MW-FD, A Failure Detector Algorithm with QoS, and an Analysis Towards Failure Detection as a Service

Alejandro Zlatko Tomsic

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Examination Committee

Chairperson: Prof. Luís Eduardo Teixeira Rodrigues
Supervisor: Prof. João Coelho Garcia
Member of the Committee: Prof. Johan Montelius

September 2013

Acknowledgements

I would like to thank my supervisor, João Garcia, for his constant help, support, guidance and understanding during this work and the past two years.

I would also like to express my gratitude to my supervisor at LIP6, Pierre Sens, and my team of co-workers, Julien Sopena and Luciana Arantes, for helping me pursuing my goals and guiding me through the process of this work, for motivating and encouraging me.

Finally, I would like to express my gratitude to my family, the EMDC colleagues, my friends and my girlfriend, who have been my strongest support.

Lisbon, September 2013

Alejandro Zlatko Tomsic

European Master in Distributed Computing, EMDC

This thesis is part of the curricula of the European Master in Distributed Computing (EMDC), a joint program among Royal Institute of Technology, Sweden (KTH), Universitat Politecnica de Catalunya, Spain (UPC), and Instituto Superior Técnico, Portugal (IST) supported by the European Community via the Erasmus Mundus program.

My track in this program has been as follows:

First and second semester of studies: IST

Third semester of studies: KTH

Fourth semester of studies: IST (Internship at LIP6 Paris)

To my parents.

Resumo

Hoje em dia, os sistemas distribuídos fornecem recursos a um grande número de aplicações. São desenhados para fornecer serviços fiáveis e contínuos independentemente da falha de alguns dos seus componentes. Porém, estes sistemas estão sujeitos a uma panóplia de falhas e, por isso, a detecção de falhas tem um papel central no desenho desses sistemas. Muitas aplicações têm restricções temporais e requerem detectores de falhas que incluam qualidade de serviço (QoS) com garantias temporais quantitativas. Habitualmente, as aplicações incluem implementações ad-hoc de detectores de falhas (FD). No entanto, não existem métodos sistemáticos de fornecer um serviço de detecção de falhas a um único computador com diferentes aplicações com requisitos variados. Neste trabalho, introduzimos o *Multiple Windows Failure Detector* (MW-FD). Este detector tem uma QoS melhorada em relação aos algoritmos existentes. Em particular, o MW-FD reduz o número de erros (falsas detecções) por unidade de tempo. Também analisámos a ideia de ter múltiplas aplicações ou máquinas virtuais, com diferentes requisitos em termos de QoS da detecção de falhas, executando-se numa única máquina usando um único detector de falhas como um serviço partilhado. Cada aplicação especifica os seus requisitos em termos de velocidade e precisão com que o FD detecta as falhas. O módulo FD pretende dar a cada aplicação a ilusão de um FD dedicado que satisfaz as suas necessidades particulares de QoS ao mesmo tempo que minimiza o número de mensagens trocadas na rede. Os resultados obtidos mostram que há melhorias de QoS sempre que se utiliza o nosso algoritmo, variando um pouco consoante as condições de rede e condições de teste. Mais, analisámos como, quando usando um serviço partilhado de detecção de falhas, os requisitos de QoS das aplicações devem ser adaptados de modo a coexistirem, mostrando que as aplicações com requisitos de QoS mais fracos beneficiam daquelas com requisitos mais fortes. A carga total imposta à rede é reduzida com o uso de um serviço partilhado comparado com o uso de um detector de falhas por aplicação.

Abstract

Distributed systems provide resources to a large number of applications today. They are designed to provide reliable and continuous services despite the failures of some of their components. Nevertheless, they are subject to a wide variety of failures and, therefore, failure detection plays a central role in the engineering of such dependable systems. Furthermore, many applications have timing constraints and require failure detectors that provide quality of service (QoS) with some quantitative timeliness guarantees. Moreover, applications usually implement their own ad-hoc failure detection modules. However, there is no systematic way of providing a failure detection service to a single host in the face of different application requirements.

In this work, we introduce the Multiple Windows Failure Detector (MW-FD). This failure detector presents an improved the QoS when compared to existing FD algorithms. Namely, it reduces the number of mistakes (false detections) made per unit of time. We also analyse the idea of multiple applications or virtual machines, with different QoS requirements in terms of failure detection, running on a single host using a single FD as a shared service. Each application specifies its own needs in terms of speed and accuracy at which the FD detects crashes. The FD module is meant to give, to each application, the ilusion of a dedicated FD that satisfies its particular needs on QoS while minimizing the number of messages exchanged in the network.

The obtained results show that there is always an improvement in terms of QoS when using our proposed algorithm, which varies in magnitude according to network conditions and test environments. Additionally, we have analysed how, when using a shared failure detection service, the required QoS of applications should be modified in order to coexist, showing that applications with weaker QoS requirements benefit from the ones with stronger ones by obtaining an improved QoS. Furthermore, the overall load imposed on the network is reduced when using the shared service, compared to the case of utilising one failure detector per application.

Keywords

Palavras Chave

Detectores de Falhas

Qualidade de Serviço

Tolerância a Falhas

Algoritmos Distribuídos

Fiabilidade

Estabilidade

Keywords

Failure Detectors

Quality of Service

Fault Tolerance

Distributed Algorithms

Reliability

Quiescence

Contents

List of Figures

List of Tables

Acronyms

VM Virtual Machine

FD Failure Detector

ED Exponential Distribution

MW Multiple Windows

NTP Network Time Protocol

DNS Domain Name System

MW Multiple Windows

QoS Quality of Service

1 Introduction

Distributed systems are being increasingly important for providing resources to a large number of user applications. They do so by using virtual services across LAN or WAN networks (Lischka & Karl 2009; Yu, Yi, Rexford, & Chiang 2008; Vaquero, Rodero-Merino, Caceres, & Lindner 2008). They integrate hardware, software and services and are required to meet high standards on quality of service (QoS).

A distributed environment is subject to a wide variety of failures. E.g., application crashes, operating system crashes, device driver crashes, application deadlocks, application livelocks, hardware failures, among others. At any given moment, some servers may be active, some busy or very slow, and some may be offline for maintenance, or may even have crashed. Therefore, these environments present dynamic and changing conditions. As an example, PlanetLab is a distributed system scattered around the globe that supports the development of new network and distributed services. It currently consists of 1169 nodes at 552 sites (PlanetLab). In this scenario, while a large number of nodes may be inactive at any time, it is not possible to know their exact status (active, slow, offline, or down) and, as a consequence, it is impractical to attempt to utilise them without any guidance.

As distributed systems are complex, it is important to address this kind of situations by providing information about service conditions and available resources. Fault-tolerant distributed systems are designed to provide reliable and continuous services despite the failures of some of their components (Nunes & Jansch-Pôrto 2002; Stefanakos 2008; Terpstra, Kangasharju, Leng, & Buchmann 2007; Bahl, Chandra, Greenberg, Kandula, Maltz, & Zhang 2007; Gupta, Chandra, & Goldszmidt 2001; Delporte-Gallet, Fauconnier, & Guerraoui 2002). A common way to handle machine and process crashes involves two steps: 1) Detect the failure; and 2) Recover, by restarting or failing over the crashed component. Failure recovery has received much attention, e.g., using periodic checkpoints, an entire VM can be failed over in one second (Cully, Lefebvre, Meyer, Feeley, Hutchinson, & Warfield 2008); finer-grained components such as processes

or threads can be restarted even faster (Candea, Kawamoto, Fujiki, Friedman, & Fox 2004; Candea, Cutler, & Fox 2004). Interestingly, failure detection has received less attention.

This work focuses on failure detectors (FD), an essential building block for distributed systems. Failure detection plays a central role in the engineering of dependable distributed systems. I.e., it is currently being used in a wide variety of settings, such as network communication protocols (Braden 1989), computer cluster management (Pfister 1998) and group membership protocols (Birman & Renesse 1994; Dolev & Malki 1996; Babaoglu, Baker, Davoli, & Giachini 1994; Moser, Melliar-Smith, Agarwal, Budhia, & Lingley-Papadopoulos 1996; van Renesse, Birman, & Maffeis 1996). Effective failure detection is essential to provide an acceptable Quality of Service (QoS) to applications and, therefore, it is necessary to find an optimized FD that can detect failures in a timely and accurate way before a generic FD service can be implemented for distributed applications.

A failure detector provides some information on which processes have crashed. This information, typically given in the form of a list of suspects, is not always up-to-date or correct. A failure detector may take a long time to start suspecting a process that has crashed and it may erroneously suspect a process that has not crashed. The reason behind this is that in asynchronous (without bound on the process execution speed, message-passing delay or message loss) distributed systems there exists the impossibility of precisely determining whether a remote process has failed or has just been very slow (Chandra & Toueg 1996).

Many applications have timing constraints. They require a FD that provides a quality of service (QoS) with some quantitative timeliness guarantees. A failure detector that starts suspecting a process one hour after it crashed can be used to solve asynchronous consensus, but is useless to an application that needs to solve many instances of consensus per minute. To ensure acceptable QoS for a FD, parameters should be properly tuned to deliver a desirable QoS to the upper layers, as the QoS of FD greatly influences the QoS that upper layers are able to provide.

It is well-known that, when detecting failures, there exists an inherent tradeoff between 1) *conservative* failure detection, i.e., reducing the risk of wrongly suspecting a correct process, and 2) *aggressive* failure detection, i.e., quickly detecting the occurrence of a real crash. There exists a continuum of valid choices between these two extremes, and an appropriate choice is strongly related to each application's requirements on QoS.

Nowadays, applications usually implement their own ad-hoc failure detection modules. These modules periodically send messages between hosts and use some sort of timeout mechanism. E.g., in a commonly used approach, monitored processes send messages at a certain rate and monitoring processes wait for a fixed amount of time for messages in order to timeout (or not). However, there is no systematic way of setting or adjusting the message sending rate or the length of the timeout in the face of different application requirements. Many people have been advocating that failure detection should be provided as a service (Bertier, Marin, & Sens 2002; Bondavalli, Chiaradonna, DiGiandomenico, & Grandoni 1996; Casimiro & Veríssimo 2001; Chandra & Toueg 1996; Charron-Bost, Défago, & Schiper 2002), similar to IP address lookup (DNS) or time synchronization (NTP). Unfortunately, in spite of important technical breakthroughs, this view has met little success so far. One of the major obstacles to building a failure detection service is that simultaneously running distributed applications with different quality-of-service requirements must be able to tune the service to meet their own needs without interfering with each other.

Finally, large scale distributed-applications have created the need for failure detectors which are efficient in terms of network load and therefore, failure detectors targeting this type of applications should address scalability and efficiency (Fakhouri, Goldszmidt, Kalantar, Pershing, & Gupta 2001; van Renesse, Minsky, & Hayden 1998).

1.1 Motivation

The main goal of this thesis is to propose an algorithm that performs better than existing solutions in the presence of unstable network conditions. Namely, we introduce an algorithm that reduces the amount of mistakes made and is more accurate than the most popular failure detectors. A second objective of this work is to study the requirements of a scenario where different applications and virtual machines running on a single physical computer share a failure detection service that satisfies their particular needs in terms of failure detection QoS, while reducing the amount of messages exchanged through the network.

1.2 Contributions

In this work, we propose and use a modified version of the FD algorithm developed by Chen et al. (Chen, Toueg, & Aguilera 2002). This framework provides a systematic way of choosing sending rates and timeouts to meet QoS requirements specified by applications according to network conditions. Our modified version of Chen FD algorithm improves the QoS of the original one, namely, by decreasing the number of mistakes (false detections) made per unit of time and the probability of making mistakes for each given detection time. We also analyse the idea of multiple applications or virtual machines, with different QoS requirements in terms of failure detection, running on a single host using a single failure detector (FD) as a shared service. In such scenario, a crash of a remote host (or process) should be reported by the FD module to all applications monitoring the failed one. We analise how a FD service with such characteristics should provide quality of service (QoS) guarantees to applications. Each application running on a host should be able to specify its own needs on speed and accuracy at which the FD detects crashes. The FD module should give, to each application, the illusion of a dedicated FD that satisfies their particular needs on QoS. Furthermore, each physical machine running the failure detection service should attempt to minimize the number of messages exchanged in the network.

1.3 Results

We have evaluated and compared the QoS of our algorithm to the most important existing ones in terms of mistake rate and query accuracy probability. These algorithms and concepts will be introduced in Chapter 2. The obtained results show that there is always an improvement in terms of QoS when using our proposed algorithm, which varies in magnitude according to network conditions and test environments. Additionally, we have evaluated how, when using a shared failure detection service, the required QoS of applications should be modified in order to coexist. The results show that applications with weaker requirements benefit from the ones with stronger ones by getting an improved resulting QoS. Furthermore, the overall amount of messages sent through the network is reduced when using the shared service, compared to the case of utilising one failure detector per application.

1.4 Structure of the Document

The rest of this document is organized as follows. Chapter 2 provides an introduction to the different technical areas related to our work. Chapter 3 introduces the Multiple Windows Failure Detector (MW-FD) and Chapter 4 evaluates it, and compares it to other failure detection algorithms. Afterwards, in Chapter 5 we present our study on combining multiple application's QoS requirements on a single failure detection service. Finally, in Chapter 6 we conclude this document by summarizing its main points and proposing possible directions of future work.

2 Related Work

In this chapter we introduce background concepts and existing solutions which are important to understand our work. Namely, we introduce time-related models in distributed systems, the *consensus problem*, failure detectors, the utilised model, the QoS specification followed, and the most relevant types of failure detectors. Furthermore, we describe the most important algorithms for failure detection and their implementation. This algorithms are further used to compare with the failure detection algorithm we developed.

2.1 Time-Related Models in Distributed Systems

When characterizing a distributed system, it is very important to understand the behavior of its components with respect to the passage of time, i.e., its processes and network links. In the following subsections, we introduce the time-related models according to the assumptions one can make (if any) regarding communication delays and process execution times.

2.1.1 Asynchronous System

A distributed system is asynchronous if there is no bound on network delay, clock drift, or the time needed by a process to execute a step. Stating that a system is asynchronous is making *n*o timing assumptions whatsoever about its communication links and processes' speed (Guerraoui & Rodrigues 2006).

2.1.2 Synchronous System

Assuming a synchronous system is assuming a system which presents all the following properties (Guerraoui & Rodrigues 2006):

- *Synchronous computation*: The time taken by a process to execute a step is bounded, i.e. it is always smaller than this upper-bound. In this context, a step of a process may involve one or more of the following actions: the delivery of a message (possibly sent by another process), a local computation, and the sending of a message to some (or many) other process(es).
- *Synchronous communication*: There is an upper-bound in the time a message takes to be delivered to the destination process, from the moment it is sent by its sender. No message among two processes can take longer than this bound to be transmitted within the system.
- *Synchronous physical clocks*: Each process has a local clock. There is a known upper-bound in the deviation each of these clocks can present with respect to a global real-time clock.

2.1.3 Partially-Synchronous System

A partially-synchronous system is one that presents, *normally* or during *most* periods of time, a synchronous behavior. During these periods of synchronous behavior, it is possible to assume time boundaries that are not exceeded. In a partially-synchronous system, there are also times in which these timing assumptions do not hold and the system behaves asynchronously. These periods might be caused by network overloads or process slowdowns, maybe due to a shortage of memory needed for its execution. The notion of a partially-synchronous system captures the fact that there are times when a system may behave asynchronously, and that there is no bound in the duration of synchronous or asynchronous periods of behavior. Nevertheless, there are some periods of time when the system behaves synchronously which are *long enough* for a process to make some progress or finish its execution (Guerraoui & Rodrigues 2006).

2.1.4 Timing Assumptions in Real Systems and the Consensus Problem

When assuming a synchronous system, the major limitation is the coverage of the model. It is very hard to build a system where timing assumptions always hold. In such a system, the network and processors load and scheduling mechanisms used should be carefully studied and understood. In a distributed large-scale system deployed across the Internet, very large time estimates should be considered and used in order to capture real time-bounds on executing a step

by a process or to send a message through the system's network. This worst-case upper-bounds are much higher than the average values and any application based on them, in practice, would be extremely slow.

The asynchronous model is attractive and is currently popular for several reasons: it is simple; applications created on the basis of the asynchronous model are more portable than others that make specific timing assumptions; and in practice, variable network loads and process slowdowns are common sources of asynchrony. Furthermore, there is no need to make any timing assumptions on the processes or network links comprising the system. Even when this approach is unquestionably appealing, it is well known that some distributed problems can not be solved under the asynchronous model (Fischer, Lynch, & Paterson 1985; Chandra, Hadzilacos, Toueg, & Charron-Bost 1996). Among them, the most important is agreement in distributed systems in any of its forms, like Consensus (Lamport, Shostak, & Pease 1982), Total-Order and Atomic Broadcast (Birman, Schiper, & Stephenson 1991). These agreement problems are equivalent (Correia, Neves, & Veríssimo 2006; Milosevic, Hutle, & Schiper 2011) in asynchronous systems with crash failures. Additionally, they are core building blocks for many distributed applications today (Burrows 2006; John, Katz-Bassett, Krishnamurthy, Anderson, & Venkataramani 2008; Lee & Thekkath 1996; Saito, Frølund, Veitch, Merchant, & Spence 2004). The impossibility of solving such problems in an asynchronous system is caused by the fact that under this model it is very difficult to determine whether a process has crashed or has only been slow (Chandra & Toueg 1996). Furthermore, for a system to be safe under asynchrony, the totality of its components have to be safe under asynchrony (Aguilera & Walfish 2009).

The partially-synchronous model is suitable for representing most real systems, but hard to implement in practice. Given normal work and network loads, one would expect a system to make progress at a certain expected pace and, hence, to work as a synchronous system. It is also expected in reality to experience moments of network congestion and workloads that would make a system violate the timing bounds expected during synchronous periods. Furthermore, under the partially-synchronous model, the problems of consensus and atomic broadcast can be solved (Dwork, Lynch, & Stockmeyer 1988) in theory. In practice, many attempts have been made (Dolev, Lynch, Pinter, Stark, & Weihl 1986; Toueg 1984; Bridgland & Watro 1987; Dolev, Dwork, & Stockmeyer 1987; Dwork, Lynch, & Stockmeyer 1988), some of which intended to implement partial synchrony. Nevertheless, the impossibility of solving this agreement problems

remained a major obstacle to the use of the asynchronous model of computation for fault-tolerant distributed systems.

In the following subsection, we will explain how these problems were circumvented extending the asynchronous model to achieve partial-synchrony. Unsurprisingly, the enabling component for achieving this is the *failure detector* abstraction.

2.2 Failure Detector

In this section, we introduce the failure detector. Moreover, we describe the failure detection model and the definition of the Quality of Service (QoS) specification for failure detection as introduced by Chen et al. (Chen, Toueg, & Aguilera 2002). Finally, we present related work on failure detectors with QoS.

A *reliable failure detector* is one that is always accurate in detecting the failure of a process. It answers processes' queries with either a response of *Trust* which can only be a hint, or *Fail*. A result of Fail means that the detector has determined that the process has crashed. A process that has crashed stays that way and, by definition, never makes further progress.

An *unreliable failure detector* is one that provides information that is not necessarily accurate. Most failure detectors fall into this category (Chen, Toueg, & Aguilera 2002; Bertier, Marin, & Sens 2002; Chandra & Toueg 1996). An unreliable failure detector may take a long time to start suspecting a process that has crashed and it may erroneously suspect a process that has not crashed (due to the impossibility to know in a real system whether a process has crashed or has been merely slow, or due to message losses and delays). It may produce one of two values when given the identity of a process: *Trust* or *Suspect*, which is only a *hint* on the state of the monitored process.

Throughout this work, we only consider unreliable failure detectors that may suspect a process that has not failed (they may be inaccurate), and not suspect a process that has in fact failed (they may be incomplete). Reliable failure detectors are not considered as they require the system to be synchronous, and few practical systems are.

As a consequence of the consensus problem (see section 2.1.4), the asynchronous model is not sufficient for systems that need to solve some form of agreement. Chandra and Toueg

provided the first formal definition of *unreliable failure detectors* (Chandra & Toueg 1996). In this work, the authors show that it is possible to broaden the applicability of the asynchronous model of computation by encapsulating partial synchrony assumptions in the unreliability of failure detectors. By augmenting the asynchronous model with an unreliable failure detector, they showed that it is possible to solve the different kinds of agreement problems, namely, consensus and atomic broadcast. An unreliable failure detector can make an infinite number of mistakes. I.e., it can add or remove *correct* processes from its list of suspects. Moreover, some correct processes may be erroneously suspected by all the other processes throughout the entire execution.

The weakest unreliable failure detector that can be used to solve agreement under the asynchronous model is represented by $\Diamond\mathcal{W}$. It meets (at least) the following properties (Chandra & Toueg 1996):

- *eventual completeness.* There is a time after which every process that crashes is permanently suspected by some correct process.

- *eventual accuracy.* There is a time after which some correct process is never suspected by any correct process.

The two properties require that eventually some conditions hold forever. This requirement is not that strong in real environments. It is acceptable that this conditions hold for *sufficiently long* periods of time for applications to make progress, e.g., in consensus, the time needed for correct processes to decide or, in atomic broadcast, for correct processes to deliver a message.

2.2.1 Implementation of Unreliable Failure Detectors

In this section, we present a classification of failure detectors according to the way a monitoring process exchanges messages with a monitored process (in the considered model, q and p respectively). We distinguish *heartbeat* failure detectors from *pinging* ones and discuss about their characteristics.

2.2.1.1 Pinging Failure Detectors

In a pinging failure detector, a process q monitors a process p by periodically sending *Are you alive?* messages. Upon the reception of an *Are you alive?* message, the monitored process p replies with an *I am alive* message. If process q times out on process p (meaning that a significantly long period of time has elapsed since the last *Are you alive?* message was sent by q), it starts suspecting p. If q later receives an *I am alive* message from p, it stops suspecting p. This implementation is defined by two parameters:

- *the interrogation interval* Δ_i*:* this is the time between two emissions of an *Are you alive?* message.
- *the timeout delay* Δ_{to}*:* this is the time between the emission of an *Are you alive?* message by q to p , and the time where q starts suspecting p. This state of suspicion continues until q receives a new *I am alive* message from p.

2.2.1.2 Heartbeat Failure Detectors

This is the most popular strategy for implementing failure detectors. The strategy works as follows: process p periodically sends *I am alive* messages, that we call *heartbeats* in this context, to process q. If process q does not receive a message from p for a certain amount of time, it starts suspecting process p. If q later receives a heartbeat from p, q then stops suspecting p. This implementation is defined by two parameters:

- *the heartbeat interval* Δ_i: it is the time between two emissions of a *heartbeat* sent by p.
- *the timeout delay* Δ_{to}*:* it is the time between the last reception of a heartbeat message from p at q, and the time where q starts suspecting p until a heartbeat from p is received.

2.2.1.3 Strategies comparison

Heartbeat failure detectors have many advantages over pinging failure detectors. The first advantage is that they send half of the messages pinging detectors send for providing the same detection quality.

The second advantage is the related to quality of the estimation of the timeout Δ_{to} that process q waits for a message from p until it starts suspecting p. In order to estimate Δ_{to}, a heartbeat failure detector only needs to consider the transmission delay of *I am alive* messages, whereas a pinging failure detector needs to take into consideration several factors, namely, the transmission delay of *Are you alive?* messages, the delay in the monitored process from the moment it receives these messages until it replies, and the transmission delay of *I am alive* responses. Therefore, it is simpler and less error prone to make a good estimation by heartbeating than by using the pinging method.

Throughout the rest of this document, we will only focus on unreliable failure detectors, which is the type of the algorithm presented in this work, and of the most relevant existing solutions.

2.3 Quality of Service (QoS) Specification for Failure Detectors

In this subsection, we present the notions introduced by Chen et al. (Chen, Toueg, & Aguilera 2002) on QoS for failure detectors. This work was the first comprehensive and systematic study on the subject. It proposed some metrics that quantify *speed* and *accuracy* of failure detectors. The failure detector model and the most relevant metrics are introduced in the following sections.

2.3.1 Introduction

The approach used by Chandra and Toueg was later used and generalized in other works (Aguilera, Chen, & Toueg 2000; Aguilera, Chen, & Toueg 1999; Aguilera., Chen, & Toueg 2000). In each of these works, failure detectors were specified in terms of their *eventual behavior* (e.g. a process that crashes is eventually suspected). Such specifications can be suitable for asynchronous systems where there is no timing assumption at all. However, many real applications have timing constraints and, for such applications failure detectors with eventual guarantees do not suffice. Applications that have timing constraints require failure detectors that provide a *quality of service (QoS)* with some quantitative timeliness guarantees.

2.3.2 Model for QoS Specification

A system of two processes, p and q, is considered. The failure detector at q monitors p, and p never crashes. Real time is continuous and ranges from 0 to ∞.

At any given time t, the output of the failure detector in q can be either S, suspect, or T, trust. Whenever the output of the failure detector in q changes, we say that a *transition* occurs: an *S-transition* occurs when the output of q changes from T to S; and a T-transition happens when the output of the failure detector at q changes from S to T. Only a finite number of transitions can take place during a finite period of time.

Only failure detectors in *steady state* are considered. A failure detector's state depends on many factors, such as an *initial condition* (whether it starts in a trust or suspect state) and for how long it has been running. As time passes, the effect of the initial condition diminishes and, eventually, the failure detector reaches steady state. In steady state, the behavior of the failure detector does not change. Typically, this state is reached rapidly. E.g., the failure detector here implemented reaches steady state after the first heartbeat is received (see Chapter 3).

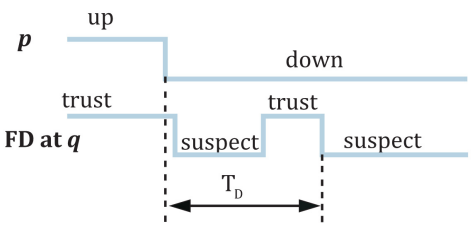
2.3.3 QoS Metrics for Failure Detectors

QoS metrics measure how *fast* and *accurate* a failure detector is. These metrics are applicable to all failure detectors, regardless of how they are implemented. The most relevant metrics, as introduced by Chen et al. are described in this section. Note that the first one is related to a failure detector's *speed*, while the remaining relate to its *accuracy*.

- ***Detection Time (T_D):*** it is the time that elapses from the moment that process p crashes until the failure detector at q detects the failure and starts suspecting p for ever. More precisely, T_D measures the time that elapses from the moment that the crash of p occurs to the moment when the final S-transition occurs (at q) and there are no further transitions (see figure 2.1).

- ***Query Accuracy Probability (P_A):*** this is the probability that the failure detector's output is correct at a random time. This metric is useful for applications that interact with the failure detector by querying it at random times. See Figures 2.3 and 2.4.

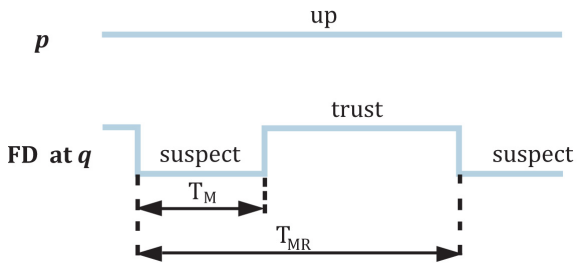
Figure 2.1: Detection Time T_D



- ***Average Mistake Rate (T_{MR}):*** this measures the rate at which a failure detector makes mistakes, i.e., it is the number of S-transitions per unit of time. This is an important metric for long-lived applications where a mistake results in a costly interrupt, such as group membership applications (Schiper & Toueg 2008) and cluster management (Correia, Neves, & Veríssimo 2006) (see Figure 2.2).

- ***Average Mistake Duration (T_M):*** this measures the time a failure detector takes, on average, to correct a mistake. More precisely, T_M measures the average time that elapses from an S-transition to the next T-transition (remember that in the considered model p never crashes and, therefore, an S-transition output by q represents a mistake, see Figure 2.2). The T_M metric is useful for applications that operate in a degraded mode when a process is incorrectly suspected.

Figure 2.2: Mistake Duration T_M and Mistake Rate T_{MR}



The *Query Accuracy Probability (P_A)* metric is not sufficient to fully describe the accuracy of a failure detector. Figure 2.4 shows two failure detectors, FD_1 and FD_2, which have the same P_A, but FD_2 makes mistakes more frequently than FD_1, i.e., its T_{MR} is higher. For some applications, a false suspicion may cause a costly interrupt and the T_{MR} is an important accuracy metric. One can also note, from Figure 2.3, that two failure detectors can have the same T_{MR}, but different P_A. Given any two of this three accuracy metrics, it is possible to

derive the remaining one (Chen, Toueg, & Aguilera 2002). We use this metrics in our evaluation (Chapter 4) to compare different failure detectors.

Figure 2.3: FD_1 and FD_2 have the same T_{MR}, but different P_A.

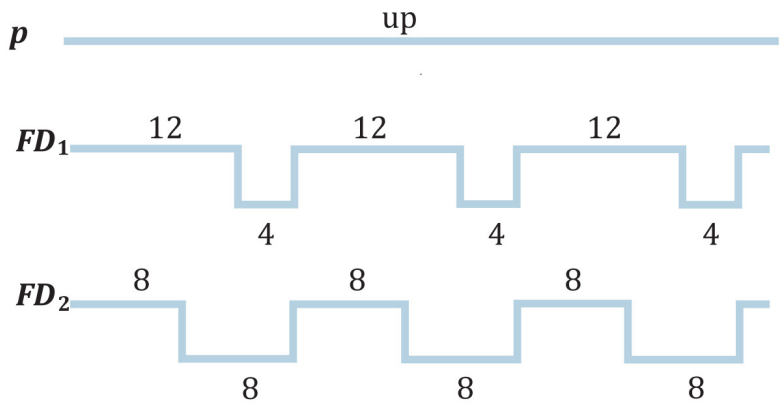
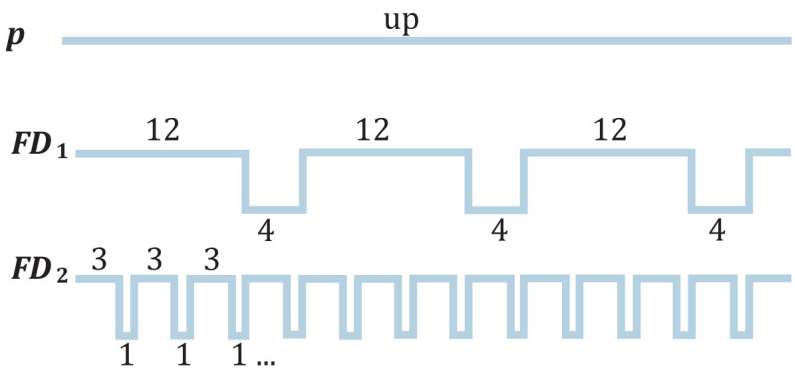


Figure 2.4: FD_1 and FD_2 have the same P_A, but different T_{MR}.



2.4 Failure Detection Algorithms

2.4.1 A Simple Heartbeat Failure Detector Algorithm and its Drawbacks

Heartbeat-based failure detection algorithms that were commonly used in the past worked as follows: At a regular time interval Δ_i, process p sends a heartbeat message to q; when q receives a heartbeat message, it starts a timer with a fixed timeout value Δ_{to}. If the timer expires before q receives a heartbeat message from p, then q starts suspecting p.

This algorithm presents two drawbacks. One regarding its accuracy and another one, its detection time. Consider the ith heartbeat message, m_i. Intuitively, the probability of a premature timeout on m_i should depend solely on m_i and, in particular, on its delay. With the common algorithm, however, the probability of a premature timeout on m_i also depends on the heartbeat m_{i-1} that precedes m_i. In fact, the timer for m_i is started upon the receipt of m_{i-1} and, if m_{i-1} is *fast* (i.e., its transmission time on the network is short), the timer for m_i starts early, increasing the probability of a premature timeout on m_i. This dependency on past heartbeats is undesirable.

To see the second problem, suppose p sends a heartbeat just before it crashes and let d be the delay of this last heartbeat. Using the common algorithm, q would permanently suspect p only $d + \Delta_{to}$ time units after the crash of p. Therefore, the worst-case detection time for this algorithm is the maximum message delay plus Δ_{to}. This is impractical because, in the majority of systems, the maximum message delay is orders of magnitude larger than the average message delay.

The source of the above problems lays in the fact that, even though the heartbeats are sent at regular intervals, the timers to catch them expire at irregular times. Namely, at the receipt times of the heartbeats plus a fixed Δ_{to}.

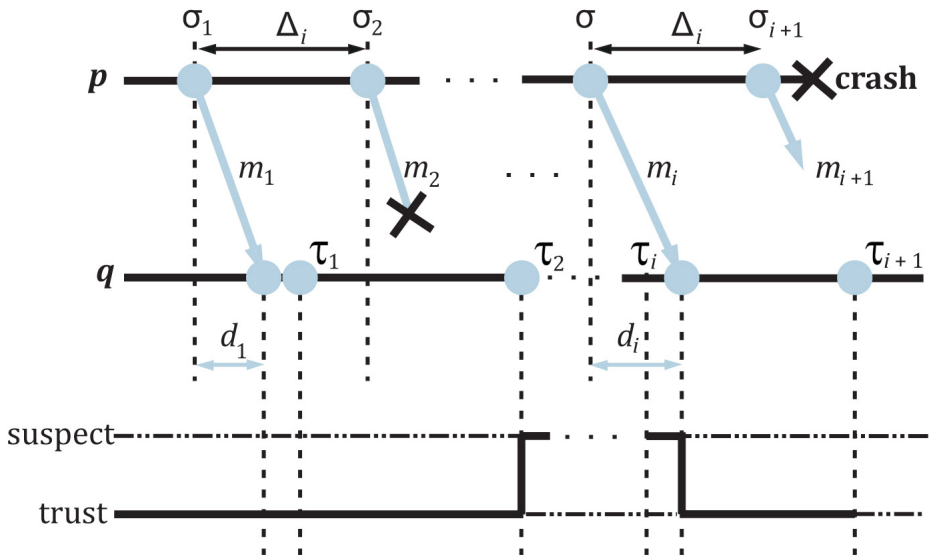
2.4.2 A Heartbeat Failure Detector Algorithm with Freshness Points

Chen et. al (Chen, Toueg, & Aguilera 2002) proposed an algorithm that eliminates these problems. In this new algorithm, the probability of a premature timeout on heartbeat m_i does not depend on the behavior of the heartbeats that precede m_i and the detection time does not depend on the maximum message delay. This algorithm is presented in the following section (2.4).

The improved algorithm works as follows: process p sends heartbeat messages m_1; m_2; . . . to q periodically every Δ_i time units (as in the simple algorithm). To determine whether to suspect p, q uses a sequence τ_1; τ_2; . . . of fixed time points, called *freshness points*, obtained by shifting the sending time of the heartbeat messages by a fixed parameter Δ_{to}. More precisely, $\tau_i = \sigma_i + \Delta_{to}$, where σ_i is the time when m_i is sent. For any time t, let i be so that $t \in [\tau_i;\tau_{i+1})$; then, q trusts p at time t if and only if q has received heartbeat m_i or m_j, where $j \geq i$. See figure

2.5. In contrast to the common algorithm (see section 2.4.1), the new algorithm guarantees an

Figure 2.5: a heartbeat failure detection algorithm with freshness points.



upper bound on the detection time. Moreover, in their work, the authors show that, among all failure detectors that send heartbeats at the same rate (they use the same network bandwidth) and present the same upper bound on detection time, the new algorithm has the best query accuracy probability (Chen, Toueg, & Aguilera 2002). This algorithm is the foundation for our work and other related algorithms (Bertier, Marin, & Sens 2002; Deianov & Toueg 2000; Bertier, Marin, & Sens 2003).

2.5 Real Implementations of Failure Detector Algorithms

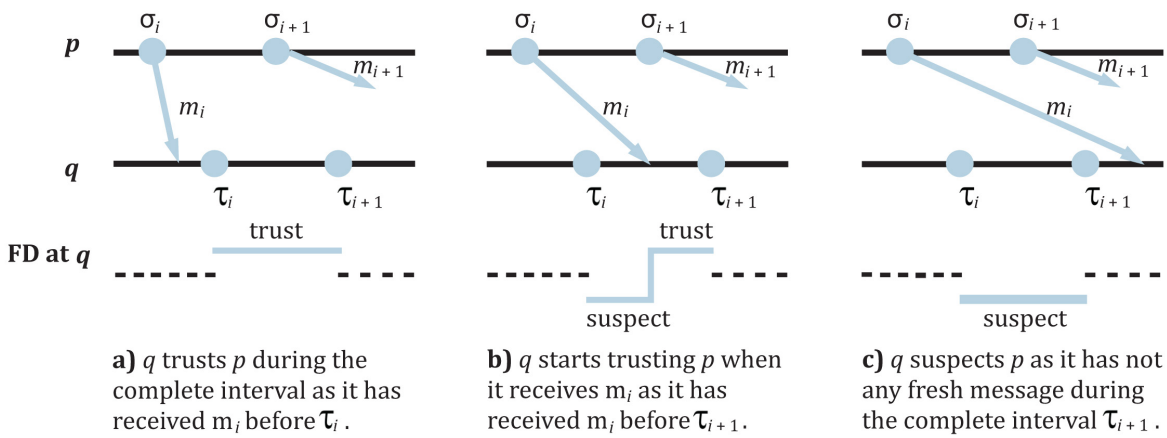
In this section, we present the most relevant failure detection algorithms to our work. This algorithms are later used in to compare our algorithm with existing solutions in Chapter 4.

2.5.1 Chen's Failure Detector

Chen et al. developed the failure detector introduced in section 2.4.2. The detailed implementation of the algorithm works as follows (Chen, Toueg, & Aguilera 2002). The monitored process p periodically sends heartbeat messages (tagged with its sequence number i) m_1;m_2;m_3;... to q every Δ_i time units, where Δ_i is a parameter of the algorithm. σ_i denotes

the sending time of message m_i. The monitoring process q shifts the σ_is forward by Δ_{to}, the other parameter of the algorithm, to obtain the sequence of times $\tau_1 < \tau_2 < \tau_3 < ...$, where $\tau_i = \sigma_i + \Delta_{to}$. Process q uses the τ_is and the times it receives heartbeat messages to determine whether to trust or suspect p, as follows: Consider time period $[\tau_i; \tau_{i+1})$, at time τ_i, q checks whether it has received some message m_j with $j \geq i$. If so, q trusts p during the entire period $[\tau_i;\tau_{i+1})$ (see Figure 2.6.a). If not, q starts suspecting p. In a suspect state, if at some time before τ_{i+1}, q receives some message m_j with $j \geq i$, then q starts trusting p from that time until τ_{i+1} (see Figure 2.6.b). If, by time τ_{i+1}, q has not received any message m_j with $j \geq i$, then q suspects p during the entire period (see Figure 2.6.c). The detailed algorithm with parameters Δ_i and Δ_{to} is depicted in Algorithm 1. Note that from time τ_i to τ_{i+1}, only messages m_j with

Figure 2.6: Chen failure detector. Possible outputs in one period $[\tau_i; \tau_{i+1})$



$j \geq i$ can affect the output of the failure detector at q. For this reason, τ_i is called a *freshness point*. From time τ_i to τ_{i+1}, messages m_j with $j \geq i$ are still fresh (useful). Therefore, the algorithm is characterized by the following property: q trusts p at time t if and only if q received a message that is still *fresh* at time t. This property immediately implies that the failure detector reaches its steady state very quickly: It does so at time τ_1, i.e., Δ_{to} time units after the first heartbeat message is sent. This is because, after time τ_j, the state of process q only depends on what happens at or after time σ_j (the time when the jth heartbeat message is sent).

2.5.1.1 Chen's Estimation of Freshness Points

So far (in Section 2.5.1), we have implicitly assumed that the clocks of p and q were synchronized, as q sets the *freshness points* τs by shifting the sending times in p, σs, by a constant Δ_i. Even when it is possible to build systems where clocks are synchronized (Verissimo & Raynal 1999), this assumption is not general.

When clocks are not synchronized, it is not possible to use the sending times in p to compute the values of arrival times τs at q. In this scenario, a different mechanism has to be used in order to compute the *expected arrival times* EAs of heartbeats at q. This expected arrival times are then used to compute the freshness points τs:

$$\tau_{l+1} = EA_{l+1} + \Delta_i \tag{2.1}$$

where Δ_i is a constant safety margin chosen by the user based on her needs on detection time T_D.

Chen's method to compute *expected arrival times* EAs considers the n previous messages (for some n), denoted m'_1,m'_2,...,m'_n. Let s_1,s_2,...,s_n be the sequence number of those messages and A_1,A_2,...,A_n their receipt times at q. Then, EA_{l+1} (where l is the largest sequence number of heartbeats received so far) is estimated by:

$$EA_{l+1} \approx \frac{1}{n}\left(\sum_{i=1}^{n} A'_i - \Delta_{to} s_i\right) + (l+1)\Delta_{to} \tag{2.2}$$

This equation first normalizes each A'_i by shifting it backwards $\Delta_{to} s_i$ time units. Then, an average of the A'_is is computed and, finally, this computed average is shifted forward by $(l+1)\Delta_{to}$.

2.5.2 Bertier Failure Detector

Bertier et al. introduced a failure detector principally intended for LAN environments (Bertier, Marin, & Sens 2002; Bertier, Marin, & Sens 2003). Their algorithm uses the same mechanism as Chen for estimating expected arrival times EAs (see Equation 2.2), but a dynamic way of computing freshness points based on Jacobson's estimation (Paxson & Allman 2000),

Algorithm 1 Chen's Failure Detector Algorithm

Process p: ▷ Using p's local clock

for all $i \geq 1$ **do**
  at time $i \cdot \Delta_i$ send heartbeat m_i to q
end for

Process q: ▷ Using q's local clock

Initialization:
  $\tau_0 = 0$;
  l = -1; ▷ l keeps the largest sequence number of messages received by q
  $A = \{0\}$; ▷ an array of size n that contains the last arrival dates of messages from p
  $EA_0 = 0$;

upon τ_{l+1} = the current time:
  output $\leftarrow S$; ▷ suspect p since no message received is still fresh at this time

upon receive message m_j at time t:
if $j>l$ **then** ▷ Received a message with a higher sequence number
  $l \leftarrow j$;
  $A \leftarrow A \cup \{t\}$
  Compute EA_{l+1} ▷ using the Equation 2.2
  $\tau_{l+1} \leftarrow EA_{l+1} + \Delta_i$ ▷ set the next freshness point using Equation 2.1
  if $t<\tau_{l+1}$ **then**
    output $\leftarrow T$;
  end if
end if

which is used in TCP to estimate the delay after which a transceiver retransmits a message. As in Chen's FD, the arrival times of the n previous messages are kept in order to compute EAs. Jacobson's estimation supposes that the behavior of the system is not constant, and it is used in this algorithm to adapt the safety margin Δ_{to} each time a heartbeat is received.

2.5.2.1 Bertier's Estimation of Safety Margins

The adaptation of the safety margin Δ_{to} is done as a function of the *error* in the last estimation. Parameter γ represents the importance of a new measure with respect to the previous ones. The *delay* represents the estimate margin, and *var* the magnitude between errors. β and ϕ are used to ponder the variance and typical values are β and $\phi = 4$. Upon the reception of message m_l, the estimation of $\Delta_{to,l+1}$ is computed as follows:

$$error_l = A_l - EA_l - delay_l \tag{2.3}$$

$$delay_{l+1} = delay_l + \gamma.error_l \tag{2.4}$$

$$var_{l+1} = var_l + \gamma.(|error_l| - var_l) \tag{2.5}$$

$$\Delta_{to,l+1} = \beta \dot{d}elay_{l+1} + \phi.var_{l+1} \tag{2.6}$$

Whenever a message m_l is received (where l is the largest sequence number seen by the failure detector at q at a given moment), $\Delta_{to,l+1}$ is computed using Equiations 2.3-2.6 and EA_{l+1} is calculated using Equation 2.2. With this two values computed, the next freshness point τ_{l+1} is computed exactly as in Equation 2.1. The timeout for message m_{l+1}, activated by q when it receives m_l, expires at the freshness point τ_{l+1}. The complete algorithm is presented in the following section.

2.5.2.2 Bertier's Algorithm

In this section, we describe the implementation of the algorithm (see Algorithm 2).

Process p periodically sends heartbeats to process q. Namely, it sends a heartbeat every Δ_i time units.

Whenever process q receives a heartbeat m_l, it obtains an estimation of the freshness point for m_{l+1} by calculating the expected arrival time EA_{l+1} and dynamic safety margin $\Delta_{to,l+1}$. If process p is currently being suspected by q, it is trusted. On the other hand, if the last freshness point expires (meaning that the local clock of q has reached τ_{l+1}), then q starts suspecting p.

Algorithm 2 Bertier's Failure Detector Algorithm

Process p: ▷ Using p's local clock

1: **for all** $i \geq 1$ **do**
2: at time $i \cdot \Delta_i$ send heartbeat m_i to q
3: **end for**

Process q: ▷ Using q's local clock

4: ***Initialization***:
5: $\tau_0 = 0$;
6: l = -1; ▷ l keeps the largest sequence number of messages received by q
7: $EA_0 = 0$;
8: $delay_0 = initialvalue$;
9: $\Delta_{to,0} = 0$;
10: $var_0 = 0$;
11: $error_0 = 0$;
12: $A = \{0\}$; ▷ an array of size n that contains the last arrival dates of messages from p

13: ***upon*** τ_{l+1} = the current time:
14: $output \leftarrow S$; ▷ suspect p since no message received is still fresh at this time

15: ***upon*** receive message m_j at time t:
16: **if** $j>l$ **then** ▷ Received a message with a higher sequence number
17: $l \leftarrow j$;
18: $A \leftarrow A \cup \{t\}$
19: $EA_{l+1} \leftarrow EA_l + \frac{1}{n}(A_l - A_{l-n-1})$ ▷ derived from the Equation 2.2
20: $error_l \leftarrow A_l - EA_l - delay_l$
21: $delay_{l+1} \leftarrow delay_l + \gamma.error_l$
22: $var_{l+1} \leftarrow var_l + \gamma.(|error_l| - var_l)$
23: $\Delta_{to,l+1} \leftarrow \beta.delay_{l+1} + \phi.var_{l+1}$
24: $\tau_{l+1} \leftarrow EA_{l+1} + \Delta_{to}$ ▷ set the next freshness point using Equation 2.1
25: **if** $t<\tau_{l+1}$ **and** $output = S$ **then**
26: $output \leftarrow T$;
27: **end if**
28: **end if**

2.5.3 The ϕ Accrual Failure Detector

2.5.3.1 Definition of an Accrual Failure Detector

Accrual failure detectors (Hayashibara, Defago, Yared, & Katayama 2004; Xiong, Vasilakos, Yang, Wei, Qiao, & Wu 2011) output suspicion information on a continuous scale instead of information of a binary nature (trust or suspect). The higher the value an accrual failure detector outputs, the higher the probability that the monitored process has actually crashed.

An *accrual failure detector* is defined as a failure detector that outputs a value associated with the monitored process. In the simplified model considered (two processes p and q, where q monitors p), the output of the failure detector at q over time can be represented by the function $suspLevel_p(t) \geq 0$ (suspicion level of p). This function must satisfy three properties. The first property specifies the function's output when p has crashed. The remaining two properties specify the output if p is correct:

1) ***Accruement:*** When process p has crashed, $suspLevel_p(t)$ tends to infinity with the passage of time.

2) ***Unknown upper-bound:*** If process p is correct, then $suspLevel_p(t)$ is bounded.

3) ***Reset:*** If process p is correct, then for any time t_0, $suspLevel_p(t) = 0$ for some time $t \geq t_0$.

The *accruement* property ensures that, eventually, a faulty process is permanently suspected (regardless of the suspicion threshold). The *unknown upper-bound* property ensures that, using a dynamic threshold, a correct process is never suspected eventually. Finally, the *reset* property ensures that a correct process that is suspected is eventually trusted again (regardless of the suspicion threshold).

2.5.3.2 Accrual Failure Detector Architecture

On the monitoring side (in our model, from q's perspective), the implementation of failure detectors can be decomposed in three steps, as follows:

1) *Monitoring:* the monitoring process gathers information from the monitored one(s), usually through the network, such as heartbeat arrival times or query-response delays (as in 2.2.1.1).

2) *Interpretation:* the monitoring information is used and interpreted, for instance to decide that a process should be suspected.

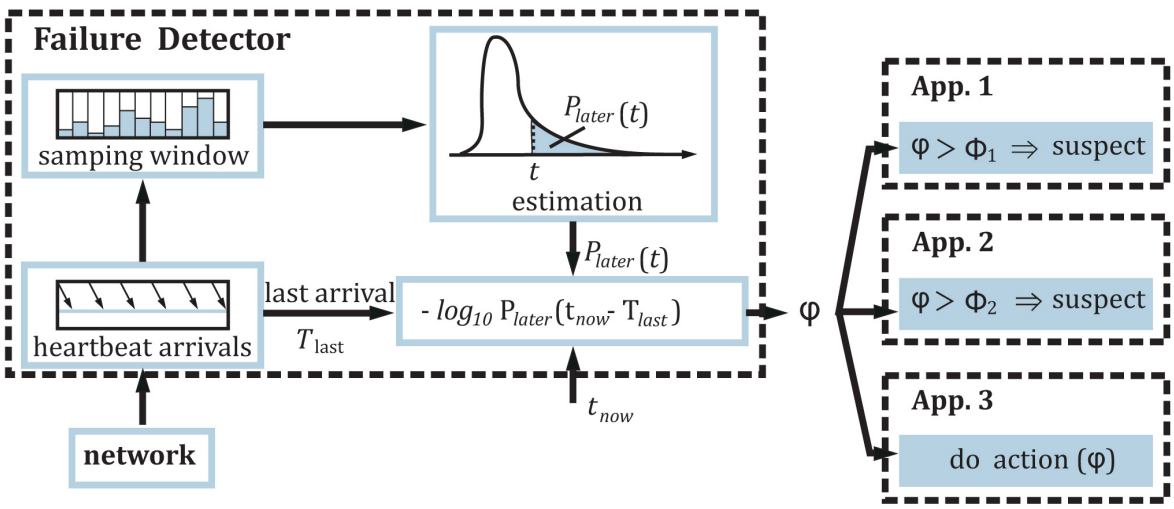
3) *Action*: actions are executed as a response to triggered suspicions. This is normally done within applications.

The main difference between traditional failure detectors and accrual failure detectors is which component of the system is responsible for which of these actions. In the failure detector algorithms presented so far, the monitoring and the interpretation steps are performed by the failure detector itself and the output they provide to applications is of binary nature; trust or suspect. Furthermore, timing-out means suspecting the monitored process and therefore, the monitoring information is already being interpreted. Applications are left with what to do with suspicions. In contrast, accrual failure detectors avoid the interpretation of monitoring information. Some value is associated with each process that represents a suspicion level. This value is then left for the applications to interpret.

2.5.3.3 The ϕ Accrual Failure Detector Algorithm

In this section, we present the practical implementation of the ϕ failure detector (Hayashibara, Defago, Yared, & Katayama 2004).

Figure 2.7: Information flow in the implementation of the ϕ failure detector.



In ϕ FD, the suspicion level is given by a value called Φ, expressed on a scale that is dynamically adjusted to reflect current network conditions. Let T_{last} denote the time when the most recent heartbeat was received, T_{now} the current time, and $P_{later}(t)$ the probability of a heartbeat arriving more than t time units after the previously received one. Then, the value of ϕ is calculated as follows:

$$\phi(T_{now}) = -\log_{10}(P_{later}(T_{now} - T_{last})) \tag{2.7}$$

In this context, ϕ has the following meaning. Given a threshold Φ, if the failure detector suspects p when $\phi \geq \Phi$, then the probability that the ϕ failure detector makes a mistake is about $\frac{1}{10^{\Phi}}$. In order to ensure the *reset* property, the value of ϕ is set to 0 by q upon the receipt of each heartbeat.

The estimation of ϕ is done as follows. When heartbeats arrive, their arrival times are stored in a sampling window (as Chen's and Bertier's FD algorithm, described in sections 2.5.1 and 2.5.2 respectively). These past samples are used to determine the distribution of interarrival times. Finally, the distribution is used to compute the current value of ϕ. The estimation of the distribution of interarrival times assumes that they follow a *normal distribution.* This is an approximation based on the assumption that heartbeat inter-arrival times are influenced by a very large number of independent unknown factors (central limit theorem). The parameters of the distribution are estimated by determining the mean μ and the variance σ^2 of the samples. Then, the probability $P_{later}(t)$ that a given heartbeat will arrive more than t time units later than the previous heartbeat is given by the following equation:

$$P_{later}(t) = \frac{1}{\sigma\sqrt{2\pi}} \int_t^{\infty} \mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}} \mathrm{d}x \tag{2.8}$$

$$= 1 - F(t) \tag{2.9}$$

where $F(t)$ is the cumulative distribution function of a normal distribution with mean μ and variance σ^2. Finally, the value of ϕ at time T_{now} is computed by applying Equation 2.7. This process is repeated by q for every new heartbeat received.

The overall mechanism is described in Algorithm 3 and depicted in figure 2.7.

Algorithm 3 The ϕ Failure Detector Algorithm

Process p: ▷ Using p's local clock

for all $i \geq 1$ **do**
  at time $i \cdot \Delta_i$ send heartbeat m_i to q
end for

Process q: ▷ Using q's local clock

Initialization:
  $\tau_0 = 0$;
  $A = \{0\}$; ▷ an array of size n that contains the last arrival dates of messages from p
  $\mu = 0$; ▷ keeps the mean of the the last n messages
  $\sigma = 0$; ▷ keeps the standard deviation of the the last n messages
  l = -1; ▷ l keeps the largest sequence number of messages received by q

upon τ_{l+1} = the current time:
  output $\leftarrow S$; ▷ suspect p since no message received is still fresh at this time

upon receive message m_j at time t:
if $j>l$ **then** ▷ Received a message with a higher sequence number
  $l \leftarrow j$;
  $A \leftarrow A \cup \{t\}$
  $\mu \leftarrow \mu(A)$ ▷ Compute the mean of the values in A
  $\sigma \leftarrow \sigma(A)$ ▷ Compute the standard deviation of the values in A
  $t_\Phi \leftarrow \phi^{-1}(\mu, \sigma^2, \Phi)$ ▷ t_Φ makes $\phi(t_\Phi) = \Phi$, computed reversing Equation 2.8
  $\tau_{l+1} \leftarrow t_\Phi$ ▷ set the next freshness point τ_{l+1} to t_Φ
  if $t<\tau_{l+1}$ **then**
    output $\leftarrow T$;
  end if
end if

2.5.4 Exponential Distribution Failure Detector (ED FD)

The Exponential Distribution failure detector (ED FD) (Xiong, Vasilakos, Yang, Wei, Qiao, & Wu 2011) is based on the same principle as the ϕ accrual failure detector. The difference lies in the distribution considered for message delays, i.e., the ϕ failure detector considers a normal distribution whereas the ED FD considers an exponential one. In ED FD, the suspicion level is given by a value called e_d, which is calculated as follows:

$$e_d = F(T_{now} - T_{last}) \tag{2.10}$$

$$F(t) = 1 - \mathrm{e}^{-\frac{1}{\mu}t} \tag{2.11}$$

where T_{now}, T_{last} and μ have the same meaning as in the ϕ accrual failure detector (section 2.5.3).

In the scope of this work, the authors sampled the behavior of different network scenarios (a cluster group, a wireless network, a wired LAN, and a WAN) and concluded that the exponential model is more suitable for representing network delays than its normal counterpart. The ED FD algorithm is exactly as the one depicted in Algorithm **??**, with the difference that, in line 18, t_ϕ is computed by reversing Equation 2.11.

Summary

In this chapter we have introduced related work in the area of failure detection for distributed systems. We started by introducing some important concepts to understand this work. Then, we briefly surveyed some of the most relevant practical systems that are related to our work.

The algorithms presented throughout this section are able to provide QoS guarantees to applications when the network behaves according to some probability distribution. Nevertheless, in the presence of bursts of lost messages, this algorithms output a large number of false positives until the network stabilizes. This occurs because the estimation of expected arrival times and freshness points by using these probabilistic approaches are dependent on the past history of observed arrival times of messages. This dependence on a big number of past samples prevents these algorithms from quickly reacting to sudden changes. In the next chapter, we further explain this problem and introduce our algorithm, which addresses unstable network behaviors.

3 Multiple Windows Failure Detector

In this chapter we introduce the *Multiple Windows Failure Detector* (MW FD) and the idea behind it.

3.1 Introduction - Dealing with Bursty Traffic

In some scenarios, the probabilistic behavior of the network (message delay and message loss) can change, e.g., a corporate network can present different behaviors according to the time of the day, i.e., at peak work hours, the amount of message dropped by the network may increase and message delays can vary significantly from message to message, as the network is more congested. On the other hand, at night, when there is barely no work traffic, the probability of losing messages decreases, and message delays are more stable. The algorithms presented in Chapter 2 can adapt their behavior to changing network conditions. Some of them achieve this by their own nature, namely, the ones presented in sections 2.5.2, 2.5.3 and 2.5.4. Chen's failure detector (see section 2.5.1) can be made adaptive by recomputing the heartbeat interval Δ_i and timeout Δ_{to} every certain period of time (the procedure for this computation will be introduced in section 5.1.1). This way, Chen's failure detector can adapt to gradually changing network conditions.

There are cases where network conditions change very frequently due to bursty traffic. This is more likely to happen in WAN scenarios, where message delays and losses are affected by a bigger amount of factors. Chen's algorithm can be adapted with the above mentioned mechanism in the cases that the following conditions hold (Chen, Toueg, & Aguilera 2002):.

1) the *occurrences* of bursts are independent of each other and follow some slowly changing probabilistic distribution.

2) the *duration* of each burst is short (smaller than the heartbeat interval Δ_i).

In this case, the situation is no different from the previous one, as heartbeats behave independently of each other, according to some new slowly changing probability distribution that takes into account the occurrence of bursts.

When 1) or 2) do not hold, some mechanism to estimate the current behavior of the network and adapt to it is needed. In the following section, we present our algorithm. It introduces a mechanism to tackle this problem.

3.2 Rationale

In order to adapt to bursty-traffic conditions, we propose the use of two components for the estimation of espected arrival times EAs and freshness points τs. Namely:

1) a ***short-term*** component that considers only the most recent messages, which is used to quickly react to sudden changes in network conditions, maybe due to bursty traffic, and

2) a ***long-term*** component that considers a bigger amount of recently received messages, that is not sensitive to momentary fluctuations, which is used to make more conservative estimations when the recent messages have been fast.

Whenever a message m_l is received by q, both components are used to estimate the freshness point τ_{i+1} for m_{l+1}, as we will explain in the following section.

3.3 The Algorithm

The *Multiple Windows Failure Detector* (MW FD) algorithm is a variation of Chen's failure detector (introduced in section 2.5.1). The main difference lays in the fact that the MW FD keeps two arrays, $A(n_1)$ and $A(n_2)$ (of sizes n_1 and n_2 respectively), of recently received heartbeat arrival times instead of keeping only one. Whenever a message m_l sent by p is received by q, q adds the arrival time of message m_l to $A(n_1)$ and to $A(n_2)$. The following step is to compute, using the values stored in $A(n_1)$ and to $A(n_2)$, the expected arrival times $EA_{l+1}(n_1)$ and $EA_{l+1}(n_2)$. The key of the algorithm is that, from the EAs computed, it uses the maximum of these estimations for the computation of the freshness point τ_{l+1}:

$$\tau_{l+1} = max\left(EA_{l+1}\left(n_1\right), EA_{l+1}\left(n_2\right)\right) + \Delta_{to} \tag{3.1}$$

where Δ_{to} is a constant safety margin as specified by the QoS requirements on T_D (see section 2.3). Finally, if message m_{l+1} is not received before time $t = \tau_{l+1}$, q starts suspecting p. The rest of the process occurs exactly as in Chen's failure detector (section 2.5.1).

Intuitively, given two window sizes WS_1 and WS_2, our algorithm should be able to make less mistakes than Chen's algorithm when using any of WS_1 or WS_2, as for each analysed sample, the MW-FD computes the maximum of the expected arrival times that would be computed by Chen's algorithm for each window size. This implies that this algorithm will only make the mistakes that Chen's FD would make when using both window sizes WS_1 and WS_2. This means:

$$Mistakes(MW_{WS_1,WS_2}) = Mistakes(Chen_{WS_1}) \cap Mistakes(Chen_{WS_2}) \tag{3.2}$$

We will analyse this with an example in our evaluation chapter (see section 4.3.5).

Algorithm 4 Multiple Windows Failure Detector Algorithm

Process p: ▷ Using p's local clock

1: **for all** $i \geq 1$ **do**
2: at time $i \cdot \Delta_i$ send heartbeat m_i to q
3: **end for**

Process q: ▷ Using q's local clock

4: ***Initialization***:
5: $\tau_0 = 0$;
6: l = -1; ▷ keeps the largest sequence number of messages received by q
7: $\boldsymbol{A(n_1) = \{0\}}$; ▷ **contains the last n_1 arrival dates of messages from p**
8: $\boldsymbol{A(n_2) = \{0\}}$; ▷ **contains the last n_2 arrival dates of messages from p**
9: $EA_0 = EA_{l+1}(n_1) = EA_{l+1}(n_2) = 0$;

10: ***upon*** τ_{l+1} = the current time:
11: *output* $\leftarrow S$; ▷ suspect p since no message received is still fresh at this time

12: ***upon*** receive message m_j at time t:
13: **if** $j>l$ **then** ▷ Received a message with a higher sequence number
14: $l \leftarrow j$;
15: $\boldsymbol{A \leftarrow A(n_1) \cup \{t\}}$
16: $\boldsymbol{A \leftarrow A(n_2) \cup \{t\}}$
17: **Compute** $\boldsymbol{EA(n_1)_{l+1}}$ **and** $\boldsymbol{EA(n_2)_{l+1}}$ ▷ **using Equation 2.2**
18: $\boldsymbol{EA_{l+1} \leftarrow max(EA_{l+1}(n_1), EA_{l+1}(n_2))}$
19: $\tau_{l+1} \leftarrow EA_{l+1} + \Delta_{to}$ ▷ set the next freshness point using Equation 2.1
20: **if** $t<\tau_{l+1}$ **then**
21: *output* $\leftarrow T$;
22: **end if**
23: **end if**

3.4 Benefits of Using Two Windows

Intuitively, our algorithm is meant to, and should, work better than Chen's FD mainly in the presence bursty traffic and rapid changes in network conditions, because of the reasons explained in section 3.3.

In our implementation it is easy to see that, by picking the maximum of the estimations on expected arrival times for expected heartbeats, the failure detector at q becomes more tolerant, i.e., more *conservative*. This occurs because for each heartbeat, our failure detector will wait for the maximum of the times estimated by each window, a fact that directly reduces the probability of making mistakes. At first sight, this would suggest that this algorithm would not be able to work in very *agressive* ranges of detection, where the required T_D is very small. Our measurements show that this is not the case, and that this algorithm performs better than all the ones we compared it to in terms of amount of mistakes (T_{MR}) and accuracy (P_A), even in the aggressive ranges. We will provide empirical evidence of this fact in our evaluation chapter (Chapter 4).

4 Evaluation

In this chapter, we present evaluation results of the tests we performed to study the performance of our algorithm and the algorithms presented in section 2.5. Namely, we evaluate and comparatively analyze the performances of MW FD, the ϕ accrual FD, ED FD, Chen's FD, and Bertier's FD when ran over traces taken from two experimental environments, a WAN and a LAN.

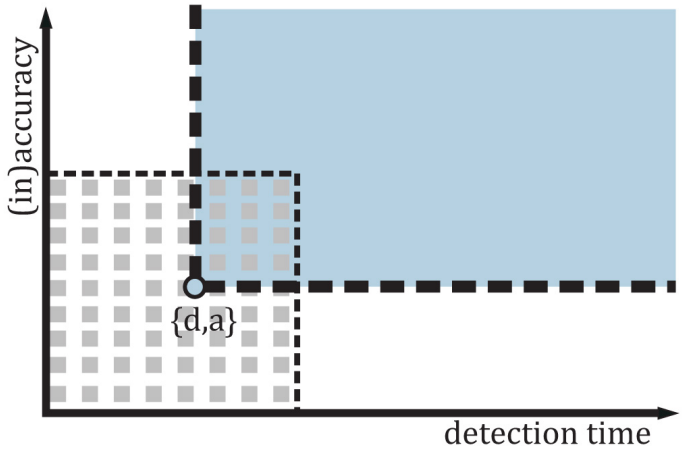
4.1 About the experiments

All experiments were performed on traces of experiments ran on two computers. They worked as follows: one process p periodically sends heartbeat messages to another process q for an arbitrarily long period of time. When heartbeats are received, their arrival times are logged by the monitoring computer q. Then, these logged arrival times are used to replay the execution for each FD algorithm, i.e., for multiple values of the configuration parameter of each FD (e.g., safety margin for Chen's FD and MW FD, or threshold Φ for the ϕ and the ED FD). Therefore, all failure detectors were compared in the same experimental conditions: the same network model, heartbeat history, and experiment parameters (namely, inter-sending intervals, sliding window sizes and communication delays). Thus, this way of experimenting provided the same experimental environment to all failure detectors. Heartbeat messages were sent using the UDP/IP protocol. During the experiments the average CPU load was nearly constant and below the computers' full capacity.

In our experiments, we have used the approach introduced by the developers of the ϕ FD. It consists in determining whether a failure detector can be parametrized to match a given set of QoS requirements. We consider a space of QoS defined by detection time on one axis and an accuracy metric (e.g., T_{MR}, or P_A) on the other. We then measure the area covered by each failure detector when we vary their parameter from a highly *aggressive* (i.e., short T_D) behavior

to a *conservative* (i.e., long T_D) one. The area covered by a failure detector is the region that corresponds to a set of QoS requirements that can it can fulfill (see Figure 4.1). During the experiment with each FD, different values (on accuracy vs. speed) were obtained by varying each's algorithm respective parameters. By using these points, we plotted the curves we present in this chapter.

Figure 4.1: the QoS area that a FD is able to satisfy, given a determined QoS point (d,a) obtained experimentally



All FDs use a sliding window that stores past samples for their computations. Namely, expected arrival times (EAs) and freshness points (τs) in Bertier's, Chen's, and the MW FD; and ϕ and e_d values for the ϕ accrual FD and the ED-FD respectively. We started our analysis once sliding windows are full, as the failure detectors are considered to be unstable during the warm-up period.

It is not possible to obtain information regarding the arrival times of lost messages (see Figure 2.5.c). In order to ensure the effectiveness of the proposed approach, and considering the influence of message loss, we use the time series theory to fill in the gaps, where a gap represents the information regarding the arrival time of a missing message. In detail, we fill in each gap with a value for its arrival time computed by $A_i = (\Delta_i + A_{i-1})$ (Nunes & Jansch-Pôrto 2002), where Δ_i is the average inter-sending interval, as seen by q, of the last n messages, where n is the window size of stored samples. This approach was previously used in similar analysis on failure detectors (Xiong, Vasilakos, Yang, Wei, Qiao, & Wu 2011; Xiong, Vasilakos, Wu, Yang, Rindos, Zhou, Song, & Pan 2012). This way, when using very small window sizes in algorithms that compute expected arrival times and freshness points (namely, Chen's FD, MW-FD and

Bertier's FD), the resulting detection time does not vary significantly. If this technique were not used, in the presence of multiple adjacent gaps, a small sized window would tend to increase the value of the estimation of the time at which the arrival of the next message is expected. This increase would be proportional to the number of adjacent gaps observed. This is a desired characteristic as a bound in detection time is important to applications.

After discarding a warm-up period, we have measured the following three key QoS metrics during the entire execution: detection time T_D, mistake rate T_{MR} and query accuracy probability P_A. Comparing parametric failure detectors is not an easy task as, depending on the values set for their parameters, their behavior may vary significantly. A mistaken approach which is commonly adopted consists in defining and setting arbitrary values for these parameters and then compare FDs which are parametrized differently. With high probability, this would lead to erroneous conclusions, i.e., that one FD is better in terms of detection time while the other FD is more accurate.

The main parameters were configured as follows: For Chen FD, the parameters were set to be the same as in (Chen, Toueg, & Aguilera 2002); $\Delta_{to} \in [0, 2500]$; For the ϕ FD, $\Phi \in [0.1, 16]$; For Bertier FD, the parameters are set the same as in (Bertier, Marin, & Sens 2002; Bertier, Marin, & Sens 2003); $\beta = 1$, $\phi = 4$, $\gamma = 0.1$. In each experiment, the other basic experimental parameters of FDs are the same.

4.2 Tests Scenarios

For our experiments, we utilised two traces. One taken from a WAN scenario, and the other one, from a LAN scenario.

4.2.1 WAN Scenario

This experiments involved two computers: one located in Switzerland, and the other in Japan. They communicated through a normal intercontinental Internet connection. We used exactly the same trace files as in (Hayashibara, Defago, Yared, & Katayama 2004), which are publicly available (JAIST). Neither machine failed during the experiment.

- *Computer p (monitored)*: The sending host was located in Switzerland, at the Swiss Federal Institute of Technology in Lausanne (EPFL).

- *Computer q (monitoring)*: The receiving host was located in Japan, at the Japan Advanced Institute of Science and Technology (JAIST).

All messages were transmitted using the UDP/IP protocol. By using the *traceroute* command, it was observed that most of the traffic was routed through the United States, rather than directly between Asia and Europe.

4.2.1.1 Samples

The experiment lasted for exactly one week. Heartbeat messages were generated at a rate of one heartbeat every 100 ms. The measured average sending rate 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 and 5.822.521 were received (around 0.4% of messages were lost). Message losses tended to occur in bursts. The longest burst was 1093 heartbeats long (i.e., it lasted for about 2 minutes). 814 different bursts of consecutively lost messages occurred. The mean of inter-arrival times observed by q was 103.9 ms with a standard deviation of 0.19 ms. The distribution of the inter-arrival times is represented in figures 4.2 and 4.3, which were obtained from JAIST's website.

A low-frequency ping process ran along with the experiment in order to obtain an estimation of the round-trip time among computers, and for being certain that the network was connected at all times.

Figure 4.3 presents a different view of inter-arrival times. It relates inter-arrival times with the heartbeat's arrival moment. Very long intervals are not depicted. The first horizontal line of points at the bottom of the graph represents *normal* heartbeats that presented an inter-arrival time of around 100ms. The second (thinner) line represents intervals obtained after a single heartbeat was lost. There is a period (April 6 and 7) where more messages were lost. According to the authors, this was likely caused by an outbreak of the *W32/Netsky.T@mm* Internet worm (dates coincided).

Figure 4.2: Distribution of heartbeat inter-arrival times as seen by the monitoring process q

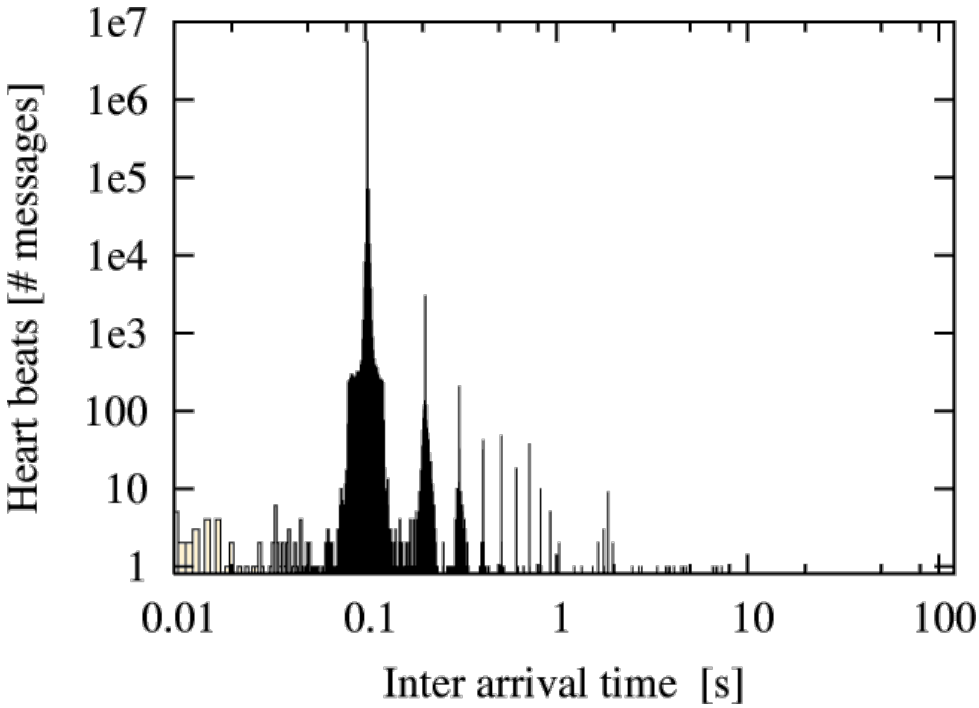
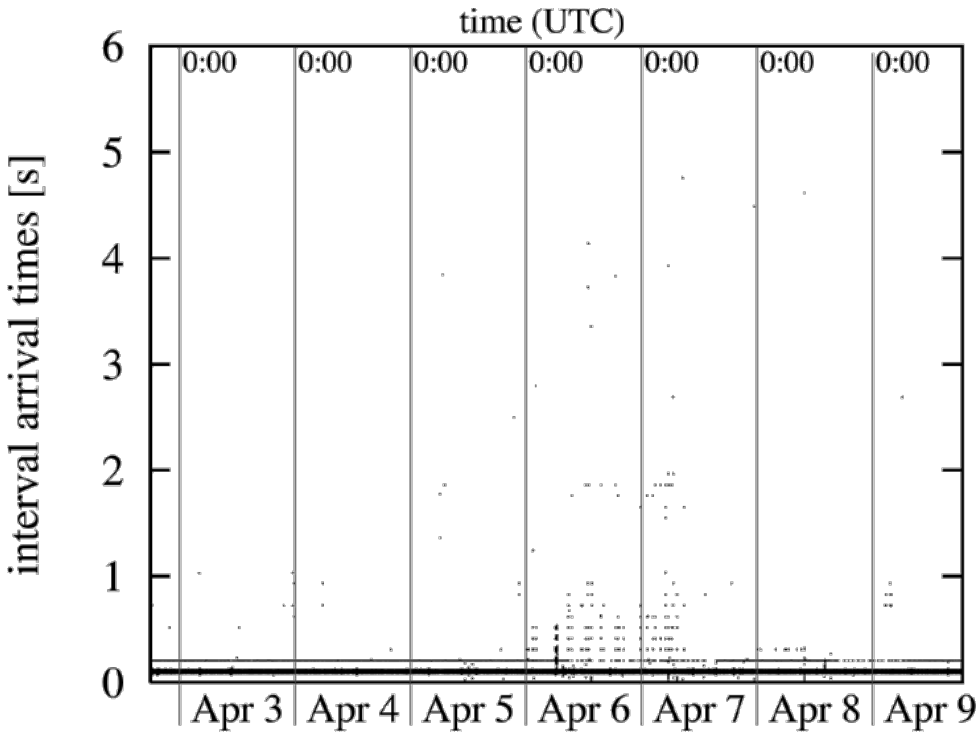


Figure 4.3: Inter-arrival times and their moment of occurrence, as seen by q



4.2.1.2 Round-trip times

During the experiment, the round-trip time (RTT) was also measured, but at a low rate. The RTT was, on average, 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.

4.2.2 LAN Scenario

This scenario is practically the same, with the difference that computers are interconnected through a LAN cable. The experiment used two identical computers located at JAIST and connected through a single unshared 100 Mbps Ethernet hub. The heartbeat interval Δ_i was set to 20ms and the experiment run for a little more than a day. 7.104.446 samples were collected. Not a single heartbeat was lost. The largest interval between the reception of two heartbeats was about 1.5 seconds. Nevertheless, the variance was very small. The average transmission delay was around 100 μs.

4.3 Experiments

4.3.1 Chen FD - Window Sizes

The first experiment measures the effect of the window size of Chen's failure detector, as in the original algorithm (using only one window for storing past samples), on its speed to accuracy ratio. We set the window from a very small size (one sample) to a very large one (10.000 samples) and measured the accuracy obtained with Chen's failure detector when run over both WAN and LAN traces.

4.3.1.1 Experiment Results

Figure 4.4 shows the results on mistake rate T_{MR} vs. detection time T_D in the WAN scenario. T_{MR} is represented on the vertical axes, expressed in logarithmic scale, and T_D in the horizontal axes. Figure 4.5 shows the results on query accuracy probability P_A vs. detection time T_D in the same scenario. Figures 4.6 and 4.7 show the same study in our tested LAN scenario.

The results clearly show that in both scenarios Chen's algorithm behaves better when using a window of a small size, in terms of both T_{MR} and P_A, and it's performance decreases as the window size increases. This is an interesting result as related publications which compared this failure detector or Bertier's (which uses the same mechanism with sliding windows for estimating arrival times) to their work, namely accrual failure detectors, utilised a window of size 1000 for these (and their) algorithms in their experiments. Furthermore, accrual failure detectors benefit from big window sizes (Hayashibara, Defago, Yared, & Katayama 2004; Xiong, Vasilakos, Yang, Wei, Qiao, & Wu 2011; Xiong, Vasilakos, Wu, Yang, Rindos, Zhou, Song, & Pan 2012).

Figure 4.4: Chen's Algorithm T_{MR} vs. T_D with different Window Sizes in a WAN scenario

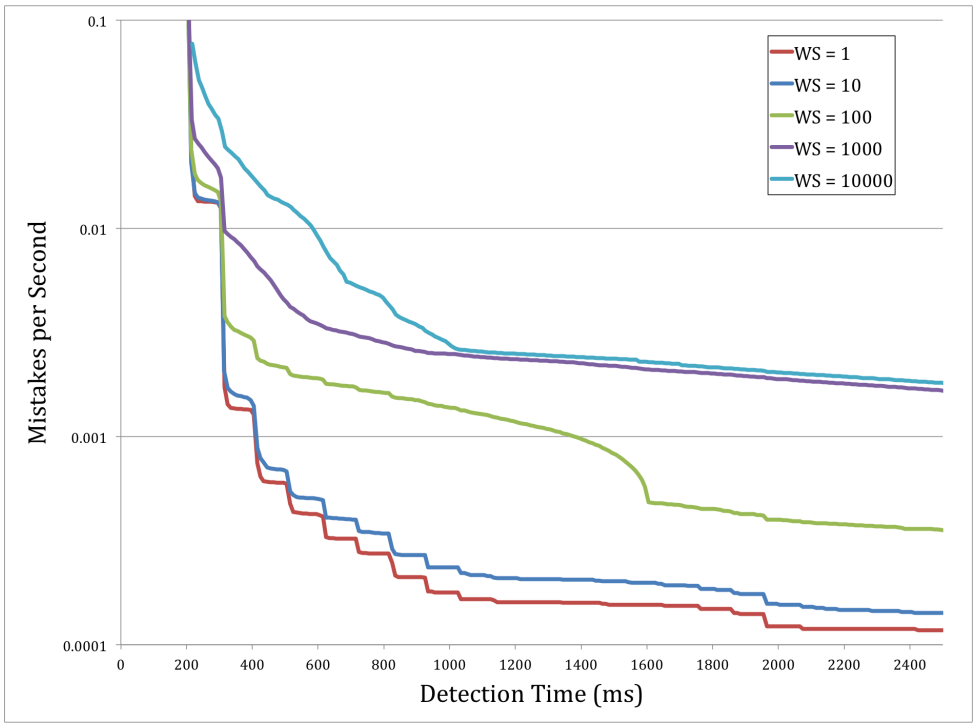


Figure 4.5: Chen's Algorithm P_A vs. T_D with different Window Sizes in a WAN scenario

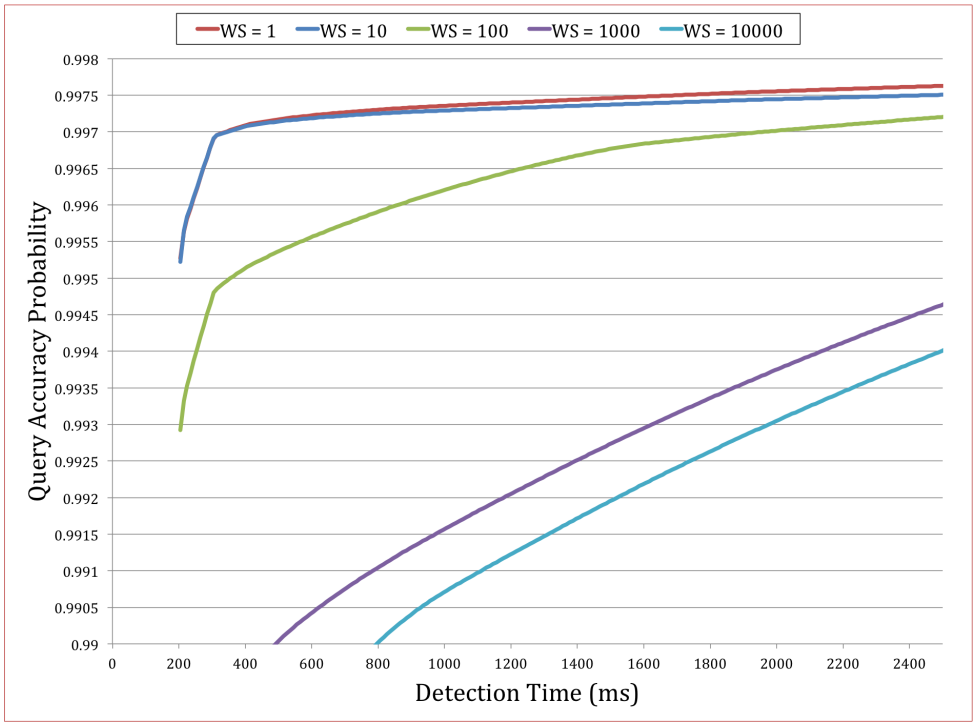


Figure 4.6: Chen's Algorithm T_{MR} vs. T_D with different Window Sizes in a LAN scenario

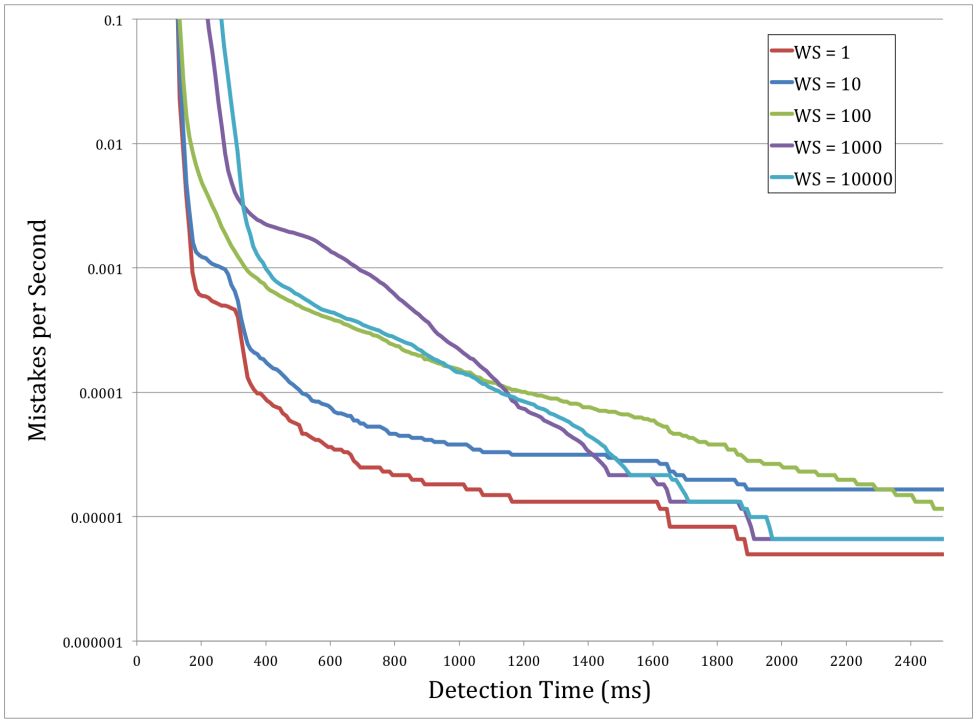
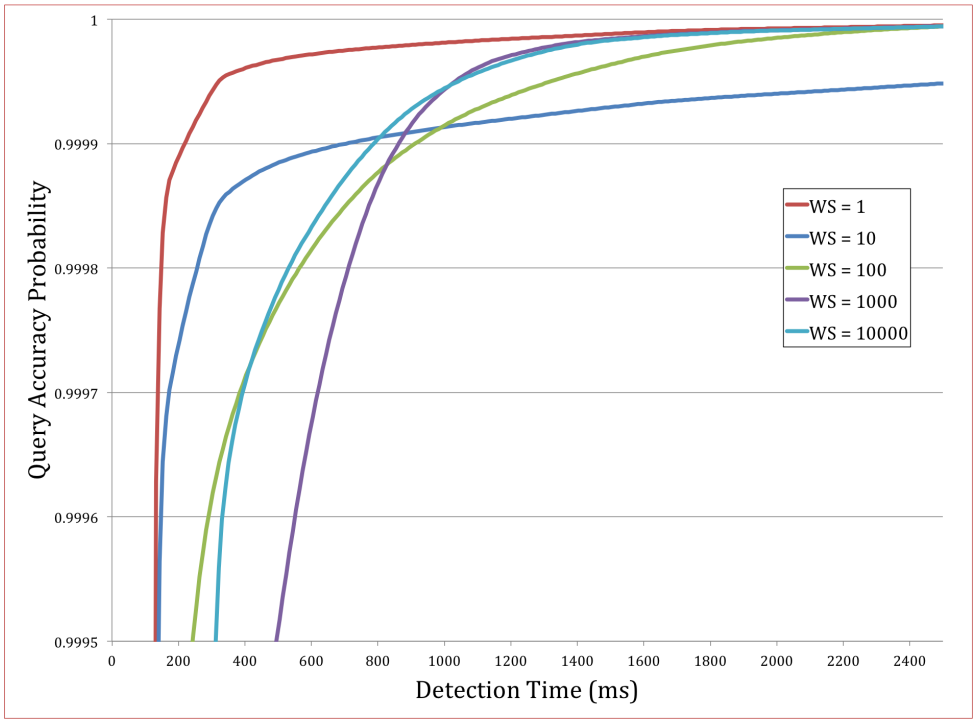


Figure 4.7: Chen's Algorithm P_A vs. T_D with different Window Sizes in a LAN scenario



4.3.2 MW-FD - Window Sizes

This experiment measures the effect of window sizes on the performance of our developed algorithm, the Multiple Windows failure detector. We set both windows from a very small size (one sample) to a very large one (10.000 samples) and measured the accuracy obtained with our failure detector when run over both WAN and LAN traces.

4.3.2.1 Experiment Results

Figure 4.8 shows the results on mistake rate T_{MR} vs. detection time T_D in the WAN scenario. T_{MR} is represented on the vertical axes, expressed in logarithmic scale, and T_D in the horizontal axes. Figure 4.9 shows the results on query accuracy probability P_A vs. detection time T_D in the same scenario. Figures 4.10 and 4.11 show the same study in our tested LAN scenario.

The results clearly show that, in terms of both T_{MR} and P_A, in both test scenarios our algorithm behaves better when using:

1) *one small window* (the smaller,the better) and

2) *one big window* (the bigger, the better).

The results also show that the performance to accuracy ratio of the algorithm decreases as the size of the small window increases, and as the size of the big window decreases. Besides, it's also noticeable from the figures that curves for tests which share the same size for the small window tend to behave similarly. This is due to the better accuracy of Chen's failure detector when using small windows (see section 4.3.2). We also conclude from this analysis that the increase in accuracy obtained by our algorithm is negligible for each value of T_D for sizes of the long term (big) window bigger than 1.000. For the small window, the experiments suggest that the best size is one (1). This captures the idea introduced in section 3.2 that one component should be reactive to very recent behavior. In this case, results show that the best is only to consider the inter-arrival time of the last message (which's value is $t_l - t_{l-1}$) for quickly reacting to changes in network conditions. Note that, at first sight, this would suggest that if there have been messages lost between the last received sample and the previous one the detection time would grow significantly. This is not the case as we use, for all algorithms implemented

Figure 4.8: MW-FD T_{MR} vs. T_D with different Window Sizes in a WAN scenario

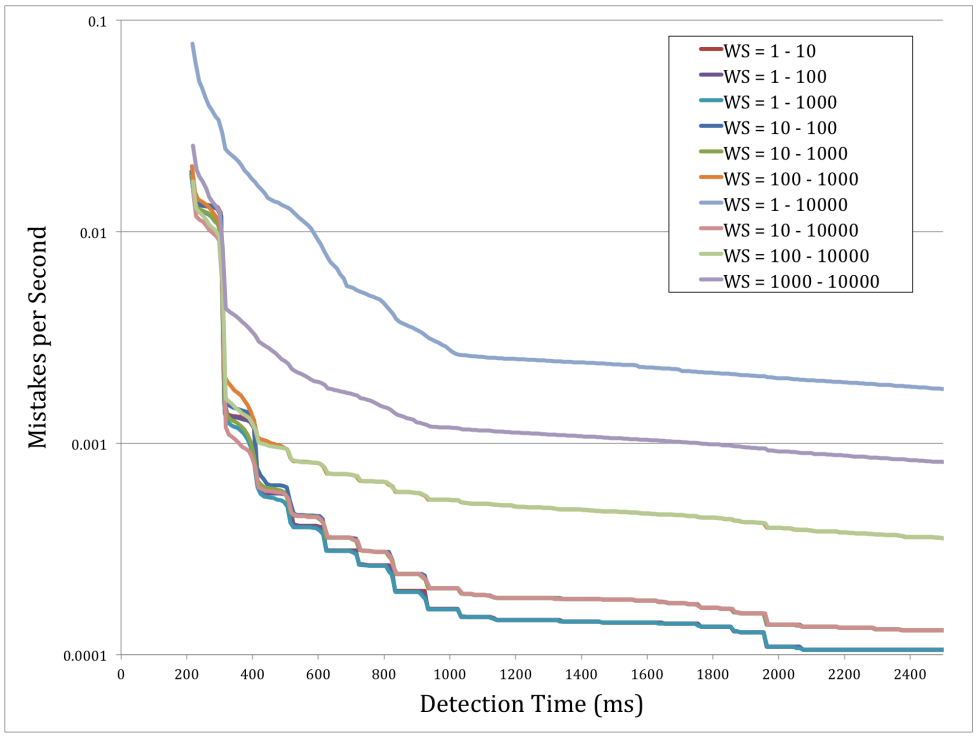


Figure 4.9: MW-FD P_A vs. T_D with different Window Sizes in a WAN scenario

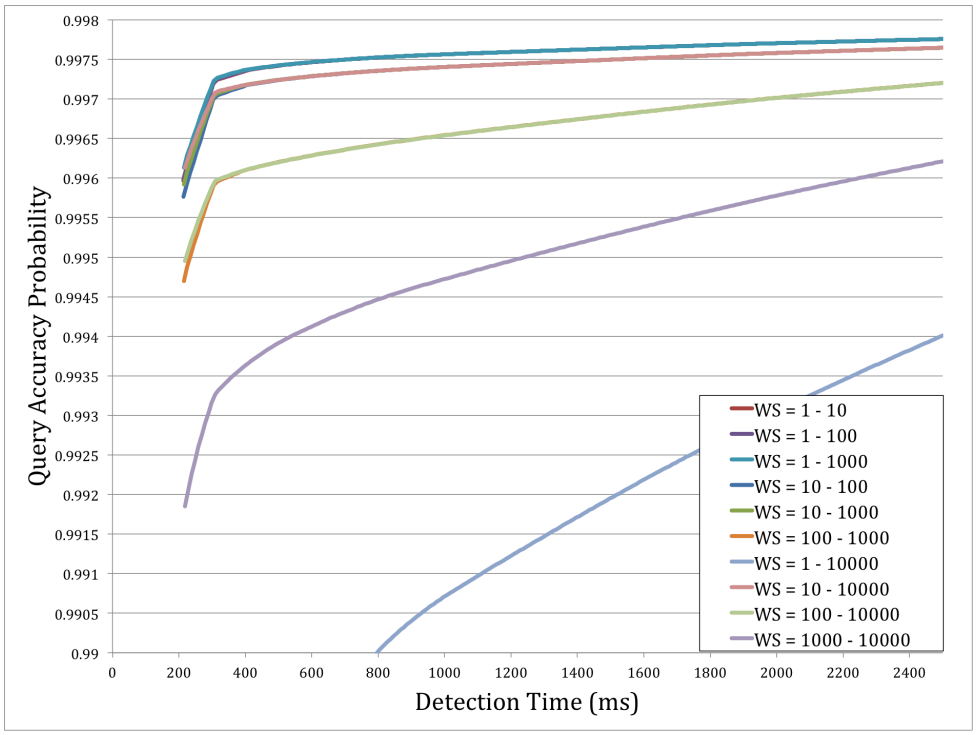


Figure 4.10: MW-FD T_{MR} vs. T_D with different Window Sizes in a LAN scenario

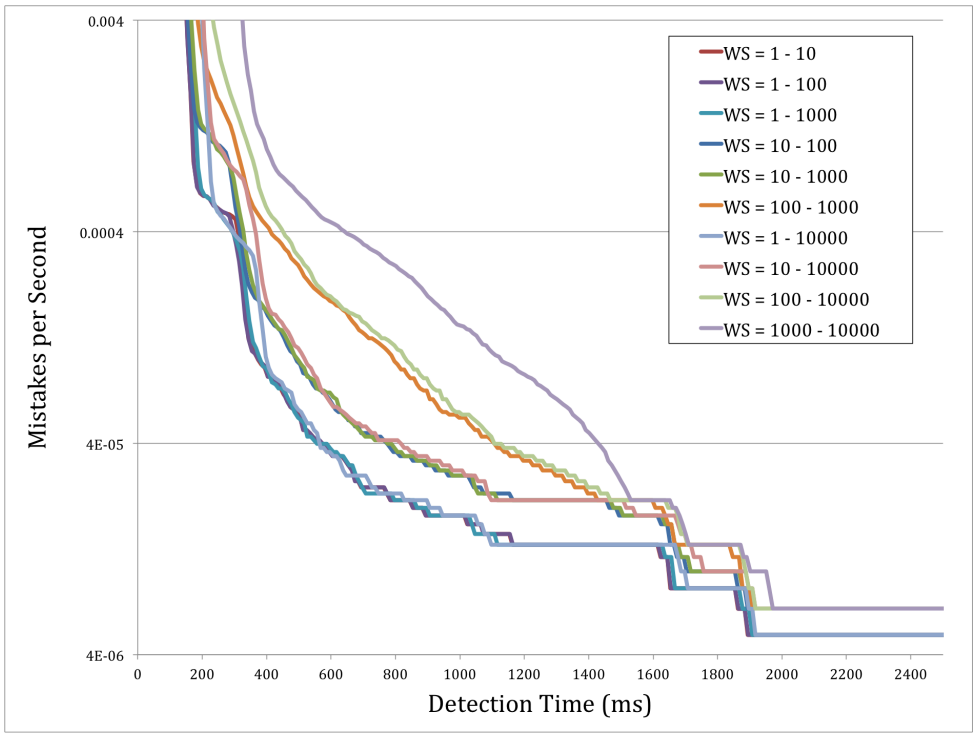
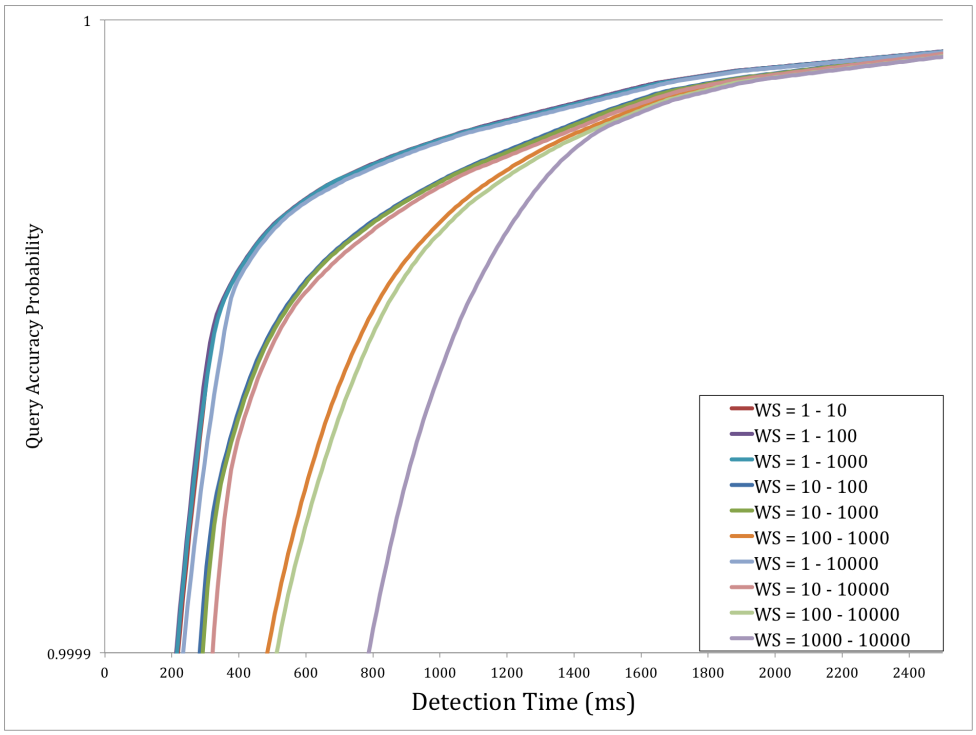


Figure 4.11: MW-FD P_A vs. T_D with different Window Sizes in a LAN scenario



in our work, the gap filling mechanism explained in section 4.1. Therefore, detection time for each sample is bounded by the maximum network delay. Finally, results suggest that the improvement obtained for long-term window sizes bigger than 1.000, the improvement obtained in performance is negligible. The best values are obtained when using windows of sizes 1 and 1.000.

4.3.3 Comparison to Other Algorithms

In this experiment, we compare the behavior of the MW-FD with four well known failure detectors. Namely, the failure detectors of Chen et al., Bertier et al., and the two accrual failure detectors; ϕ and ED FD. In this experiment we intend to show that the MW-FD presents the best detection time to accuracy ratio. Chen and the MW failure detectors share a common tuning parameter, the safety margin Δ_{to}, which we vary in our experiments to get the different values of detection time. The tuning parameter for the accrual failure detectors was the threshold Φ. Unlike the rest of the failure detectors, Bertier's has no tunning parameter. For this reason, its behavior is plotted as a single point on the figures. Finally, the values of window sizes for the rest of the failure detectors were set to:

- 1 and 1.000 for Chen's failure detector. These values were chosen as 1 is the best value according to our experiment in section 4.3.1 and 1.000 is the commonly used value used in related work experiments.
- 1000 for the ϕ and the ED failure detectors. These failure detectors benefit from using large window sizes. However, we have tested and discovered that for window sizes beyond 1.000 samples, the improvement these algorithms obtain is negligible. Furthermore, this is the value used by their authors in their respective works (Hayashibara, Defago, Yared, & Katayama 2004; Xiong, Vasilakos, Yang, Wei, Qiao, & Wu 2011).
- 1000 for Bertier's FD, as that is the value their authors use in their work (Bertier, Marin, & Sens 2002; Bertier, Marin, & Sens 2003).
- One window of size 1 and one window of size 1000 for our algorithm, the MW-FD. These values were chosen as a consequence of the results obtained in the experiments presented in section 4.3.2, which show that this is the best configuration in terms of resulting accuracy to detection time ratio.

In order to measure the detection time T_D of the accrual failure detectors, we proceeded as follows: assuming that a crash would occur exactly after successfully sending a heartbeat (worst case) we measure the time elapsed until the time a failure detector would report a suspicion. With the ϕ and the ED failure detectors, we consider the threshold and reverse the computation of the accrual values (ϕ and e_d in each respective case) to obtain the equivalent timeouts. We compute this equivalent timeout each time a new heartbeat is received and take the mean value $\Delta_{to,\Phi}$. For all algorithms, we estimated the mean propagation time Δ_{tr} based on the information about the round-trip time. Then, we have estimated the average (worst-case) detection time simply as follows:

$$D_T \approx \Delta_{tr} + \Delta_{to,\Phi} \tag{4.1}$$

for accrual failure detectors, and as:

$$D_T \approx \Delta_{tr} + \Delta_i + \Delta_{to} \tag{4.2}$$

for the rest of the failure detectors.

4.3.3.1 Experiment Results

Figure 4.12 shows the results on mistake rate T_{MR} vs. detection time T_D in the WAN scenario. T_{MR} is represented on the vertical axes, expressed in logarithmic scale, and T_D in the horizontal axes. Figure 4.13 shows a zoom to the curve in the aggressive range (where 400ms $\geq T_D$) and figure 4.14 shows the conservative range ($T_D \geq$ 400ms). Figure 4.15 shows the results on query accuracy probability P_A. vs. detection time T_D in the same scenario. Figures 4.16 and 4.17 show the same study in our tested LAN scenario.

The results indicate that all algorithms follow the same tendency. Our algorithm seems to outperform the rest in both scenarios, and in both the aggressive and conservative ranges. It presents the lowest mistake rate and the best query accuracy probability for all measured detection times. Note that in both graphs, the curve of the accrual failure detector with normal distribution (ϕ) is stopped early. This is due to the rounding error preventing the graphs to the very conservative case (Xiong, Vasilakos, Yang, Wei, Qiao, & Wu 2011). This effect can be explained as follows: when using a normal or exponential distribution for representing message delays, if the values of inter-arrival times are stable, meaning that the variance is low, the probability of receiving a message at time t tends to zero as t increases. This creates an impossibility of computing values for ϕ and e_d for large detection times in the conservative range

(see Equations 2.7-2.11).

Figure 4.12: Mistake Rate T_{MR} vs. Detection Time T_D in a WAN scenario

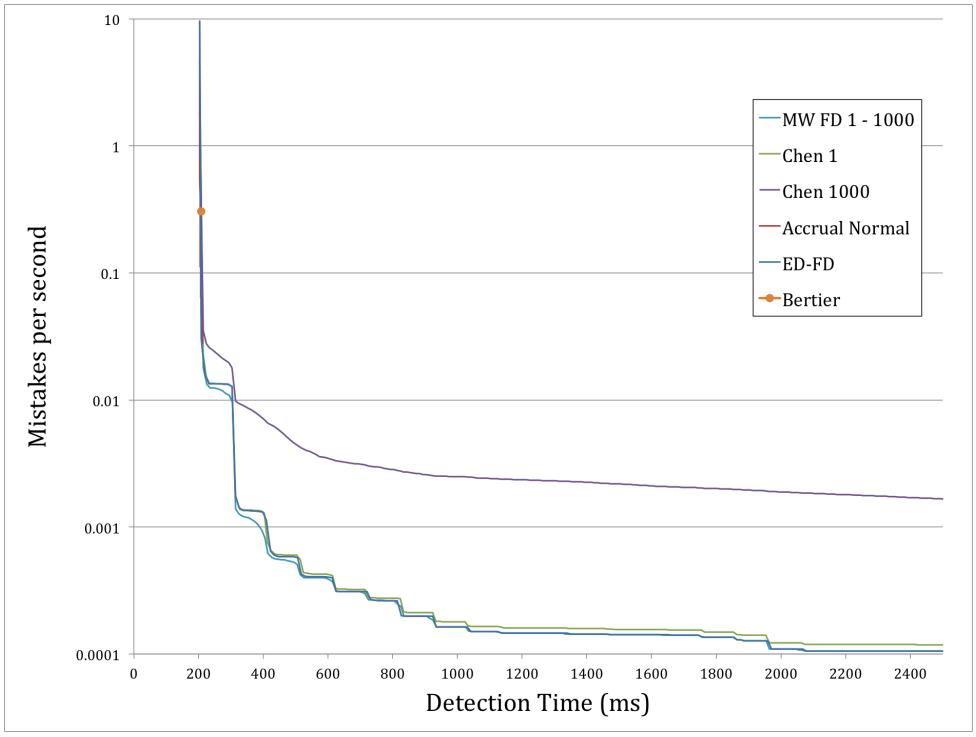


Figure 4.13: Agressive Range: T_{MR} vs. T_D in a WAN scenario

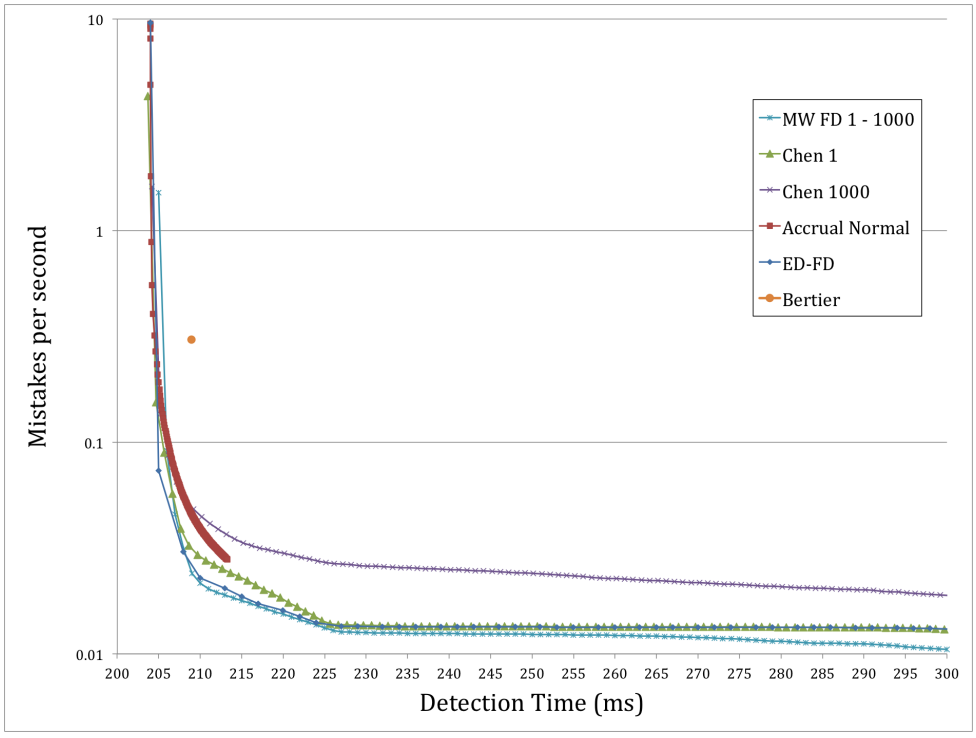


Figure 4.14: Conservative Range: T_{MR} vs. T_D in a WAN scenario

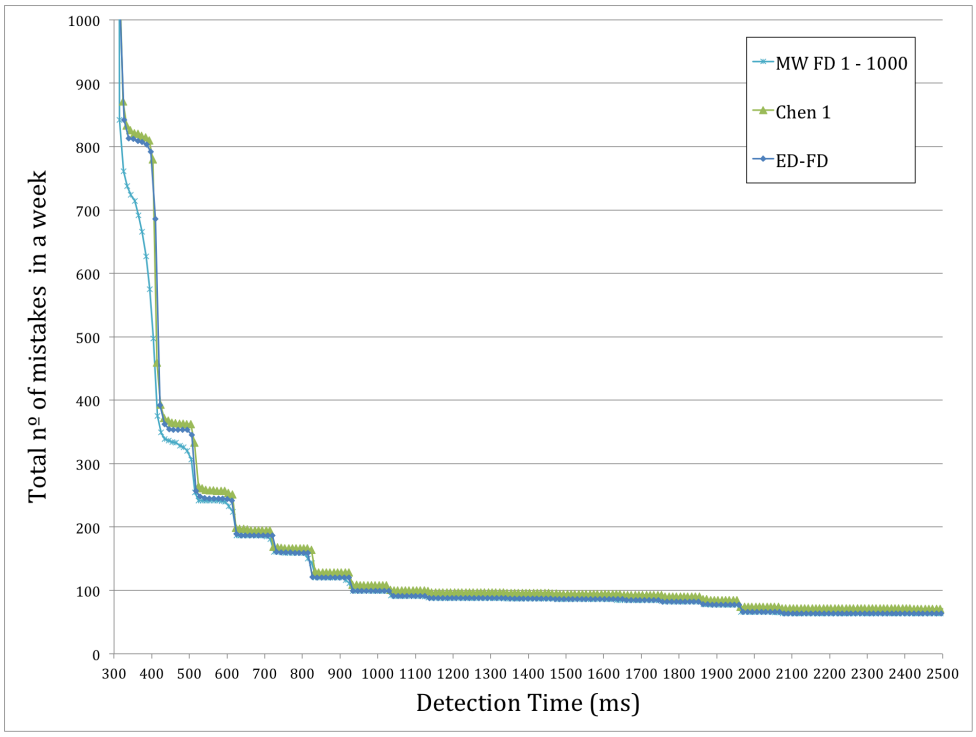


Figure 4.15: Query Accuracy Probability P_A vs. Detection Time T_D in a WAN scenario

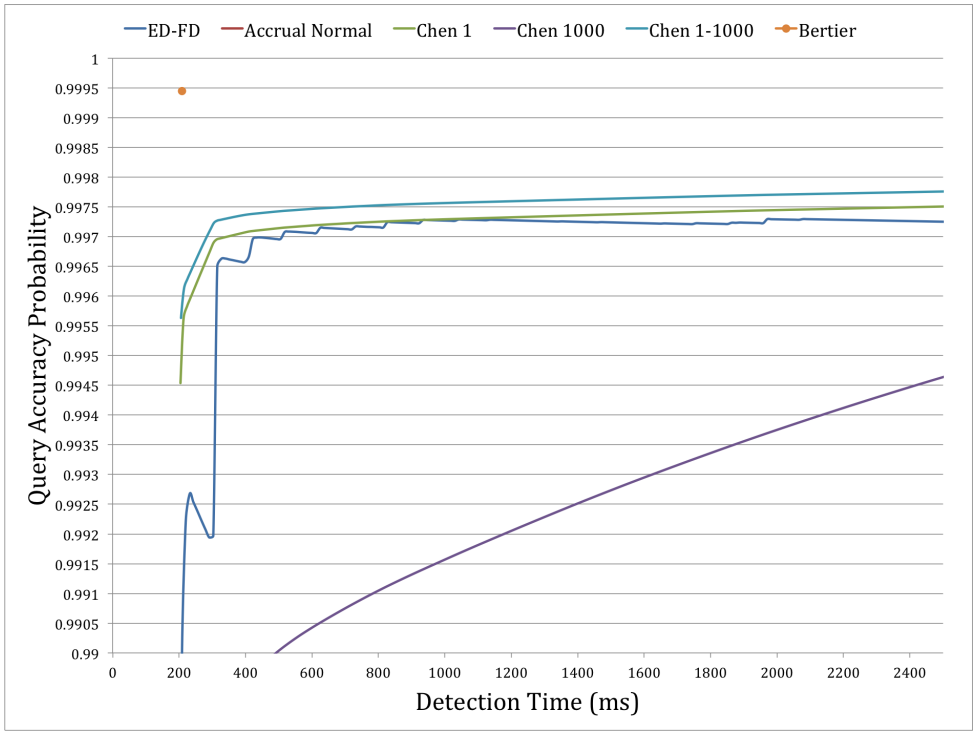


Figure 4.16: Mistake Rate T_{MR} vs. Detection Time T_D in a LAN scenario

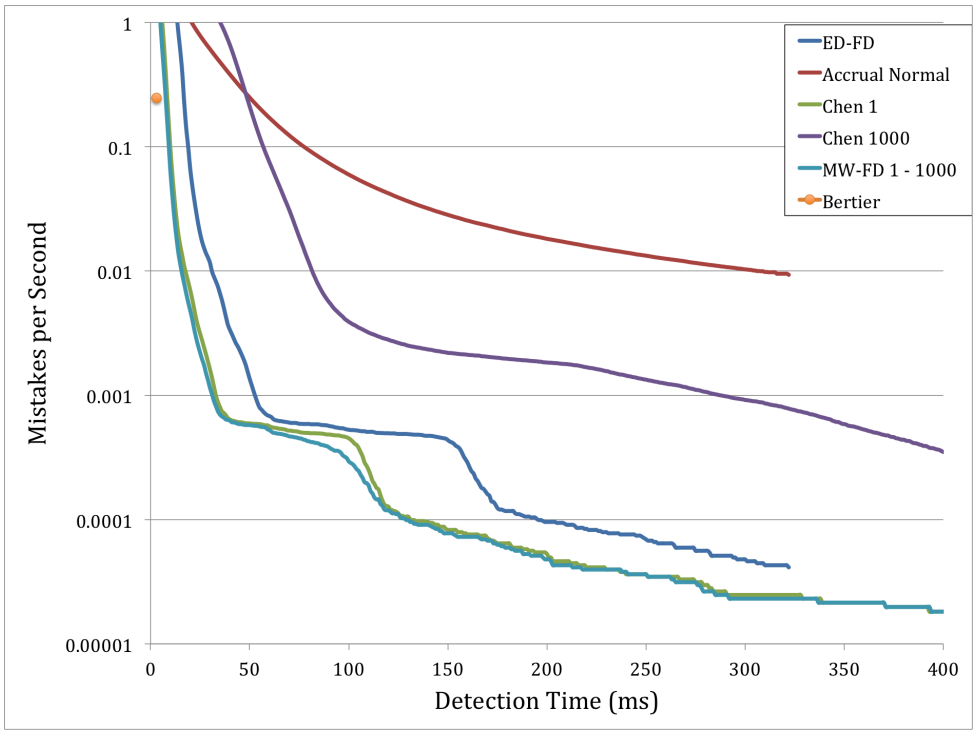
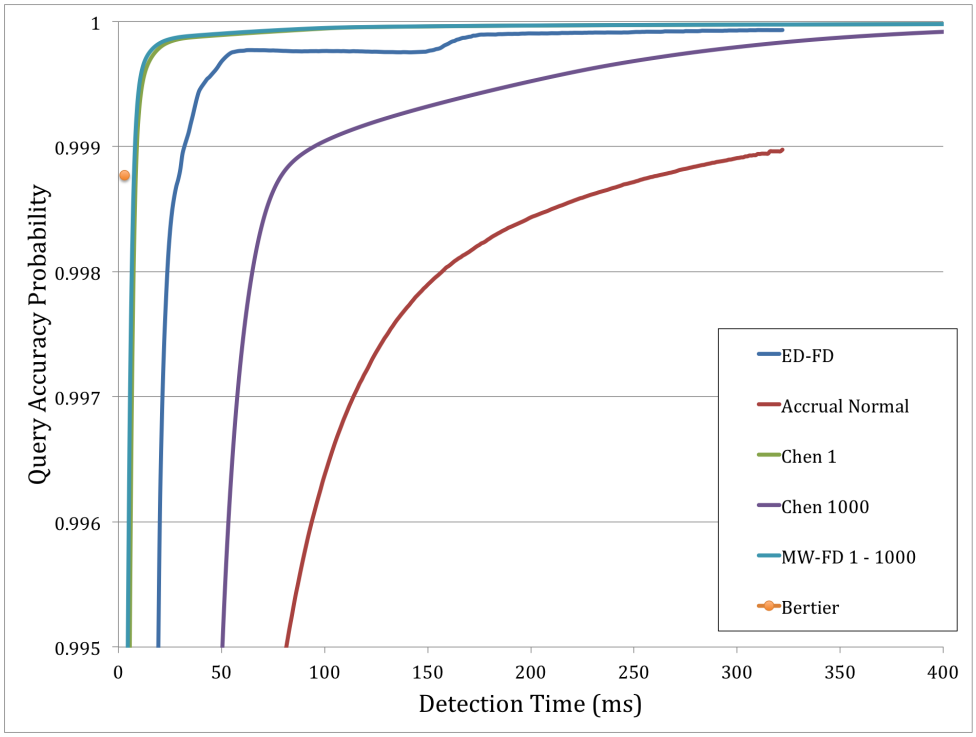


Figure 4.17: Query Accuracy Probability P_A vs. Detection Time T_D in a LAN scenario



4.3.4 Analysis for a Single Detection Time: T_D = 215ms

This experiment measures the number of mistakes each failure detector makes in the WAN scenario as time passes. We evaluate this for a unique detection time T_D=215ms. We chose this value because it makes failure detectors work in an agressive range and we are able to obtain results for most of the failure detectors. The only failure detector that can not be parametrized to obtain this T_D is Bertier's. The goal of this experiment is to further analyse the behavior of the MW-FD and to determine which conditions make this FD obtain an overall better accuracy.

Figure 4.18 shows how the total number of mistakes made by each FD increases as samples are analysed. The total number of mistakes is represented on the vertical axes and the progressive number of analysed samples on the horizontal axes (5.845.712 in total).

4.3.4.1 Experiment Results

From the results observed in figure 4.18, we split the total sample space into four subsamples, as Table 4.1 shows. First, there is one stable period that we call *Stable 1*, which starts from the beginning of the sample and lasts until the moment before the big mistake burst in the middle of the graph (on the horizontal axes) starts. After this period, there is the *Burst* period and then, the *Worm* period that coincides with the *W32/Netsky.T@mm* Internet worm (see section 4.2.1). The last *Stable 2* period starts after the *Worm* period, and lasts until the end of the sample.

Name	From Sample	To Sample
Stable	1	2900000
Burst	2900001	2930000
Worm Period	2930001	4860000
Stable 2	4860001	5845712

Table 4.1: Division of the WAN sample into smaller subsamples

Figure 4.19 shows the total number of mistakes each failure detector made on each sub-period and Figure 4.20 shows the ratio of mistake reduction obtained when using the MW FD, compared the rest of the algorithms. The results show that the MW-FD performs better in all scenarios, but particularly better during the *Burst* period. This is consequent with the algorithm's goal.

Figure 4.18: WAN: Cummulative Mistakes for fixed T_D=215 ms

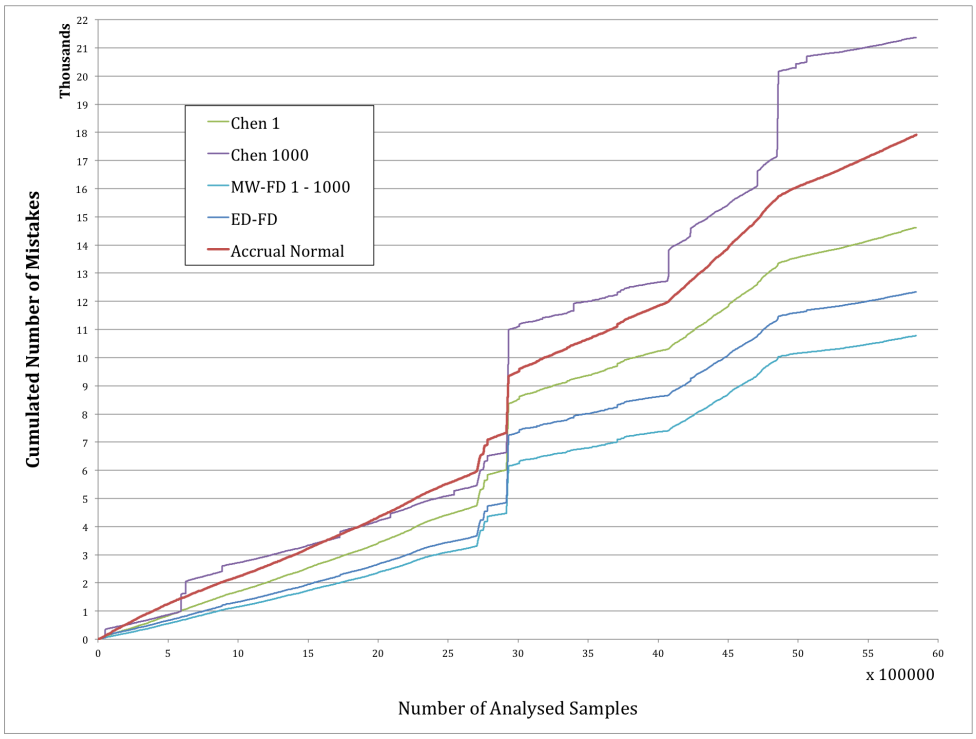


Figure 4.19: WAN: Total Mistakes made during each subsample for fixed T_D=215 ms

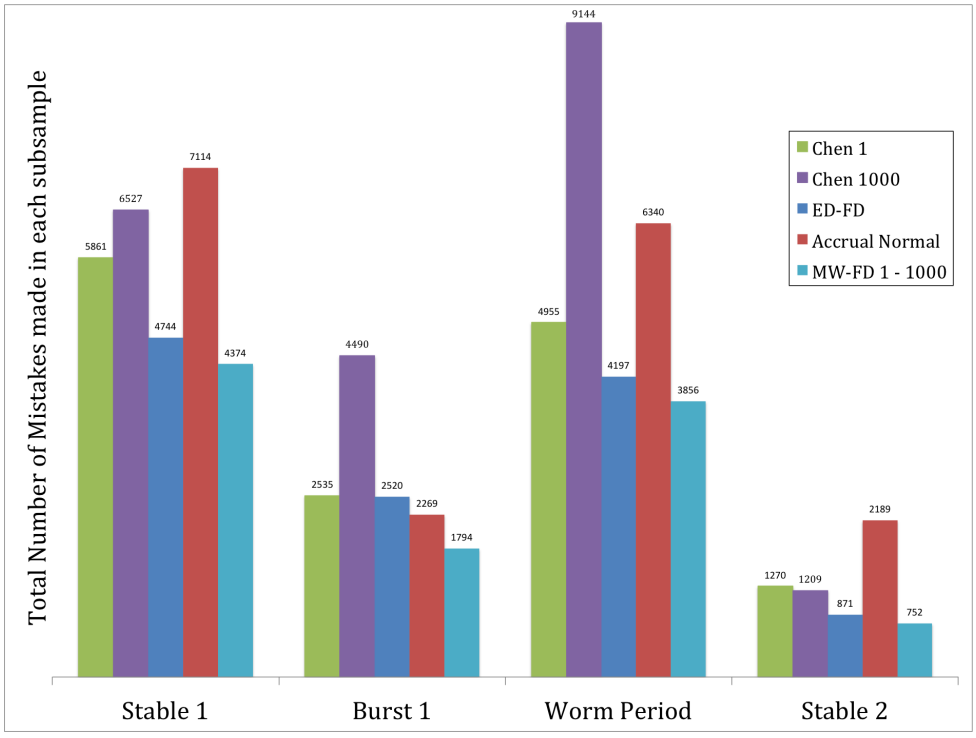
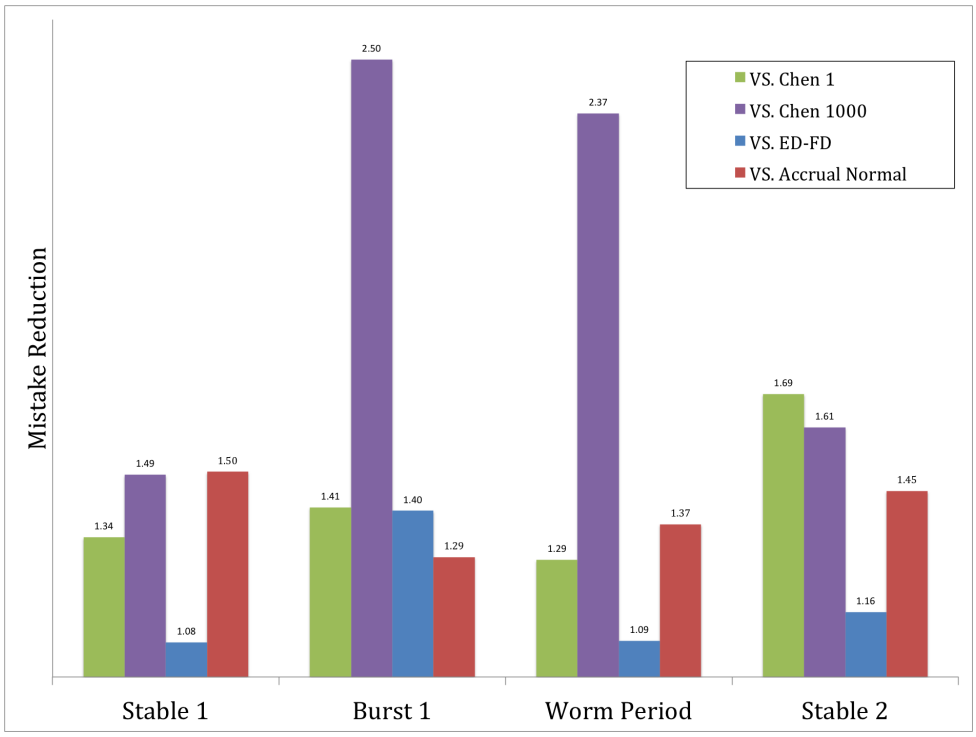


Figure 4.20: WAN: Mistake Reducion when comparing MW-FD to other algorithms T_D=215 ms



4.3.5 MW-FD vs. Chen's FD

4.3.5.1 Window Usage

When using the MW-FD, two windows are used to compute the estimation of the next freshness point τ_{i+1}. Table 4.2 shows the percent of times the estimation of each window is used in both (LAN and WAN) scenarios to compute the expected arrival time of an the next heartbeat message. The *Indistinct* column represents the cases where the estimation estimated by both windows results in the same value, when computed using millisecond precision.

Surprisingly, in the WAN scenario, most of the times, both windows give the same estimation of the expected arrival time for the next message (71.62% of the times). Furthermore, when that is not the case, the big window gives the most tolerant estimation with higher frequency than the smaller one. In the LAN scenario, both widows are similarly used. The big window presents the maximum usage. This dominance of the usage of the big window happens possibly due to the fact that normally, most messages exchanged in a network present a small delay, close to the mean. Therefore, in most of the cases, using a window of small size would cause the algorithm to expect the arrival of the next heartbeat after a short time. When using a big window, if there

are messages that have been slow in the past and their inter-arrival times are still contained in the window, they will continue to influence the expectation of the arrival of the next message for some time, preventing the overall estimation of the algorithm from being very aggressive.

-	Total Samples	Big Window	Small Window	Indistinct
WAN	100%	24.05%	4.33%	71.62%
LAN	100%	51.55%	44.09 %	4.36%

Table 4.2: Window Usage for different scenarios

4.3.5.2 Mistake Reduction

As introduced in Equation 3.2 (see section 3.3), given two window sizes WS_1 and WS_2, the MW-FD makes less mistakes than Chen's algorithm when using any of WS_1 or WS_2. In this section we ilustrate this idea with an example in the WAN scenario, where T_D=215ms, $WS_1 = 1$ and $WS_2 = 1000$. We compare *MW-FD(*WS_1, WS_2*)* to *Chen-FD(*WS_1*)* and *Chen-FD(*WS_2). Figure 4.21 shows which mistakes each failure detector make. One can note the effect of the use of two windows as the picture clearly shows that *MW-FD(*WS_1, WS_2*)* only makes the mistakes that both *Chen-FD(*WS_1*)* and *Chen-FD(*WS_2*)* make.

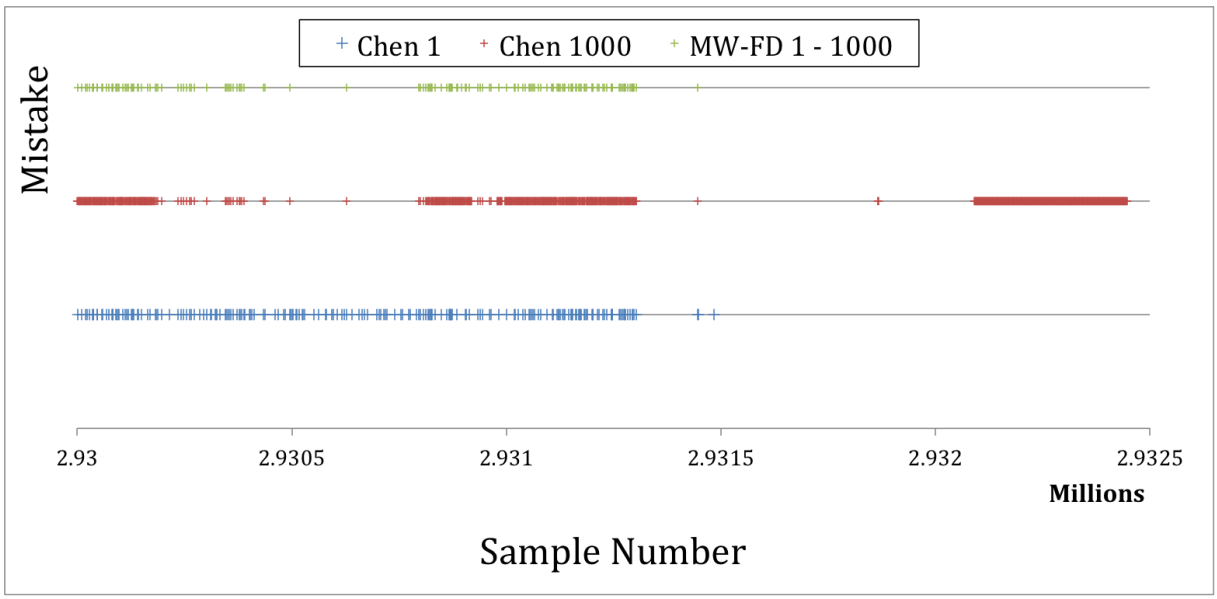


Figure 4.21: Mistakes made by Chen and MW-FD, WAN scenario

5 Towards Failure Detection as a Service

In this chapter, we study the possibility of providing a single failure detection service to multiple applications or virtual machines running on a single (same) physical computer. Each of these applications (or VMs) present their own, and probably different, requirements on failure detection QoS. In particular, we study how to combine these requirements in order to send a single heartbeat per physical machine (see Figure **??**). The goal of this analysis is to provide each application with the illusion of a dedicated failure detector that fulfills its particular requirements in terms of QoS while minimizing the load imposed on the network.

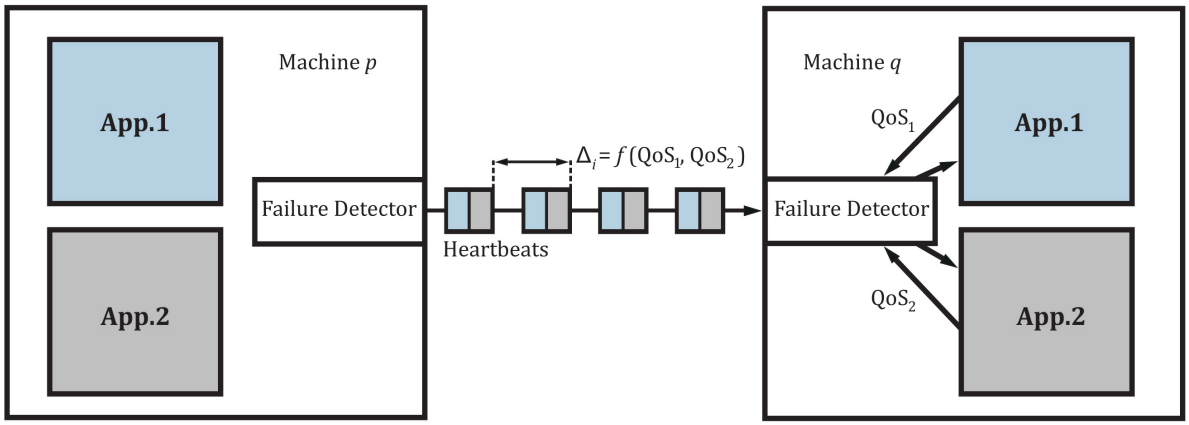


Figure 5.1: Multiple Applications using a single FD service per physical machine

5.1 Background

5.1.1 Configuring a FD to Satisfy a QoS Specification

Chen et al. (Chen, Toueg, & Aguilera 2002) introduced a mechanism for applications to express their QoS requirements and obtain, for specific network conditions, the rate Δ_i at which a monitored machine (p in our model) should send heartbeats to a monitoring machine (q) and

the timeout time Δ_{to} this machine should wait for a new heartbeat before it starts suspecting p, that make the algorithm satisfy the required QoS.

The QoS requirements are assumed to be expressed as a tuple (T_D^U, T_{MR}^U, T_M^U), where T_D^U is an upper bound on the detection time, T_{MR}^U is an upper bound on average mistake rate (or, equivalently, a lower bound on the average mistake recurrence time), and T_M^U is an upper bound on the average mistake duration. In other words, the failure detector should provide to the application, a tuple (T_D, T_{MR}, T_M) that satisfies:

- $T_D \leq T_D^U$
- $T_{MR} \leq T_{MR}^U$
- $T_M \leq T_M^U$

Given a system where the probability of losing a message is p_L and the distribution of message delays is $Pr(D \leq x)$, the failure detector should provide the following properties:

1) The detection time is bounded as follows:

$$T_D \leq \Delta_i + \Delta_{to} \tag{5.1}$$

2) The average mistake rate is:

$$E(T_{MR}) = \frac{\Delta_i}{p_S} \tag{5.2}$$

3) The average mistake duration is:

$$E(T_M) = \frac{\int_0^{\Delta_i} u(x)dx}{p_S} \tag{5.3}$$

where p_S is the probability of an *S-transition* occurring at time τ_i, for any $i \geq 2$, and $u(x)$ is the probability that q suspects p at time $\tau_i + x$, for $x \in [0, \Delta_i)$.

From these definitions and conditions, Chen et al. introduced a configuration mechanism based on a programing problem that can be solved by using a numerical method. This mechanism is used to configure their algorithm, and ours, in order to meet QoS requirements. The explanation of how this problem was formulated can be found in (Chen, Toueg, & Aguilera 2002). The configuration procedure takes as inputs :

1) the QoS requirements. Namely, the tuple (T_D^U, T_{MR}^U, T_M^U) , and

2) the probabilistic behavior of the heartbeat messages. Namely p_L, the probability of a message being dropped by the network; and $V(D)$, the variance of message delays.

Then, it outputs the failure detector parameters Δ_i and Δ_{to} so that the failure detector satisfies the application's specific QoS requirements. Furthermore, to minimize the network bandwidth used, the configuration procedure finds the largest inter-sending interval $\Delta_{i,max}$ that satisfies these QoS requirements. It works as follows (Chen, Toueg, & Aguilera 2002):

- *Step 1*: Compute
$$\gamma' = \frac{(1-p_L)(D_T^U)^2}{V(D)+(D_T^U)^2} \tag{5.4}$$
and let
$$\Delta_{i,max} = min(\gamma' \dot{D}_T^U, T_M^U). \tag{5.5}$$
If $\Delta_{i,max} = 0$, then the QoS cannot be achieved and stop; else continue.

- *Step 2*: Let
$$f(\Delta_i) = \Delta_i \prod_{j=1}^{\lceil T_D^U/\Delta_i \rceil - 1} \frac{V(D)+(T_D^U - j\Delta_i)^2}{V(D)+p_L(T_D^U - j\Delta_i)^2} \tag{5.6}$$
Find the largest $\Delta_i \leq \Delta_{i,max}$ such that $f(\Delta_i) \leq T_{MR}^U$. Such an Δ_i always exist and can be computed using a numerical method.

- *Step 3*: Set $\Delta_{to} = T_D^U - \Delta_i$ and output Δ_i and Δ_{to}

Figure **??** illustrates the configuration process described in this section. The information output by this process, plus the expected arrival times EAs (computed as explained in section 2.5.1.1) are used by the MW-FD (or Chen's FD) for estimating expected arrival times. Finally, note that it is possible to run the configuration procedure periodically in order to make the algorithm adaptive to changes in the probabilistic behavior of the network.

5.1.1.1 Estimating p_L and $V(D)$

It is easy to estimate p_L and $V(D)$ using heartbeat messages; p_L can be estimated by using the sequence numbers of the heartbeat messages to count the number of missing heartbeats and then dividing this count by the highest sequence number received so far. To estimate $V(D)$, the mechanism is as follows; when p sends a heartbeat m, it timestamps m with the sending time S and, when q receives m, q records the receipt time A. In this way, $A-S$ is the delay of m. Then

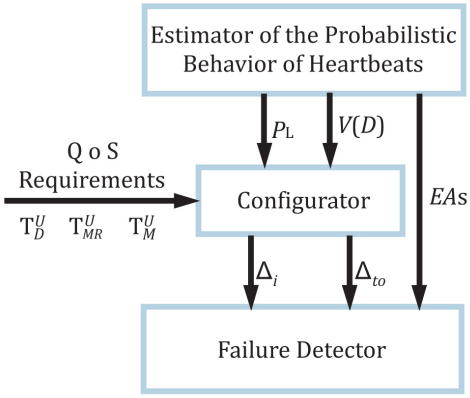


Figure 5.2: Meeting QoS Requirements

the average and variance of $A-S$ are computed for multiple past heartbeat messages, obtaining accurate estimates for $V(D)$. This process works even though the clocks are not synchronized as the procedure estimates $V(D)$ by computing the variance of $A-S$ of multiple heartbeat messages, where A is the time (with respect to q's local clock) when q receives m and S is the time (with respect to p's local clock) when p sends m. When clocks are not synchronized, $A-S$ is the delay of m plus the skew between the clocks of p and q. Thus, the variance of $A-S$ is the same as the variance $V(D)$ of message delays.

5.2 Existing Solutions

As presented in the previous section, Chen et. al (Chen, Toueg, & Aguilera 2002) introduced the explained mechanism for their developed failure detector to meet application specific QoS requirements. Nevertheless, their work does not address the issue of adapting the heartbeat sending interval and timeout in the presence of many applications or virtual machines running on the same physical machine, sharing a single failure detection service.

Accrual failure detectors (Hayashibara, Defago, Yared, & Katayama 2004; Xiong, Vasilakos, Yang, Wei, Qiao, & Wu 2011), as introduced in sections 2.5.3 and 2.5.4, do not output an interpreted result ($Trust$ or $Suspect$). Instead, they output a value on a continue scale that indicates the probability that a monitored process has crashed. Applications using such failure detectors can set different thresholds (maybe multiple thresholds per application) on the suspicion level that would make them trigger some action. Nevertheless, we intend our QoS specification to be expressed in a higher level. Namely, as a tuple (T_D^U, T_{MR}^U, T_M^U). Furthermore, our devel-

oped algorithm, which presents the best performance vs. accuracy ratio (see section 4.3.3) can be parametrized with the configuration procedure described in the previous section, as it is a variation of Chen's algorithm.

Deianov et al. (Deianov & Toueg 2000) introduced a failure detection service based on Chen's FD for multiple applications running on a single machine. Nevertheless, in their work, they do not explain how to adapt the heartbeat interval and timeout in the presence of different QoS provided by applications.

Bertier et al. (Bertier, Marin, & Sens 2002; Bertier, Marin, & Sens 2003) introduced the failure detector presented in section 2.5.2. In their work, they express that they use, in their implementation, an adaptation layer to fulfill the requirements of different applications with different QoS specifications. Despite of this claim, they do not explain how this is achieved.

5.3 Determining Δ_i and Δ_{to} from Multiple QoS Requirements

In this section, we explain how to combine the QoS requirements of multiple applications running on a single physical computer in order to determine the Δ_i and Δ_{to} values for the heartbeats of a physical machine running a failure detection service for multiple applications.

5.3.1 Measuring the Impact of Single Parameters on Δ_i and Δ_{to}

We propose that applications express their QoS requirements as proposed by Chen et al. (Chen, Toueg, & Aguilera 2002), as a tuple (T_D^U, T_{MR}^U, T_M^U). In this section we study how the variation of each of the parameters in the tuple affects the resulting Δ_i and Δ_{to}.

5.3.1.1 Varying Detection Time

Figure 5.3 shows the impact of varying the detection time T_D on the resulting Δ_i and Δ_{to}, computed by the mechanism introduced in section 5.1.1. The graph shows that, as T_D grows, both Δ_i and Δ_{to} grow linearly. This is obvious as $T_D = \Delta_i + \Delta_{to}$. The ratio at which they grow is determined by the remaining QoS parameters.

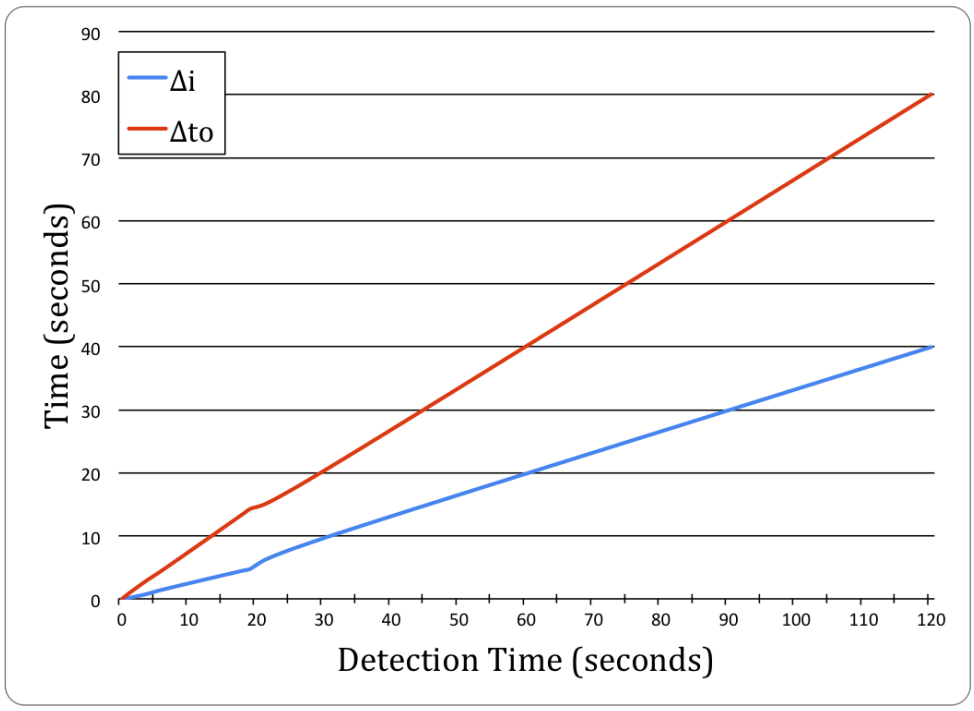


Figure 5.3: Varying Detection Time, for T_{MR} = 1/month — T_M = 5s.

5.3.1.2 Varying Mistake Rate

Figure 5.4 shows the impact of varying the detection time T_{MR} on the resulting Δ_i and Δ_{to}. The graph shows that, as T_{MR} grows, Δ_i decreases and Δ_{to} grows. This happens as the requirement becomes more constrained when T_{MR} grows (less mistakes are allowed). After a certain point (in this example, when T_{MR}=56s), Δ_i and Δ_{to} remain constant. The ratio at which Δ_i and Δ_{to} vary, as the point where they start remaining constant is determined by the remaining QoS parameters.

5.3.1.3 Varying Mistake Duration

Figure 5.5 shows the impact of varying the detection time T_M on the resulting Δ_i and Δ_{to}. The graph shows that, as T_M grows, Δ_i grows and Δ_{to} decreases. This happens as the requirement becomes more relaxed when T_M grows. After a certain point (in this example, when T_M=10s), Δ_i and Δ_{to} remain constant. The ratio at which Δ_i and Δ_{to} vary, as the point where they start remaining constant is determined by the remaining QoS parameters.

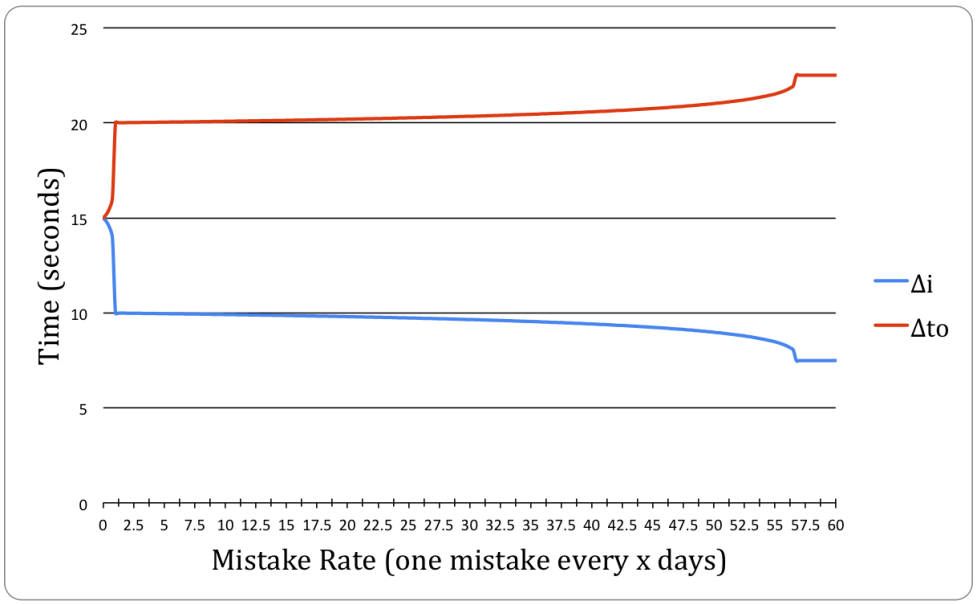


Figure 5.4: Varying Mistake Rate, for $T_D = 30$s — $T_M = 5$s.

5.4 Δ_i and Δ_{to} for multiple QoS

In this section we propose a solution for adapting Δ_i and Δ_{to} to multiple applications with different QoS requirements running on the same physical computer while minimizing the number of sent messages through the network (see Figure **??**). Considering n applications or VMs running on a single physical machine, the procedure works as follows:

- *Step 1:* For each application app_j, where $j = 1, ..., n$, input the QoS requirements tuple and compute $\Delta_{i,j}$ and $\Delta_{to,j}$ using Chen's equation (introduced in section 5.1.1).
- *Step 2:* From all computed $\Delta_{i,j}$, compute $\Delta_{i,min} = min(\Delta_{i,1}, ..., \Delta_{i,n})$ and use it for the physical machine (and for every app_j).
- *Step 3:* Use, for each app_j, $\Delta_{to,j} = T_{D,j} - \Delta_{i,min}$, where $T_{D,j}$ is the detection time of app_j and $\Delta_{i,min}$ is the minimum of the heartbeat intervals computed in step 1.
- *Step 4:* The FD service uses $\Delta_{i,min}$ for sending heartbeats and computes freshness points $\tau_{i,j}$ differently for each app_j by using each $\Delta_{to,j}$ and timeout differently for each app_j.

5.4.1 Consequences on the QoS of different applications

By using this mechanism, for each app_j:

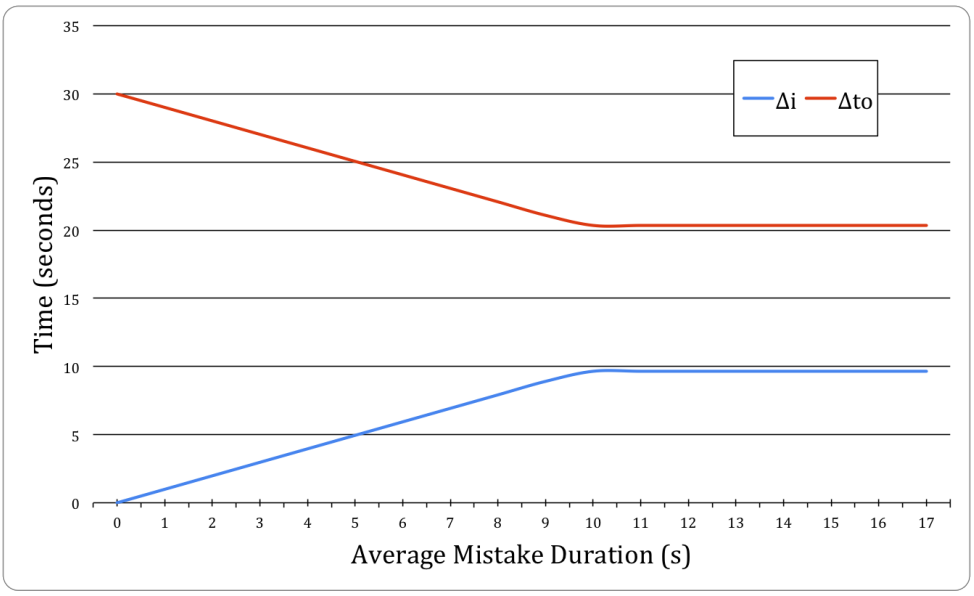


Figure 5.5: Varying Mistake Duration, for $T_{MR} = 1/\text{month}$ — $T_D = 30\text{s}$.

- the detection time is maintained, exactly, as $T_D = \Delta_i + \Delta_{to}$,
- the applications that obtain a modified (adapted) QoS reduce their mistake rate. This happens because, when decreasing the Δ_i and increasing Δ_{to}, the T_{MR} is reduced as a consequence of the illustrated in figure 5.4, and
- the applications that obtain a modified QoS also reduce their mistake duration. This happens because, when decreasing the Δ_i and increasing Δ_{to}, the T_{MR} is reduced as a consequence of the illustrated in figure 5.5

As a conclusion, using the shortest heartbeat interval and adapting the timeout to meet exactly the detection time required by each application improves the QoS of the *adapted* applications (namely, the applications which do not present $\Delta_{i,min}$) in terms of mistake duration and mistake rate and, therefore, this mechanism maintains of improves the overall failure detection QoS that a single service provides to all applications. Furthermore, network traffic is reduced from the case of using a single failure detector per application, because in that case, for each app_j a heartbeat should be sent every $\Delta_{i,j}$.

In future work, we would like to experimentally evaluate the use of this technique, in terms of resulting QoS as well as in reduction of network traffic used for failure detection.

6 Conclusions and Future Work

In this work, we introduced the Multiple Windows Failure Detector (MW-FD), an algorithm for failure detection in distributed systems which provides QoS guarantees. Our experiments in both WAN and LAN scenarios indicate that our failure detector presents a better QoS when comparing to existing FD algorithms. Namely, it reduces the number of mistakes (false detections) made per unit of time. The difference varies in magnitude according to network conditions and test environments.

We have also studied the idea of multiple applications or virtual machines, with different QoS requirements in terms of failure detection, running on a single host using a single FD as a shared service. We have proposed a mechanism for adapting heartbeat inter-sending rates and timeout values of applications coexisting in a single physical machine that use a single failure detection mechanism. Our proposed idea implies that when using a shared failure detection service, applications with weaker QoS requirements benefit from the ones with stronger ones by obtaining an improved QoS. Furthermore, the overall load imposed on the network is reduced when using the shared service, compared to the case of utilising a failure detector per application.

An empirical analysis on resulting QoS of applications using the service as well as a study on how network traffic is reduced by using this approach are possible directions of future work.

References

(1989). Requirements for internet hosts - communication layers.

Aguilera, M. K., W. Chen, & S. Toueg (1999, June). Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theor. Comput. Sci. 220*(1), 3–30.

Aguilera, M. K., W. Chen, & S. Toueg (2000, April). Failure detection and consensus in the crash-recovery model. *Distrib. Comput. 13*(2), 99–125.

Aguilera., M. K., W. Chen, & S. Toueg (2000, April). On quiescent reliable communication. *SIAM J. Comput. 29*(6), 2040–2073.

Aguilera, M. K. & M. Walfish (2009). No time for asynchrony. In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, Berkeley, CA, USA, pp. 3–3. USENIX Association.

Attiya, H., A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, & R. Reischuk (1987). Achievable cases in an asynchronous environment. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pp. 337–346.

Babaoglu, O., M. G. Baker, R. Davoli, & L. A. Giachini (1994). Relacs: A communications infrastructure for constructing reliable applications in large-scale distributed systems. Technical report.

Bahl, P., R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, & M. Zhang (2007, August). Towards highly reliable enterprise network services via inference of multi-level dependencies. *SIGCOMM Comput. Commun. Rev. 37*(4), 13–24.

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

Bertier, M., O. Marin, & P. Sens (2003). Performance analysis of a hierarchical failure detector. In *In Proceedings of the International Conference on Dependable Systems and Networks*, pp. 635–644.

Birman, K., A. Schiper, & P. Stephenson (1991, August). Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst. 9*(3), 272–314.

Birman, K. P. & R. V. Renesse (1994). *Reliable Distributed Computing with the ISIS Toolkit*. Los Alamitos, CA, USA: IEEE Computer Society Press.

Bondavalli, A., S. Chiaradonna, F. DiGiandomenico, & F. Grandoni (1996). Discriminating fault rate and persistency to improve fault treatment. Technical report, Paris, France, France.

Bridgland, M. F. & R. J. Watro (1987). Fault-tolerant decision making in totally asynchronous distributed systems. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, New York, NY, USA, pp. 52–63. ACM.

Burrows, M. (2006). The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, Berkeley, CA, USA, pp. 335–350. USENIX Association.

Candea, G., J. Cutler, & A. Fox (2004, March). Improving availability with recursive microreboots: a soft-state system case study. *Perform. Eval. 56*(1-4), 213–248.

Candea, G., S. Kawamoto, Y. Fujiki, G. Friedman, & A. Fox (2004). Microreboot — a technique for cheap recovery. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, pp. 3–3. USENIX Association.

Casimiro, A. & P. Veríssimo (2001, October). Using the timely computing base for dependable qos adaptation. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, New Orleans, USA, pp. 208–217. IEEE Computer Society Press.

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

Chandra, T. D., V. Hadzilacos, S. Toueg, & B. Charron-Bost (1996). On the impossibility of group membership. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, New York, NY, USA, pp. 322–330. ACM.

Chandra, T. D. & S. Toueg (1996, March). Unreliable failure detectors for reliable distributed systems. *J. ACM 43*(2), 225–267.

Charron-Bost, B., X. Défago, & A. Schiper (2002). Broadcasting messages in fault-tolerant distributed systems: The benefit of handling input-triggered and output-triggered suspicions differently. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, SRDS '02, Washington, DC, USA, pp. 244–. IEEE Computer Society.

Chen, W., S. Toueg, & M. K. Aguilera (2002, May). On the quality of service of failure detectors. *IEEE Trans. Comput. 51*(5), 561–580.

Correia, M., N. F. Neves, & P. Veríssimo (2006, January). From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *Comput. J. 49*(1), 82–96.

Cully, B., G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, & A. Warfield (2008). Remus: high availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, Berkeley, CA, USA, pp. 161–174. USENIX Association.

Deianov, B. & S. Toueg (2000, June). Failure detector service for dependable computing. *Proc, 2000 Int'l Conf. Dependable Systems and Neteworks*, B14–B15.

Delporte-Gallet, C., H. Fauconnier, & R. Guerraoui (2002). A realistic look at failure detectors. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, Washington, DC, USA, pp. 345–353. IEEE Computer Society.

Dolev, D., C. Dwork, & L. Stockmeyer (1987, January). On the minimal synchronism needed for distributed consensus. *J. ACM 34*(1), 77–97.

Dolev, D., N. A. Lynch, S. S. Pinter, E. W. Stark, & W. E. Weihl (1986, May). Reaching approximate agreement in the presence of faults. *J. ACM 33*(3), 499–516.

Dolev, D. & D. Malki (1996, April). The transis approach to high availability cluster communication. *Commun. ACM 39*(4), 64–70.

Dwork, C., N. Lynch, & L. Stockmeyer (1988, April). Consensus in the presence of partial synchrony. *J. ACM 35*(2), 288–323.

Fakhouri, S. A., G. Goldszmidt, M. Kalantar, J. A. Pershing, & I. Gupta (2001). Gulfstream - a system for dynamic topology management in multi-domain server farms. In *Proceedings of the 3rd IEEE International Conference on Cluster Computing*, CLUSTER '01,

Washington, DC, USA, pp. 55–. IEEE Computer Society.

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

Guerraoui, R. & L. Rodrigues (2006). *Introduction to Reliable Distributed Programming*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Gupta, I., T. D. Chandra, & G. S. Goldszmidt (2001). On scalable and efficient distributed failure detectors. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, PODC '01, New York, NY, USA, pp. 170–179. ACM.

Hayashibara, N., X. Defago, R. Yared, & T. Katayama (2004). The phi; accrual failure detector. In *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*, pp. 66–78.

JAIST. http://ddsg.jaist.ac.jp/en/jst/data.html.

John, J. P., E. Katz-Bassett, A. Krishnamurthy, T. Anderson, & A. Venkataramani (2008). Consensus routing: the internet as a distributed system. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, Berkeley, CA, USA, pp. 351–364. USENIX Association.

Lamport, L., R. Shostak, & M. Pease (1982, July). The byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4*(3), 382–401.

Lee, E. K. & C. A. Thekkath (1996, September). Petal: distributed virtual disks. *SIGOPS Oper. Syst. Rev. 30*(5), 84–92.

Lischka, J. & H. Karl (2009). A virtual network mapping algorithm based on subgraph isomorphism detection. In *Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, VISA '09, New York, NY, USA, pp. 81–88. ACM.

Milosevic, Z., M. Hutle, & A. Schiper (2011). On the reduction of atomic broadcast to consensus with byzantine faults. In *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems*, SRDS '11, Washington, DC, USA, pp. 235–244. IEEE Computer Society.

Moser, L. E., P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, & C. A. Lingley-Papadopoulos (1996, April). Totem: a fault-tolerant multicast group communication system. *Commun. ACM 39*(4), 54–63.

Nunes, R. C. & I. Jansch-Pôrto (2002). Modeling communication delays in distributed systems using time series. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, SRDS '02, Washington, DC, USA, pp. 268–. IEEE Computer Society.

Paxson, V. & M. Allman (2000). Computing tcp's retransmission timer.

Pfister, G. (1998). *In Search of Clusters* (second ed. ed.). Prentice Hall.

PlanetLab. http://www.planet-lab.eu/.

Saito, Y., S. Frølund, A. Veitch, A. Merchant, & S. Spence (2004, October). Fab: building distributed enterprise disk arrays from commodity components. *SIGARCH Comput. Archit. News 32*(5), 48–58.

Schiper, N. & S. Toueg (2008). A robust and lightweight stable leader election service for dynamic systems. In *DSN*, pp. 207–216. IEEE Computer Society.

Stefanakos, S. (2008, December). Reliable routings in networks with generalized link failure events. *IEEE/ACM Trans. Netw. 16*(6), 1331–1339.

Terpstra, W. W., J. Kangasharju, C. Leng, & A. P. Buchmann (2007, August). Bubblestorm: resilient, probabilistic, and exhaustive peer-to-peer search. *SIGCOMM Comput. Commun. Rev. 37*(4), 49–60.

Toueg, S. (1984). Randomized byzantine agreements. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, PODC '84, New York, NY, USA, pp. 163–178. ACM.

van Renesse, R., K. P. Birman, & S. Maffeis (1996, April). Horus: a flexible group communication system. *Commun. ACM 39*(4), 76–83.

van Renesse, R., Y. Minsky, & M. Hayden (1998). A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, London, UK, UK, pp. 55–70. Springer-Verlag.

Vaquero, L. M., L. Rodero-Merino, J. Caceres, & M. Lindner (2008, December). A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev. 39*(1), 50–55.

Verissimo, P. & M. Raynal (1999). Time in distributed system models and algorithms. In *Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems*, London, UK, UK, pp. 1–32. Springer-Verlag.

Xiong, N., A. V. Vasilakos, J. Wu, Y. R. Yang, A. Rindos, Y. Zhou, W.-Z. Song, & Y. Pan (2012). A self-tuning failure detection scheme for cloud computing service. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, Washington, DC, USA, pp. 668–679. IEEE Computer Society.

Xiong, N., A. V. Vasilakos, Y. R. Yang, S. Wei, C. Qiao, & J. Wu (2011). General traffic-feature analysis for an effective failure detector in fault-tolerant wired and wireless networks. Technical report.

Yu, M., Y. Yi, J. Rexford, & M. Chiang (2008, March). Rethinking virtual network embedding: substrate support for path splitting and migration. *SIGCOMM Comput. Commun. Rev. 38*(2), 17–29.