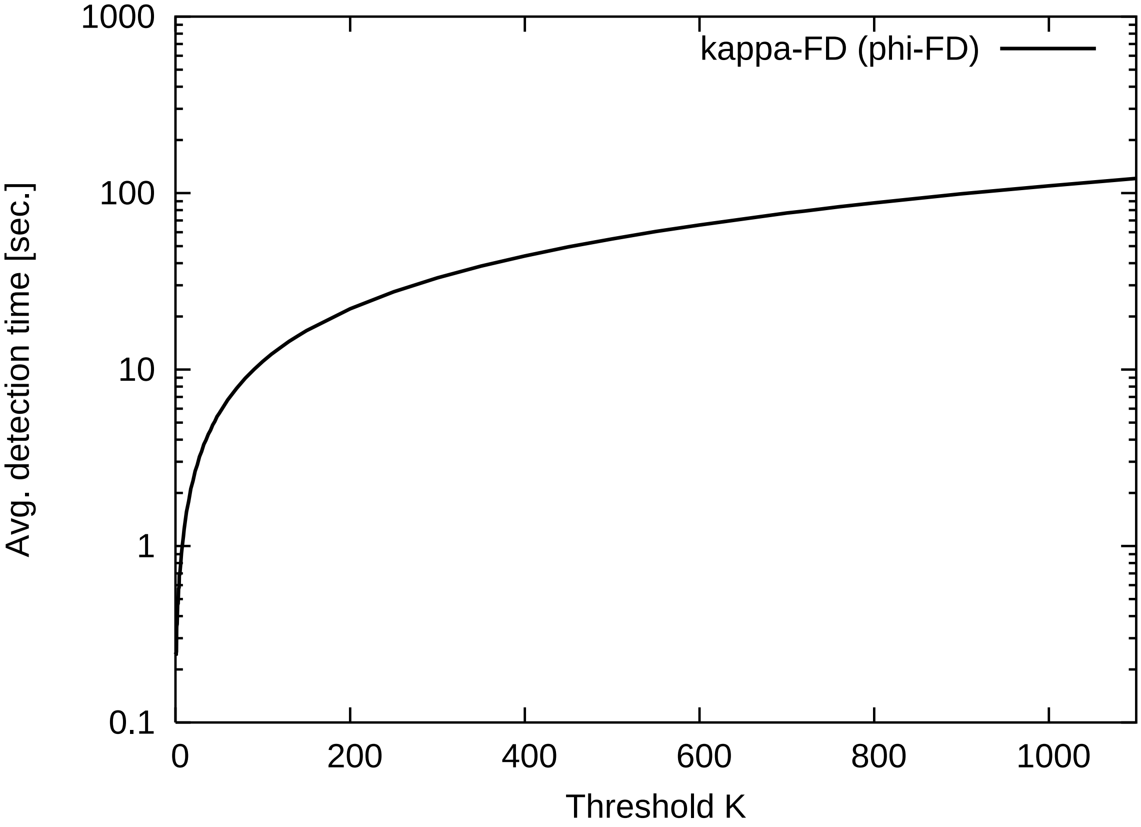

Figure 6.2: K_q vs. Average detection time (y-axis scale is logarithmic)

Figure 6.1 illustrates the fact that, as the threshold increased, fewer wrong suspicions were generated, until no suspicions were generated during the one week period of the experiment. It also shows that the curve suddenly sink to around $\lambda_M = 0.0001$ at the left side and then goes to the right side while gently eliminating wrong suspicions.

With higher thresholds, some lost messages no longer caused wrong suspicions, until $\overline{K_p} = 1100$, beyond which not a single wrong suspicion was generated, where $0.5 \leq K_p \leq 800.0$, and a single wrong suspicion was generated, where $900.0 \leq K_p \leq 1000.0$, during the whole duration of the experiment. Evidently, a longer experimentation period or a different environment would almost certainly yield different values for the threshold $\overline{K_p}$, and hence the value is not particularly important. What is important is simply that such a value exists, and that could be observed under real-world conditions.

The figure allows us to make some other observations. The mistake rate decreases gradually for threshold values $K_p \geq 2.5$. Also the κ failure detector generates less than 10 mistakes per a day in $K_p \geq 17.5$. This shows that fine-tuning the κ-failure detector is possible for conservative failure detection.

Figure 6.2 shows the increment of the average detection time with K_p. In fact, it gradually

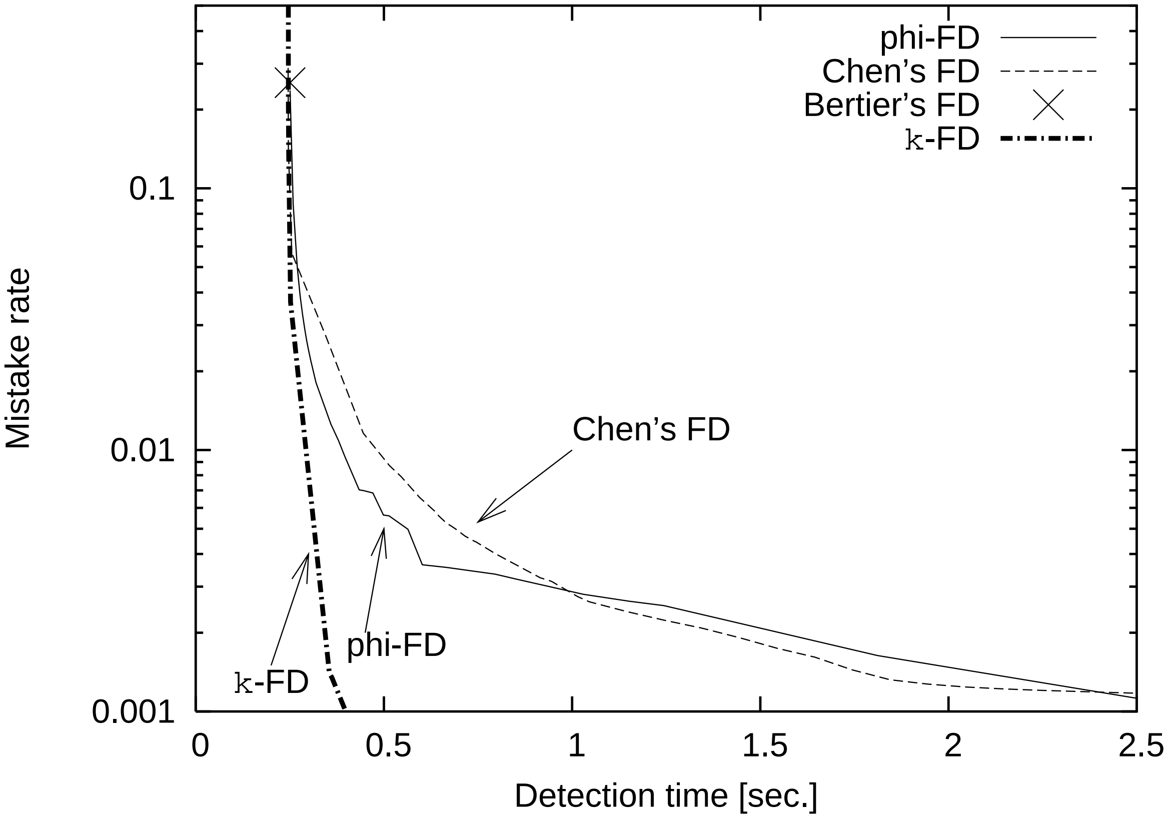

Figure 6.3: The result of the comparison highlighting the aggressive range.

increases according to the increment of K_p. $\overline{K_p}$ corresponding to the average detection time of 120.898 seconds. It can avoid recognizing the longest message loss ($1,094$ losses), whose duration is 113.321 seconds, as a wrong suspicion. If reasonably good quality of the failure detection is needed - such as 10 mistake per a day, the failure detector needs more than about 2.0 sec. (with $K_p \geq 17.5$) in average detection time in this environment.

Comparison with other failure detectors (Scenario 2)

The behavior of the κ failure detector compared with other adaptive failure detectors, Chen's FD [CTA02], Bertier's FD [BMS02] and the φ failure detector appeared in an earlier chapter.

The result in Figure 6.3 shows that this implementation of the κ failure detector has better performance than other failure detectors in the aggressive range. While, in the conservative range (see Fig. 6.4), Chen's failure detector is little better than the κ failure detector, with an average detection time of only about 85 seconds. However, the difference between them is negligible. Interestingly, both eliminate all wrong suspicions at an average detection time of 110 [sec.]. This corresponds to the maximum number of message losses ($1,094$). The φ and Bertier's failure detectors, do not have a conservative setting.

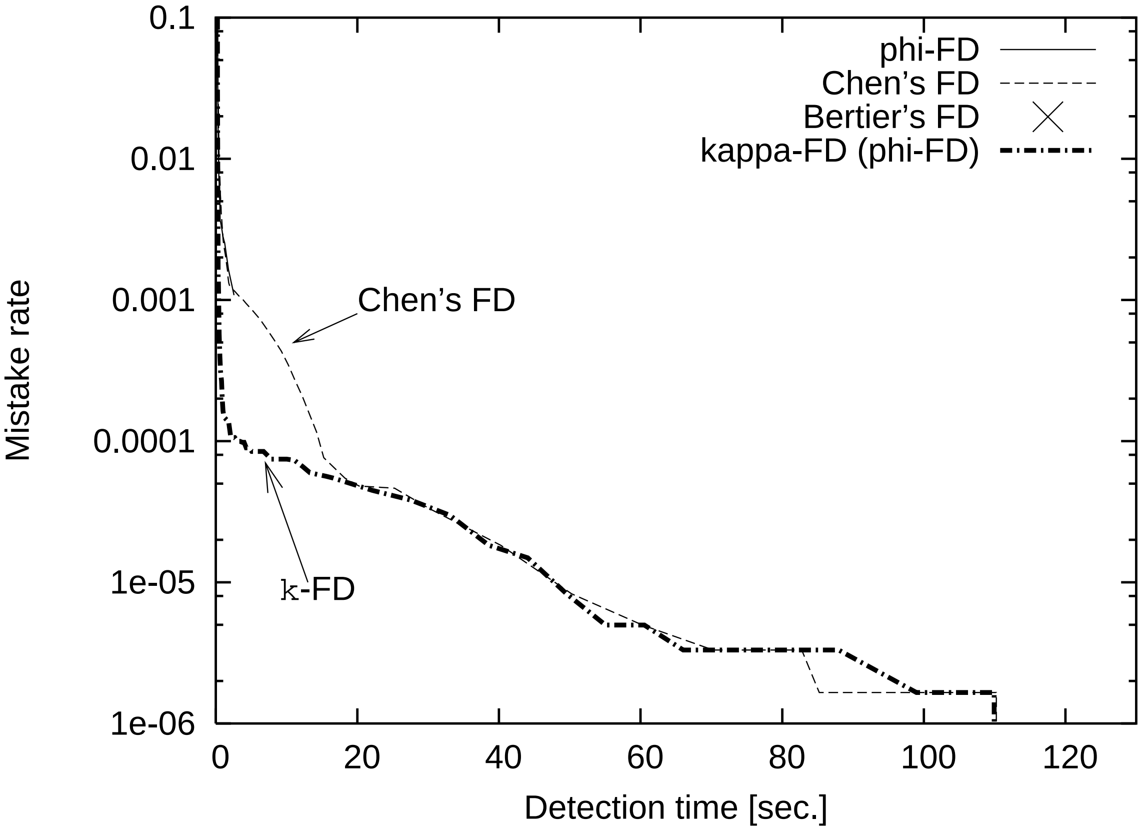

Figure 6.4: The result of the comparison highlighting the conservative range.

This implementation of the κ failure detector uses the failure detection mechanism of the φ failure detector as a contribution function. The difference is that the κ failure detector takes account of each of two consecutive heartbeats to compute the expected arrival time of the next heartbeat. If some of them are lost, it computes the virtual arrival time of each lost message which reflects the expected arrival time. While, in the φ failure detector, the time between the reception of two heartbeats simply contributes to the computation of the expected arrival time of the next heartbeat. In this case, the κ failure detector has a better result than the φ failure detector in accuracy and average detection time.

6.3.3 Discussion

The average detection time and the mistake rate λ_M in the κ-failure detector have been measured. In this measurement, the behavior of the failure detector was observed when the threshold K_p increased. It was found that at the threshold $K_p = 1100.0$ the κ-failure detector no longer had wrong suspicions.

To build a conservative failure detector, you should set a larger K_p for the failure detector. In fact, the failure detector approaches the failure detector of class $\mathcal{P}$ if K_p are increased. However,

K_p and λ_M are trade-offs. The κ failure detector with $K_p \geq 1100.0$ could be a failure detector of class $\mathcal{P}$ in the experimentation over the internet. Moreover, with $K_p \geq 450.0$ can have less than 1 wrong suspicion per a day. It takes a long time to detect failures ($K_p = 450.0$ corresponds to an average detection time of 49.400 sec.) but such a threshold can be set if the fail over constant is really serious for the system or the application.

In the implementation mentioned in Sect.6.2.1, the distribution of heartbeat arrival intervals which are random variables is assumed to be a normal. The heartbeat arrival intervals are crowded around the mean and the variance is about 0.0002 in the experimentation. The value of κ is a cumulative value among arrival intervals. The κ failure detector evaluates each interval between a pair of heartbeats. Thus, the scale is a reasonable choice. However, the distribution of heartbeat arrival intervals can be completely different in some other internet. The problem of which distribution is appropriate is still an open question and an interesting topic.

Then, the performance of the κ failure detector was compared with other adaptive failure detectors. The result of the κ failure detector was better than others in the aggressive range. In the conservative range, Chen's failure detector improved its performance, and the results of it and the κ failure detector become almost the same. Note that the κ failure detector also has the advantage of providing its service for many applications simultaneously because it is an instance of accrual failure detectors.

Bertier's failure detector is also interesting but, no α can be set, because α is computed automatically using the round-trip estimation algorithm [Jac88] and other parameters are not compatible to parameters in the κ failure detector. Also the failure detector is designed for local-area networks. For this reason, it does not work well in experiments over the Internet.

In the case of both adaptive failure detectors (Chen's FD and Bertier's FD), applications decide the interval for sending heartbeat messages. In contrast, applications had no choice on the sending interval in the implementation and experiments presented here. However, the sending time of heartbeats should be decided by the system or its requirements (e.g., traffic on heartbeats is required to be less than 1% of the bandwidth), otherwise some applications can paint the bandwidth out by its requirement. However, the impact on accuracy of failure detectors by changing emission time, is still interesting.

As we said, the κ failure detector is a framework for implementing accrual failure detectors. In the implementation presented here, the failure detection mechanism of the φ failure detector is used as a contribution function of the κ failure detector. On the other hand, other mechanisms (i.e., [CTA02, BMS02, SM01], etc.) can be used in the κ failure detector. It is not considered that this implementation, based on the φ failure detector, can always have a good result. The other failure detection mechanisms can be seen as contribution functions in the different environments. For instance, Bertier's paper [BMS02] shows that their failure detector had a better result than Chen's. In this environment, their failure detection mechanism can be used.

6.4 Summary

In this chapter, a novel approach to implementing a tunable conservative failure detection in distributed systems has been presented. The κ failure detector presented in this chapter addresses the problem of conservative failure detection by taking account of message losses and short-lived network partitions. In addition, the failure detector outputs information on a continuous scale rather than using the traditional "trust-or-suspect" model. This improves its flexibility as applications can trigger suspicions based on their own requirements, without interfering with each other.

The κ failure detector was described as a generic concept whereby a loss-intolerant detection strategy can be used as the basis for computing the contribution of a single heartbeat. Yet, the combination of contributions makes it possible to set a threshold so that consecutive message losses are tolerated.

The chapter describes an implementation of the κ failure detector, where the contribution of a heartbeat is based on the φ failure detector [HDK03b, HDK03a] described in earlier work. The resulting implementation is compared with the failure detector of Chen et al. [CTA02]. The results in this work show that the κ failure detector behaves as expected in the conservative range, since it can be set so that message losses do not trigger wrong suspicions. Also, when setting κ for aggressive failure detection, it was found that its performance was better than other failure detectors. In particular, it might be interesting to evaluate other contribution functions, but this is left for future work.

Chapter 7

Conclusion

7.1 Research Assessment

The goal of this work was to implement a failure detection service as a generic service for large-scale distributed systems. Several approaches to a failure detection service have been developed but none have yet provided such a service. It was considered that the first step of the work was to survey and classify existing failure detection techniques. Several problems were found. Specifically, no approach could adapt simultaneously to network conditions and application requirements. Hence it was decided that the problem of flexibility for both adaptations would be the target for this dissertation.

In this dissertation, the concept, the definitions and mechanisms of accrual failure detectors were first proposed. Then two instances of accrual failure detector, called the φ failure detector and the κ failure detector were considered. Both were implemented and their performance measured with some parameters. The measurements used the sequence of heartbeat messages taken over three weeks in an experiment between Japan and Switzerland. [1]. In fact, they are well fitted to the network conditions if the parameters are set appropriately. On the other hand, they can adapt to diverse application requirements, because they provide a degree of suspicion about corresponding process to applications. Thus, the approach of this dissertation can address the problem of flexibility. In the rest of the chapter, these issues were presented more concretely.

Taxonomy and survey of failure detectors. Currently, some papers have presented failure detection services in large-scale distributed systems. However, they lack some important features in the context of large-scale systems. Literature on failure detectors has been surveyed and problems pointed out. Each approach partially covered these problems but none covered them

[1] All experiments in the dissertation used the experimental results.

all. In fact, the problem of flexibility in failure detectors is not actually addressed. This work was very important as a starting point for implementing a generic failure detection service for large-scale distributed systems.

Accrual failure detectors. The notion of accrual failure detectors was developed. In short, accrual failure detectors can adapt to network conditions and the diverse requirements of applications running simultaneously, in an efficient manner. This principle has been applied in designing two instances of accrual failure detectors.

The first, called the φ failure detector, is aimed at *aggressive* failure detection and works as follows. Each failure detector module outputs a degree of suspicion. While, applications have their own threshold Φ_p for a process p and determine their own decision with respect to suspicion or trust, in p by comparison with Φ_p and the φ_p output from the module. For the experiments on the φ failure detector, an experiment on heartbeat messages over the Internet was carried out. The results show that the φ failure detector performed reasonably if its parameters were well-tuned. The performance of the φ failure detector was also of the same order as that of the other adaptive failure detectors. This approach was able to address the problem of flexibility.

Second, failure detector needs for the proper handling of message losses in a bursty network condition were considered, unlike other implementations which either model losses based on unrealistic assumptions, or simply ignore the problem entirely. Existing approaches, including the φ failure detector, cannot meet the needs of *conservative* failure detection. To address this, the κ failure detector was developed as an extension to the φ failure detector. The κ failure detector can handle message losses. The failure detector module outputs information on a continuous scale, which is the contribution of the heartbeat messages. Hence the κ failure detector can make conservative failure detection. This improves its flexibility as applications can trigger suspicions based on their requirements. The κfailure detector was implemented and its performance compared with that of the adaptive failure detector proposed by Chen *et al.* [CTA02]. The results of experiments presented here show that the κfailure detector behaves in a wide range, from aggressive to conservative. The performance of the κ failure detector is comparable to Chen 庁 s if the parameter κ in the failure detector is set for aggressive failure detection. If both have the conservative setting, the result is that the κ failure detector is comparable to Chen's. It was confirmed the κ failure detector does incur any extra cost compared with the others.

A peer connection with two failure detector modules was assumed when the accrual failure detectors, the φ failure detector and the κ failure detector were developed. They are core techniques for implementing a failure detection service. A failure detection service was then designed for a pragmatic large-scale system using these failure detectors. The design clarifies the role of accrual failure detectors in the service and shows that accrual failure detectors can be implemented as a part of the service.

7.2 Open Questions and Future Directions

Important techniques mentioned above have been discovered and concomitantly new interesting challenges found that are important for reaching the goal of the work. Such challenges are described in this section.

Combining accrual failure detectors and information propagation. Accrual failure detectors could be used as a mechanism between every two processes that interact directly. A threshold can then be set locally, and information on suspicions propagated as usual. There might be better ways of taking full advantage of the particularity of accrual failure detectors.

Adaptation for the network condition. The φ failure detector and the κ failure detector were both implemented. Both estimated the degree to which some process is suspected of having crashed, using the normal distribution as a first attempt because arrival intervals, except message losses, gather around the mean. In fact, the distribution of arrival intervals including message losses actually does not come close to a normal distribution. Accrual failure detectors compute the degree using some scale which should be implemented as a plug-in because the traffic pattern is different at least between working times and nights in the same environment. An interesting question is how much the scale influences the average mistake-rate.

Properties of accrual failure detectors In this dissertation, the completeness of the κ failure detector is discussed and proved. However, a proof of the accuracy property has not been presented. Intuitively, it can be proved that a failure detector of class $\diamond\mathcal{P}$ can be realized by accrual failure detectors in a partially synchronous system model. This issue would be interesting for improving algorithms for agreement problems.

Design and implementation of the failure detection service. There are lots of problems in implementing a failure detection service. For instance, programmers must be given the facility to program using the service (e.g., programming interface) and the service needs a mechanism for fault tolerance. Too many parameters allowing the failure detection to be set is not so big an advantage. Programmers worry which setting (e.g., the emission time of heartbeats) is better for the application. It is important to distinguish between useful parameters for programmers and others.

In this dissertation is was assumed that the failure detector module was implemented as a process. On the other hand, Wiesmann *et al.* discussed the idea of using standard interfaces (e.g., Simple Network Management Protocol (SNMP)) for implementing failure detectors [WDS03, Rei02, Mül04]. Failure detectors can be associated with switches and routers using SNMP for detecting failures, propagating information and estimating message transmission times. This means that failure detectors can detect link failures physically.

References

[ACT99] M. K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, June 1999.

[BCBT96] 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.

[BMS02] 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.

[BMS03] 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.

[BO83] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proc. of 2nd Symp. on Principles of Distributed Computing (PODC)*, pages 27–30, Montreal, Canada, August 1983.

[CBDS02] B. Charron-Bost, X. Défago, and A. Schiper. Broadcasting messages in fault-tolerant distributed systems: the benefit of handling input-triggered and output-triggered suspicions differently. In *Proc. of the 21st IEEE Int'l Symposium on Reliable Distributed Systems (SRDS-21)*, pages 244–249, Osaka, Japan, October 2002.

[CKV01] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):427–469, May 2001.

[CRV95] F. Cosquer, L. Rodrigues, and P. Verissimo. Using tailored failure suspectors to support distributed cooperative applications. In *Proc. 7th IASTED/ISMM Intl. Conf. on Parallel and Distributed Computing and Systems*, pages 352–356, Washington, D.C., USA, October 1995.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[CTA02] 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.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchrony needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, jan 1987.

[Déf00] X. Défago. *Agreement-related Problems: From Semi-Passive Replication to Totally Ordered Broadcast*. PhD thesis, EPFL, Lausanne, Switzerland, 2000.

[DFS99] X. Défago, P. Felber, and A. Schiper. Optimization techniques for replicating CORBA objects. In *Proc. of the 4th IEEE Int'l Workshop on Object-oriented Real-time Dependable Systems(WORDS'99)*, pages 2–8, Santa Barbara, CA, USA, January 1999.

[DGM02] 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.

[DLS88] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[DSS98] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proc. 17th IEEE Intl. Symp. on Reliable Distributed Systems (SRDS-17)*, pages 43–50, West Lafayette, IN, USA, October 1998.

[FDGO99] 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.

[Fet01] C. Fetzer. Enforcing perfect failure detector. In *Proc. 21st IEEE Intl. Conf. on Distributed Computing Systems (ICDCS-21)*, pages 350–357, Mesa, AZ, USA, April 2001.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid. *Intl. Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[FLP85] 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.

[Fri02] R. Friedman. Fuzzy group membership. In A. Schiper, A. A. Shvartsman, H. Weatherspoon, and B. Y. Zhao, editors, *Future Directions in Distributed Computing: Research and Posit ion Papers (FuDiCo 2002)*, volume 2584 of *LNCS*, pages 114–118, Bertinoro, Italy, June 2002. Springer-Verlag Heidelberg.

[FRT01] 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.

[GCG01] 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.

[GKG02] I. Gupta, A.-M. Kermarrec, and A. J. Ganesh. Efficient epidemic-style protocols for reliable and scalable multicast. In *21st Symposium on Reliable Distributed Systems (SRDS'02)*, pages 180–189, Osaka, Japan, October 2002.

[GKM03] 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, February 2003.

[GS01] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41, January 2001.

[HCK02] 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.

[HDK03a] N. Hayashibara, X. Défago, and T. Katayama. Implementation and performance analysis of the φ-failure detector. Research Report IS-RR-2003-013, Japan Adv. Inst. of Sci. and Tech., Ishikawa, Japan, September 2003.

[HDK03b] 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*, pages 22–27, Sorrento, Italy, October 2003.

[Jac88] V. Jacobson. Congestion avoidance and control. In *Proc. of ACM SIGCOMM'88*, Stanford, CA, USA, August 1988.

[KMG03] A.-M. Kermarrec, L. Massoulié, and A. J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Trans. on Parallel and Distributed Systems*, 14(3):248–258, 2003.

[KSMD02] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe: A group membership service for WANs. *ACM Transactions on Computer Systems*, 20(3):1–48, August 2002.

[MMR03] A. Mostefaoui, E. Mourgaya, and M. Raynal. Asynchronous implementation of failure detectors. In *Proc. Intl. Conf. on Dependable Systems and Networks (DSN'03)*, pages 351–360, San Francisco, California, USA, June 2003.

[Mül04] M. Müller. Performance evaluation of a fialure detector using snmp, February 2004. Semester project, EPFL, Switzerland.

[Rei02] F. Reichenbach. Service snmp de détection de faute pour des systèm répartis. Master's thesis, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, February 2002. written in French.

[SDS01] 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, December 2001.

[SFK$^+$98] 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.

[SM95] Laura S. Sabel and Keith Marzullo. Election vs. consensus in asynchronous systems. Technical Report TR95-1488, 17, 1995.

[SM01] 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.

[USS03] P. Urbán, I. Schnayderman, and A. Schiper. Comparison of failure detectors and group membership: Performa nce study of two atomic broadcast algorithms. In *Proc. Intl. Conf. on Dependable Systems and Networks (DNS'03)*, pages 645–654, San Francisco, CA, USA, June 2003.

[vRMH98] 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.

[WDS03] M. Wiesmann, X. Défago, and A. Schiper. Group communication based on standard interfaces. In *the 2nd IEEE Intl. Symposium on Network Computing and Applications (NCA-03)*, pages 140–147, Cambridge, MA, USA, April 2003.

Appendix A

Implementations of Accrual Failure Detectors

A.1 φ-Failure Detector

In this section, we describe an implementation of the φ failure detector. There are two main tasks in the failure detector. Task 1 gathers data and calculates values, such as mean μ and variance δ of sampled data, which are needed to compute φ. Task 2 compute φ using values calculated in the previous task. We assume that heartbeat arrival times follow the normal distribution. Thus, we define functions for computing φ as follows:

$$P_{acc}(t) \overset{\text{def}}{=} \frac{1}{\sigma\sqrt{2\pi}} \int\limits_{-\infty}^{t} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx$$

$$\varphi(t) \overset{\text{def}}{=} -log_{10}(1 - P_{acc}(t))$$

A.2 κ-Failure Detector

We describe the implementation details of the κ failure detector shown in Chapter 6. Our implementation of the κ failure detector composes the φ failure detector. Thus, the contribution function $c^i(t)$, which computes κ in the function κ(t), is defined as follows:

$$c^i(t) \overset{\text{def}}{=} \varphi(t)$$

Therefore, the κ failure detector implementation is similar to the φ failure detector imple-

Algorithm 1 The φ-Failure Detector

```
Initialization:
    $s_p := -1$                                   {keep the largest sequence number}
    $WS := a\ constant$                           {the window size is a constant}
    $LA_p := 0$                                   {the arrival time in the previous receipt}
    $\Delta_{H_p^i} := 0$                         {inter-arrival time}
    $sum_p := sqr_p := 0$                         summation and std. deviation of $\Delta_{H_p^i}$

Task 1: {sampling data}
    **upon** receive heartbeat $H_p^i$
        **if** $i > s_p$ **then**
            $\Delta_{H_p^i} := A_p^i - LA_p$
            $LA_p := A_p^i$
            $s_p := i$
            $sum_p := \sum_{j=i-(WS-1)}^{i} \Delta_{H_p^j}$
            $sqr_p := \sum_{j=i-(WS-1)}^{i} (\Delta_{H_p^j})^2$
            $\sigma_p^2 := \sqrt{\frac{sqr_p}{WS} - \left(\frac{sum_p}{WS}\right)^2}$
            $\mu_p := \frac{sum_p}{WS}$
        **end if**

Task 2: {calculation for $\varphi$}
    **upon** receive request from $p$ about $q$ at time $t$
        $\varphi_p$ := $\varphi(t)$
        **return** $\varphi_p$
```

mentation. Specially, the Task 1 is exactly same between them. So we skip it in the description of the κ failure detector implementation (see Algorithm 2).

Algorithm 2 κ-Failure Detector

```
Initialization:
  $s_p := -1$                    {Keep the largest sequence number}
  $ws := a\ constant$            {Window size is constant}
  $LA_p := 0$                    {Arrival time of the previous receipt}
  $sum_p := sqr_p := 0$
  $\Delta_{H_p^i} := 0$          {Arrival interval}

Task 1: {Sampling data}
  upon Receive heartbeat $H_p^i$

Task 2: {calculation for $\kappa$}
  upon Receive request from $p$ about $q$ at time $t$
    $\kappa_p := \kappa(t)$      {$k$ is computed by th e contribution function}
    return $\kappa_p$
```

Appendix B

Adaptive Failure Detectors

Important equations and informal descriptions of adaptive failure detectors used in the dissertation were appeared in Section 3.2.4. In this chapter, we introduce implementations of these failure detectors as pseudocode notations.

B.1 Chen's failure detector

Chen *et al.* have proposed several algorithms of failure detectors for synchronized and unsynchronized system [CTA02]. Now, we assume unsynchronized distributed systems, thus, we choose the failure detector algorithm NFD-E and implemented it (see Algorithm 3). In this failure detector, a sending interval δ_i and a safety margin α are computed based on a QoS requirement suite (e.g., upper bound of T_D, lower bound of T_{MR}, etc.) before it starts to monitor. It means that α is a constant value during the execution of the failure detector.

B.2 Bertier's failure detector

Bertier's failure detector [BMS02] combines Chen's failure detector and Jacobson RTT estimation algorithm [Jac88]. Therefore, we skip to describe same lines as Chen's ones. In fact, Algorithm 4 corresponds to line 6 to 13 in Algorithm 3.

In this failure detector, a safety margin α is dynamically adjusted by Jacobson's algorithm. Thus, the failure detector only allows to set parameters of Jacobson's algorithm for estimating an appropriate α. There are three parameters allowed to set in Bertier's failure detector: γ represents the importance of the new measure with respect to the previous ones, β and ϕ allow to ponder the variance of arrival times.

Algorithm 3 Chen's failure detector with NFD-E

```
Process $p$: using $p$'s local clock
  $\forall i \geq 1$, at time $t\Delta_i$, send heartbeat $m_i$ to $q$;
Process $q$: using $q$'s local clock
Initialization:
  $\tau_0$ = 0;
  $l = -1$; $\{l$ keeps the largest sequence number in all messages $q$ has received so far$\}$
  upon $\tau_{l+1}$ = the current time; $\{$if the current time reaches $\tau_{l+1}$, then none of the messages received is still fresh$\}$
  $output \leftarrow suspect$; $\{$suspect $p$ since no message received is still fresh at this time$\}$
  upon receive message $m_j$ at time $t$:
  **if** $j > l$ **then**
    $l \leftarrow j$;
    $\tau_{l+1} \leftarrow EA_{l+1} + \alpha$; $\{$set the next freshness point $\tau_{l+1}$ using the expected arrival time at $m_{l+1}\}$
    **if** $t < \tau_{l+1}$ **then**
      $output \leftarrow trust$; $\{$trust $p$ since $m_l$ is still fresh at time $t\}$
    **end if**
  **end if**
```

Algorithm 4 Bertier's failure detector

```
  upon receive message $m_j$ at time $t$:
  **if** $j > l$ **then**
    $l \leftarrow j$;
    $error_l \leftarrow delay_l + \gamma error_l$;
    $var_{l+1} \leftarrow var_l + \gamma(|error_l| - var_l)$;
    $\alpha_{l+1} \leftarrow \beta delay_{l+1} + \phi var_{l+1}$;
    $\tau_{l+1} \leftarrow EA_{l+1} + \alpha_{l+1}$; $\{$set the next freshness point $\tau_{l+1}$ using the expected arrival time at $m_{l+1}\}$
    **if** $t < \tau_{l+1}$ **then**
      $output \leftarrow trust$; $\{$trust $p$ since $m_l$ is still fresh at time $t\}$
    **end if**
  **end if**
```

Publications

[1] N. Hayashibara, A. Cherif and T. Katayama, “Failure Detectors for Large-Scale Distributed Systems”, In Proc. of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS-21), the Int’l Workshop on Self-Repairing and Self-Configurable Distributed Systems (RCDS’2002), pp.404-409, Osaka, Japan, Oct., 2002

[2] N. Hayashibara, P. Urbán, A. Schiper and T. Katayama, “Performance Comparison Between the Paxos and Chandra-Toueg Consensus Algorithms”, In Proc. of the 2002 International Arab Conference on Information Technology (ACIT’2002), vol.1 pp.526-533, Doha, Qatar, Dec., 2002,

[3] X. Défago, N. Hayashibara and T. Katayama, “On the Design of a Failure Detection Service for Large-Scale Distributed Systems”, In Proc. of Int’l Symposium on Towards Peta-Bit Ultra-Networks(ISBN4-9900330-3-5), pp.88-95, Sep., 2003

[4] N. Hayashibara, X. Défago and T. Katayama, “Two-ways Adaptive Failure Detection with the φ-Failure Detector”, In Proc. of the Workshop on Adaptive Distributed Systems (WADiS03). (In conjunction with the 17th International Symposium on Distributed Computing (DISC 2003)), pp.22-27, Sorrento, Italy, Oct., 2003

[5] N. Hayashibara, X. Défago, R. Yared and T. Katayama, “The φ Accrual Failure Detector”, In Proc. of the 23rd IEEE Symposium on Reliable Distributed Systems (SRDS-23), Florianópolis, Brazil, Oct., 2004, To appear

[6] P. Urbán, N. Hayashibara, A. Schiper and T. Katayama, “Performance Comparison of a Rotating Coordinator and a Leader based Consensus Algorithm”, In Proc. of the 23rd IEEE Symposium on Reliable Distributed Systems (SRDS-23), Florianópolis, Brazil, Oct., 2004, To appear

[7] X. Défago, P. Urbán, N. Hayashibara and T. Katayama, “On Accrual Failure Detectors”, Submitted to the 8th International Conference on Principles of Distributed Systems, Grenoble, France, Dec., 2004