UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Support for Dependable and Adaptive Distributed Systems and Applications

Mônica Lopes Muniz Corrêa Dixit

DOUTORAMENTO EM INFORMÁTICA
ESPECIALIDADE ENGENHARIA INFORMÁTICA

2011

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Support for Dependable and Adaptive Distributed Systems and Applications

Mônica Lopes Muniz Corrêa Dixit

DOUTORAMENTO EM INFORMÁTICA
ESPECIALIDADE ENGENHARIA INFORMÁTICA

2011

Tese orientada pelo Prof. Doutor António Casimiro Costa

Abstract

Distributed applications executing in uncertain environments, like the Internet, need to make timing/synchrony assumptions (for instance, about the maximum message transmission delay), in order to make progress. In the case of adaptive systems these temporal bounds should be computed at runtime, using probabilistic or specifically designed ad hoc approaches, typically with the objective of improving the application performance. From a dependability perspective, however, the concern is to secure some properties on which the application can rely.

This thesis addresses the problem of supporting adaptive systems and applications in stochastic environments, from a dependability perspective: maintaining the correctness of system properties after adaptation. The idea behind dependable adaptation consists in ensuring that the assumed bounds for fundamental variables (e.g., network delays) are secured with a known and constant probability.

Assuming that during its lifetime a system alternates periods where its temporal behavior is well characterized (stable phases), with transition periods where a variation of the network conditions occurs (transient phases), the proposed approach is based on the following: if the environment is generically characterized in analytical terms and it is possible to detect the alternation of these stable and transient phases, then it is possible to effectively and dependably adapt applications.

Based on this idea, the thesis introduces *Adaptare*, a framework for supporting dependable adaptation in stochastic environments. An extensive evaluation of *Adaptare* is provided, assessing the correctness and effectiveness of the implemented mechanisms. The results indicate that the proposed strategies and methodologies are indeed effective to support dependable adaptation of distributed systems and applications.

Finally, the applicability of *Adaptare* is evaluated in the context of two fundamental problems in distributed systems: consensus and failure detection. The thesis proposes solutions for these problems based on modular architectures in which *Adaptare* is used as a middleware for dependable adaptation of assumed timeouts.

Keywords: Adaptive distributed systems, Dependability, QoS adaptation, Stochastic network characterization, Adaptive consensus, Adaptive failure detector

Resumo

Aplicações distribuídas que executam em ambientes incertos, como a Internet, baseiam-se em pressupostos sobre tempo/sincronia (por exemplo, assumem um tempo máximo para a transmissão de mensagens) a fim de assegurar progresso. No caso de sistemas adaptativos, esses limites temporais devem ser calculados em tempo de execução, usando abordagens probabilísticas ou desenhadas de forma específica e ad hoc, tipicamente visando melhorar o desempenho da aplicação. Sob o ponto de vista da confiabilidade, no entanto, o objetivo é garantir algumas propriedades nas quais a aplicação pode confiar.

Esta tese aborda o problema de suportar sistemas adaptativos e aplicações que operam em ambientes estocásticos, numa perspectiva de confiabilidade: mantendo a correção das propriedades do sistema após a adaptação. A ideia da adaptação confiável consiste em garantir que os limites assumidos para variáveis fundamentais (por exemplo, latências de transmissão) são assegurados com uma probabilidade conhecida e constante.

Supondo que durante a execução o sistema alterna períodos nos quais o seu comportamento temporal é bem caracterizado (fases estáveis), com períodos de transição durante os quais ocorrem variações das condições da rede (fases transientes), a abordagem proposta baseia-se no seguinte: se o ambiente é genericamente caracterizado em termos analíticos e é possível detetar a alternância entre fases estáveis e transientes, então é possível adaptar as aplicações de forma efetiva e confiável.

Com base nesta ideia, a tese apresenta uma plataforma para suportar a adaptação confiável em ambientes estocásticos, denominada *Adaptare*. A tese contém uma extensa avaliação do *Adaptare*, que foi realizada para verificar a correção e eficácia dos mecanismos desenvolvidos. Os resultados indicam que as estratégias e metodologias propostas são de facto efetivas para suportar a adaptação confiável de sistemas e aplicações distribuídas.

Finalmente, a aplicabilidade do *Adaptare* é avaliada no contexto de dois problemas fundamentais em sistemas distribuídos: consenso e deteção de falhas. A tese propõe soluções para estes problemas baseadas em arquiteturas modulares nas quais o *Adaptare* é usado como um middleware para a adaptação confiável de *timeouts*.

Palavras Chave: Sistemas distribuídos adaptativos, Confiabilidade, Adaptação de QoS, Caracterização estocástica da rede, Consenso adaptativo, Detector de falhas adaptativo

Resumo Alargado

Sistemas e aplicações estão se tornando cada vez mais distribuídos, por exemplo, através da Internet. O crescimento do uso de dispositivos móveis e distribuídos resulta em ambientes complexos (incluindo redes e plataformas computacionais) que tendem a ser incertos, tornando impraticável, ou até mesmo incorreto confiar em limites temporais fixos, previamente configurados. No entanto, em diversos domínios de aplicação (por exemplo, automação industrial, serviços interativos na Internet e aplicações veiculares) existem fortes requisitos para a operação dentro de limites temporais bem definidos, além de uma crescente necessidade de garantir a confiabilidade. Apesar da impossibilidade de oferecer garantias de tempo-real em tais cenários, soluções baseadas no melhor esforço não são suficientes em alguns dos domínios de aplicação referidos. Uma possível forma de lidar com a incerteza temporal do ambiente e, ao mesmo tempo, garantir requisitos de confiabilidade consiste em assegurar que as aplicações se adaptam aos recursos disponíveis, e que essa adaptação é feita de forma confiável. Ou seja, as aplicações permanecem corretas como resultado da adaptação.

O objetivo fundamental desta tese é mostrar que é possível desenvolver soluções que suportem e facilitem a adaptação confiável de sistemas e aplicações, apesar das incertezas no ambiente operacional. Em ambientes sujeitos à incerteza, a abordagem clássica de assumir limites fixos variáveis temporais não é apropriada. Se forem assumidos limites muito elevados, com o objetivo de evitar faltas temporais (que ocorrem quando os limites assumidos são violados), o desempenho do sistema poderá ser comprometido devido a atrasos na execução. Por outro lado, se forem assumidos limites reduzidos para obter um melhor desempenho, a probabilidade de ocorrerem faltas temporais tenderá a ser maior, o que pode ter um impacto negativo na correção do sistema, ou até mesmo no desempenho (caso existam atrasos decorrentes das faltas).

Esta tese defende que nesses ambientes a confiabilidade deve ser medida em função da capacidade do sistema de garantir alguns limites, adaptando-se às mudanças nas condições observadas. Portanto, enquanto outros trabalhos que estudam sistemas adaptativos estão focados essencialmente no desempenho das aplicações, o nosso principal

interesse é garantir confiabilidade, que é medida em termos de cobertura dos pressupostos, ou seja, dos limites assumidos para variáveis temporais.

Casimiro & Verissimo (2001) definiram os princípios arquitetónicos e funcionais para a adaptação confiável. A ideia é centrada em duas premissas essenciais:

- Os limites assumidos para variáveis fundamentais (como prazos ou períodos para execução de ações) não são estaticamente configurados, mas adaptados ao longo da execução;

- O processo de adaptação assegura que, ao longo do tempo, os limites assumidos são tais que as variáveis temporais permanecem dentro desses limites com uma probabilidade conhecida e constante.

Por exemplo, considerando uma aplicação ou protocolo que define um limite temporal associado à latência de transmissão de mensagens (*timeout*), esta aplicação vai adaptar o valor do *timeout* ao longo da execução, com o objetivo de assegurar que a probabilidade de receber mensagens dentro do limite estabelecido permanecerá acima de um valor pré-definido. Portanto, quando as latências de transmissão de mensagens aumentarem ou diminuírem, o *timeout* também aumentará ou diminuirá na medida exata do que é necessário para garantir a estabilidade desejada do valor de probabilidade. Por outras palavras, essa aplicação vai garantir a estabilidade da cobertura de um pressuposto (Verissimo & Casimiro, 2002).

Assim, a qualidade de serviço (QoS) não é mais expressa como um valor único, (por exemplo um prazo que deve ser satisfeito), mas como um par $\langle limite\ temporal, cobertura\rangle$, no qual a cobertura deve ser garantida, enquanto o limite temporal pode variar como resultado da adaptação, de acordo com as condições verificadas no ambiente. Esta tese adota uma abordagem prática, que procura garantir que a cobertura observada é sempre maior do que o especificado, enquanto os limites para as variáveis temporais são configurados com o menor valor possível.

No entanto, só é possível garantir a cobertura se algumas condições forem impostas (assumidas) para o comportamento do ambiente. Se os tempos para a transmissão de mensagens na rede puderem variar de forma arbitrária, então qualquer observação ou caracterização do ambiente será inútil, no sentido de que nada pode ser inferido relativamente ao comportamento futuro. Felizmente, esse não é o caso usual. É possível que ocorram variações instantâneas ou de curto prazo, que são imprevisíveis e impossíveis de caracterizar, mas no integral do tempo de execução as variações que ocorrem seguem normalmente um padrão, permitindo caracterizar probabilisticamente o estado operacional e derivar os limites que devem ser utilizados para garantir a cobertura esperada.

Nesse sentido, Casimiro & Verissimo (2001) adotaram um modelo muito simples, assumindo apenas que as redes se comportam estocasticamente, mas sem fazerem pressupostos sobre as distribuições de probabilidade que descrevem seu comportamento. Assim, os limites eram calculados de forma conservadora, a fim de serem corretos e independentes de distribuições específicas. Isso foi suficiente para ilustrar a ideia da adaptação confiável.

Esta tese defende a possibilidade de definir um modelo mais forte, com hipóteses mais agressivas, com o objetivo de alcançar resultados com viabilidade prática. Assume-se que os sistemas alternam períodos estáveis, durante os quais o comportamento temporal do ambiente é bem caracterizado, e períodos de transição, durante os quais ocorrem variações nas condições do ambiente, e que a alternância entre períodos estáveis e períodos de transição pode ser detetada. Assim, se for possível concretizar mecanismos para distinguir estas fases, então será também possível calcular dinamicamente limites temporais melhores (menores) quando uma fase estável é detetada, e limites temporais mais conservadores, mas ainda assim confiáveis, durante os períodos de transição.

Com base nesta ideia, a tese propõe um modelo de sistema para suportar a adaptação confiável de aplicações que operam em ambientes estocásticos, garantindo a cobertura de variáveis temporais. O modelo e arquitetura propostos são definidos a partir da especificação do conjunto de premissas adotado, da apresentação da metodologia para adaptação confiável, e da discussão teórica das condições necessárias para a viabilidade da abordagem proposta.

A tese apresenta *Adaptare* - uma plataforma para suportar a adaptação confiável. A plataforma *Adaptare* emprega mecanismos estatísticos para caracterizar probabilisticamente as condições da rede e, assim, derivar limites confiáveis para variáveis temporais. A tese descreve detalhadamente a arquitetura, operação e concretização do *Adaptare*.

A concretização desta plataforma possibilitou a avaliação e validação do modelo e arquitetura propostos, incluindo: (i) a verificação da correção dos mecanismos concretizados para monitorização e caracterização da rede; (ii) a validação das hipóteses consideradas em diferentes cenários e redes; (iii) a análise da complexidade e do tempo de execução para o cálculo de limites confiáveis; e (iv) a comparação, em termos de complexidade e confiabilidade, da solução proposta com outras alternativas para adaptação utilizadas na literatura.

A tese demonstra a utilidade e aplicabilidade das soluções propostas em sistemas reais, através do projeto, concretização e avaliação de protocolos adaptativos que utilizam o *Adaptare* como um middleware de suporte à adaptação confiável. A tese aborda, nomeadamente, dois problemas fundamentais na área de sistemas distribuídos: consenso e deteção de falhas.

A partir de um protocolo de consenso desenvolvido para operar em redes sem fio ad hoc que utiliza um *timeout* estático, a tese demonstra experimentalmente a necessidade da utilização de *timeouts* adaptativos nessas redes para a melhoria de desempenho do protocolo, e descreve a metodologia seguida para transformar o protocolo considerado numa solução adaptativa. Nessa solução, o *Adaptare* é utilizado como um serviço que fornece *timeouts* confiáveis.

Finalmente, a tese apresenta um detetor de falhas autónomo e adaptativo denominado *Adaptare-FD*, orientado à confiabilidade. O *Adaptare-FD* ajusta *timeouts* e períodos de interrogação em tempo de execução, de acordo com as condições da rede e requisitos de qualidade de serviço especificados pela aplicação cliente. O *Adaptare-FD* tem uma arquitetura modular e também utiliza o *Adaptare* como um middleware para o cálculo de *timeouts*.

Assim, esta tese é um trabalho coerente sobre as arquiteturas, modelos e mecanismos que apoiam a concretização de aplicações adaptativas confiáveis, considerando-se os exemplos de consenso e deteção de falhas adaptativos em ambientes estocásticos para ilustrar os benefícios fornecidos pela abordagem proposta.

Acknowledgements

Many people helped me during this work, directly or indirectly, and I am profoundly grateful to all of them.

First of all, my sincerely thanks to my advisor, professor António Casimiro. Thank you for giving me this opportunity and for believing in my work. I appreciated all your support, wise advices and guidance. You truly worked with me, so thank you for everything.

Thanks to all professors and students from the Navigators research group, I learned a lot from you. In particular I thank professor Paulo Veríssimo, for his leadership, enthusiasm, and hard work.

Special thanks to my husband Rudra Dixit, for being an endless source of love, patience (so much patience!), peace, and joy. You are the best person I ever met, and life wouldn't be so easy and funny without you.

An enormous thanks to Giuliana Veronese. Thank you so much for taking care of me in my first (and scary) months in Portugal, and for being the best friend that I could possibly imagine since then. You are more than a friend, you are a sister to me.

Thanks to the very good friends that I made during these years: Vini, Lê, João, Bruno, Marcírio, Pati, André, Pedro, Simão. Thanks for all the laughs and encouragement!

I also would like to thank to my family, to whom I dedicate this thesis. Here I repeat what I said in my master thesis, because it is the very truth. I always did everything that I wanted, and only what I wanted, because I knew that no matter what, you would always be there for me. I will never be able to thank you enough for this.

Finally, I gratefully acknowledge the financial support from Fundação da Faculdade de Ciências da Universidade de Lisboa through the Hidenets project, and from Fundação para a Ciência e a Tecnologia de Portugal.

À minha família, pelo incentivo e fé.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Computer systems and applications are becoming increasingly distributed, for example, over the Internet. The resulting pervasiveness and ubiquity of computing devices leads to complex environments (including networks and computational platforms) that tend to be unpredictable, essentially asynchronous, making it impractical, or even incorrect, to rely on aprioristic time-related bounds. On the other hand, in several application domains (e.g., home and factory automation, interactive services over the Internet, vehicular applications) there are strong requirements for timely operation and increased concerns with dependability assurance. Unfortunately, hard real-time guarantees cannot be given in such settings, and best-effort soft real-time guarantees will not be sufficient in some of the application domains mentioned above. One possible way to cope with the uncertain timeliness of the environment while meeting dependability constraints, consists in ensuring that applications *adapt* to the available resources, *and* do that in a *dependable* way, that is, they remain correct as a result of adaptation.

1.1 Motivation and objectives

The scenario in open distributed systems is subject to important factors of change in the way in which timeliness requirements are handled, face to the uncertainty observed in complex operational environments. On one hand, traditionally weak models, such as asynchronous, are limited to be used in practical systems due to impossibility results that derive from their lack of notion of time. On the other hand, traditionally strong models, such as synchronous, do not correctly represent the behavior of complex networked environments. To circumvent these problems, models of partial synchrony are used in practice, typically enriching the asynchronous model with the necessary timing/synchrony assumptions to make possible the design and implementation of distributed systems. However, variations in the network behavior may invalidate these assumptions, ultimately compromising the system correctness. Clearly, the key issue here is the lack, or loss thereof, of long term coverage of time-related assumptions.

We propose to overcome this problem by securing coverage stability through dependable adaptation. Therefore, while other works addressing adaptive systems are mainly concerned with performance, our main concern is dependability, which is measured in terms of coverage of assumed bounds for time-related variables. Casimiro & Verissimo (2001) introduced the architectural and functional principles for dependable QoS adaptation. In essence, their idea lies in two very simple premises:

- Assumed bounds for fundamental variables (e.g., deadlines and timeouts) are not aprioristic constants, but instead are adapted throughout the execution;
- The adaptation process ensures that, over time, the assumed bounds are such that those variables remain within these bounds with a known and constant probability.

For instance, consider an application or protocol that defines a timeout value based on the assumed message round-trip delay. This application will adapt the timeout during the execution with the objective of ensuring that the probability of receiving timely messages will stay close to some predefined value. Therefore, when message delays increase or decrease, the timeout will also increase or decrease in the exact measure of what is needed to ensure the desired stability of the probability value. In other words, this application will secure a coverage stability property (Verissimo & Casimiro, 2002).

However, achieving coverage stability is only possible if some limits are imposed (assumed) on how the environment behaves. If message delays can vary in some arbitrary fashion, then any observation or characterization of the environment will be useless, in the sense that nothing can be inferred with respect to the future behavior. Fortunately, this is not the usual case. There may be instantaneous or short-term variations that are unpredictable and impossible to characterize, but medium to long-term variations typically follow some pattern, allowing to probabilistically characterize the current operational state and to derive the bounds that must be used for achieving coverage stability.

In that sense, Casimiro & Verissimo (2001) adopted a very simple and weak model, assuming just that networks behave stochastically, but the probability distributions that describe their behavior are unknown. In other words, they assumed that there is an envelope for the probabilistic behavior, which is known and characterizable, and includes all the possible probabilistic behaviors. Therefore, bounds are conservatively selected in order to be correct regardless of specific distributions. This was sufficient to illustrate the feasibility of the dependable QoS adaptation approach.

In this thesis we defend that it is possible to use a stronger model, with more aggressive assumptions, in order to achieve practical results on the adaptation process, and thus to develop solutions that can be applied to effectively address real and fundamental problems of dependable distributed systems. We advance on the work presented in Casimiro & Verissimo (2001) by leveraging on the assumption that a system alternates stable periods, during which the environment is probabilistically characterizable, and transition periods, in which a variation of the environment conditions occurs, and that the changes between stable and transition periods can be detected. Therefore, if we can implement mechanisms to distinguish these periods, then it is possible to dynamically compute improved (tighter) time bounds when a stable period is detected, and conservative but still dependable bounds during transition periods. The relevance of computing improved bounds lies in the fact that they make the approach for dependable adaptation practical. This is the central idea of our work, and it motivates the first objective of this thesis.

Objective 1: *To define a system model and architecture to support dependable adaptation of applications operating in stochastic environments, securing coverage stability of time-related variables.*

Once the system model and architecture are formally specified, it is possible to define the specific mechanisms used to implement the methodology for dependable adaptation, in order to assess the benefits of our approach in practice. The actual implementation of the selected mechanisms will be useful to evaluate the improvements that can be achieved by the proposed approach for adaptation. Thus, our second objective is formulated as follows.

Objective 2: *To define and validate the mechanisms for network monitoring and characterization used to achieve dependable adaptation. These mechanisms will be implemented into a framework for supporting dependable adaptation of applications operating in stochastic environments. The evaluation of the proposed model and mechanisms includes: (i) verifying the correctness of the defined mechanisms; (ii) validating our assumptions under different scenarios and networks; (iii) analyzing the overhead imposed by the implemented framework; and (iv) comparing our solution for adaptation with other currently used alternatives, in terms of complexity and achieved dependability.*

Finally, it is fundamental to demonstrate the usefulness of our approach and the applicability of the proposed solutions in real systems. In fact, this is stated by our third and final objective.

Objective 3: *To design, implement and evaluate distributed adaptive protocols that use the proposed framework as a middleware for dependable adaptation.*

1.2 Contributions

The main contributions of this thesis are summarized bellow.

- Definition of a system model (set of assumptions) and methodology for supporting dependable adaptation based on continuous network monitoring and probabilistic characterization.

- Proposal and implementation of *Adaptare*. *Adaptare* is a framework that employs statistical mechanisms to probabilistically characterize the network conditions, and to derive dependable bounds for time-related variables, such as message delays. In this thesis we describe *Adaptare*'s architecture, operation, and implementation.

- Description and analysis of an extensive set of experimental results used to thoroughly quantify and evaluate the benefits of the proposed approach and the correctness of the mechanisms implemented by *Adaptare*.

- Implementation and evaluation of an adaptive consensus protocol. Starting from a static timeout-based consensus protocol developed to operate in wireless ad hoc networks, we first experimentally demonstrate the need for adaptive timeouts in such dynamic environments in order to improve the algorithm's performance. Then we describe our methodology to transform this protocol into an adaptive solution, using *Adaptare* as a timeout provisioning service. We quantify the benefits of timeout adaptation by comparing both versions of the protocol.

- Proposal, implementation and evaluation of *Adaptare-FD*, an autonomic and adaptive dependability-oriented failure detector. *Adaptare-FD* has a modular architecture and it uses *Adaptare* as a middleware for timeout computation. *Adaptare-FD* adjusts both timeouts and interrogation periods in runtime, according to specified QoS parameters and the network conditions.

1.3 Structure of the thesis

The remaining of this thesis is organized as follows.

Chapter 2: Context and Related Work. In this chapter we present the fundamental concepts and relevant works from the research areas related to this thesis. We divide the chapter's content in three main parts. In the first part, we describe traditional models of synchrony, discuss their strengths and weaknesses, and present the stochastic model, which is the basis for the model considered in this thesis. The second part of the chapter presents several works related to network probabilistic characterization.

The different conclusions from those works motivated our approach for continuous network monitoring and characterization. In the last part of the chapter we present the components of QoS architectures and contrast them with the mechanisms proposed in our methodology for dependable QoS adaptation. We also summarize some related work with focus on adaptive consensus and failure detection.

Chapter 3: Adapting for Dependability. This chapter introduces our approach for dependable QoS adaptation. First, we state the objective of adaptation in terms of achieving a stable coverage of system properties, and discuss our assumptions regarding network behavior and application dynamics. Then, we explain in detail our methodology to achieve dependable adaptation and analyze the necessary conditions for the correctness of the proposed solution.

Chapter 4: *Adaptare*. In this chapter we introduce *Adaptare*, a framework for supporting dependable adaptation of applications operating in stochastic environments. We describe its modular architecture, and how the implemented mechanisms execute together for the computation of dependable bounds. We provide implementation details, including the specification of the API that allows the integration between client applications and *Adaptare*. Finally, we present an extensive evaluation, based on synthetic and real traces from different scenarios, which aims at validating our mechanisms and assumptions and characterizing the benefits of using our approach, in comparison to the conservative solution followed in Casimiro & Verissimo (2001). The overhead of *Adaptare* is assessed both through analytical and empirical studies. We also compare *Adaptare* with other well-known approaches for adaptation, identifying the conditions in which applying our solution would be more appropriate.

Chapter 5: Timeout-based Adaptive Consensus. In this chapter we exemplify how adaptive protocols can be easily implemented using *Adaptare*. Following a pragmatic approach, we take a static timeout-based consensus protocol for wireless ad hoc networks introduced in Moniz *et al.* (2009) and describe our methodology to transform it into an adaptive consensus protocol, in which timeouts are adjusted according to *Adaptare*'s indication. We quantify the benefits of using adaptive timeouts by comparing the achieved performance of the adaptive and original versions of this protocol.

We believe that the methodology presented in this chapter is generally applicable for the design and implementation of other timeout-based adaptive protocols.

Chapter 6: *Adaptare-FD.* In this chapter we introduce an autonomic and adaptive failure detector, called *Adaptare-FD*. Our failure detector adapts both timeouts and interrogation periods according to QoS parameters specified in its interface. Timeouts are provided by the *Adaptare* framework. We present *Adaptare-FD* architecture, and evaluate its performance by comparing its operation with other timeout-based adaptive failure detectors, in a varied set of network conditions. In particular, we analyze the differences between *Adaptare-FD* and Chen's failure detector (Chen *et al.*, 2002), since they both adapt operational parameters to meet the required QoS level. From this analysis, it was possible to identify the conditions under which each approach should be applied, and to conclude that *Adaptare-FD* is more suitable to systems with high availability requirements.

Chapter 7: Conclusions and Future Research Directions. This chapter concludes this thesis, summarizing our main contributions, and identifying interesting open issues that could be addressed to extend this work in a future research.

Chapter 2

Context and Related Work

This chapter provides the context for the work presented in this thesis, including fundamental concepts and previous related work. It is organized in three main sections. Section 2.1 presents the most important models of synchrony proposed in the literature, and the stochastic model in which our work is based. In Section 2.2 we review several works that address the problem of probabilistic network characterization. We show that these works achieved varied results, suggesting that network delays may be characterized by a wide range of different probability distributions, which motivated our approach for monitoring and runtime characterization of the environment. Finally, Section 2.3 discusses important concepts related to quality of service (QoS) in distributed systems. We first present the typical components of QoS architectures, distinguishing our definition for QoS and the mechanisms that we implement from those generalized architectures. Finally, we survey related work that defines QoS in terms of time-related properties and proposes adaptive solutions for fundamental problems in distributed systems.

2.1 Synchrony in distributed systems

The design of a distributed protocol or algorithm requires the definition of the considered model for the execution environment. This model specifies what one can expect from the environment, i.e., the set of assumptions on which the system designer can rely when devising a new protocol. The model may cover different aspects of the system, such as timing properties and potential faults (Lamport & Lynch, 1990).

The solution proposed in this thesis is focused on monitoring and characterizing time-related variables, in particular network latencies, in order to drive the adaptation process of distributed applications. Therefore, we are primarily concerned with timeliness properties of a given system or network, which are formally specified by the model of synchrony. In this section we review several models of synchrony adopted in the design of distributed systems.

2.1.1 Synchronous model

The synchronous model is characterized by the existence of known bounds on communication delays and on relative speeds of different processors. Moreover, the rate of drift of local clocks is also known and bounded (Verissimo & Rodrigues, 2001).

Building distributed protocols to operate in a synchronous system is relatively simple, because it is known *a priori* how long processes have to wait for a given message to be received or an operation to be completed, before making some decision that depends on that message or operation. For example, if a message is not received within the known delay, the protocol can correctly assume that either the message was lost, or it was never sent due to a fault in the sending process.

Raynal (2002) surveys various consensus protocols designed to operate in synchronous systems. Informally, consensus is an agreement problem in which each process p_i proposes an initial value v_i, and all correct processes have to agree on a common value v, which has to be one of the proposed values.

However, environments such as the Internet, or wireless ad hoc networks, are dynamic and intrinsically not synchronous: among other factors, variations on the network load, process load, scheduling policies and routing protocols may affect the temporal behavior of the system. Thus, the synchronous model is not appropriate for the design of applications and protocols operating in such environments, since assumed temporal bounds may be violated, compromising the system correctness.

2.1.2 Asynchronous model

In the asynchronous model, communication delays and process speeds are not bounded. There is no notion of time, which means that nothing can be expected in terms of timeliness under this model. This is the weakest and safest model of synchrony, since there are no timing assumptions to be violated. Moreover, a protocol designed to correctly operate in asynchronous systems will be correct in any system with stronger synchrony properties.

However, Fischer *et al.* (1985) have shown that the consensus problem does not have a deterministic solution in asynchronous systems prone to crash failures - a result known as the FLP impossibility. This is a very important result, since the solution for many fundamental problems of distributed systems, such as atomic broadcast, leader election or clock synchronization, relies on the ability to achieve some form of agreement among a set of processes.

Fortunately, practical distributed systems are not fully asynchronous. In fact, these systems behave synchronously most of the time, when appropriate upper bounds for communication delays hold. Moreover, periods of synchrony are in general long enough to allow the execution of distributed protocols such as consensus. This synchrony is only occasionally disturbed by some external factor. Because of that, it is possible to assume intermediate models of synchrony. The remaining of this section presents some of these models, which are based on augmenting the asynchronous model with additional synchrony assumptions or components, in order to circumvent the FLP impossibility result.

2.1.3 Asynchronous model with failure detectors

Failure detectors are distributed oracles that monitor a set of processes and indicate which processes are suspected of being crashed. In this model, each process has a failure detector module that monitors other processes in the system and maintains a dynamic list of suspected processes. They are used in asynchronous systems, and the necessary synchrony assumption is encapsulated by the failure detector.

Typically, processes are monitored either by responding to periodic queries (pull style failure detector) or by sending periodic heartbeats (push style failure detector).

The responses/heartbeats are received by the monitoring module in each process, and they indicate that the sending process is alive.

When a response/heartbeat is not received within a given amount of time (called the failure detector timeout) the monitoring process suspects that the sending (monitored) process is crashed. Since the system is asynchronous, it is possible that messages are delayed and arrive after the timeout expiration. In this case the failure detector will mistakenly suspect a correct process, from the time of timeout expiration until the reception of a valid message. A received response/heartbeat is valid if it arrives before the timeout expiration for the reception of the next response/heartbeat. Otherwise, this response/heartbeat is considered an old message and it is discarded by the failure detector. When a valid message is received, the sending process is removed from the suspected list.

Moreover, if the monitored process crashes, the monitor process will only detect this failure when the timeout for the next expected response/heartbeat from the monitored crashed process expires. Thus, there is a period of time during which the failure detector will not suspect a crashed process, referred as detection time.

This model was introduced in Chandra & Toueg (1996) with the concept of unreliable failure detectors, since they can make mistakes by suspecting correct processes, or not suspecting crashed processes immediately, due to the uncertainty of the environment. Failure detectors are classified according to their completeness and accuracy properties, where completeness refers to the ability of eventually suspecting all crashed processes and accuracy restricts the mistakes made by the failure detector. There are different levels of completeness and accuracy, as follows.

- *Strong Completeness.* Eventually every process that crashes is permanently suspected by every correct process.
- *Weak Completeness.* Eventually every process that crashes is permanently suspected by some correct process.
- *Strong Accuracy.* No process is suspected before it crashes.
- *Weak Accuracy.* Some correct process is never suspected.

- *Eventual Strong Accuracy.* There is a time after which correct processes are not suspected by any correct process.

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

Table 2.1 presents the eight classes of failure detectors defined in Chandra & Toueg (1996), which are derived from the combination of the above properties.

Completeness	**Accuracy**			
	Strong	**Weak**	**Eventual Strong**	**Eventual Weak**
Strong	P Perfect	S Strong	$\Diamond$P Eventually Perfect	$\Diamond$S Eventually Strong
Weak	Q	W Weak	$\Diamond$Q	$\Diamond$W Eventually Weak

Table 2.1: Classes of unreliable failure detectors.

The asynchronous model augmented with a failure detector is one of the most widely used approaches to circumvent the FLP result. Chandra & Toueg (1996) propose a consensus protocol that uses a $\Diamond$S failure detector, assuming an asynchronous system, processes prone to crash failures, majority of correct processes and reliable channels. This protocol provides the principles of many other existent consensus algorithms. A $\Diamond$S-based consensus protocol is presented in Schiper (1997), based on the same set of assumptions. It has been proved that $\Diamond$W is the weakest failure detector for solving consensus in asynchronous systems subject to crash failures with a majority of correct processes (Chandra *et al.*, 1996), and that the $\Diamond$S and the $\Diamond$W failure detectors are equivalent (Chandra & Toueg, 1996).

Some researches focus on relaxing the assumptions about the environment (for example, the assumption that the majority of processes is correct), proposing protocols based on stronger failure detectors specifications. Mostefaoui & Raynal (2000) define a family S_x of failure detectors, which includes the class S ($x = 1$) and satisfies a perpetual accuracy property, which states that there are x correct processes that are never suspected to be crashed ($1 \leq x \leq n - f$). The authors propose a uniform consensus protocol that works with any failure detector of this family. We formally introduce the consensus problem in Section 2.3.2.1. The agreement property of consensus states that no two correct processes decide differently. In order to prevent the possibility that a process decides on a different value before crashing, in the uniform consensus this property is changed to: no two processes (correct or not) decide differently.

Friedman *et al.* (2004) define a class P^f of failure detectors and propose a consensus protocol using failure detectors of class $P^f \times \diamondsuit S$. Class P^f includes all the failure detectors that satisfy the strong completeness property and a f-accuracy property, which states that at any time, no more than $n-f-1$ alive processes are suspected. Class $P^f \times \diamondsuit S$ is composed by the failure detectors that satisfy the properties defined by $\diamondsuit S$ and P^f. The protocol solves the consensus problem in asynchronous distributed systems prone to $f < n$ crash failures.

More recent works in this research area propose adaptive failure detectors, which are more suitable to the increasingly dynamic environments in which distributed systems operate nowadays (Bertier *et al.*, 2002; Chen *et al.*, 2002; Falai & Bondavalli, 2005; Nunes & Jansch-Porto, 2004). We discuss adaptive solutions in Section 2.3. In particular, adaptive failure detectors are presented in Section 2.3.2.2.

2.1.4 Partially synchronous model

The concept of partial synchrony was introduced in Dwork *et al.* (1988). The communication system is defined as partially synchronous in two cases:

- If there is an unknown upper bound Δ on message delivery time;
- If there is a known upper bound Δ on message delivery time which holds from some unknown point on - the so-called global stabilization time (GST).

In theory, this model states that if Δ is known, message delays are bounded by Δ from time GST onward, where GST is unknown. This means that after GST, Δ holds forever, which is a very strong assumption. Dwork *et al.* (1988) point out that, in practice, there is an upper bound on the amount of time after GST required for executing a protocol. Thus, it is only necessary that Δ holds from GST up to the required time to execute the protocol.

Based on this model, Dwork *et al.* (1988) present several consensus protocols for both cases of partial synchrony and four different fault models: fail-stop, omission, authenticated byzantine and byzantine. The correctness conditions of these protocols are separated into safety and termination. The proposed consensus algorithms are indulgent: they always satisfy the safety conditions, no matter how asynchronously the system behaves, while termination conditions eventually hold.

2.1.5 Timed asynchronous model

Cristian and Fetzer propose the timed asynchronous distributed system model in Cristian & Fetzer (1999). This model enriches the asynchronous model with the assumption that processes have access to hardware clocks with a constant maximum drift rate. Hence, it is possible to provide timed services under this model (based on assumed bounds for communication and processing delays), and detect when the assumed time bounds are violated. Communication has omission/performance failure semantics, and processes have crash/performance failure semantics. A performance failure occurs whenever the upper bound for communication or processing delay is violated. Processes can recover from failures and there is no bound on the frequency of failures of timed services.

Fetzer (2003) proposes a new class of failure detectors, called timed-perfect failure detector class, based on this model. The output value of a timed-perfect failure detector module that monitors a process p_i is either crashed (p_i is crashed), up (p_i is not crashed) or recovering (p_i is recovering from a crash). Timed-perfect failure detectors are defined by three properties (crash accuracy, up accuracy and recovering accuracy), and by a finite constant ($DD \geq 0$) called detection delay. These properties together state that a process that is crashed for more than DD time units must be classified as *crashed* as long as it stays crashed. Moreover, if the status of a process changes (e.g., after it recovers from a crash), the timed-perfect failure detector has DD time units to correct the classification of this process.

A consensus protocol based on the timed asynchronous distributed system model is presented in Fetzer & Cristian (1995). In this work, the authors assume that: (i) systems alternate between periods of "majority-stability", in which the majority of the processes is up, timely, and can communicate in a timely manner, and short periods of instability, and (ii) after an unstable period there is a time interval of some minimum length in which the system will be stable. The proposed consensus protocol requires that the system is majority stable, and its termination property states that each non-crashed process decides in a finite number of steps.

2.1.6 Wormholes model

From the observation that systems properties vary not only with time (e.g., the partially synchronous model assumes that the system is asynchronous before GST and synchronous after GST), but also that system components may provide different properties, Verissimo (2006) proposes the wormholes model. Basically, this model assumes that different parts of the system have different properties. Thus, the main difference between the wormholes model and the other models described so far is that it assumes that system components are heterogeneous, which allows the design of hybrid distributed systems. Taking the definition of synchrony assumptions as example, while the wormholes model may consider that at a given time some components of the system are synchronous, while others are asynchronous (heterogeneous model), the other models of synchrony presented in this chapter make assumptions that refer to the whole system (homogeneous model).

Verissimo (2006) argues that the existence of synchronous components in the system enables the construction and implementation of algorithms to deal with problems that are not solvable in an asynchronous environment, such as consensus and failure detection. In this model, the system is divided in two parts: a payload system, where applications are executed, and a wormhole subsystem. The payload system may assume any set of faults and synchronism degree. Processes executing in the payload system (payload processes) communicate through payload channels. The wormhole subsystem has stronger properties than the payload system. If the wormhole processes (which run in the wormhole subsystem) are able to communicate amongst themselves through wormhole channels, then the wormhole subsystem is said to be distributed. Otherwise it will be a wormhole with a local scope. Payload processes interact with the wormhole using a well-defined interface, to request services which can only be executed in the wormhole subsystem, due to its stronger properties.

The Timely Computing Base model (TCB) presented in Verissimo *et al.* (2000) uses the wormhole approach to address the problem of having "applications with synchrony requirements running on environments with uncertain timeliness". In this model, the system has components that are synchronous enough to perform timed actions - the TCB modules. The TCB subsystem (a distributed wormhole), composed by the TCB modules and the control channel, is synchronous. Thus, it is used to provide services that cannot be executed in the payload asynchronous system, such as timely execution, duration measurement and timing failure detection.

In particular, the authors propose a perfect timing failure detector (pTFD), defined by two properties: all timing failures are detected by the TCB in a known bounded interval from its occurrence (timed strong completeness) and no timely action results in a timing failure (timed strong accuracy), if they are executed and terminate more than a known amount of time before the deadline.

2.1.7 Stochastic model

The stochastic model is adopted in Casimiro & Verissimo (2001), and it inspired our work. This model assumes that fundamental time-related variables, such as message delays, follow some probability distribution, which may change over time. Therefore, it is not possible to assume fixed upper bounds (that will always hold) as it is done with synchronous models, and differently from asynchronous models it is possible to state bounds that will hold with some probability.

This assumption allows to estimate values for time-related variables with a given probability. Thus, synchrony guarantees are oriented to an expected coverage, which is the probability that synchrony assumptions hold. Casimiro & Verissimo (2001) classify this as a model of partial synchrony. Note that while the traditional partially synchronous model introduced in Dwork *et al.* (1988) guarantees that communication delays are always bounded by some Δ after the global stabilization time, in the stochastic model there is a probability $P < 1$ that Δ holds, which is the coverage of the assumed bound.

Despite adopting a probabilistic model, Casimiro & Verissimo (2001) do not assume any specific probability distribution, proposing mechanisms that are correct for all distributions. As presented in Chapter 3, we advance on this work by assuming that particular probability distributions may be assumed over time, which appropriately describe the stochastic behavior of time-related variables. Given the specific knowledge of the distribution, we can compute more accurate bounds (closer to real observations), with the same coverage.

2.2 Network characterization

The work presented in this thesis is focused on monitoring and probabilistically characterizing communication delays in order to support the adaptation of distributed applications in a dependable way. Thus, although synchrony refers to both message and processing delays, we are concerned with the former. The characterization of communication delays in IP-based networks is a widely studied subject. In this section, we provide an overview of several works that address this problem.

One of the simplest statistical approaches to characterize communication delays in distributed systems is the time series modeling and analysis. No probability distribution is considered: samples are modeled as a time series, which is defined as "a set of measurements of a stochastic variable taken sequentially in time" (Bowerman & O'Connel, 1993). A methodology to model round-trip communication delays as a time series is presented in Nunes & Jansch-Pôrto (2002).

The approach followed in Jifeng *et al.* (2004) proposes a multiple model method to predict the Internet end-to-end delays. This method is based on using different models for prediction, and combining their results into a single output. In Jifeng *et al.* (2004), the model set is composed by different auto regressive models based on time series built from measured samples. More solutions based on time series modeling are surveyed in Yang *et al.* (2004). The limitation of this approach is that in a time series the samples must be spaced at uniform time intervals, but in many applications monitoring is not performed with fixed periods. For example, tools that monitor TCP packets should not model them as a time series, since they are not received at a constant rate.

As discussed in Section 2.1.7, Casimiro & Verissimo (2001) simply assume that the environment behaves stochastically, but no assumptions are made about the specific probability distribution for communication delays. Interestingly, over the last few years a number of works have addressed the problem of probabilistically characterizing the delays in IP-based networks using real measured data, allowing to conclude that empirically observed delays may be characterized by well-known distributions. Some examples are:

- Markopoulou *et al.* (2006) analyze packet loss and delays over the Internet. The work focuses on wide-area backbone networks, representing long-distance communication. The probabilistic characterization of delays shows that the exponential distribution would be a good fit for the observed behavior.
- The study performed in Mukherjee (1992) over different Internet paths concludes that network delays follow a shifted Gamma distribution with shape and scale parameters changing over time, depending on the network load.
- Piratla *et al.* (2004) employ several statistical procedures to assess the probability distributions that best characterize the delays of packets in a data stream, for different sending rates and packet sizes. In this study, the shifted Gamma and Beta distributions are indicated as the best fit.
- Elteto & Molnar (1999) analyze round trip delays collected in the Ericsson Corporate Network. The samples were analyzed using the Kolmogorov-Smirnov test, and the results show that round trip delays can be well approximated by a truncated normal distribution.
- Zhang & He (2007) model end-to-end delays in wide area networks using statistical methods. Their tests consider the Pareto, normal and lognormal distribution. The results show that the Pareto distribution is more appropriate to model end-to-end delays.
- Hernandez & Phillips (2006) propose a so-called Weibull mixture model, which is based on the combination of Weibull distributions, to characterize end-to-end network delays. The model was validated using measurements collected in the Internet.

Some of these works also recognize that probability distributions may change over time (Piratla *et al.*, 2004), depending on the load or other sporadic occurrences, like failures or route changes. Therefore, in order to provide reliable information to applications in runtime, it becomes necessary to detect changes in the distribution.

Fortunately, there is also considerable work and well-known approaches and mechanisms to address this problem. Among others, we can find approaches based on time-exponentially weighted moving histograms (Menth *et al.*, 2006), on the Kolmogorov-Smirnov test (Elteto & Molnar, 1999) or the Mann-Kendall test (Chen *et al.*, 2006). For instance, Elteto & Molnar (1999) apply the Kolmogorov-Smirnov test on round-trip times to detect changes in a delay process. Between these changes, the process can be assumed to be stationary with constant delay distribution. This assumption was confirmed by the trend analysis test.

From the discussion above, we can conclude that: (i) there is not a single probability distribution that represents end-to-end delays in all distributed systems; and (ii) even for one specific monitored environment, delays are not stationary and the distribution changes over time.

All these possibilities motivated the design and development of an effectively generic framework for automatic and dependable adaptation. Our work is built upon the stochastic model adopted in Casimiro & Verissimo (2001), but we propose mechanisms to identify the probability distribution that best describes the observed network behavior in order to compute more precise bounds for temporal variables.

In Chapter 4 we introduce *Adaptare*, a framework specifically characterized by: (i) modularly accommodating various mechanisms to analyze the environment conditions and dealing with several probability distributions; (ii) automatically determining the most suitable methods and the best fitting probability distributions. Note that this combination allows achieving more accurate characterizations of the environment state at a given time, and along the timeline, which is a crucial result of our work.

2.3 QoS assurance in uncertain environments

Quality of service (QoS) is a commonly used term in computational systems, which may have different meanings, depending on the application context. Extensive research has been done in this area for different contexts, such as multimedia applications, distributed databases systems, industrial applications and telecommunication systems.

In the context of our work, QoS is defined and measured in terms of coverage assurance. The concept of coverage of assumptions was introduced in Powell (1992), and it refers to the probability that assumptions on which the system is based when it is deployed in a real setting will hold. Thus, in our work QoS assurance boils down to satisfying a required probability value (a coverage) specified by a user. We focus on securing the coverage of network latency assumptions, by constantly monitoring the available QoS and providing dependable bounds for network latency according to the observed conditions, that is, bounds that will be secured with the required coverage. These bounds are then used in the adaptation of client applications. Because of that, the adaptation process is said to be dependability-oriented.

The mechanisms that we implement to achieve this goal do not require a fully fledged QoS infrastructure to operate with. Nevertheless, they can be seen as typical components found in most QoS architectures, namely for QoS monitoring and QoS adaptation. Therefore, in the following section we review concepts and work in the area of QoS-oriented systems and architectures. Then, in Section 2.3.2 we present several adaptive solutions for different problems and applications where QoS is related to timeliness properties, as we consider in this thesis.

2.3.1 QoS-oriented systems and architectures

There exist different system architectural approaches to provide end-to-end quality of service guarantees in distributed systems. Aurrecoechea *et al.* (1998) present a nice survey of QoS architectures and introduce a generalized framework that describe their main components. Although the survey is focused on QoS for distributed multimedia systems, it describes generic mechanisms for QoS management that can be applied in a large variety of distributed applications.

A QoS architecture must provide the necessary interfaces for applications to specify the expected QoS level. Typically, QoS specifications are formalized by service level agreements (SLA). Examples of QoS parameters that may be specified in multimedia and other applications are throughput rates, delays, jitter, loss rates, among others. More generic parameters are: level of service, which expresses the required QoS in qualitative terms (e.g., deterministic or best-effort); QoS management policy, which determines the actions performed in case of QoS violations; and cost of service, which specifies the price for the required QoS level.

Different mechanisms are employed to guarantee that the required QoS is provided. The systematization proposed in Aurrecoechea *et al.* (1998) categorizes these mechanisms as follows:

- QoS provision mechanisms: responsible for static resource management. They translate QoS specified requirements into low-level system parameters, apply admission control tests that verify if there are enough available resources to meet the required QoS, and execute reservation protocols to allocate the necessary resources.
- QoS control mechanisms: perform dynamic resource management. They encompass real-time traffic control tasks, such as flow shaping, flow scheduling, and flow synchronization.
- QoS management mechanisms: also related to dynamic resource management, ensuring that the contracted QoS is sustained. QoS management mechanisms are subdivided into: (i) QoS monitoring, where each system layer monitors the QoS offered by the lower layer; (ii) QoS maintenance, which compares the current offered QoS with the contracted one and performs corrective actions, if possible and necessary; (iii) QoS degradation, which informs the user if the contracted QoS cannot be maintained; (iv) QoS availability, which allows applications to specify monitoring intervals for QoS parameters such as delay, loss, and bandwidth; and (v) QoS scalability, which defines the adaptation mechanisms to react to variations in the observed end-to-end QoS.

In this area, several works focus on developing QoS architectures for multimedia applications executing in dynamic best-effort environments, such as the Internet, when resource reservation is not possible. For example, the solution proposed in Bhatti & Knight (1999) analyzes a snapshot of the network conditions (defined by a set of measured QoS parameters) and defines which application modes are feasible on that context. Thus, the application can switch to a better operation mode in order to maintain the expected QoS level. QoS is specified in terms of possible operation modes (QoS specification). A so-called QoSEngine translates application modes to low level network parameters, and vice-versa (QoS provision). These parameters are constantly monitored (QoS control) and the application periodically receives information regarding the QoS offered by the network in terms of feasible modes (QoS management). A more complex model based on a fuzzy controller for dynamic QoS adaptation is presented in Koliver *et al.* (2002). Based on a quality degree metric, and how it is differently perceived at the sender and receiver sides, system parameters (e.g., bit rate) are adjusted to meet some expected QoS level, according to the fuzzy controller indications.

Solutions for QoS assurance developed in the context of a given application tend to be too specific, requiring the configuration of parameters which are tightly related to the application semantics. The works described above are two examples of QoS architectures for multimedia applications, which although effective, are very application-dependent, and require significant *a priori* configuration. Designers must specify not only the parameters to quantify the provided QoS (such as the so-called quality degree metric in Koliver *et al.* (2002)), but also their possible ranges, the set of QoS levels, and the mapping between them. Moreover, these works do not assess the dynamics of the environment. No attempt is made in order to understand and model the dynamic behavior of the measured parameters, neither to predict any future states.

A significantly different and more generic way of dealing with QoS in distributed systems was proposed by Gorender *et al.* (2007). They define QoS in terms of timeliness of communication channels, as we do in our work, and propose an adaptive programming model for fault-tolerant distributed computing, built on top of a QoS-based system. A component called QoS Provider (QoSP) exposes functions to specify and further assess the synchrony of all communication channels in the system. Using the QoSP functions, it is possible to create a communication channel, change and obtain its QoS (timely/untimely), and obtain the expected delay for a message transfer. Moreover, the QoSP modules monitor all timely channels to check if their QoS suffered any modification. In the proposed programming model, each process p_i has access to three sets: $down_i$, $live_i$ and $uncertain_i$. These sets are made up of processes identities and can evolve dynamically, according to the observed synchrony. The semantics of these sets is the following: if $p_j \in down_i$, p_i can safely consider p_j as being crashed; if $p_j \in live_i$, p_i is given a hint that it can currently consider p_j as not crashed; and when $p_j \in uncertain_i$, p_i has no information on the current state (crashed or live) of p_j. The sets $live$ and $uncertain$ are maintained by a so-called state detector, according to the information delivered by the QoSP. The $down$ set is maintained by a failure detector, using the information provided by the QoSP and the state detector. The authors demonstrate the achievable benefits of using this model by defining a $\diamond S$ consensus protocol that adapts to the current QoS using the sets, and using the majority assumption only when needed. Thus, while other $\diamond S$ consensus protocols require a majority of correct processes ($f < n/2$), the proposed protocol allows bypassing this bound when $|uncertain_i| < n$.

Differently from the solutions described above, our approach does not require an underlying QoS-oriented infrastructure to operate. On the contrary, it directly builds on a probabilistic system model to implement the following QoS mechanisms:

- QoS control, which comprises the monitoring and probabilistic characterization of network delays; and

- QoS management, which provides dependable information about the network behavior to client applications, driving the adaptation process in order to preserve the required dependability level.

2.3.2 QoS monitoring and adaptation

In the dynamic environments considered in this work, monitoring is of fundamental importance. In fact, the development of solutions for monitoring and network traffic analysis is an active research area (Tzagkarakis *et al.*, 2009). Advances in this area are relevant to support several properties of autonomous systems, like self-organization, self-adaptation, and other self-* properties (Babaoglu *et al.*, 2005).

Providing QoS guarantees for the communication in spite of the uncertain or probabilistic nature of networks is a problem with a wide scope, which can be addressed from many different perspectives. In Casimiro & Verissimo (2001), the fundamental architectural and functional principles for dependable QoS adaptation were introduced, providing relevant background for our work. In that work the authors analyze why systems would fail as a result of timing assumptions being violated, as it may happen in environments with weak synchrony. A relevant effect is *decreased coverage* of some time bound, when the number of timing failures goes beyond an assumed limit.

In systems in which time bounds must remain fixed, other variables should be adapted to avoid the decreased coverage of these bounds. For example, in Krishnamurthy *et al.* (2001) the idea is to use replication for timing fault tolerance. The number of used service replicas is dynamically selected to ensure timely responses with a given probability.

Another approach to deal with decreased coverage of timing assumptions, which we follow in our work, is to explore the ability of applications to use adaptive time-related bounds for temporal variables (e.g., communication timeouts). The remaining of this section focuses on two such cases, or application problems: adaptive consensus and adaptive failure detection. These are fundamental distributed systems building blocks, whose implementation can be made adaptive with respect to the timeliness conditions, in order to sustain some expected QoS level.

2.3.2.1 Adaptive consensus

The consensus problem is formally defined by the following properties (Chandra *et al.*, 1996):

- Validity: If a correct process decides v, then v was proposed by some process.
- Agreement: no two correct processes decide differently; and
- Termination: All correct processes eventually decide.

As discussed in Section 2.1, this problem has no deterministic solution in fully asynchronous systems prone to crash failures. Thus, consensus protocols are designed for system models augmented with some synchrony assumptions, such as partial synchrony or failure detectors. Since the synchrony assumptions may not hold in runtime (due to lack of assumption coverage), designers efforts are focused on always ensuring the safety of protocols, even when the network does not present the expected synchrony. Liveness is achieved when the system exhibits the necessary synchrony level.

A typical consensus protocol executes in rounds, where participant processes rely on a timeout to limit the amount of time they should wait for messages from other processes. This timeout can be explicitly used in the consensus algorithm, or implicitly used, when it is encapsulated in a failure detector. The timeout value is derived from the assumed synchrony - the upper bound on message delays - and it is an important parameter with direct impact on the achieved performance (for example, consensus latency).

The study presented in Borran *et al.* (2008) confirms the importance of using appropriate timeouts. The authors evaluate the performance of the Paxos consensus protocol (Lamport, 1998) extended with a communication layer for wireless ad hoc networks, using different timeout values. The evaluation results show that the timeout value has considerable impact on the consensus latency and throughput. The protocol is based on a fixed timeout, which must be properly adjusted for a given execution platform. The authors indicate that adaptive timeouts could be used when message delays are unknown, but do not point to any concrete solution that could be used for that purpose.

When the actual timeout value that is suitable for the system is unknown, a usual approach is to initiate the protocol with a small timeout. Since this value may be too small, leading to timer expiration, it will be increased every time this happens (or when the protocol detects that a mistake was made due to the small timeout). The reason behind this approach is that the timeout will eventually become large enough to represent an upper bound on message delays, and the protocol will terminate. Examples of consensus protocols that implement this technique are found in Freiling & Völzer (2006) and Aguilera *et al.* (2004). The drawback of this approach is that a small period of network instability may lead to an excessively large timeout value, possibly compromising the overall performance of the protocol in subsequent executions.

Sampaio *et al.* (2003), Sampaio & Brasileiro (2005), and Sampaio *et al.* (2005) propose consensus protocols which are adaptive according to system timeliness. The authors extend the consensus protocol introduced in Chandra & Toueg (1996) with slowness oracles in order to improve the consensus execution time. Slowness oracles provide information about the responsiveness of system processes, which is measured from the round trip delays of messages exchanged during the consensus execution. Processes send their slowness information within their consensus messages, so their slowness oracles keep a consistent global view of processes responsiveness. The consensus protocol executes in rounds based on the rotating coordinator paradigm, and uses a failure detector to encapsulate synchrony assumptions. In Sampaio *et al.* (2003), the slowness oracles indicate the best (most responsive) process to be the coordinator of the first round of each consensus execution. The coordinators of the next rounds follow a round robin schedule on the ordered list of processes. This idea is extended in Sampaio & Brasileiro (2005) by implementing an adaptive process ordering module. This module uses the slowness oracle information to determine the order of processes to be the coordinators for all rounds in the consensus execution. Finally, in Sampaio *et al.* (2005) timing information is used not only by the slowness oracle, but also to set the failure detector timeout. Two failure detector implementations are used: a push style failure detector (FD-Push), and a pull style failure detector (FD-Pull). The FD-Push increases its timeout by 1 ms whenever a mistake is detected. In the FD-Pull, if a process p_i receives a response (*"I'm alive"* message) from a process p_j that is currently in its suspected list, p_i computes the round trip delay corresponding to this response, which is used as the new timeout for p_j.

In Chapter 5 we discuss the importance of using adaptive timeouts in distributed protocols operating in dynamic environments. We present our methodology to create an adaptive version of a timeout-based consensus protocol designed to operate in wireless ad hoc networks, using the *Adaptare* framework. Our evaluation compares the performance of the original protocol and its adaptive version, and shows the benefits of using adaptive timeouts.

2.3.2.2 Adaptive failure detectors

Failure detectors are oracles that provide information about the status of processes in a distributed system. As explained in Section 2.1.3, processes are constantly monitored through the exchange of periodic messages (either spontaneously as a heartbeat, or in response to query messages from monitoring modules). This approach relies on timeouts in order to determine if processes are alive or crashed: processes are suspected of being crashed if heartbeats/responses are not received within the timeout. Timeout selection is hence a fundamental issue in the configuration of failure detectors, with high impact on the quality of failure detection: a too high timeout compromises the detection latency, while a too small timeout increases the number of mistakes (false suspicions).

The first systematic study of the QoS of failure detectors was presented in Chen *et al.* (2002), where the authors define a set of QoS metrics, which are independent of failure detector implementations, as follows:

- Detection time (T_D): time that elapses from p_j's crash to the time when p_i starts suspecting p_j permanently;
- Mistake recurrence time (T_{MR}): time between two consecutive false suspicions;
- Mistake duration (T_M): time taken by the failure detector to correct a false suspicion.

Those metrics are generally used to measure and compare the overall quality of service of failure detectors. Falai & Bondavalli (2005) and Nunes & Jansch-Porto (2004) perform comparative studies of adaptive failure detectors in which different approaches for timeout estimation are used, with the objective of improving the provided QoS. In both works timeouts are derived from an estimator based on a number of observed delays, and different mechanisms to compute safety margins.

Timeout estimators can vary on their complexity. Some estimators employ simple methods, like using the delay of the last received message (Falai & Bondavalli, 2005) or the average delay of the last n received messages (Bertier *et al.*, 2002; Falai & Bondavalli, 2005; Nunes & Jansch-Porto, 2004), while others rely on more elaborated approaches, for example, applying an auto regressive model to build a time series from the last delays, and using this model to predict the delay of the next message (Falai & Bondavalli, 2005; Nunes & Jansch-Porto, 2004).

Safety margins may be static or dynamic. Static safety margins, as used in Chen *et al.* (2002), although simpler, require some *a priori* analysis of the execution environment in order to be appropriately defined. The flexibility provided by dynamic safety margins studied in Bertier *et al.* (2002), Falai & Bondavalli (2005) and Nunes & Jansch-Porto (2004) makes them more suitable to network environments susceptible to frequent changes.

The failure detector presented in Bertier *et al.* (2002) implements an adaptation layer to dynamically adjust not only the timeout value, but also the interrogation period (frequency of monitoring). Timeouts are computed by the average of last n observed delays, plus a safety margin based in the Jacobson's algorithm (Jacobson, 1988). The interrogation period is adapted through an heuristic that considers both network load, network capacities and applications needs. More specifically, two QoS values are analyzed: the required quality, which is the minimum quality of detection (specified in terms of detection time) required by the application, and the ability quality, which is the maximum quality of detection that the network can provide (depending on the current load and available bandwidth). The interrogation period is then computed to secure the quality of detection speficied by the minimum of those two values.

Fetzer *et al.* (2001) observe that the monitoring approach implemented by failure detector modules requires the exchange of control messages which are sent in addition to regular application messages, wasting network resources. They address this issue with a so-called "lazy" adaptive failure detection protocol which relies as much as possible on application messages to monitor other processes: control messages are used only when no application message is sent to the monitored processes. Every time a monitored process p_j receives either an application message or a control message from a monitor process p_i, it sends a confirmation message back to p_i (ack message). If p_i does not receive this ack message within an adaptive timeout, it suspects that p_j is crashed. The protocol is adaptive in the sense that the overhead of control messages is reduced when the applications activity increases, and the timeout is adjusted in runtime according to the highest observed round trip delay. The authors point out that (i) when the system becomes partially synchronous (i.e., there is a time after which communication and processing delays are bounded), the proposed protocol implements an eventually perfect failure detector, and (ii) using the maximum observed round trip delay as timeout reduces the number of mistakes. Nevertheless, this approach presents the same problem observed in some adaptive consensus protocols discussed in the previous section: the timeout never decreases and a small period of network instability (with high latencies) may result in a too large timeout being adopted for the rest of the execution, compromising the detection time of the failure detector thereof.

The adaptive failure detectors presented above share a significant weakness: the adaptation process is not QoS-driven. Apart from the solution proposed by Fetzer *et al.* (2001), the algorithms for timeout estimation require *a priori* configuration of operational parameters that are not explicitly related to QoS metrics, but have direct influence on them. Thus, in order to meet a desired QoS level, some underlying parameters must be carefully adjusted for a specific execution environment. Whenever the network changes or a different QoS is required, these parameters must be reconfigured.

Chen *et al.* (2002) introduce the first QoS-driven failure detector. They assume that message delays and losses present a probabilistic behavior, and use a configuration module to compute the timeout and interrogation period based on simple parameters for the characterization of network delays (loss probability, expected value, and variance), and QoS metrics given as input. The QoS metrics are: an upper bound on the detection time (T_D^U), a lower bound on the average mistake recurrence time (T_{MR}^L), and an upper bound on the average mistake duration (T_M^U). Although the algorithms and evaluation presented in Chen *et al.* (2002) do not contemplate adaptive failure detectors, the authors argue that their failure detector could be made adaptive by periodically re-executing the configuration steps, which would compute new values for the timeout and interrogation period based on network parameters estimated from the most recent heartbeats.

Very recently, a new QoS-driven failure detection approach was introduced in de Sá & de Araújo Macêdo (2010), which is based on the feedback control theory to compute and adjust both timeout and interrogation period in runtime, according to measured delays and required QoS parameters. Timeouts are computed from an estimator that uses past measured delays, plus a safety margin based on the observed accuracy of the detection service. The novelty and fundamental difference of the approach with respect to the state of the art lies in the way the interrogation period is computed. The solution relies on a feedback control loop mechanism that estimates the relationship between delays and the interrogation period. Given the good results achieved with this approach, the authors argue that it is particularly attractive for the implementation of autonomic systems.

In Chapter 6 we present *Adaptare-FD*, a dependability-oriented adaptive failure detector built on top of *Adaptare*. *Adaptare-FD* receives as input an upper bound on the detection time (T_D^U) and a lower bound on the coverage of failure detection (C^L), and adjusts the timeout and the interrogation period to meet the required QoS level. We evaluate *Adaptare-FD* in comparison with an adaptive version of Chen's failure detector and with several failure detectors based on different timeout estimators, proposed in Falai & Bondavalli (2005) and Nunes & Jansch-Porto (2004).

2.4 Summary

In this chapter we presented an overview of fundamental concepts and of several works in the relevant research areas.

We revisited several models of synchrony proposed in the literature to represent timeliness properties in distributed systems. In Chapter 3 we describe in details our approach for dependable adaptation, which is based and refines the stochastic model presented in this chapter.

We motivated the need for monitoring mechanisms by reviewing several works that perform network characterization and achieve very divergent results. Our framework for network characterization and adaptation support, *Adaptare*, is introduced in Chapter 4.

Finally, in this thesis we are concerned with dependability guarantees and QoS adaptation. Quality of service is a term with different meanings in the literature. We discussed several works that consider QoS in terms of network latencies as we do in our work, and propose adaptive solutions for different problems. In particular, we focused on adaptive consensus and failure detector protocols, due to their importance for the design of distributed systems. Our solutions for these problems, using *Adaptare* as a dependable timeout provisioning service, aim at evaluating the benefits of our approach for dependable adaptation in practical applications. They are presented in Chapters 5 and 6, respectively.

Chapter 3

Adapting for Dependability

Distributed protocols executing in uncertain environments, like the Internet or ambient computing systems, should dynamically adapt to environment changes in order to preserve Quality of Service (QoS). Furthermore, the adaptation process should be dependable, if correctness of protocol properties is to be maintained.

In our work we assume that during its lifetime, a system alternates periods where its temporal behavior is well characterized (stable phases), with transition periods during which a variation of the environment conditions occurs (transient phases). Our method is based on the following: if the environment is generically characterized in analytical terms, and we can detect the alternation of these stable and transient phases, we can improve the effectiveness and dependability of QoS adaptation.

This chapter discusses the theoretical foundations of our approach for dependable adaptation, presenting the assumptions and methodology that we propose in this thesis. In Section 3.1 we define dependable adaptation and describe its objective. Section 3.2 states the assumptions that we made about execution environments and applications. In Section 3.3 we introduce our approach for dependable adaptation, explaining the main activities involved in the adaptation process. Section 3.4 presents a more detailed view of our methodology for adaptation, including its operation and reaction according to changes in the execution environment. Finally, in Section 3.5 we analyze the necessary conditions for the correctness of our approach, and discuss how the dynamics of environments and applications may affect the dependability of the adaptation process.

3.1 Dependability goal

The fundamental objective of our work is to provide the means for adaptive applications to behave dependably despite temporal uncertainties in the operating environment. In these settings, the classical design approach of assuming fixed upper bounds for temporal variables, such as communication delays, is typically not appropriate. One possibility is to set very high upper bounds, preventing the occurrence of timing faults (i.e., preventing assumed bounds to be violated) but possibly compromising system performance by slowing down the execution. The other option is to set more aggressive (lower) bounds for performance reasons, which increases the probability of timing faults and, consequently, may have a negative impact on system correctness.

We argue that in these environments, dependability must be equated through the ability of the system to secure some bounds while adapting to changing conditions. As explained in Verissimo & Casimiro (2002), for dependable adaptation to be achieved it is sufficient to secure a *coverage stability* property. In concrete terms, this means that given a system assumption, the effective coverage of this assumption (the probability that the assumption is satisfied) will stay close to some assumed value during the system execution, and that difference is bounded.

For example, an assumption could be "the round-trip delay is bounded by T_{RTT}". There is a probability that this assumption will hold during some observation interval (coverage of the assumption). This coverage will vary due to changes in the environment conditions (e.g., load variations over time). On the other hand, the application can adapt the assumed bound T_{RTT} to increase or decrease its coverage. The objective of dependable adaptation is to select the adequate bound T_{RTT}, so that its effective coverage will be controlled in some manner.

This means that QoS is no longer expressed as a single value, a time bound to be satisfied, but as a $\langle bound, coverage \rangle$ pair, in which the coverage should remain constant while the bound may vary as a result of adaptation, to meet the conditions of the environment. A practical approach, which is the one we adopt in this work, is to ensure that the observed coverage is always higher than the specified one, while the bounds for the random variables are as small as possible.

Nevertheless, deciding when and how to adapt depends on how the environment is assumed to behave and on the concrete approaches for monitoring this behavior.

These issues are addressed in the remaining of this chapter, where we describe what we expect from the execution environment and from the applications, and how our methodology for dependable adaptation should operate on the considered model.

3.2 Assumptions

In Chapter 2 we presented the stochastic model, in which the environment is assumed to behave stochastically. Based on this model, Casimiro & Verissimo (2001) introduce the architectural and functional principles for dependable adaptation. The authors consider that the probability distribution that describe the environment behavior is unknown, and propose generic mechanisms (valid for all distributions) to support adaptation. In this thesis we advance on their work by extending the stochastic model with less conservative assumptions about the environment dynamics, in order to support improved and still dependable adaptation. We make the following assumptions:

- **Interleaved stochastic behavior:** We assume that the environment alternates stable periods, during which it follows some specific probability distribution, with transition periods, where the distribution is unknown or cannot be characterized. This assumption is supported by the results of many works (Hernandez & Phillips, 2006; Papagiannaki *et al.*, 2003; Piratla *et al.*, 2004).

- **Sufficient stability:** We assume that the dynamics of environment changes is not arbitrarily fast, i.e., there is a minimum duration for stable periods before a transition period occurs. This is a mandatory assumption for any application that needs to recognize the state of a dynamic environment.

Additionally, we also need to make assumptions concerning application-level behavior and the availability of resources to perform the needed computations in support of adaptation decisions.

- **Sufficient activity:** Mechanisms for network monitoring and characterization are based on the analysis of a set of sequential measures (samples) of the stochastic variable under observation. This set of samples is called *history*. Thus, we assume that there is sufficient system activity, allowing enough samples to be obtained in order to fill the history, as required to feed probabilistic recognition mechanisms. For instance, if message round-trip durations are being observed, then there will be statistically sufficient and independent message transmissions to allow characterizing the state of the environment. Obviously, system activity depends on the application. We believe that this is an acceptable assumption in most practical interactive and reactive systems.

- **Resource availability:** The system has the computational resources (processor, memory, etc.) which are needed to execute the detection and recognition mechanisms in a sufficiently fast manner.

Interestingly, it is easy to see that all these assumptions are somehow interdependent. As with any control framework, for a good quality of control it is necessary to ensure that the controller system is sufficiently fast with respect to the dynamics of the controlled system. In our case, the required resources and application activity depend on the effective dynamics of the environment. A balance between these three aspects must exist so that it becomes possible to dependably adapt the application.

Fortunately, the systems we want to address usually present intensive interaction patterns (with the environment or between distributed components) in sufficiently stable settings, hence meeting the requirements implied by our assumptions: the evaluation presented in Chapter 4 aimed at raising evidence that these requirements indeed hold in practical settings. Using traces of real systems' executions we implicitly test the satisfaction of sufficient activity, sufficient stability and interleaved stochastic behavior assumptions. To reason about resource availability, we provide execution measurements in specific computational platforms and conduct a complexity analysis of the implemented algorithms.

Our methodology for supporting dependable adaptation is described in the next sections. Then, in Section 3.5 we discuss the required conditions in terms of network dynamics and applications activity for the correctness of the proposed approach.

3.3 Environment recognition and adaptation

The adaptation process proposed in this thesis is composed by two activities: identification of the current environment conditions and QoS adaptation.

- **Environment recognition:** The environment conditions can be inferred by analyzing a real-time data flow representing, for example, the end-to-end message delays in a network. When the analytical description of the data is not known, we need to determine the model that best describes the data. Using statistics, the data may be represented by a cumulative distribution function (CDF), defining the probability distribution of a real random variable X. We realize that the data models can be so complex that they cannot be described in terms of simple well-defined probability distributions. Whenever the model that describes the data is known, the problem is reduced to estimating parameters of a known model from the available data.

- **QoS adaptation:** Once the best fitting distribution (together with its parameters) has been identified, its statistics properties can be exploited to find a pair $\langle bound, coverage\rangle$ that will satisfy the coverage objective for the assumed bound throughout the execution. Compared to the method defined in Casimiro & Verissimo (2001), the pair obtained with our approach is better, since the coverage stability objective can be reached using tighter (and thus lower) bounds during periods of system stability.

3.4 The adaptation approach

As introduced in Section 3.2, we assume an interleaved stochastic behavior of the environment, i.e., we consider that the system alternates *stable periods*, during which the statistical process that generates the data flow is under control and we can compute the corresponding distribution using an appropriate number of samples (history), with *transient periods*, in which the environment conditions are changing, then the associated statistical process is actually varying and no fixed distribution can describe its real behavior.

Our approach for dependable adaptation works as follows. During transient periods, a conservative approach is adopted, by setting the pair $\langle bound, coverage\rangle$ using the one-sided inequality of probability theory (Allen, 1990), as in Casimiro & Verissimo (2001). The computed bound is a conservative value that holds for all distributions. As soon as a stable phase is detected, which is equivalent to being able to identify a probability distribution that describes the analyzed history within some predefined error margin, an improved (lower) bound can be computed according to the identified distribution. Both conservative and improved bounds are *safe* from the perspective of securing the coverage stability property.

The mechanisms that are used to identify proper probability distributions, and thus stable phases, are based on mathematical and statistical methods for the analysis of sample input data. We call them *phase detection mechanisms*, since they implicitly indicate if the environment is in a transient or in a stable phase: they test the history of measured samples against a finite set of probability distributions in order to determine the best fitting distribution. In reality, even if no distribution is detected, it is still possible for the system to be stable, but its behavior is described by some distribution not considered in the mechanisms. However, since we cannot test for all probability distributions, if the analyzed history does not fit in any of the considered distributions, the environment is assumed to be in a transient period.

The phase detection mechanisms must be continuously executed, processing new incoming data, and producing indications of the actual state of the environment. Consequently, time bounds are also continuously produced, and continuously changing, even if the environment remains in a stable phase. This directly results from small changes in distribution parameters, steaming from the continuous update of the set of sample values used in the analysis, which does not necessarily imply a phase change. In other words, every time a new observation of the stochastic variable is obtained, the history is updated and a new bound is computed. This means that several bounds are provided throughout a single phase (either stable or transient), due to variations in parameters that are derived from the history. In that sense, even when the environment changes, if this change is smooth enough, it is possible that distributions and their corresponding parameters are continuously adjusted according to new incoming samples in a way that the characterization remains stable.

Figure 3.1 shows how our adaptation process operates with the help of an example scenario, identified by the following temporal events:

- Before time T0 the environment is stable and improved bounds are computed according to the detected probability distribution.

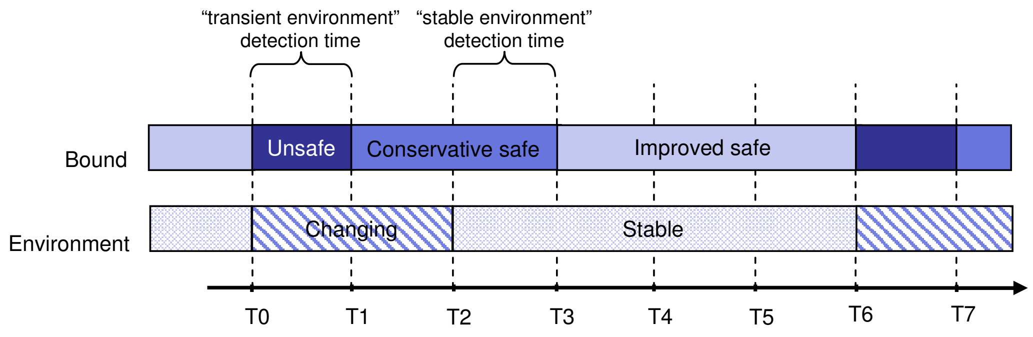


Figure 3.1: Adaptation operation.

- At time T0 the environment starts to change. The current assumed bound, computed for the previous stable period, is no longer safe (we discuss this situation in detail below).

- At time T1 the phase detection mechanisms detect the transient phase and the bound is updated to a safe but conservative value as in Casimiro & Verissimo (2001). This bound varies during the transient phase as new samples are obtained, following the conservative approach.

- At time T2 the environment reaches a new stable phase.

- At time T3 the phase detection mechanisms start identifying the stable phase. During the stable period, improved bounds (tailored for the corresponding distribution) can be continuously computed, as new samples from this phase are collected.

- At time T6 the environment conditions start changing again.

- At time T7 the phase detection mechanisms detect that the environment is changing and the bound is set to a new conservative safe value, and so on.

Note that there is an alternation of periods during which the bound is unsafe ([T0;T1]), safe but conservative ([T1;T3]), and improved and safe ([T3;T6]). Note also that the relative lengths of the represented intervals do not reflect reality. The objective is just to exemplify the alternation of phases, not their durations. Given the alternation of these phases, the effectiveness of our approach clearly depends on: (i) the "stable environment" detection time, which is the time needed to detect that the environment has reached a new stable configuration; and (ii) the "transient environment" detection time, which is the time that it takes to detect that the environment is changing.

The "stable environment" detection time does not affect the correctness of our approach: while stability is not detected, the application will rely on a conservative but safe bound, which means that the system dependability is not compromised. The only problem caused by a large "stable environment" detection time is that the conservative bound has a negative impact on the system performance, which could be improved if a better (improved safe) bound was used.

The "transient environment" detection time is particularly important. During the unsafe periods, the environment is changing but the bound is tailored for a particular set of conditions that does not hold anymore. In other words, *the "transient environment" detection time has direct impact on the dependability of adaptation.* In real environments it is not possible to *a priori* quantify the impact of detection time on the achievable dependability. However, it is possible to measure the achieved coverage in order to verify if the requirements are satisfied (which we do in our evaluation in Chapter 4) and, provided that the stated assumptions are met, this impact will be negligible. If the ratio between the duration of safe periods ([T1;T6]) and unsafe periods ([T0;T1]) is large enough, then the long term assurance of the coverage stability property can be guaranteed. Therefore, to prevent that the impact of unsafe periods compromises the dependability of the proposed solution, two conditions must be fulfilled:

- The system must be stable for sufficiently long periods of time. This is precisely what we state in the *sufficient stability* assumption (see Section 3.2) and therefore this may be considered as given by assumption.

- The "transient environment" detection time must be just as small as needed to ensure the large ratio mentioned above. This is achieved both by construction (e.g., implementing phase detection mechanisms with fast reaction to variations) and based on the assumption of *sufficient activity* (i.e., there will be enough observation points to ensure a fast detection).

In summary, the conditions above state that our approach requires sufficient stability from networks and sufficient activity from applications, which leads to the following question: *how much is sufficient?* This question is discussed in the next section, where we assess the necessary conditions to ensure the correctness of the proposed methodology.

3.5 Securing dependability

Our approach for dependable adaptation relies on assumptions about network stability and application activity, that we presume to be "sufficient". Despite being difficult or even impossible to verify if these assumptions are met in some real system, we believe that understanding their correlation and influence on the dependability assurance may help system designers to properly configure their applications, taking full advantage of the adaptation process. For the analysis presented in this section, we consider the following:

- A time-related stochastic variable X is monitored. X takes value x_i at time t_i. X values are measured and used to fill the history of samples, which is further analyzed by the phase detection mechanisms.
- At time t_i there is an upper bound b_i on X, which must hold at least with coverage C_{min} (given by the client application). This bound is adjusted in runtime according to the output of the phase detection mechanisms, in order to secure the required coverage C_{min}.
- The history of samples has a fixed size h. Whenever a new sample x_i is measured, the history is updated by removing the oldest sample and adding the new one. The updated history is then analyzed by the phase detection mechanisms, and a new bound b_i is computed.
- A timing fault occurs at time t_i if X takes a value x_i which is higher than the current (last computed) bound b_{i-i}.

- The observed coverage C is then a function of timing faults and the total number of observations (measured samples of X):

$$C = 1 - \frac{\text{number of timing faults}}{\text{number of measured samples}}$$

- We say that the system is dependable in a given observation interval if the achieved coverage during this time interval is at least as high as the required coverage:

$$C \geq C_{min}$$

Clearly, system dependability depends on the number of timing faults. Thus, it is fundamental to understand the conditions that lead to these faults. A timing fault may be caused by two different factors:

1. A bound b_i (computed at time t_i) is required to hold with coverage C_{min}. Thus, there is a probability $P = 1 - C_{min}$ that, at time t_{i+1}, X takes a value x_{i+1} that is higher than b_i, producing a timing fault. Note that this may occur during a period of system stability: it is only related to the required coverage.

2. If the system starts to change at time t_{i+i}, the current bound b_i is no longer safe, since it was computed for stable conditions that are not verified anymore. Then, it is possible that X takes a value x_{i+1} which is higher than b_i, causing a timing fault. In fact, while the phase detection mechanisms are not able to detect that the system is changing ("transient environment" detection time), the possibility of timing faults remains.

In the first case, timing faults can be seen as expectable, given the coverage that is specified by the client application. They are *acceptable* faults, under the probabilistic perspective we assume in this work. In order to minimize these faults, the application should require a higher coverage.

In the second case, timing faults derive from changes in the environment. As we cannot anticipate when the environment will start to change, those timing faults are unavoidable. The best we can do is to minimize the number of those faults by implementing mechanisms with fast reaction to changes. As discussed in the previous section, the “transient environment” detection time must be as small as possible. Take as example a history composed by h samples, all of them collected during a stable phase, which is detected by the phase detection mechanisms. Then, the environment starts to change. New observations from this transient period will be periodically collected and added to the history. This means that during a period of time (until h observations from the transient phase are obtained) the history will be composed by samples from both the previous stable and the current transient phases. Since the phase detection mechanisms are re-executed every time a new sample is added to the history, at some point, the history will have enough observations from the transient period, so the phase detection mechanisms will correct the characterization from stable (previous phase) to transient (current phase). The more observations are required to make this correction, the longer will be the “transient environment” detection time. Thus, in order to reduce the number of faults derived from environment changes, it is necessary to minimize the number of observations that must be in the history so that the phase detection mechanisms are able to detect that a transient period has started.

As timing faults caused by environment changes are unavoidable, the maximum coverage that can be secured by our approach is limited. Assuming that the phase detection mechanisms require n observations from a transient period in order to detect the changing environment and compute a conservative safe bound, the upper bound on the securable coverage C^U is:

$$C^U = 1 - \frac{\text{number of transient periods} \times \text{n}}{\text{number of measured samples}}$$

In the best possible solution, one sample from a transient period in the history would be enough to recognize that the system is no longer stable ($n = 1$). Interestingly, in our practical experiments presented in the next chapters we could observe that usually our approach does not produce consecutive timing faults, which suggests that our phase detection mechanisms are as reactive as possible, detecting transient periods from a single sample in the history. In terms of real time, the ‘transient environment” detection time will be dictated by the application activity - the detection will happen as soon as a new sample is measured.

While we realize that it is not possible for the application designer to quantify how many times the environment will suffer significant changes, we also believe that the discussion above is relevant to understand how system dependability is affected by the environment dynamics and applications activity. In a real system, assuming that our mechanisms are correct, if the coverage is not secured, it means that the environment is more dynamic than it was expected. In this case, a possible solution would be to increase the application activity or artificially create activity (despite other negative effects that could arise). By doing that, the stochastic variable would be sampled more frequently, improving the achieved coverage. Obviously, there is a limit for the number of measurements an application can perform in a given period of time. When system activity cannot be increased and the environment dynamics does not allow to secure the required coverage, it means that the sufficient stability assumption does not hold, and thus our solution is not applicable to that environment.

3.6 Summary

In this chapter we addressed the problem of supporting adaptive systems and applications in stochastic environments, from a dependability perspective: maintaining the correctness of system properties after adaptation. Therefore, while other works addressing adaptive systems are mainly concerned with performance, our main concern is dependability.

The concepts and methodology for dependable adaptation presented in this chapter constitute the basis of our work. Although it is not possible to know, *a priori*, if our assumptions (particularly "sufficient stability" and "sufficient activity") are met in real systems, we discussed and analyzed the necessary conditions for the correctness of our approach. Furthermore, through this analysis we were able to establish the theoretical upper bound on the securable coverage in terms of environment stability and application activity.

In the next chapter we introduce and evaluate *Adaptare*, a framework for supporting dependable adaptation, which implements the methodology proposed in this chapter.

Notes

An early version of our methodology for dependable adaptation, containing parts of the theory presented in this chapter, appeared in "A framework for dependable QoS adaptation in probabilistic environments", Dixit, Casimiro, Verissimo, Lollini, and Bondavalli, "Proceedings of the 23rd ACM Symposium on Applied Computing", Fortaleza, Brazil, March 2008 (Casimiro *et al.*, 2008).

This theory, in particular the model proposed in this chapter, was described in "*Adaptare*: Supporting automatic and dependable adaptation in dynamic environments", to appear in "ACM Transactions on Autonomous and Adaptive Systems" (Dixit *et al.*, 2011).

Chapter 4

Adaptare

In this chapter we present *Adaptare* - a complete framework that implements the ideas for automatic and dependable adaptation presented in the previous chapter. The chapter is composed by four main sections. Section 4.1 presents *Adaptare*'s architecture. Algorithms and implementation details are discussed in Section 4.2. The API that allows for *Adaptare*'s integration with client applications is described in Section 4.3. The modular design of *Adaptare* makes it easily extensible, thus it is possible to add new mechanisms and algorithms according to system requirements. However, our implementation is complete enough to be successfully applied in a variety of systems, as we demonstrate in Section 4.4. Our evaluation is based on synthetic data flows generated from probability distributions, as well as on real data traces collected in various Internet-based environments. We also compare our solution with other approaches for adaptation and we show that *Adaptare*, albeit more complex, is very effective, allowing protocols to adapt to the available resources in a dependable way.

4.1 Architecture

Adaptare is a framework developed to support adaptive systems and applications operating in stochastic environments, driving the adaptation process according to dependability requirements specified by the client applications.

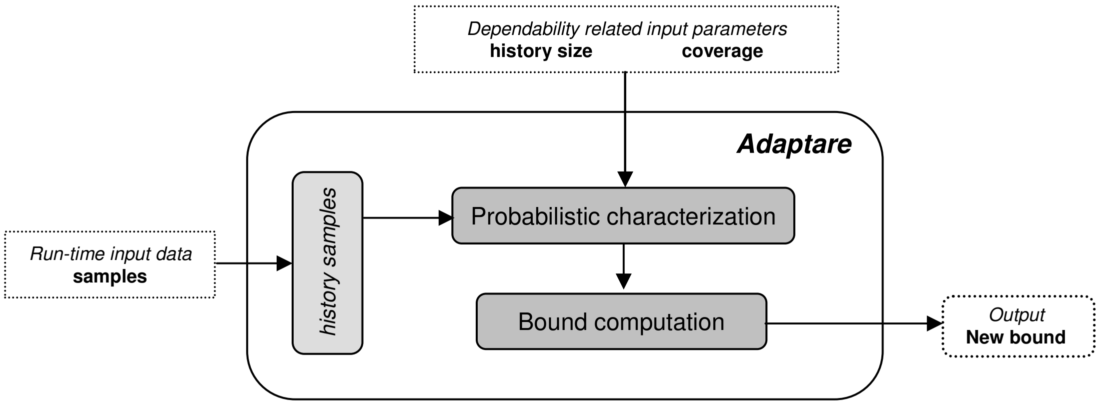


Figure 4.1: *Adaptare* overview.

Figure 4.1 presents an overview of *Adaptare*'s operation. Three parameters must be provided at the framework interface:

- **samples**: recent measurements of the monitored temporal variable;
- **history size** h: number of samples that will be analyzed by the framework; and
- **coverage**: required coverage for the computed bound, which is the probability that this bound will hold given the current system conditions.

Then, *Adaptare* performs a probabilistic analysis of the samples in the history, in order to determine an upper bound on the observed variable with the given coverage. This upper bound is returned to the client application.

Probabilistic characterization. *Adaptare* employs a set of statistical methods for the analysis of input samples in order to determine whether the system is in a stable period and, if this is true, which probability distribution better describes the current state.

Bound computation. Once the environment characterization is completed, *Adaptare* is able to compute a new dependable upper bound for the observed variable. For stable phases, this bound is derived from the cumulative distribution function (CDF) of the identified probability distribution. During transient periods, *Adaptare* computes a conservative bound which holds for all distributions.

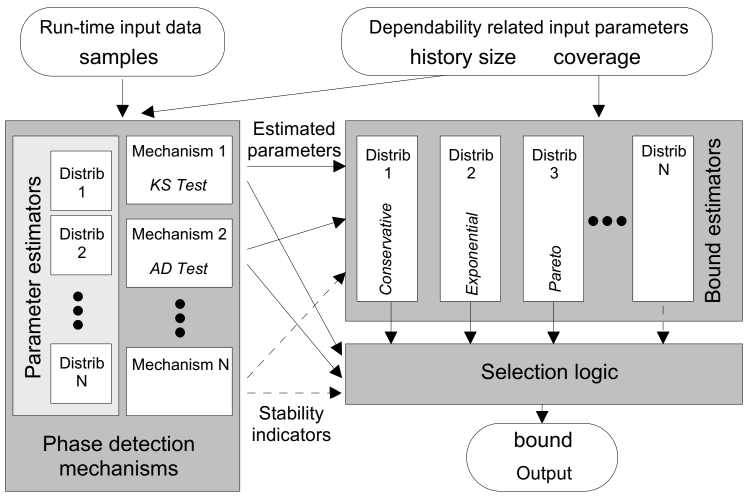


Figure 4.2: *Adaptare*'s architecture.

The scheme depicted in Figure 4.2 shows the architecture of *Adaptare*. Internally, the service admits the use of several phase detection mechanisms, as well as several bound estimators, corresponding to different probability distributions. This openness creates the possibility of extending the framework with different phase detection mechanisms, as well as probability distributions which are found to be more suitable to some specific environment. Because of these possible multiple phase detection mechanisms, several bounds may be selectable as candidates to the output value. Therefore, a selection logic must be implemented to choose only one of the available bounds. The following section addresses the implementation of all these internal mechanisms.

4.2 Implementation

4.2.1 Phase detection mechanisms

We implemented two goodness-of-fit (GoF) tests as phase detection mechanisms: the Kolmogorov-Smirnov (KS) test and the Anderson-Darling (AD) test. GoF tests are formal statistical procedures used to assess the underlying distribution of a data set. A stable period with distribution F is detected when some GoF tests establish the

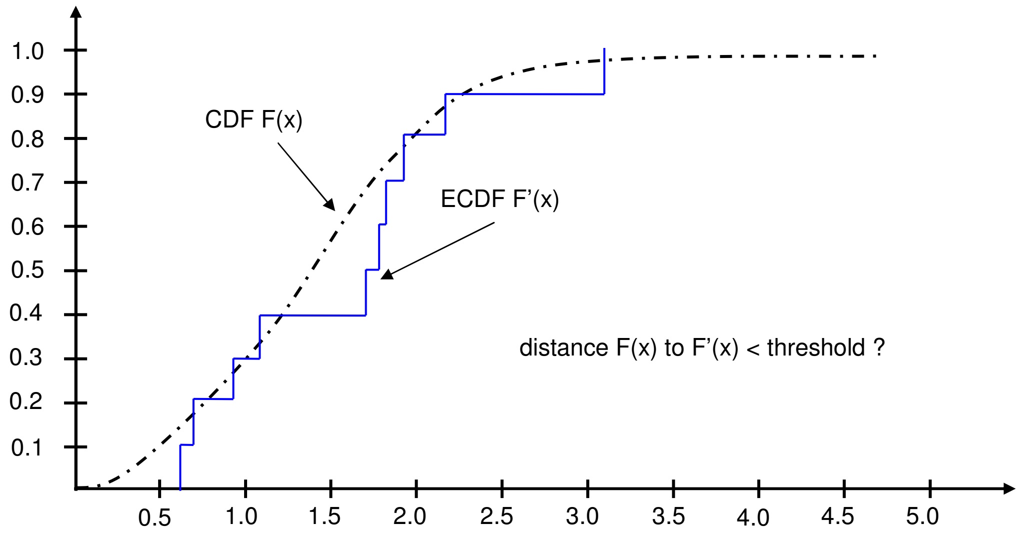


Figure 4.3: GoF distance test operation.

goodness of fit between the postulated distribution F and the evidence contained in the experimental observations (Trivedi, 2002). Both AD and KS are distance tests based on the comparison of the cumulative distribution function (CDF) of the assumed distribution F and the empirical cumulative distribution function (ECDF), which is a CDF built from the input samples, as shown in Figure 4.3. If the assumed distribution F is correct, its CDF closely follows the empirical CDF (the "distance" between them is less than a given threshold).

Each phase detection mechanism performs the following steps:

1. Given a sample of size n, order the sample points to satisfy $x_1 \leq x_2 \leq ... \leq x_n$;

2. Assume a distribution F with CDF $F_0(x)$ (the parameters of the postulated distribution F are previously estimated using the methods described in Section 4.2.2);

3. Compute the method statistic S_m;

4. If $S_m \leq s_{n;\alpha}$, the test accepts that the sample points follow the assumed distribution F with significance level α, and a stable phase is detected. Otherwise, a transient phase is detected. In hypothesis testing, the significance level defines the probability of rejecting the null hypothesis when it is correct. In the context

of *Adaptare*, the null hypothesis is that the analyzed data follows the assumed distribution F, thus α is the probability that a stable period is wrongly recognized as a transient one. The value of $s_{n;\alpha}$ is obtained from published tables of critical values (existing for both KS and AD tests).

The statistic S_m for the KS method is known as D_n, and it is computed as:

$$D_n = \max_x |\hat{F}_n(x) - F_0(x)|,$$

where $\hat{F}_n(x)$ is an empirical distribution function built from the history samples:

$$\hat{F}_n(x) = \frac{\text{number of values in the history that are } \leq x}{n}.$$

For the AD method, the statistic is known as A^2, and it is calculated as:

$$A^2 = -n - S,$$

where:

$$S = \sum_{i=1}^{n} \frac{(2i-1)}{n} [\ln F_0(x_i) + \ln(1 - F_0(x_{n+1-i}))].$$

Both algorithms described above assume a probability distribution F and verify if the given sample points come from this distribution. The significance level used by *Adaptare* in both AD and KS tests is $\alpha = 0.01$, which means that there is a probability of 1% that a stable phase following one of the considered distributions is not detected. This is the lowest significance level in the tables of critical values, thus the confidence on the stability detection provided by *Adaptare* is as high as possible. We test five distributions: exponential, shifted exponential, Pareto, Weibull and uniform. The set of distributions was defined based on other works that address the statistical characterization of network delays, e.g., Bolot (1993); Downey (2001); Hernandez & Phillips (2006); Markopoulou *et al.* (2006); Papagiannaki *et al.* (2003); Tickoo & Sikdar (2004). Note that the framework can be extended to be able to deal with more distributions, if they are appropriate for the random variable which will be analyzed. If the phase detection mechanisms do not identify a stable phase with any of the tested distributions, they assume that the environment is changing and a transient phase is detected.

The main advantage of the KS test in comparison to the AD test is that the critical values for the KS statistic are independent of specific distributions: there is a single table of critical values, which is valid for all distributions. However, the test has some limitations: it tends to be more sensitive near the center of the distribution than at the tails, and if the distribution parameters must be estimated from the data to be tested (which is what we do in our implementation), the results can be compromised (Jain, 1991). Due to this limitation, there are some works that propose different KS critical values to specific distributions with unknown parameters, which are estimated from the sample. In our experiments with the KS test we used these modified tables for the exponential and shifted exponential (Trivedi, 2002), Pareto (Porter *et al.*, 1992) and Weibull (Evans *et al.*, 1989) distributions. However, the modified KS tables for the Weibull, exponential and shifted exponential distributions do not have critical values for a history size higher than 30. Thus, in these cases ($h > 30$), the standard KS table (Trivedi, 2002) is used. Regarding the Pareto distribution, the table of critical values presented in Porter *et al.* (1992) is limited to a set of shape parameters in the interval $[0.5, 4.0]$. If the estimated shape parameter is within this interval, it is rounded for the closest defined value and the modified table is applied. Otherwise, the standard KS table is used. The test for the uniform distribution is also performed using the standard KS table, since, to our knowledge, there is no modified table for this distribution in the literature.

In the AD test, there is no problem in estimating the distribution parameters from the sample points. However, this test is only available for a few specific distributions, since the critical values depend on the assumed distribution and there are published tables for only a limited number of distributions. AD critical values for the Weibull distribution can be found in Stephens (1976). For the exponential and shifted exponential distributions, we implemented the modified statistic and used the critical values proposed in Stephens (1974). Like in the KS test, AD critical values for the Pareto distribution are limited to shape parameters in the interval $[0.5, 4.0]$. Considering that there is no general table of AD critical values (independent of the tested distribution), if the estimated shape parameter is not in this interval, the AD phase detection mechanism does not test the Pareto distribution. Finally, critical values of the AD test for the uniform distribution are presented in Rahman *et al.* (2006).

The tables of critical values for both KS and AD tests used in *Adaptare* are presented in Appendix A.

For a given input sample, it is possible that a phase detection mechanism identifies more than one distribution, due to similarities between distributions, and uncertainty of the statistical methods and parameters estimation. In those cases the mechanisms return the detected distribution with the lowest statistic value, which means that the sample data are closer to that distribution.

4.2.2 Parameters estimators

Both KS and AD tests need to estimate distribution parameters in order to execute their statistical tests. There are various methods, both numerical and graphical, for estimating the parameters of a probability distribution. From a statistical point of view, the method of maximum likelihood estimation (MLE) is considered to be one of the most robust techniques for parameter estimation.

The principle of the MLE method is to select as an estimation of a parameter θ the value for which the observed sample is most "likely" to occur (Balakrishnan & Basu, 1995; Trivedi, 2002). We applied this method to estimate exponential, shifted exponential, Pareto and uniform parameters. For the Weibull distribution, the MLE method produces equations that are impossible to solve in closed form: they must be simultaneously solved using iterative algorithms, which have the disadvantage of being very time-consuming. Since execution time is an important factor in our framework, we decided to estimate Weibull parameters through linear regression, instead of using MLE. We implemented the method of least squares, which requires that a straight line be fit to a set of data points, minimizing the sum of the squares of the y-coordinate deviations from it (Trivedi, 2002).

Table 4.1 presents the equations for parameters estimation, where $\{t_1, t_2, ..., t_n\}$ is the *ordered* sample history. To estimate Weibull parameters, consider that $x_i = \ln(t_i)$, $y_i = \ln(-\ln(1-F(t_i)))$, and $F(x_i) = (i-0.3)/(n+0.4)$ (approximation of the median rankings). These equations are derived from the Weibull CDF, using the method of

Distribution	**Parameters estimators**
Exponential	$\hat{\lambda} = \frac{1}{\bar{t}}$
Pareto	$\hat{k} = t_{min}$ $\hat{\alpha} = \frac{n}{\sum_{i=1}^{n} \ln \frac{t_i}{\hat{k}}}$
Shifted Exponential	$\hat{\lambda} = n\frac{(\bar{t} - t_{min})}{n-1}$ $\hat{\gamma} = t_{min} - \frac{\hat{\gamma}}{n}$
Weibull	$\hat{\gamma} = \frac{\sum_{i=1}^{n} x_i y_i - \frac{\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n}}{\sum_{i=1}^{n} x_i^2 - \frac{(\sum_{i=1}^{n} x_i)^2}{n}}$ $\hat{\alpha} = e^{-\frac{\bar{y} - \hat{\gamma}\bar{t}}{\hat{\gamma}}}$
Uniform	$\hat{a} = t_{min}$ $\hat{b} = t_{max}$
Conservative	$E(\hat{D}) = \frac{\sum_{i=1}^{n} t_i}{n}$ $V(\hat{D}) = \frac{\sum_{i=1}^{n} (t_i - \bar{t})^2}{n-1}$

Table 4.1: Parameters estimators.

regression on Y. Due to its complexity, we will not present the derivation process in this document, but a complete explanation can be found in ReliaSoft (2006).

4.2.3 Bound estimators

Depending on the output of the phase detection mechanisms, one of the bounds computed by the implemented bound estimators will be selected as the *Adaptare* output. The current implementation has six bound estimators, as shown in Table 4.2: the conservative one, which is based on the one-sided inequality of probability theory (Allen, 1990), and provides a conservative bound which holds for all probability distributions; and estimators for the exponential, shifted exponential, Pareto, Weibull and uniform

Distribution	**CDF**	**Minimum bound** t
Exponential	$F_X(t) = 1 - e^{-\lambda t}$	$t = \frac{1}{\lambda} \ln \frac{1}{1-C}$
Pareto	$F_X(t) = 1 - (\frac{k}{t})^{\lambda}$	$t = \frac{k}{\sqrt[\lambda]{1-C}}$
Shifted Exponential	$F_X(t) = 1 - e^{\frac{(-t+\gamma)}{\lambda}}$	$t = \gamma + \lambda \ln \frac{1}{1-C}$
Weibull	$F_X(t) = 1 - e^{-(\frac{t}{\lambda})^{\gamma}}$	$t = \lambda \sqrt[\gamma]{\ln \frac{1}{1-C}}$
Uniform	$F_X(t) = \frac{t-b}{b-a}$	$t = C(b-a) + a$
Conservative	N/A	$t = E(D) + \sqrt{\frac{V(D)}{1-C} - V(D)}$

Table 4.2: Bound estimators for a required coverage C.

distributions.

The bound estimators are derived from the distributions' CDF. We recall that the CDF represents the probability that the random variable X takes on a value less than or equal to x (for every real number x):

$$F_X(x) = P(X \leq x)$$

Our objective is to ensure that a given bound is safe, i.e., that the real delay will be less than or equal to the assumed bound, with a certain minimum probability (the expected coverage). Thus, in the CDF function, X is the observed delay, x is the assumed bound provided by the framework, and $F_X(x)$ is the expected coverage. For example, the exponential CDF is:

$$F_X(x) = 1 - e^{-\lambda x} = C$$

For a given coverage C, a safe bound for the exponential distribution can be computed by isolating x:

$$x = \frac{1}{\lambda} \ln \frac{1}{1-C}$$

4.2.4 Selection logic

As explained in Section 4.2.1, the phase detection mechanisms are individually executed and each one indicates the environment condition and returns one distribution which best characterizes the analyzed history whenever a stable phase is detected. The selection logic receives these results and is responsible for selecting one of them as the output of the framework (see Figure 4.2). Since our objective is to produce improved safe bounds, *Adaptare* returns the lowest bound to the client application.

4.3 *Adaptare* as a service

The current implementation of *Adaptare* is freely available as a library in `http://www.navigators.di.fc.ul.pt/software/Adaptare.jar`. We defined a very simple programming model, in order to facilitate the integration with client applications. Thus, *Adaptare* can be easily used as a monitoring service for supporting dependable adaptation, as a plug-and-play solution. The library encapsulates the framework functionalities, which are provided to client applications through a well-defined API (Application Programming Interface), as follows:

- *static double GetBound(double cov, int h, double[] samples)*

  For specific cases in which an application needs to use the service only sporadically, it does not need to register as a client. It should invoke this static method, providing the required coverage, the history size and the sample points. The service will calculate a new bound and return it to the application.

- *int register()*

  This method registers the calling application as a service client, returning a unique identifier (for this service instance) that should be provided by the application to perform other operations. Applications are required to implement a simple interface in order to be registered as a client of this service. This is an implementation of the Observer design pattern (Gamma *et al.*, 1995), and is required for the notification of QoS changes.

- *double getBound(int id, double cov, int h)*

  This function returns a new bound to be used by the calling application. It receives as input the application identifier (so the service can determine the sample points which belong to this application), the expected coverage and the number of most recent sample points that should be used to compute the new bound.

- *void notifyQoSChanges(int id, double minCov, double maxCov, int h, double bound)*

  If an application wants to be notified about QoS changes, it should use this non-blocking function, providing its identifier, the acceptable coverage interval (minimum and maximum expected coverage), the history size and the current bound that is being used. The relation between the history size and the frequency of sample provision is a responsibility of the application. Whenever the adaptation service perceives that the coverage of this application is out of the specified interval, it will produce a new secure bound and return to the application by calling a function from the interface that clients must implement.

- *double addSamples(int id, double[] samples)*

  A registered application can add one or more sample points to its history by calling this function. If the application did not ask to be notified about QoS changes (by calling the *notifyQoSChanges* function), this function will just add the sample points and return -1. Otherwise, the service will verify if the coverage of the current bound used by this application is within the desired interval. If it is not, a new secure bound will be produced and sent to the application, and the service will associate this new bound to this application.

4.4 Results and evaluation

One of the objectives of our work is to show the possibility of having a component which is able to characterize the current state of the environment and, based on dependability requirements, infer how time-related bounds should be adapted.

In the previous sections we have introduced *Adaptare*, describing its objectives, functionalities, implemented methods and algorithms. This section presents a set of results obtained with this framework, and a systematic evaluation focusing on the following main points:

- Validation of our implementation and demonstration of the correctness of the phase detection mechanisms by comparing their ability to characterize the current conditions of the environment using controlled traces, synthetically produced;
- Quantification of *Adaptare*'s achievements by determining the real effectiveness and improvements obtained, using real RTT (round-trip time) traces;
- Quantification of *Adaptare*'s overhead through a complexity analysis of the implemented algorithms and measurements of the effective latencies using typical computing platforms;
- Performance comparison of *Adaptare* with other well-known solutions for the problem of dynamically adapting time-related variables.

4.4.1 Analysis of the phase detection mechanisms

In this first part of our evaluation, we tested our framework using synthetic data traces in order to analyze the correctness of the AD and KS phase detection mechanisms. We generated 10 traces of 3000 samples, for each one of the five considered distributions (i.e., a total of 50 traces). The distributions parameters were configured according to Table 4.3.

Adaptare was executed for each trace using the phase detection mechanisms individually. For these executions, we defined the minimum required coverage $C = 98\%$, and history size $h = 30$ samples. According to the experiments with real traces presented in Section 4.4.2, this history size ($h = 30$) produces the best overall results. Moreover, the significance level (α) of both phase detections mechanisms, which defines the probability of not detecting a stable phase, was set to $\alpha = 0.01$. We specified four metrics to compare the AD and KS goodness-of-fit tests:

- *Stability detection:* percentage of the samples that were detected as stable phases;

Traces	Exponential (λ)	Shifted Exp (λ, γ)	Pareto (λ, k)	Weibull (λ, γ)	Uniform (a, b)
1	0.8	0.8, 100	1.0, 1.0	1.0, 2.0	10, 20
2	1.6	1.6, 200	1.4, 2.0	1.4, 2.5	20, 40
3	2.4	2.4, 300	1.8, 3.0	1.8, 3.0	30, 60
4	3.2	3.2, 400	2.2, 4.0	2.2, 3.5	50, 100
5	4.0	4.0, 500	2.5, 5.0	2.5, 4.0	70, 140
6	4.8	4.8, 600	2.8, 6.0	2.8, 4.5	80, 160
7	5.6	5.6, 700	3.1, 7.0	3.1, 5.0	100, 200
8	6.4	6.4, 800	3.4, 8.0	3.4, 5.5	200, 400
9	7.2	7.2, 900	3.7, 9.0	3.7, 6.0	300, 600
10	8.0	8.0, 1000	4.0, 10.0	4.0, 6.5	500, 1000

Table 4.3: Parameters used to generate the synthetic data traces.

- *Detection correctness:* percentage of the samples characterized as stable phases in which the framework detected the correct distribution. Given that in this phase of the evaluation we used synthetic traces generated from known distributions, we are able to quantify the correctness of the mechanisms;

- *Coverage*: achieved coverage, which corresponds to the number of measured samples whose value is less than or equal to the computed bound, divided by the total number of samples;

- *Improvement of bounds*: improvement obtained by our approach in comparison with the conservative solution proposed in Casimiro & Verissimo (2001). It represents the percentage of reduction of the conservative bounds, obtained when using *Adaptare*'s bounds.

The above metrics can be analyzed when using synthetic traces generated from stable and fixed distributions. On the other hand, other metrics that could have been considered, like "stable environment" and "transient environment" detection times, would have required synthetic traces with both stable and transient periods. Although it would have been possible to create several synthetic scenarios with varied dynamics, they would not cover all the cases. The evaluation would always be partial and it would not be possible to derive any generic conclusion. For our purposes, it was enough to perform a basic comparison of the GoF tests in steady situations to conclude that there

Distribution	Stability det.		Det. correct.		Improvement		Coverage	
	AD	KS	AD	KS	AD	KS	AD	KS
Exponential	94.53	93.53	98.64	99.19	20.16	19.90	99.58	99.59
Shifted exp.	98.98	99.07	99.18	63.04	1.98	2.12	99.10	98.96
Pareto	76.26	81.75	72.96	78.41	23.62	27.20	98.33	98.25
Weibull	98.43	97.86	79.88	92.63	32.57	35.35	99.41	99.34
Uniform	55.57	48.39	94.82	70.58	9.60	7.50	100.00	100.00

Table 4.4: Comparing phase detection mechanisms using synthetic data traces.

is not a single best GoF test for all distributions. This is clear from the results in Table 4.4. The performance of each mechanism depends on its intrinsic features and on the distributions characteristics.

Regarding the considered metrics, we should note that the *detection correctness* metric can only be measured with synthetic traces (since we know the actual distribution). Because of this, only the other three metrics are used in the evaluation with real traces, presented in Section 4.4.2.

The most important result to be observed based on the values in Table 4.4 is that the two main objectives of *Adaptare* were achieved for all distributions by both mechanisms: the bounds were improved (up to 35%) and the minimum required coverage (98%) was secured. In order to fully understand the presented values, it is necessary to perform a more detailed analysis, separated by distribution.

In the experiments with exponential traces, both mechanisms reached excellent and very similar results: they detected stability in more than 90% of the samples, and correctly characterized almost all of these stable points as exponentially distributed. This significant rate of exponential detection allowed a reduction of approximately 20% in the bounds computed by our framework (comparing with the conservative bounds).

Regarding the shifted exponential distribution, almost all points were detected as belonging to stable phases by the two mechanisms. However, the KS mechanism made incorrect characterizations (recognizing other distributions instead of the shifted exponential) in more than 35% of these points. There are at least two factors that can lead to these mistakes: the inherent uncertainty associated with parameters estimation and the probabilistic mechanisms, and the similarities between the tested distributions (e.g., every exponential distribution is also a Weibull distribution with $\lambda = 1$), which is interesting in order to verify the precision of the implemented mechanisms, but also increases the possibility of inaccurate detection. Despite all these uncertainties, the characterization was correct in the majority of the sample points. Another important observation in the shifted exponential results is that while there was a high rate of stability detection, the improvement of bounds was not significant (2%), showing that the bounds produced by *Adaptare* to the shifted exponential distribution are very close to the conservative bounds.

The tests with Pareto traces presented the lowest rate of correct detection among all distributions. This is due to the technical limitation explained in Section 4.2.1: both mechanisms have Pareto critical values only for a small set of shape parameters. Thus, besides the uncertainty already present in the parameter estimation, this estimator is rounded to one of these pre-defined values, compromising the accuracy of the characterization. However, even with this limitation, the improvement of bounds for the Pareto traces was very significant (about 25% on average).

Both mechanisms presented very good results in the experiments with Weibull traces. Excellent rate of stability detection (more than 95%), good distribution characterization (80% - 90%) and the best rate of improvement of bounds (up to 35%). These are the best overall results among the tested distributions.

Finally, the results of the tests using uniform traces show that the stability detection for this distribution is the more conservative: around 50%. The AD test correctly characterized these sample points as uniformly distributed in 94.82% of the cases. However, we did not obtain the same results with the KS test, which is a good example of the KS limitation: estimating parameters from the sample is not adequate when applying the standard test, which we do for the uniform distribution, since we did not find a modified table of KS critical values for it. Consequently, this test failed in detecting the uniform distribution in approximately 30% of the stable points. Nevertheless, both mechanisms obtained almost 10% of improvement in the produced bounds, while ensuring a coverage of 100%.

Detecting wrong distributions may lead to a lower coverage, compromising the dependability of *Adaptare*. For example, we observed that the bounds produced for the shifted exponential distribution are very close to the conservative bounds, while the bounds produced for the Weibull distribution are more aggressive, with improvements of at least 30%. Thus, if a trace that follows a shifted exponential distribution is wrongly characterized as a Weibull trace, the produced bounds may be lower than the measured samples, decreasing the coverage. However, our results show that despite the inherent errors in the characterization (due to similarities between distributions, and statistical uncertainties on the parameters estimation), *Adaptare* secures the expected coverage, i.e., the rate of mistakes is low enough not to compromise the dependability of our approach.

From these results, we conclude that *it is possible to define effective mechanisms to detect stable and transient phases and, for the stable ones, correctly characterize the observed probability distribution*.

4.4.2 Validation using real RTT measurements

This section presents a set of experiments based on real data, in order to show that our assumption about the interleaved stochastic behavior is realistic for network delays in different environments, and to demonstrate that *Adaptare* is able to characterize these behaviors and provide applications with improved and dependable bounds. We selected data traces freely available in the Internet, composed by measurements collected in different wired and wireless networks, and extracted RTT traces from them using the *tcptrace* tool. These RTT traces have been provided as input to *Adaptare* (they constitute the "input samples" of Figure 4.2). Data traces were gathered from the following sources:

- **Inmotion:** Traceset of TCP transfers between a car traveling at speeds from 5 mph to 75 mph, and an 802.11b access point. A complete description of the environment, hardware and software used to generate this traceset can be found in Gass *et al.* (2006). We downloaded the tcpdump files from Gass *et al.* (2005);

- **Umass:** A collection of wireless traces from the University of Puerto Rico. Contains wireless signal strength measurements over distances of 500 feet and one mile. For more information, see UMass Trace Repository (2006);

- **Dartmouth:** Dataset including tcpdump data for 5 years or more, for over 450 access points and several thousand users at Dartmouth College (Kotz *et al.*, 2004). Experiments details are discussed in Henderson *et al.* (2004);

Statistic (ms)	Inmotion	Umass	Dartmouth	LBNL	RON
Average interval	4.42	0.39	118.97	19.34	124.45
Minimum interval	0.00	0.00	0.00	0.00	0.00
Maximum interval	2414.91	152.15	17504.08	946.42	1853.09
Standard deviation	26.16	2.06	555.35	40.79	160.66

Table 4.5: Samples statistics for each data source.

- **LBNL:** Packet traces from two internal network locations at the Lawrence Berkeley National Laboratory (LBNL) in the USA, giving an overview of internal enterprise traffic recorded at a medium-sized site. These traces are available to download in Paxson *et al.* (2007);

- **RON:** Traces containing thousands of latency and loss samples taken on the Resilient Overlay Network (RON) testbed. More information can be found in Andersen *et al.* (2001). The dataset is available at RON (2001).

For the tests with real traces, the phase detection mechanisms were executed in parallel, as illustrated in Figure 4.2. The most important parameter that must be defined by the framework's clients is the history size. This parameter states the number of most recently collected samples that are analyzed for the environment characterization, which has a direct impact on the achievable results. For this reason, we conducted a set of experiments with different history sizes for all traces. For these experiments we have set the required coverage to $C = 0.98$, and selected four traces from each data source. The size of these traces varied from 10000 to 50000 samples.

Table 4.5 shows some statistics for the interval between sample points from each data source. These values indicate the system activity when the traces were collected, which we assume to be sufficient for the characterization performed by *Adaptare* (see "sufficient activity" assumption in Chapter 3). In any case, the results would confirm if the assumption was correct. We can observe that for each data source, the interval between samples varies during execution. There is also a significant variation among data sources: e.g., in the Umass traces the measurements are more frequent (every 0.39 milliseconds on average), while Dartmouth and RON traces present much higher intervals (more than 100 milliseconds).

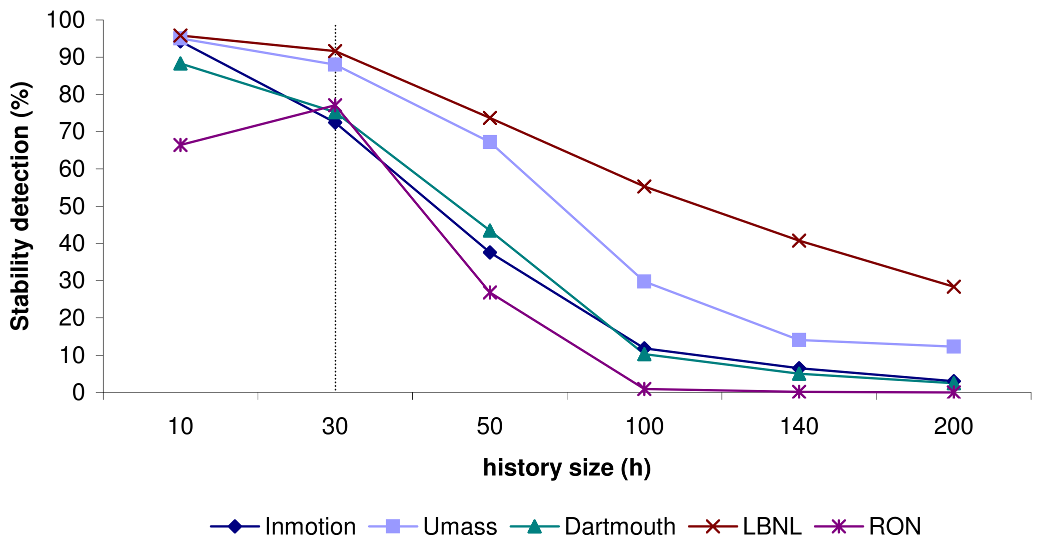


Figure 4.4: Stability detection.

The average results for each data source are presented in Figures 4.4, 4.5, and 4.6. Average values are representative enough in our evaluation because stability detection and coverage are measures of the entire trace, not individual samples. To achieve a more consolidated view, the presented results are averaged over the four values obtained for each of the traces of the given source, which were very similar. Within each trace we observed significant differences between minimum and maximum bounds produced by *Adaptare* over the entire trace. However, the variance is very small and, for all traces, our experiments shown that a confidence interval of 95% for the computed time bounds deviates from the average at most by 2%, making the average a proper indicative of the overall results.

Figure 4.4 shows the average stability detection. Each sample in a trace is characterized by *Adaptare* as either stable or unstable, thus the percentage of stable points in the total of samples determines the stability detection for a given trace.

The effects of increasing the history size are clear on the obtained values: the larger the history size is, the lower is the rate of detected stability points. Intuitively, what happens is that we are dealing with traces from real environments, subject to uncertain statistical processes, making it increasingly difficult to fit a large set of samples into one specific distribution. Taking as given that data correlation is good enough for all

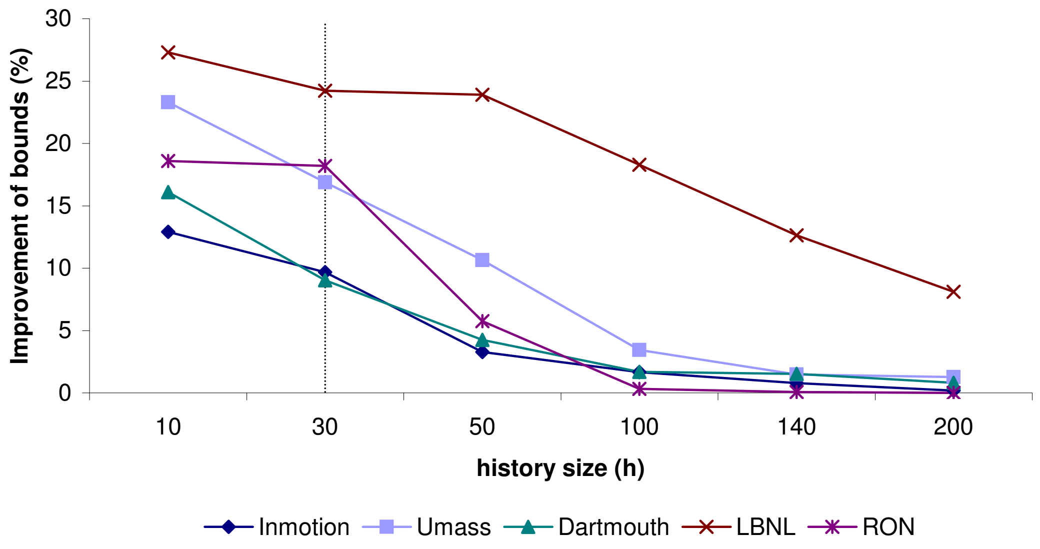


Figure 4.5: Improvement of bounds.

the considered history sizes (i.e., both the “sufficient stability” and the “sufficient activity” assumptions hold), the inability of detecting stable phases when using higher history sizes is explained by the fact that in real environments the stochastic behavior is not “pure”, i.e., it does not strictly follow one well-defined probability distribution, but approximates one or even more distributions. Therefore, fitting a big history into one specific well-defined distribution becomes impossible, which does not mean the environment is not stable, but rather that we are looking for “pure” (stable) distributions, that indeed are not there. On the other hand, fitting a small history into a specific probability distribution that is close to the real distribution becomes feasible, allowing the detection of stable phases (that exist in reality, but for approximate distributions). This is the reason why the selection of an adequate history size has a considerable impact on the results and is crucial.

Noting that the ability of correctly identifying stable phases is dependent on the correlation between the samples (they need to be collected within a sufficiently small interval of time), this result highlights the interdependence between the “sufficient stability” and the “sufficient activity” assumptions, as discussed in Chapter 3. For example, the Dartmouth and Inmotion traces have very similar stability detection, despite the significant difference in system activity (while Inmotion traces have samples

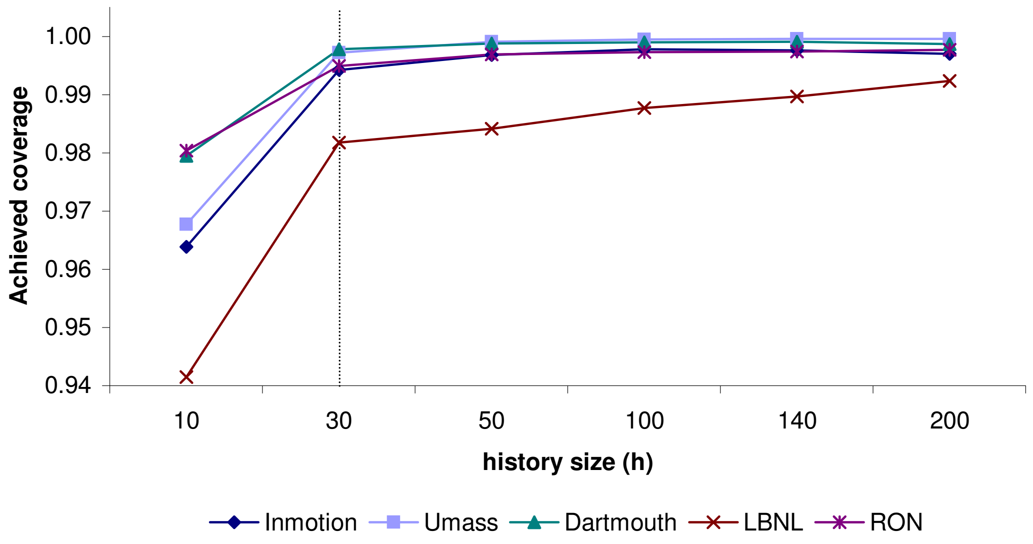


Figure 4.6: Achieved coverage.

for each 4 milliseconds, Dartmouth samples were collected with an interval of 118 milliseconds on average). This means that the stable phases in the Dartmouth environment were large enough to allow more sparse measurements (lower activity), while still achieving the same results in terms of stability detection that we observed in the Inmotion traces.

Figure 4.5 shows the improvement of bounds, which quantifies the reduction in the time bounds obtained when using *Adaptare*, in comparison with the baseline conservative solution proposed in Casimiro & Verissimo (2001).

The average coverage is presented in Figure 4.6. Each sample in the trace is matched against the bound assumed at that moment, and if the RTT is higher than the bound, a timing fault is accounted. The coverage of a given trace is defined by the proportion of non-faulty samples (those that do not lead to timing faults), over the total number of samples.

For large history sizes, the outcome of the lower rate of detected stability points is that the average improvement of bounds is modest (Figure 4.5) – *Adaptare* computes the conservative bound most of the time. On the other hand, if the history size is too small, detection mechanisms will tend to incorrectly identify stable periods that do not correspond to the real environment conditions. This erroneous behavior will lead *Adaptare* to possibly select lower bounds than it should (depending on the identified distribution and the actual state of the environment), which may compromise the dependability of our approach. This effect can be observed in our experiments with $h = 10$: in the majority of the traces, they have the highest rate of stability detection (Figure 4.4) and consequently the best improvement of bounds (Figure 4.5), but the coverage is not secured (Figure 4.6).

For history sizes $h \geq 30$, the required coverage was secured. Recall from the discussion in Chapter 3 that a timing fault will possibly occur every time the environment starts to change (alternation from stable to transient phase), affecting the achieved coverage. If the minimum coverage was secured, it means that the number of environment changes was sufficiently small, as required by the "sufficient stability" assumption, to allow this coverage.

For all performed experiments, the best results were obtained with a history size of $h = 30$, marked by the dotted vertical line in each figure: highest improvement of bounds, while securing the minimum required coverage.

These results provide sufficient evidence that there are real environments in which our assumptions hold. In these environments, *Adaptare can be successfully applied in order to make estimations through a probabilistic analysis of historical data and provide applications with better information related to the environment behavior, while satisfying the required dependability objective*.

4.4.3 Complexity analysis

One essential factor to be considered in our work is the complexity of the implemented algorithms. Although we are assuming that there are enough resources to perform the necessary computations, it is important to demonstrate that *Adaptare* has an acceptable complexity in order to reach its objectives. This section presents an informal analysis of the framework complexity and some measures of its execution time.

We show that the current implementation of *Adaptare* presents a complexity $O(m \times d \times h \log(h))$, where m is the number of phase detection mechanisms, d is the number of considered distributions, and h is the history size. We realize that because our implementation uses small values of m, d, and h, this asymptotic analysis does not provide enough information about execution time. Nevertheless, we believe

Algorithm 1: *Adaptare*'s algorithm.

```
Input: coverage, history size, measured samples.
Output: New computed bound.
1 for each phase detection mechanism $M$ do
2     Get distribution from $M$
3 Select one new bound according to the selection logic
4 return new bound
```

Algorithm 2: Phase detection mechanism's algorithm.

```
Input: Measured samples.
Output: Detected distribution.
1 detected distribution $\leftarrow$ none
2 for each distribution $D$ do
3     Estimate $D$ parameters
4     if samples fit in $D$ (with its parameters) then
5         Update detected distribution according to the mechanism's logic
6 if detected distribution = none then
7     return transient phase
8 else
9     return detected distribution
```

it is important to give an intuition of the impact of our algorithms and the influence of each variable in the framework execution. In any case, the asymptotic analysis is complemented with real measurements of the *Adaptare* framework's performance.

The general framework algorithm is described in Algorithm 1. Assuming that line 2 is a single operation, the complexity of the entire algorithm would be $O(m)$. However, the execution of a phase detection mechanism (line 2) involves more operations, as shown in Algorithm 2. For each distribution, the parameters are estimated (line 3) and the verification if the given samples fit in this distribution is performed (line 4).

Regarding parameters estimation, the exponential distribution has only one parameter (λ), which is estimated as the inverse of the sample mean. This estimation has a complexity that is linear with the history size ($O(h)$). The shifted exponential parameters (λ and γ) and the Weibull parameters (α and γ) are estimated using linear regression, which also has linear complexity with the history size $O(h)$. The k pa-

Algorithm 3: GoF test's algorithm.

```
  Input: Assumed distribution with estimated parameters, measured samples.
  Output: If the measures samples fit in the assumed distribution.
1 Sort measured samples
2 Calculate GoF statistic $S$ for assumed distribution $\hat{D}$
3 Get critical value $c_{h,\alpha}$
4 if $S \leq c_{h,\alpha}$ then
5     return true
6 else
7     return false
```

rameter of the Pareto distribution is estimated as the smallest sample value and the α parameter is computed using a MLE formula. Both estimations are performed with complexity $O(h)$. Finally, the uniform parameters a and b are estimated as the minimum and maximum sample values, leading to an $O(h)$ complexity.

For now, we will assume that the conditional test in line 4 corresponds to one operation. Thus, given that there are d distributions, and estimating the parameters of each tested distribution takes linear time $O(h)$, the complexity of this algorithm is $O(d \times h)$. In our implementation, the conditional test in line 4 is performed by GoF tests.

A general algorithm for a GoF test is presented in Algorithm 3. For both mechanisms, in order to calculate the GoF statistic (line 2), it is necessary to sort the input sample. Our implementation uses a modified mergesort with complexity $O(h \log(h))$ to sort the measured data (line 1). Moreover, the formulas presented in Section 4.2.1 to compute the GoF statistics are calculated in linear time ($O(h)$). Thus, performing a Gof test takes $h \log(h) + h$ operations, which is reduced to an asymptotic complexity of $O(h \log(h))$.

After analyzing the complexity of each part of the framework execution, we conclude that since each phase detection mechanism M is executed for each distribution D after estimating their parameters, the execution time of the current implementation of *Adaptare* is a function of $m \times d \times (h + h \log(h))$. Thus, the asymptotic complexity is $O(m \times d \times h \log(h))$, as previously mentioned. Considering a fixed number of mechanisms and distributions (currently, $m = 2$ and $d = 5$), the actual complexity of

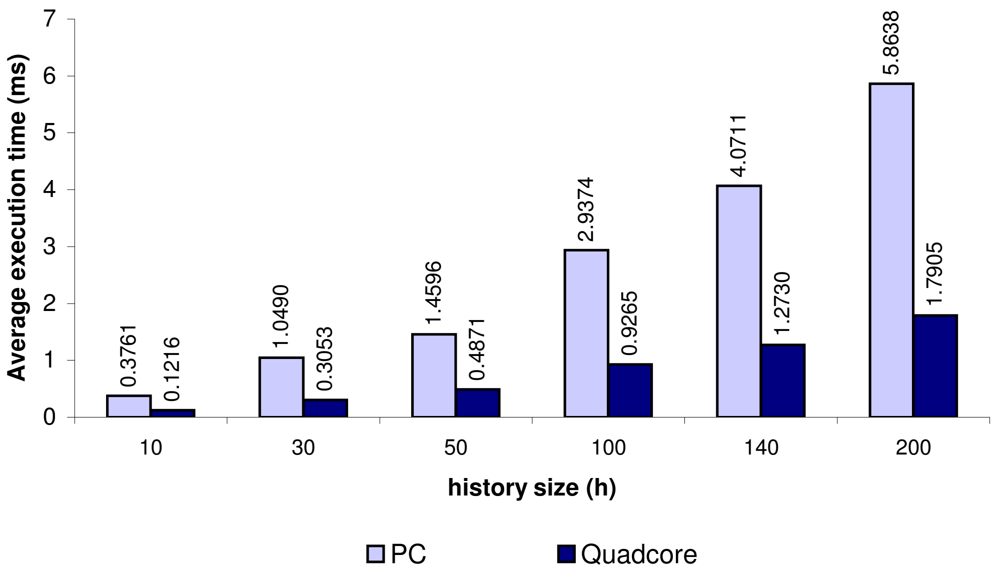


Figure 4.7: Measured execution time.

the framework execution is $O(h\log(h))$. It is important to note that this complexity is imposed by the sort operation required by both mechanisms.

To conclude our analysis, we measured the time that *Adaptare* takes to compute a new bound, for different history sizes. These measurements were executed in two different platforms: (i) a Dell Optiplex GX520 PC, with 2GB of RAM and one 2.8GHz Pentium 4 processor, and (ii) a 64-bit 2.3GHz quadcore Xeon machine. The initial point of the measured time interval is when the framework has the history samples available in a list, while the final point is defined to be immediately after a new bound is produced. We varied the history size from 10 to 200 and used an Inmotion trace composed by 8200 points. *Adaptare* was executed for each new sample point added to the history, computing approximately 8000 new bounds. Figure 4.7 presents the average execution times.

The analysis of the execution time can be helpful in order to determine how a certain application should use the framework. On average, *Adaptare* takes up to six milliseconds to analyze the samples and compute a new bound, depending on the history size and on the execution platform. We believe that these are reasonable values for many practical applications. Note that these values reflect the user level / application perception of the framework execution time, which includes all typical overheads (interference of other tasks, context switches, etc.), usual in multiprocess preemptive systems.

Nevertheless, when considering embedded applications or other systems with constrained resources, these values may be a considerable tradeoff for the improvements that may be achieved. In these cases, we propose two simple and combinable measures that a designer can employ in order to reduce the overhead of executing *Adaptare*. The first measure consists in executing the framework more sparingly. Ideally, a new bound would be computed whenever a new sample is added to the history. However, if new data is added to the history too frequently, it will be wise to execute *Adaptare* more parsimoniously, in fixed intervals, or when a reasonable amount of new samples is added to the history. The second measure consists in disabling the recognition of some probability distributions, which can be safely done without reducing the performance of *Adaptare* when it is known that these distributions are never detected. With this measure, the execution times in Figure 4.7, which correspond to a fully-fledged framework testing five different distributions, could be significantly reduced.

4.4.4 Comparing *Adaptare* to other adaptive solutions

Adaptare was designed to be easily integrated with client applications, driving their adaptation while guaranteeing the required dependability level. To achieve this, *Adaptare* executes complex and costly analysis mechanisms, as discussed in the previous section. Therefore, it is important to evaluate the relative benefits of *Adaptare* in comparison with simpler and cheaper mechanisms.

We selected two simpler approaches to compare with: the RTT estimation algorithm performed by TCP, which has constant execution time ($O(1)$), and a method based on the WinMean estimator plus a safety margin, of linear complexity $O(h)$. Recall that *Adaptare*'s execution has a complexity of $O(hlog(h))$, dictated by the sorting algorithm executed within the phase detection mechanisms.

The TCP protocol formally defines a simple method for setting the retransmission timer of each non-acknowledged packet, based on the observed RTTs and their variations. The algorithm, presented in Algorithm 4, was proposed in Jacobson (1988) and is formalized in RFC 2988. In this section we refer to it as the *TCP-RTT* approach.

Algorithm 4: TCP's algorithm to compute the retransmission timer.

```
Output: New retransmission timer RTO.
1  $k \leftarrow 4$, $\alpha \leftarrow 1/8$, $\beta \leftarrow 1/4$
2  Wait for ACK of sent message m
3  if ACK is not received within RTO then
4      Retransmit message m
5      $RTO \leftarrow 2 \cdot RTO$
6  else
7      $R \leftarrow RTT$ of sent message
8      if R is the first measured RTT then
9          $SRTT \leftarrow R$
10         $RTTVAR \leftarrow R/2$
11         $RTO \leftarrow SRTT + k \cdot RTTVAR$
12     else
13         $RTTVAR \leftarrow RTTVAR(1-\beta) + \beta|SRTT - R|$
14         $SRTT \leftarrow SRTT(1-\alpha) + \alpha \cdot R$
15         $RTO \leftarrow SRTT + k \cdot RTTVAR$
16 return $RTO$
```

Algorithm 5: Mean-Jac estimation algorithm.

```
Input: Measured samples, current safety margin, current estimator.
Output: New estimator.
1  $\alpha \leftarrow 1/4$
2  $mean \leftarrow$ samples average
3  $margin \leftarrow margin + \alpha(|lastSample - estimator| - margin)$
4  $estimator \leftarrow mean + margin$
5  return $estimator$
```

The algorithm based on the WinMean estimator plus a safety margin is presented in Algorithm 5. The WinMean estimator simply computes the average of history samples. The safety margin is also based on the Jacobson algorithm (Jacobson, 1988). This approach is applied to adjust failure detectors timeouts in Nunes & Jansch-Porto (2004) and Falai & Bondavalli (2005), and it will be referred as the *Mean-Jac* approach.

For the comparative experiments we configured *Adaptare* with $C = 98\%$ and $h = 30$, considering the positive results previously achieved with this history size. Then, we applied the real traces from the five different sources as input for the different

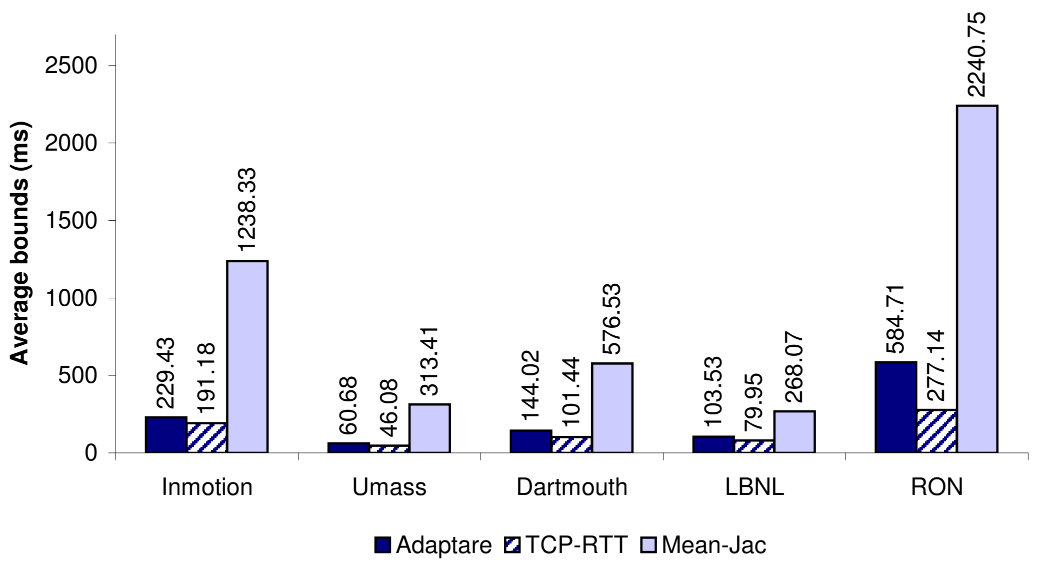


Figure 4.8: Comparing *Adaptare* with different solutions - computed bounds.

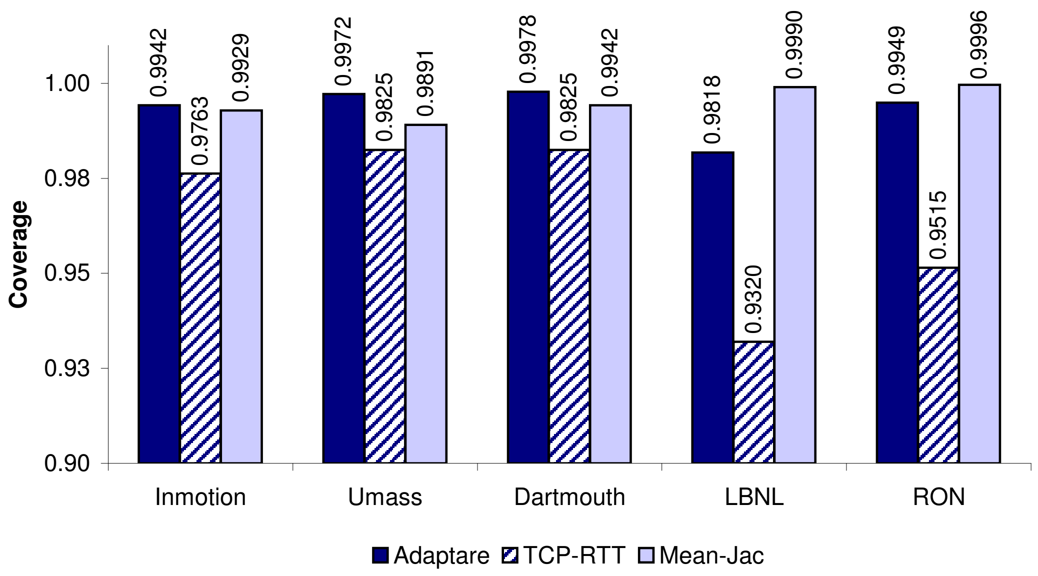


Figure 4.9: Comparing *Adaptare* with different solutions - achieved coverage.

approaches, collecting the bounds produced by all of them, based on the same input. The results are summarized in Figures 4.8 and 4.9.

When comparing *Adaptare* with the *Mean-Jac* method, we verify that *Adaptare* provided significantly lower bounds (*Mean-Jac* bounds are on average 2.5 to 5 times higher) without any negative impact on the achieved coverage. In fact, in most cases the coverage provided by *Adaptare* was even better than the coverage provided by the *Mean-Jac* approach. In any case, and for both approaches, the required minimum coverage was always secured.

The results achieved when applying the *TCP-RTT* approach were noticeably different from those achieved with the *Mean-Jac* approach. The *TCP-RTT* algorithm provided smaller bounds (reductions roughly ranging between 15% and 50% when comparing with *Adaptare*), clearly indicating that the *TCP-RTT* method is more "agressive" than *Adaptare*. The reverse side of the coin is that in terms of coverage, the results clearly indicate that *TCP-RTT* approach produces more timing faults than those produced with *Adaptare*. In fact, the minimum coverage was not achieved by the *TCP-RTT* algorithm in three of the five tested environments. *Adaptare* satisfies coverage requirements (achieving comparatively higher coverages) at the cost of slightly higher timeouts. Furthermore, the results presented in Figure 4.10 show that when the minimum coverage requested to *Adaptare* is reduced to be leveled with the average coverage provided by the *TCP-RTT* approach, the average timeouts of both approaches will also become nearly the same. This result indicates that despite being very simple, the *TCP-RTT* estimation method is very efficient. However, it is important to stress that, unlike with *Adaptare*, in the *TCP-RTT* approach it is not possible to request a desired coverage, which is a clear disadvantage under a dependability adaptation perspective.

In order to correctly interpret these results, it is necessary to retain that if the objective was to merely compute small bounds, then an aggressive method would achieve that easily. However, if bounds are too small, this will typically have negative effects. For instance, in the case of the TCP protocol, every timing fault resulting from a badly dimensioned timeout implies one unnecessary retransmission, and a consequent waste of network resources. Furthermore, successive timing faults may imply instability of higher level applications or protocols.

Therefore, for comparison purposes, it is necessary to consider both the achieved bounds and the achieved coverage, altogether. In general, *Adaptare* was able to compute relative small bounds *and* secure the expected dependability level. These results indicate that *Adaptare* is indeed able to properly react to environment changes and compute more effective and dependable bounds than with the other two approaches:

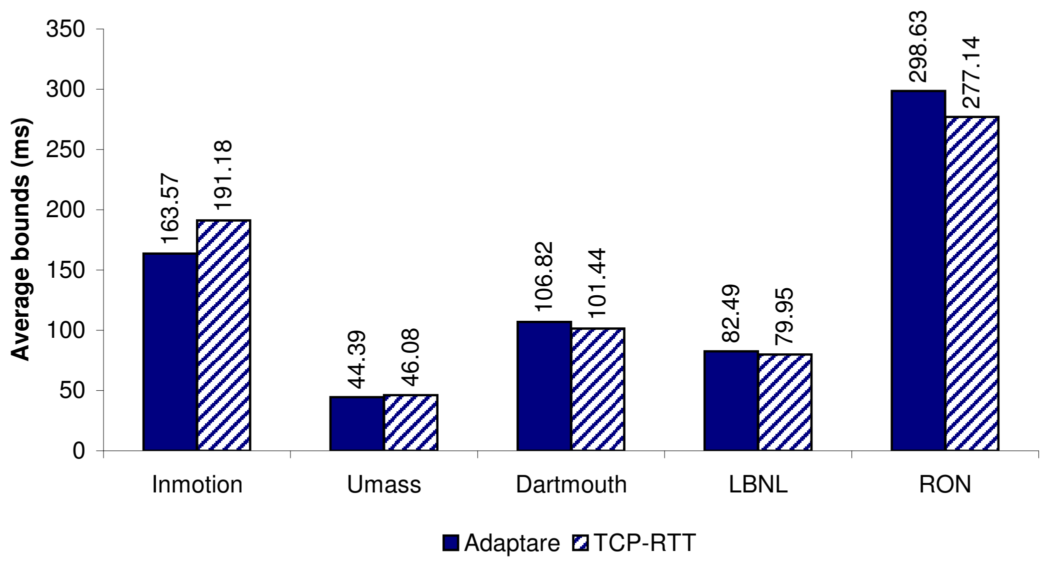


Figure 4.10: Comparing *Adaptare* and *TCP-RTT* bounds, for the same coverage.

producing small bounds, while still keeping the number of timing faults within the accepted limits.

The price to pay for such optimized solution is that it requires more resources, in terms of storage space and particularly memory operations. The *TCP-RTT* approach is based only on the last measured RTT and estimation error, so it does not need any extra storage. The bound computation is straightforward, requiring a total of 15 arithmetic operations. In the *Mean-Jac* approach, the average computation performed by the Win-Mean estimator requires h arithmetic operations, while the Jacobson safety margin is calculated in constant time, using 5 arithmetic operations. Thus, this approach executes $h + 5$ operations and requires storage space for keeping the history samples. In contrast, *Adaptare* is more complex, due to the number of operations executed by the phase detection mechanisms to perform the network characterization. Regarding execution overhead, and as presented in Section 4.4.3, *Adaptare* imposes overheads in the millisecond order, while the execution overhead of simple approaches like *TCP-RTT* or *Mean-Jac* is comparably negligible.

Overall, deciding which approach is the better will necessarily depend on the application objectives and requirements and on the amount of available resources. Nothing comes for free, but we have shown that among the three adaptive methods evaluated in this section, *Adaptare* is the best solution for adaptive systems, in particular for those with dependability concerns.

4.5 Summary

In this chapter we introduced *Adaptare* - a framework for supporting automatic and dependable adaptation in stochastic environments. We evaluated *Adaptare* using synthetic data traces generated from specific distributions, and real data traces available in the Internet, which were collected by different applications, operating in different environments. Our results attest the correctness of the implemented mechanisms, and the benefits of using *Adaptare*, in comparison with other approaches for adaptation.

The fundamental conclusion to derive from the experiments reported in this chapter is that it is possible to define effective mechanisms to detect stable and transient periods and, for the stable ones, correctly identify the observed probability distribution. Because of that, *Adaptare* constitutes a valuable approach to achieve dependable adaptation and, at the same time, obtain improved time bounds.

The next chapters assess *Adaptare*'s applicability, by presenting adaptive solutions for two very relevant problems in distributed systems: consensus and failure detection. Those solutions were designed based on modular architectures in which *Adaptare* is used as a service for timeout provisioning, driving the adaptation process in a dependable way.

Notes

A preliminary version of *Adaptare* was used to support adaptation in the context of transactional systems. The framework was used as a middleware to support transactions with relaxed temporal requirements. This work appeared in "Using Experimental Measurements to Assess Dependable Adaptation Support Mechanisms for Timed Transactions", Dixit, Casimiro, Laranjeiro, and Vieira, "Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems, with Proceedings of the 27th IEEE Symposium on Reliable Distributed Systems", Napoli, Italy, October 2008. As we decided to focus on monitoring and characterization of network delays, those results are not reported in this thesis. Interested readers may refer to Dixit *et al.* (2008).

Most of the content of this chapter has been reported as a FCUL technical report in "A Probabilistic Framework for Automatic and Dependable Adaptation in Dynamic Environments", Dixit, Casimiro, Verissimo, Lollini, and Bondavalli, DI/FCUL TR-2009-19, January 2010 (Dixit *et al.*, 2009). After some revisions and improvements, the work "Adaptare: Supporting automatic and dependable adaptation in dynamic environments" was accepted for publication in the "ACM Transactions on Autonomous and Adaptive Systems", in March 2011 (Dixit *et al.*, 2011).

Chapter 5

Timeout-based Adaptive Consensus

Algorithms for solving distributed systems problems often use timeouts as a means to achieve progress. They are designed in a way that safety is always preserved despite timeouts being too small or too large. A conservatively large static timeout value is usually selected, such that the overall system performance is acceptable in the normal case. This approach is good enough for applications that execute in stable environments, but it may compromise the system's performance in more dynamic settings, such as wireless networks. In these cases, it is expected that adaptive solutions that automatically adjust timeouts according to the observed network conditions will perform better.

In this chapter we describe a pragmatic approach to transform a static timeout-based consensus protocol into an adaptive protocol for best performance. The presented solution performs well despite changes in network conditions and without sacrificing safety, thus clearly illustrating the importance and usefulness of adaptation and autonomic behavior.

In the proposed approach, *Adaptare* is used to analyze message delays among consensus processes, and to provide timeout values derived from the performed analysis. As *Adaptare* was designed to produce dependable values, a distinguishing feature of this approach is that we have confidence on the assumed timeouts, which is in contrast with ad hoc approaches for timeout selection that are found in the literature (see Chapter 2). Using *Adaptare* as an independent service also makes it easier to apply the proposed approach to transform other static protocols into dynamic ones. In this sense, our methodology is generally applicable, and this chapter contributes with the basic principles and guidelines that must be followed for that purpose.

The chapter is organized as follows. The motivation for using adaptive timeouts in distributed protocols executing in dynamic environments is presented in Section 5.1. In Section 5.2 we describe the static timeout-based consensus protocol considered in this chapter. In Section 5.3 we experimentally justify the need for adaptive timeouts by demonstrating how the execution time of each round of this algorithm degrades as the number of consensus processes (and consequently the network load) increases. Section 5.4 describes our methodology to transform this static timeout-based consensus protocol into an adaptive solution. Implementation details are given in Section 5.5. Finally, Section 5.6 presents an experimental evaluation in which we compare the performance of the static and adaptive versions of this consensus protocol in a wireless ad hoc network.

5.1 Motivation

The consensus problem is a fundamental building block in the design of distributed systems, as it contributes to the coordination of actions in order to achieve consistent decisions. As discussed in Chapter 2, the FLP impossibility result (Fischer *et al.*, 1985) states that there is no deterministic solution for the consensus problem in asynchronous systems prone to process crash failures. The typical approach to circumvent this impossibility is to strengthen the timing assumptions of the system, either implicitly through a failure detector, or explicitly through partial synchrony models, so that processes can resort to some timing information for making progress. This usually involves setting a timeout and, if the timeout expires, take appropriate measures (e.g., marking some process as faulty), instead of waiting indefinitely for messages that may never arrive.

Timing information, however, is inherently unreliable in asynchronous systems. Therefore, the design of distributed consensus protocols is usually centered at preserving safety regardless of the underlying timing behavior, while liveness is achieved on an eventual basis, when the system exhibits some minimum level of synchrony, as it is done in Aguilera *et al.* (2001) and Lamport (1998), for example. This design principle, as sound as it is, relegates the problem of selecting an appropriate timeout to a secondary plane, because its value has no effect on the safety of the protocol (and liveness just requires for this value to eventually become sufficiently large). Thus, this problem is often dismissed as being ‘merely’ an engineering decision, where a fixed timeout is conservatively selected based on ad hoc approaches or on empirical observations of the network. However, while in any soundly designed protocol correctness is not dependent on specific timeout values, performance is. A too small timeout will raise many false positives (if failure detection is involved) or cause too much contention (if retransmission is used). A too large timeout will hinder a quick recovery from failures.

Therefore, to reason about performance, a crucial issue is the characterization of the temporal behavior of the network on which the consensus protocol is executed. Local Area Networks (LANs) constitute a very favorable environment from a temporal perspective. Network delays are small (in the order of microseconds), very stable across different LANs, independent of the communicating nodes, and they are not much affected by contention. On the other hand, in large-scale networks (WANs) delays are much higher and they strongly depend on the locations of specific end points. Moreover, delays are not so stable over time, in particular because routes may change dynamically and global load fluctuations also have some impact on observed delays. The problem becomes even more relevant when considering the operation in wireless environments. Network delays are also much larger than in LANs, but in addition they are strongly exposed to the effects of contention (Acharya *et al.*, 2008; Jardosh *et al.*, 2005). Varying the number of nodes that actively execute a distributed protocol and communicate within a single hop distance has a clear impact on observable delays. Consequently, a correct timeout selection, namely involving dynamic adaptation, becomes more relevant in these environments.

In this chapter we illustrate the kind of improvements that may be achieved by using adaptive timeouts. We present our pragmatic approach to transform a static timeout-based consensus protocol for ad hoc wireless networks into an adaptive solution, using the services provided by the *Adaptare* framework.

5.2 Consensus protocol

The consensus protocol introduced in Moniz *et al.* (2009) solves the k-consensus problem in wireless ad hoc networks. In the k-consensus problem, each process p_i proposes an initial binary value $v_i \in \{0, 1\}$, and at least $k > \frac{n}{2}$ of them have to agree on a common proposed value, where n is the number of participant processes. The remaining processes are not required to decide, but if they decide, it must be on the same value decided by the k processes.

The work adopts the communication failure model (Santoro & Widmayer, 1989, 2007). This model does not assume end-to-end reliable delivery mechanisms, and any failure (either in a process or in a communication link) is manifested as a transmission fault. For example, a process crash will be perceived as a series of omission faults with the crashed process as sender. This model is more appropriate to represent the dynamics of wireless ad hoc networks. However, an impossibility result presented in Santoro & Widmayer (1989) states that, under the communication failure model, there is no finite time deterministic algorithm that allows n processes to reach k-agreement, if more than $n - 2$ transmission failures occur in a communication step. The protocol introduced in Moniz *et al.* (2009) circumvents this impossibility result by employing randomization to tolerate omission transmission faults. In its randomized version, the k-consensus problem is formally defined by the following properties:

- **Validity.** If all processes propose the same value v, then any process that decides, decides v.
- **Agreement.** No two correct processes decide differently.
- **Termination.** At least k processes eventually decide with probability 1.

The protocol, presented in Algorithm 6, executes in rounds. In a round, each process p_i broadcasts a message containing its identifier i, proposal value v_i, phase ϕ_i, and other variables comprising its internal state (line 7). Then it waits for messages broadcast by the other processes (lines 9-11). When messages for phase ϕ_i are received from the majority of the processes, process p_i will make progress by analyzing them, updating its state (proposal, phase, and/or status) and possibly deciding on some value or initiating a new round (lines 12-31). However, as assumed in the communication

Algorithm 6: Static timeout-based k-consensus algorithm.

```
Input: Initial binary proposal value $proposal_i \in \{0,1\}$
Output: Binary decision value $decision_i \in \{0,1\}$
$\phi_i \leftarrow 1$;
$v_i \leftarrow proposal_i$;
$status_i \leftarrow undecided$;
$V_i \leftarrow \emptyset$;
$timeout \leftarrow 10ms$;
while true do
    broadcast($\langle i, \phi_i, v_i, status_i \rangle$);
    $deadline \leftarrow$ curTime() $+timeout$;
    while ($|\{\langle *, \phi, *, * \rangle \in V_i : \phi = \phi_i\}| \leq \frac{n}{2}$ and curTime() $< deadline$) do
        $M_i \leftarrow$ receive();
        $V_i \leftarrow V_i \cup M_i$;
    while $\exists_{\langle *, \phi, v, status \rangle \in V_i} : \phi > \phi_i$ do
        $\phi_i \leftarrow \phi$;
        $v_i \leftarrow v$;
        $status_i \leftarrow status$;
    if $|\{\langle *, \phi, *, * \rangle \in V_i : \phi = \phi_i\}| > \frac{n}{2}$ then
        if $\phi_i \bmod 2 = 1$ then                                /* odd phase */
            if $\exists_{v \in \{0,1\}} : |\{\langle *, \phi, v, * \rangle \in V_i : \phi = \phi_i\}| > \frac{n}{2}$ then
                $v_i \leftarrow v$;
            else
                $v_i \leftarrow \perp$;
        else                                                        /* even phase */
            if $\exists_{v \in \{0,1\}} : |\{\langle *, \phi, v, * \rangle \in V_i : \phi = \phi_i\}| > \frac{n}{2}$ then
                $status_i \leftarrow decided$;
            if $\exists_{v \in \{0,1\}} : |\{\langle *, \phi, v, * \rangle \in V_i : \phi = \phi_i\}| \geq 1$ then
                $v_i \leftarrow v$;
            else
                $v_i \leftarrow$ coin$_i$();
        $\phi_i \leftarrow \phi_i + 1$;
    if $status_i = decided$ then
        $decision_i \leftarrow v_i$;
```

failure model, some messages that a process is supposed to receive may be lost. This may delay p_i (since it will need to wait for slower messages) or may even prevent progress to be done (if too many messages are lost). Therefore, in each round, process p_i waits for messages only during a pre-defined amount of time. If not enough messages are received during this time interval, its state does not change and p_i restarts the round by retransmitting the original message. For implementing the protocol, a timeout mechanism should be used to bound round durations and trigger retransmissions. This protocol ensures safety regardless of the number of omission faults in each round, while liveness is guaranteed in rounds where the number of omission faults is $f \leq \lceil \frac{n}{2} \rceil (n-k) + k - 2$.

Detailed explanations about the algorithm's execution, as well as correctness proofs, are beyond the scope of this thesis. Interested readers may refer to Moniz *et al.* (2009).

5.3 Impact of network conditions

As mentioned in Section 5.1, caring about the selection of appropriate timeouts is often dismissed as being 'merely' an engineering decision. In fact, the concern is usually with safety, not much with performance. However, distributed protocols are only useful and significant if they perform reasonable well – safety is fundamental, but may not be enough. In this section we show that the performance of a distributed protocol can be seriously affected by varying network conditions, in particular if wireless environments are considered. This motivates the need for practical solutions to deal with such variations, based on adapting temporal variables like timeouts, aiming at achieving the best possible performance under the available conditions.

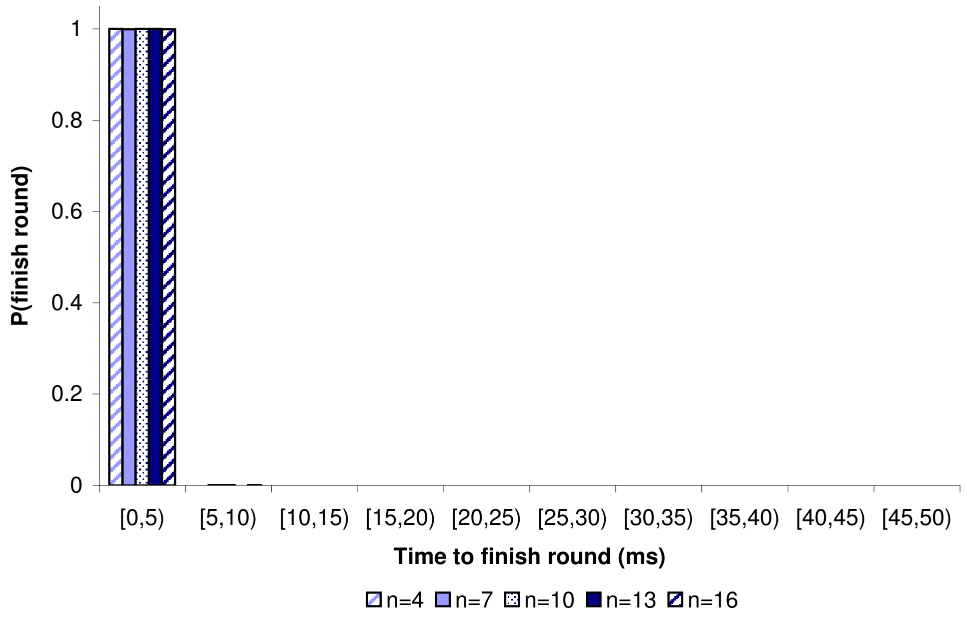
We performed practical experiments that compare the execution time of a round of the considered consensus protocol in a local area network (LAN) and in a wireless ad hoc network. The LAN experiments were performed using a cluster in our laboratory, while the Emulab testbed (White *et al.*, 2002) was used for wireless experiments. We measured the duration of each consensus round for varying load conditions. In fact, the load was simply controlled by increasing the number of processes participating in the consensus execution. The number of processes was set to 4, 7, 10, 13, and 16.

As we wanted to analyze the impact of increased load in the round execution time, the static timeout was set to a very large value (1 second). This provides enough time for all necessary messages to be received and thus gives a precise view of the round duration, even if it takes a long time. Exceptional cases of rounds in which half or more messages were lost, blocking the protocol and causing timeout expiration and retransmission, were not considered in the presented results, since they are not relevant for the purpose of reasoning about the duration of rounds.

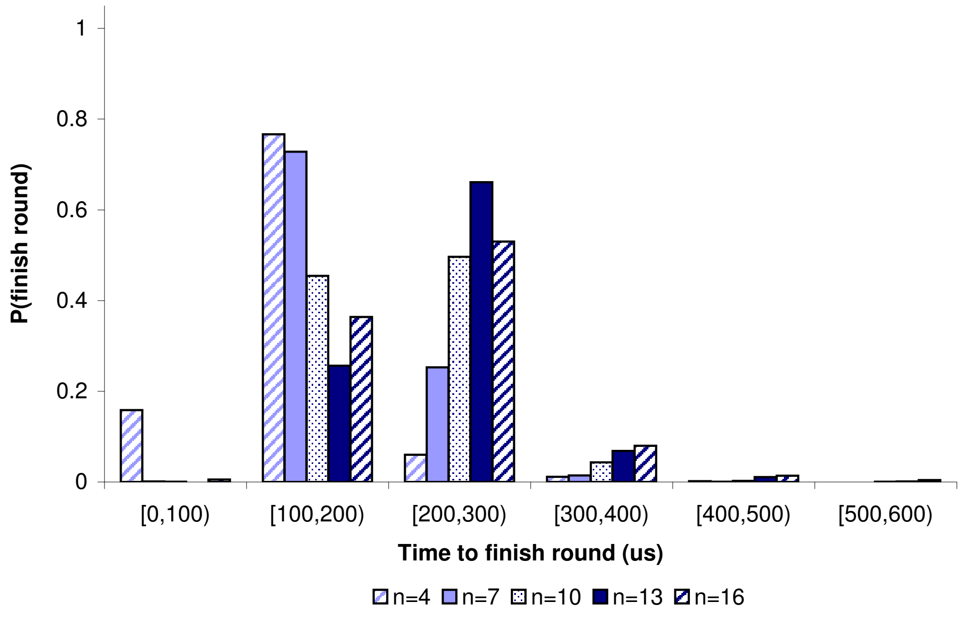
The histograms presented in Figures 5.1 and 5.2 show the distribution of round execution times in the two networking environments for the different load conditions corresponding to varying number of processes. The LAN environment is stable in the sense that a small variation in the number of processes does not affect the execution time of a round significantly (Figure 5.1(a)). Rounds are almost always completed in less than 5 milliseconds, allowing the original version of the consensus algorithm, which uses a fixed timeout of 10 milliseconds, to perform well in this network. In fact, we observed that a timeout of 600 microseconds would be sufficiently large to receive all the required messages to complete a round in these very stable conditions, as presented in Figure 5.1(b).

On the other hand, the execution of consensus rounds in the wireless network (Figure 5.2) is clearly slower, with latencies one order of magnitude higher than in LANs. The most important outcome is that the dependency on the number of participating processes becomes more relevant. While a timeout of 10 milliseconds appears to be sufficient for consensus among 4, 7 or 10 processes, for 13 or 16 processes the timeout should be of at least 15 milliseconds to prevent unnecessary retransmissions. It is easy to conclude that in the wireless setting the selection of the appropriate timeout is a crucial issue, which would not be the case in LAN environments. We must say that if a much larger number of processes were used, the contention in the LAN environment would eventually increase, leading to more visible uncertainty and larger delays. However, this would be an extreme case, whereas our aim was to observe the behavior of the network environments in typical scenarios.

It is important to note that in these experiments the consensus protocol was implemented over UDP, which means that the transport layer does not employ any kind of retransmission mechanism or congestion control. This is in accordance with the communication failure model assumed in Moniz *et al.* (2009), and implies that retransmis-



(a) Time in milliseconds.



(b) Time in microseconds.

Figure 5.1: Round execution time in a wired LAN.

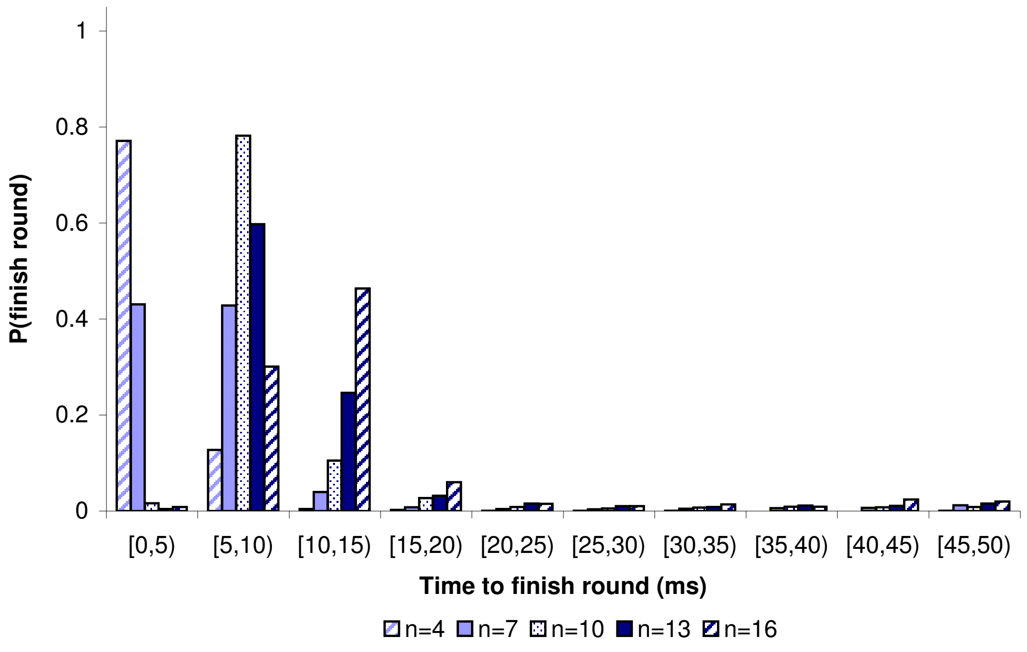


Figure 5.2: Round execution time in a wireless ad hoc network.

sions (if necessary) must be handled by the protocol. It is thus the responsibility of the protocol to select the appropriate timeout values. These must be small enough to quickly initiate a new round instead of waiting for slow (e.g., due to backoff or waiting delays of the 802.11 MAC layer) or lost messages, but large enough to avoid unnecessary retransmissions and consequent congestion, which would ultimately slow down the protocol. Moreover, these values should be adapted according to the observed conditions, that is, the protocol should be adaptive.

5.4 Achieving adaptive consensus

As usual in consensus protocols, the algorithm described in Section 5.2 relies on a timeout to decide whether messages should be retransmitted and when that should be done. In this section, we describe our approach to transform it into an adaptive consensus protocol. Our goal is to improve its performance by dynamically adjusting timeouts in response to variations on the network delays.

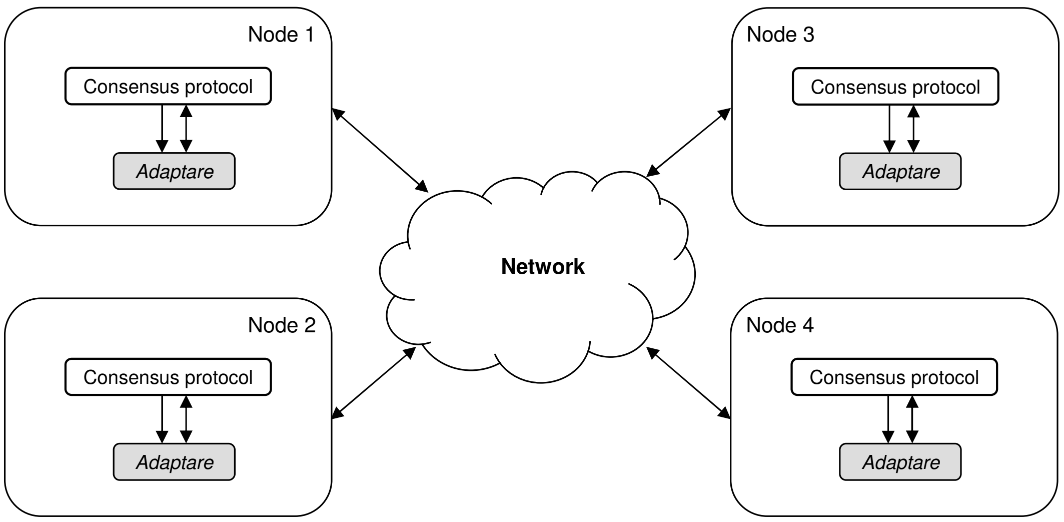


Figure 5.3: Architecture of the adaptive consensus (for $n = 4$).

Figure 5.3 shows the architecture of our solution. Each consensus process has its own instance of the timeout provisioning service (*Adaptare*), so that timeout selection decisions are fully decentralized. Furthermore, the logical separation between the timeout-based consensus protocol and timeout calculation functions makes it easier to apply the approach without the need of significant changes on protocol code, as detailed ahead. We believe that this also makes the approach more generic and suitable to be applied to build adaptive versions of other timeout-based protocols.

5.4.1 Protocol instrumentation

Typically, timeout-based protocols rely on a fixed timeout value that defines an upper bound on the time that a process should wait for some event. In order to make timeouts adaptive using *Adaptare*, it is necessary to:

- Identify the system variable that should be bounded by the timeout;
- Instrument the protocol in order to collect samples of this variable that will be fed into *Adaptare*;
- Search for places in the protocol where the timeout is used, adding a call to *Adaptare* in order to update the timeout.

In the consensus protocol that we are considering, the timeout defines how long a process executing phase ϕ_i will wait for the reception of messages of this phase from the majority of the processes, during a round. Therefore, the variable to be monitored is the time elapsed between the beginning of a phase and the reception of each message for that phase. Even those messages that arrive in later rounds should be considered, otherwise we would have information only about timely messages, compromising *Adaptare*'s analysis. It is however necessary to ensure that the latency measurement corresponding to each received message is done with respect to the beginning of the adequate phase (which can be an older one).

Given that, we instrumented the consensus protocol as shown in Algorithm 7. We keep track of the time at which each phase is initiated (lines 5, 17 and 34). When a new message is received in any round, the algorithm determines the phase to which the message belongs, calculates the amount of time elapsed since the beginning of that phase (line 13), and sends this value to *Adaptare* (line 14). Besides that, we only had to modify the consensus code so that, before a message is broadcast, a request is made to *Adaptare* to obtain the timeout value that should be used in that round (line 7). The functions `getTimeout()` (line 7) and `addSample()` (line 14) are implemented as UDP messages sent to an adaptation layer, as explained in Section 5.5.

5.4.2 Configuring *Adaptare*

One fundamental issue in the proposed approach is the correct configuration of *Adaptare*, so that dynamically obtained timeout values are appropriate to achieve the desired performance. In essence, when developing the adaptive solution it is necessary to understand which are the performance objectives of the application or protocol, and translate them into a coverage requirement, which must be provided to *Adaptare*.

Defining the optimal coverage value depends on the specific case in consideration. For some protocols it might be desirable to select timeout values that will hold most of the time, with very high probability, while in other cases it might be better to be more aggressive, using timeouts that may hold only with a smaller probability, in favor of increased reactivity. Here we explain the reasoning that we followed to determine the coverage that should be used in the case of the considered adaptive consensus protocol.

Algorithm 7: Adaptive timeout-based k-consensus algorithm.

```
Input: Initial binary proposal value $proposal_i \in \{0, 1\}$
Output: Binary decision value $decision_i \in \{0, 1\}$
$\phi_i \leftarrow 1$;
$v_i \leftarrow proposal_i$;
$status_i \leftarrow undecided$;
$V_i \leftarrow \emptyset$;
$start[\phi_i] \leftarrow$ curTime();
while true do
    $timeout \leftarrow$ getTimeout();
    broadcast($\langle i, \phi_i, v_i, status_i \rangle$);
    $deadline \leftarrow$ curTime() $+timeout$;
    while ($|\{\langle *, \phi, *, * \rangle \in V_i : \phi = \phi_i\}| \leq \frac{n}{2}$ and curTime() $< deadline$) do
        $M_i \leftarrow$ receive();
        $V_i \leftarrow V_i \cup M_i$;
        $delay \leftarrow$ curTime() $-start[phaseOf(M_i)]$;
        addSample($delay$);
    while $\exists_{\langle *, \phi, v, status \rangle \in V_i} : \phi > \phi_i$ do
        $\phi_i \leftarrow \phi$;
        $start[\phi_i] \leftarrow$ curTime();
        $v_i \leftarrow v$;
        $status_i \leftarrow status$;
    if $|\{\langle *, \phi, *, * \rangle \in V_i : \phi = \phi_i\}| > \frac{n}{2}$ then
        if $\phi_i$ mod $2 = 1$ then                                  /* odd phase */
            if $\exists_{v \in \{0,1\}} : |\{\langle *, \phi, v, * \rangle \in V_i : \phi = \phi_i\}| > \frac{n}{2}$ then
                $v_i \leftarrow v$;
            else
                $v_i \leftarrow \perp$;
        else                                                           /* even phase */
            if $\exists_{v \in \{0,1\}} : |\{\langle *, \phi, v, * \rangle \in V_i : \phi = \phi_i\}| > \frac{n}{2}$ then
                $status_i \leftarrow decided$;
            if $\exists_{v \in \{0,1\}} : |\{\langle *, \phi, v, * \rangle \in V_i : \phi = \phi_i\}| \geq 1$ then
                $v_i \leftarrow v$;
            else
                $v_i \leftarrow$ coin$_i$();
        $\phi_i \leftarrow \phi_i + 1$;
        $start[\phi_i] \leftarrow$ curTime();
    if $status_i = decided$ then
        $decision_i \leftarrow v_i$;
```

During the consensus execution, all processes will be sending and receiving messages, and will be able to make progress as long as they receive enough messages during a defined time interval after they initiate a round (by sending a message). They do not need to receive all the possible messages, just a fraction of them. In fact, they only need to receive messages from the majority of the processes, that is, more than $\frac{n}{2}$. Given that the total number of messages they could receive in a round is n (one from each process, including their own), the fraction of required timely messages is thus $\frac{\lfloor \frac{n}{2} \rfloor + 1}{n}$. In other words, we can say that the selected timeout must be such that it allows messages to be timely with a minimum probability (or coverage) given by:

$$C = \frac{\lfloor \frac{n}{2} \rfloor + 1}{n}$$

This defines the coverage value that must be used in the configuration of *Adaptare*, which will yield a timeout value that will be sufficiently large to allow progress to be made for the observed environment conditions.

Note that the timeout might not be always sufficiently large, leading to unnecessary retransmissions. This is natural when considering stochastic processes, with continuously changing environments. For instance, if at a certain moment there is an increase in the observed delays, this may lead, in a first moment, to retransmissions caused by premature timeouts. As a reaction, these increased delays will be fed into *Adaptare*, which will also provide increased timeout values to the protocol. This automatic adjustment will bring the protocol back to the expected good behavior, in which it will wait just the necessary amount of time for making progress.

One may be tempted to think that a good solution would be to simply use very large timeout values, to always allow enough messages to be received, avoiding retransmissions. However, there are two problems in doing that. First, given that messages might be lost and never arrive, this would affect liveness if half or more of the expected messages were lost in a round. Second, even if enough messages are received, waiting for them could delay consensus much more than if a new round was started. In fact, in Figure 5.2 we observe that some rounds may take a long time, in the order of tens of milliseconds, to terminate. Using well balanced timeout values, such long lasting rounds can be avoided, and the overall consensus execution time will be improved.

5.5 Implementation details

An *adaptation layer*, which is the interface between the adaptive protocol and *Adaptare*, was implemented as a lightweight Java class. This class contains the following two methods:

- The `addSample(long)` method is used to feed *Adaptare* with measured delays. The method simply updates *Adaptare*'s local history by adding the new sample and removing the oldest one, keeping a constant history size.

- The `long getTimeout()` method is used to get an updated timeout value. When called, the method requests a new bound to *Adaptare*. The desired coverage, $C = \frac{\lfloor \frac{n}{2} \rfloor + 1}{n}$, is hard coded in the implementation of this method and is just dependent on the total number of processes, a parameter that is known by the consensus processes, and which is given on startup. Note that it would be possible to allow the application to define the required coverage in each call (in runtime), which could be interesting for applications with varying performance objectives, or with several functional levels.

In our implementation, the consensus processes communicate with the adaptation layer through UDP messages, but other approaches could be used as well. There are three different types of messages that are exchanged: one feed message (`MSG_NEW_SAMPLE`), one request message (`MSG_TIMEOUT_REQUEST`), and one response message (`MSG_TIMEOUT_REPLY`).

Feeding *Adaptare*. As described in Section 5.4.1, when a consensus process receives a broadcast message, it identifies the phase to which the message belongs, and computes the elapsed time from the beginning of that phase to the reception of the message. This value is sent to the adaptation layer in a `MSG_NEW_SAMPLE` message.

Upon the reception of a `MSG_NEW_SAMPLE` message, the adaptation layer updates *Adaptare*'s history of measurements using the `addSample(long)` method.

Getting a new timeout. Before sending a broadcast message, each consensus process requests a new timeout value by sending a `MSG_TIMEOUT_REQUEST` message to the adaptation layer. This layer triggers *Adaptare*'s timeout computation by executing the `long getTimeout()` method, and replies to the consensus process with a `MSG_TIMEOUT_REPLY` message containing the new computed timeout.

5.6 Performance evaluation

We executed a set of experiments in order to quantify the improvements that are achieved by our adaptive consensus protocol. We compared the performance of the static version presented in Moniz *et al.* (2009) and of the adaptive version. In this section we present the achieved results in terms of average latency, which is the consensus execution time (i.e., the amount of time that a consensus process takes to decide), number of broadcast messages sent by each process per consensus execution, and average timeout.

The experiments were carried out on the Emulab testbed (White *et al.*, 2002). A total of 16 nodes were used, each one with the following hardware characteristics: Pentium III processor, 600 MHz of clock speed, 256 MB of RAM, and 802.11 a/b/g D-Link DWL-AG530 WLAN interface card. The operating system was the Fedora Core 4 Linux with kernel version 2.6.18.6. The nodes were located on the same physical cluster and were, at most, a few meters distant from each other. Since the Emulab environment is not isolated, and our experiments could suffer from the interference of other nodes outside our control, we executed the experiments in six different days, to mitigate possible occasional interferences on the global results.

The number of processes participating in the consensus execution was set to 4, 7, 10, 13, and 16. Processes with odd identifiers initially propose the value 1, while processes with even identifiers propose 0, guaranteeing a divergent initial proposal set.

An experiment comprises 20 consecutive executions of the static and the adaptive algorithms for a given n. We executed five experiments per day (one of each value of n, resulting in 100 executions per day of each version of the protocol), during the six different days. This gives a total of 1200 consensus executions (600 executions of the static algorithm, plus 600 executions of the adaptive algorithm).

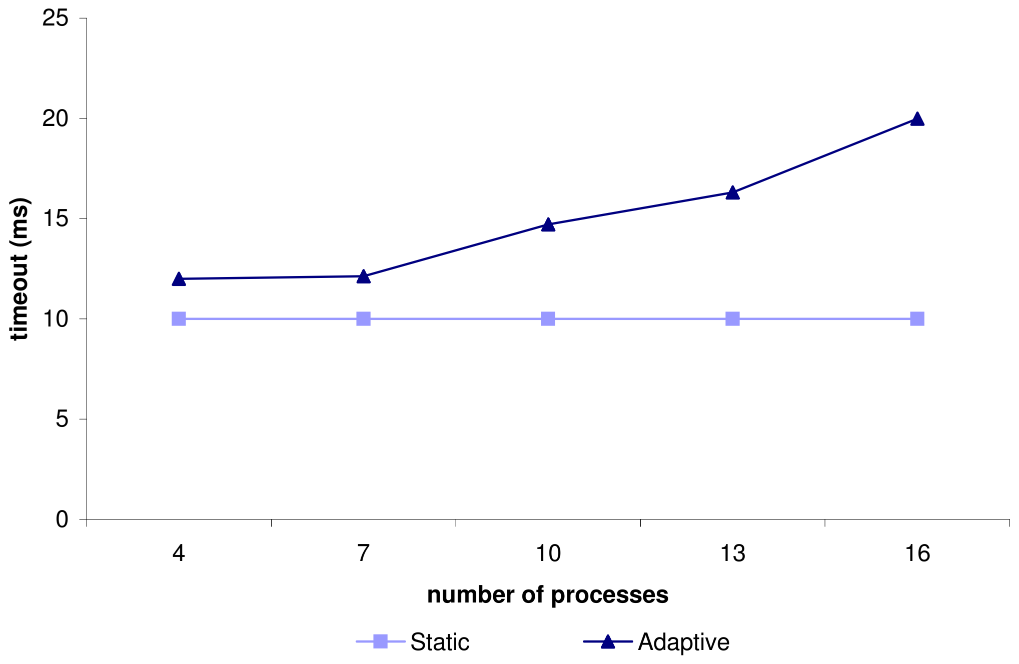


Figure 5.4: Average timeout.

The static version of the protocol was configured to use a timeout of 10 milliseconds. This value was obtained by running a set of empirical tests with different static values, being the value that provided the better results on average. The initial timeout for the adaptive version is also 10 milliseconds, but it is dynamically adjusted according to *Adaptare* outputs. *Adaptare* was configured to use a history size of $h = 30$, since the evaluation presented in Chapter 4 indicated that this value provides the best results. The required coverage was set to $C = \frac{\lfloor \frac{n}{2} \rfloor + 1}{n}$, as explained in Section 5.4.2.

Figure 5.4 shows the average timeouts. The lower timeout is the fixed value of the static algorithm, 10 milliseconds. The adaptive algorithm presents average timeouts from 12 to 20 milliseconds, depending on the number of processes participating in the consensus execution. When more processes are participating in the consensus execution, the contention in the shared medium is increased, affecting the transmission delays. Thus, timeout values computed by *Adaptare* increase with the number of consensus processes, since *Adaptare* analyzes the observed latencies to derive appropriate timeouts.

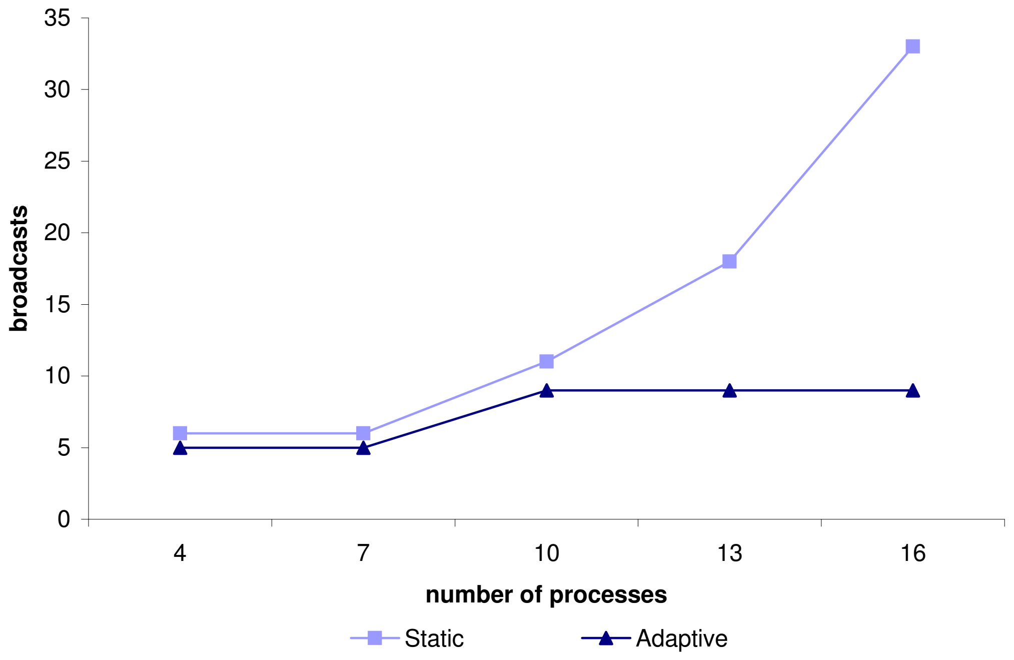


Figure 5.5: Average number of broadcasts per process.

Analyzing the number of broadcast messages sent by each process (Figure 5.5), the relation between timeouts and retransmissions is evident. The lower bound for the number of broadcasts is three, and it is achieved in fault-free executions (no omission occurs) in which the timeouts are sufficiently large to allow the reception of all required messages. These results show that in the adaptive consensus, which used higher timeouts, the number of broadcasts per process was significantly lower, up to 72% less for the case of 16 processes. This is a clear indication of the benefits that may be achieved by adapting the timeout. When the consensus protocol is executed by a higher number of processes, creating more contention and increased transmission delays, the static version uses an inadequate timeout value, whereas the adaptive version automatically adjusts the timeout to fit the environment conditions. By increasing the timeout, the adaptive version avoids premature retransmissions and hence prevents the network load to increase even more and affect negatively the observed network delays, as well as the consensus latency.

Increasing the timeout also increases the retransmission delay, when a retransmis-

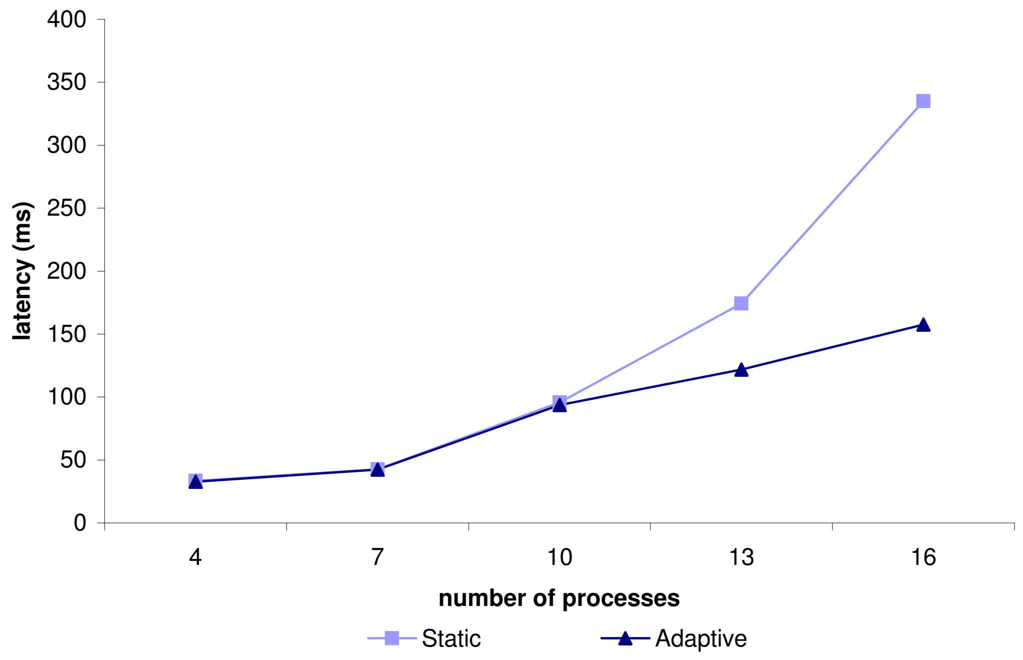


Figure 5.6: Average latency (consensus execution time).

sion must actually be performed. Ultimately, this delay could impact the consensus execution time, which is the fundamental performance indicator perceived by end users. However, our results show that the overall execution time is nevertheless better for the adaptive version, which indicates a positive trade-off in favor of the increased timeouts. In fact, Figure 5.6 shows the latency improvements for the dynamic version, which were particularly visible for the scenarios with 13 and 16 processes. This also confirms the experiments in the wireless environment presented in Section 5.3, which suggested that a timeout of 10 milliseconds would be sufficient for consensus among up to 10 processes, but too small for more processes. Both static and adaptive versions of the protocol achieved similar latencies in the scenarios with 4, 7 and 10 processes. However, in executions with 13 and 16 processes, with the adaptive timeout increased to 16 and 20 milliseconds (on average), latency improvements were of about 30% and 53%, respectively.

This set of experiments emphasizes the importance of dynamically adjusting time-related variables of distributed algorithms according to observed changes in the operating conditions. The timeout adjustments that took place in the adaptive algorithm lead to an improvement of the network load up to 72%, and of the latency up to 53%.

We believe that our contribution is relevant both from an engineering and from a practical perspective. By using *Adaptare* as a well-defined service for timeout provisioning we have shown that it is possible to easily add adaptive behavior to a previously static protocol, without the need for significant changes in the code. The practical outcome is that it becomes possible to improve performance without sacrificing other properties, namely safety. This is particularly important, as shown, in load-sensitive wireless environments.

5.7 Summary

In this chapter we addressed a problem that is sometimes disregarded, but that is practically relevant to improve the runtime performance of distributed algorithms: adapting timing variables and timeouts according to the actual conditions of the environment.

Following a pragmatic methodology, we described a simple approach to transform a static timeout-based consensus protocol for wireless ad hoc networks into an adaptive protocol. A fundamental building block of our solution was the *Adaptare* framework, which continuously provided the timeout values that should be used at a given moment during the protocol's execution.

The results of the experimental evaluation attested the benefits of using adaptive protocols in wireless networks. While the static version of the consensus algorithm can perform well only on the scenarios for which the static timeout has been determined, the adaptive version is able to adapt to any timeliness conditions.

Notes

An early version of the work presented in this chapter was reported as a FCUL technical report: "Timeout Adaptive Consensus: Improving Performance through Adaptation", Dixit, Moniz, and Casimiro, DI/FCUL TR-2010-06, November 2010 (Dixit *et al.*, 2010).

This subject was also discussed in “From static to dynamic protocols: adapting timeouts for improved performance”, Casimiro and Dixit, “Proceedings of the I Workshop on Autonomic Distributed Systems (WoSIDA’11)”, Campo Grande, Brazil, May 2011 (Casimiro & Dixit, 2011).

Finally, a more complete work with the content of this chapter will appear in “Timeout-based Adaptive Consensus: Improving Performance through Adaptation”, Dixit, Moniz, and Casimiro, “Proceedings of the 27th ACM Symposium on Applied Computing, Trento, Italy, March 2012 (Dixit *et al.*, 2012).

Chapter 6

Adaptare-FD

In Chapter 5 we demonstrated the applicability of *Adaptare* by developing and evaluating an adaptive consensus protocol in which timeout values are adjusted according to *Adaptare*'s output. In this chapter we address the problem of failure detection in distributed systems, as a second example of how *Adaptare* can be used as a building block in the design of dependable and adaptive protocols.

In distributed systems and applications, the need to detect the failure of system components in a fast and accurate way is often a fundamental issue. In order to ensure the correct operation of the failure detector according to some desired characteristics, it is necessary to deal with the environment uncertainties, concerning timeliness and reliability. Additionally, applications should be able to specify their dependability requirements, which should be directly mapped to requirements for failure detection. One possible way to cope with this problem is to reason in terms of the quality of service (QoS) of failure detectors, both in their specification and evaluation. In this chapter we propose a novel dependability-oriented approach for specifying the QoS of failure detectors, and introduce *Adaptare-FD*, an autonomic and adaptive failure detector that executes according to this new specification.

The chapter is organized as follows. Section 6.1 discusses the importance of implementing adaptive failure detectors in order to satisfy QoS requirements for failure detection in uncertain environments. In Section 6.2 we describe the system model and introduce the basic algorithm for failure detection considered in this chapter. We also present the different algorithms for adaptation implemented by the failure detectors that we later on compare with our approach. Section 6.3 introduces *Adaptare-FD*, explaining its interface, architecture, and operation. In Section 6.4 we discuss the applicability of our failure detector by comparing the dependability aspects of *Adaptare-FD* with different approaches. Section 6.5 presents the experimental evaluation of *Adaptare-FD*, in which we compare its performance with other timeout-based adaptive failure detectors.

6.1 Motivation

The problem of failure detection is interesting and difficult to solve in uncertain environments, since the quality of service (QoS) of the failure detector will depend on its ability to correctly characterize the environment state and adapt relevant parameters for failure detection. Therefore, we selected this problem as the second fundamental distributed system problem (along with consensus, addressed in Chapter 5) to focus in this thesis, in order to show the usefulness of *Adaptare*.

As presented in Chapter 2, Chandra & Toueg (1996) introduced the concept of unreliable failure detectors, which are essentially oracles that provide information about the status (alive or crashed) of system processes. They circumvent the FLP impossibility result (Fischer *et al.*, 1985) by encapsulating the temporal uncertainties observed in asynchronous systems, freeing the system designer from the need to deal with them. However, as shown in Sergent *et al.* (2001), the actual implementation of the failure detector is fundamental to the overall system performance.

Two generic performance-related attributes of failure detectors are their speed (how fast they detect a failure) and their accuracy (how well they avoid making mistakes by suspecting correct processes). With timeout-based failure detectors, these attributes depend on the timeout value and on the interrogation period (frequency of monitoring).

Therefore, a fundamental problem in the implementation of failure detectors in asynchronous and dynamic environments concerns the configuration of their operational parameters, which involves the need to handle non-functional user-level requirements and the ability to characterize the state of the operational environment. Furthermore, this has to be done continuously to handle environment changes.

One of the facets of this problem is concerned with how users should specify their requirements for failure detection, and which metrics should be used for quantifying performance. In that sense, the work in Chen *et al.* (2002) introduced a set of metrics for specifying and evaluating the QoS of failure detectors, abstracting from the specific implementation approach (see Chapter 2). These metrics are widely used to assess the QoS of failure detectors.

Moreover, to effectively address dependability concerns and deal with uncertain and changing environments, the specific solution to predict network delays and estimate timeouts becomes crucial. Our interest is thus on adaptive failure detection. Building on the basic *Push* and *Pull* styles of algorithm structuring, some works focus on algorithmic techniques to improve performance (Fetzer *et al.*, 2001), others on the provision of adequate interfaces to support the varying requirements of applications (Hayashibara *et al.*, 2004; Satzger *et al.*, 2007), others on methods for detecting environment changes and adapting parameters accordingly (Bertier *et al.*, 2002; Chen *et al.*, 2002; Falai & Bondavalli, 2005; Nunes & Jansch-Porto, 2004), and others on algorithms to secure some required QoS level (Chen *et al.*, 2002; de Sá & de Araújo Macêdo, 2010). The work presented in Chen *et al.* (2002) is particularly relevant to our purposes, since it is the most closely related work in the literature. It presents the first systematic study of the QoS of failure detectors, and proposes a failure detector (here referred as Chen's FD) that receives QoS requirements as input parameters and implements methods based on probability theory to compute interrogation periods and timeouts.

We show that our solution for dependable adaptation is useful in practice by introducing a new dependability-oriented approach for the specification of the failure detector QoS, and proposing a new failure detector, named *Adaptare-FD*. More specifically, our approach is said to be dependability-oriented since it allows QoS to be specified by means of the required coverage for failure detection. *Adaptare-FD* follows this new specification, and is thus able to exploit the mechanisms implemented in *Adaptare*. We evaluate and discuss the relative merits of *Adaptare-FD*, based on a comparative analysis with other approaches. In particular, we focus on the comparison with Chen's approach (Chen *et al.*, 2002), to reveal subtle differences that are not easily observed in a fast glance, and highlight the tradeoffs in choosing one of those solutions.

6.2 Adaptive failure detection

6.2.1 System model and basic algorithm

In order to understand the operation of the failure detectors evaluated in this chapter (including *Adaptare-FD*), it is important to specify the considered system model, which states the assumptions that are made regarding communication and process failures. We consider a distributed system with a finite set of processes $\Pi = \{p, q, ..., z\}$. Processes are interconnected through unreliable channels, which can loose messages or discard corrupted messages. Communication delays are probabilistic, following the stochastic model presented in Chapter 2. In fact, these assumptions are compatible with the ones stated in Chapter 3, on which *Adaptare* was designed. An environment satisfying the latter will also satisfy the assumptions stated here. Regarding processes, we assume that they only fail by crashing, but otherwise behave correctly.

We consider a pull-style crash failure detection model where a process *p* monitors a process *q*, by periodically sending it *"are you alive?"* query messages. Process *q* responds to every received *"are you alive?"* message with a corresponding *"I'm alive"* message. Depending on the reception instants of *q*'s responses, the output of the failure detector may be T (*trust*), or S (*suspect*). Since we adopt a pull-style failure detector model, no assumptions about synchronized clocks are needed.

Algorithm 8 shows the basic algorithm that is typically used in pull-style failure detectors, and that we consider in this chapter. The algorithm executes as follows.

- The monitoring process p is initially configured to suspect that the monitored process q is crashed, since it knows nothing about q (line 2).
- At times $\sigma_i = \sigma_{i-1} + \eta_{i-1}$, p sends an *"are you alive?"* query message mq_i to q, and updates its η_i (interrogation period) and δ_i (timeout) values (lines 7-10). η_i specifies how long p will wait to send the next query message ($\sigma_{i+1} = \sigma_i + \eta_i$), and δ_i defines the timeout for receiving the response for mq_i.
- A response m_i for query message mq_i is expected to be received between σ_i (time at which mq_i was sent) and $\tau_i = \sigma_i + \delta_i$ (timeout expiration for receiving m_i). Thus, if p does not receive the response message m_i for query message mq_i or a subsequent response message m_j ($j > i$) by τ_i, p suspects q (lines 11-13).

Algorithm 8: Failure detection algorithm.

```
Output: T (trust) or S (suspect)
1  Process p (monitoring process)
     // Initialization
2    $output \leftarrow S$
3    $RTTs \leftarrow \emptyset$
4    $\sigma_0 \leftarrow currentTime$
5    $\eta_0 \leftarrow compute\_\eta(RTTs)$
6    $\delta_0 \leftarrow compute\_\delta(RTTs)$
     // Send query messages
7    foreach $i \geq 1$, at time $\sigma_i = \sigma_{i-1} + \eta_{i-1}$ do
8        Send "are you alive?" message $mq_i$ to $q$
9        $\eta_i \leftarrow compute\_\eta(RTTs)$
10       $\delta_i \leftarrow compute\_\delta(RTTs)$
     // Check timeout expiration
11   foreach $i \geq 1$, at time $\tau_i = \sigma_i + \delta_i$ do
12       if did not receive message $m_j$ with $j \geq i$ then
13           $output \leftarrow S$
     // Receive responses
14   foreach "I'm alive" message $m_j$ received at time $t$ do
15       $rtt_j \leftarrow$ compute_rtt($m_j$)
16       $RTTs \leftarrow RTTs \cup rtt_j$
17       if $j \geq i$ and $t \in [\tau_i, \tau_{i+1})$ then
18           $output \leftarrow T$

19 Process q (monitored process)
20   foreach "are you alive?" message $mq_i$ received from $p$ do
21       send "I'm alive" message $m_i$ to $p$
```

- When p receives a response message m_j at time t, it computes m_j's RTT, i.e., the elapsed time from σ_j (time at which the query message mq_j was sent) to t (lines 14-16). Process p keeps a set of RTT values, used to compute η and δ values.

- If p receives a response message m_j at time $t \in [\tau_i, \tau_{i+1})$ and $j \geq i$, then p trusts q is alive. If $j < i$, then m_j is an old message and it should be discarded. If $j = i$, then the response for mq_i is late ($t \geq \tau_i$), but q is alive. If $j > i$, then m_j

timeout has not expired yet ($t < \tau_j$), and p can trust q is alive (lines 14,17-18).

- The monitored process q responds to every received query message mq_i with an *"I'm alive"* response message m_i, sent to the monitoring process p (lines 20-21).

The main distinguishing factor between the different adaptive failure detector approaches lies in the solutions used for computing the interrogation period, η, and the timeout, δ. These two parameters are dynamically adjusted during the execution, depending on the required QoS (usually defined in a static way during initialization) and on the observed behavior of the communication environment (which may change). Therefore, the functions $compute\_\eta(RTTs)$ and $compute\_\delta(RTTs)$ encapsulate the configuration procedures that take place during initialization, and every time a query message is sent. In that way, their values will vary according to the delays observed on the reception of response messages, which are kept in the $RTTs$ set. The different approaches for adaptation to which we compare our proposal are presented next.

6.2.2 Chen's failure detector

Chen *et al.* (2002) propose a failure detector in which the parameters are configured to meet some required QoS level, specified by three metrics: an upper bound on the detection time (T_D^U), a lower bound on the average mistake recurrence time (T_{MR}^L), and an upper bound on the average mistake duration (T_M^U). This failure detector implements a push-style algorithm, in which monitored processes send periodic heartbeats to the monitoring process spontaneously, instead of responding to query messages.

Figure 6.1 (from Chen *et al.* (2002)) presents a schematic view of Chen's FD. QoS requirements are given as input parameters. The estimator module predicts the message behavior, described by the message loss probability p_L, and by the expected value ($E(D)$) and variance ($V(D)$) of message delays.

Finally, the configurator module uses all this information to compute the interrogation period and timeout values. The authors describe several configuration procedures, which depend on different assumptions that may be made about the system in which the failure detector is used. Although the algorithms and evaluation presented in Chen *et al.* (2002) do not contemplate adaptive failure detectors, the authors pointed out that their failure detector could be made adaptive by periodically re-executing the estimator

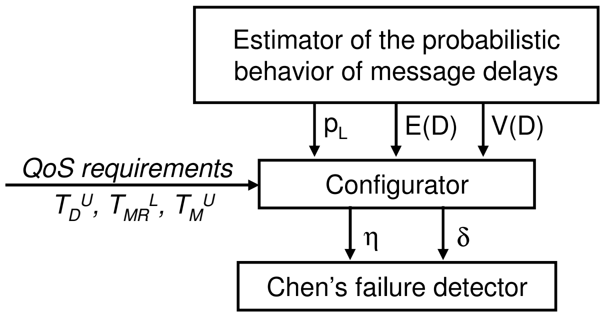


Figure 6.1: Schematic view of Chen's failure detector.

and configurator modules, computing new values for η and δ, based on p_L, $E(D)$ and $V(D)$ estimated from the most recent heartbeats.

We implemented a fully adaptive version of this failure detector with slight modifications in order to use the pull-style algorithm presented in Algorithm 8. From the different configuration procedures proposed in Chen *et al.* (2002), we adopted the one that assumes unknown message behavior and synchronized clocks. Note that we do not make any assumption about clock synchronization, but since we implemented a pull-style failure detector, both sending and receiving times are measured on the same process, thus we can safely use this procedure.

The configuration procedure to compute the interrogation period η and timeout δ performs the following steps (from Chen *et al.* (2002)):

- Compute:

$$\gamma = \frac{(1 - p_L)(T_D^U - E(D))^2}{V(D) + (T_D^U - E(D))^2},$$

  and let:

$$\eta_{max} = min(\gamma' T_M^U, T_D^U - E(D)).$$

  If $\eta_{max} = 0$, then output "QoS cannot be achieved" and stop.

- Let:

$$f(\eta) = \eta \cdot \prod_{j=1}^{\lceil (T_D^U - E(D))/\eta \rceil - 1} \frac{V(D) + (T_D^U - E(D) - j\eta)^2}{V(D) + p_L(T_D^U - E(D) - j\eta)^2}.$$

Find the largest $\eta \leq \eta_{max}$ such that $f(\eta) \geq T_{MR}^L$.

- Set $\delta = T_D^U - \eta$ and output η and δ.

This procedure, which we present here for self-containment, is based on the probability theory, and assumes that $T_D^U > E(D)$. The values of p_L, $E(D)$, and $V(D)$ are estimated based on measured RTTs of past messages. Interested readers may refer to Chen *et al.* (2002) for further information.

6.2.3 Other timeout estimation methods

Most of the timeout-based adaptive failure detectors described in the literature are based on fixed interrogation periods, and a common approach to estimate timeouts: the use of an estimator, plus a safety margin (Bertier *et al.*, 2002; Falai & Bondavalli, 2005; Nunes & Jansch-Porto, 2004).

The failure detectors used in our comparative evaluation that follow this approach combine two different estimators with two methods to compute dynamic safety margins. The estimators are defined as follows, where rtt_i is the measured RTT at time i, and $r\hat{t}t_i$ is an estimation for rtt_i.

- WinMean(n): Computes the average of the delays of the last n received messages.

$$r\hat{t}t_{t+1} = \frac{\sum_{i=t-n}^{t} rtt_i}{n}$$

- Lpf(β): A simplified ARIMA (AutoRegressive Integrated Moving Average) estimator, in which the estimated value is the exponential smoothing of the observations (Falai & Bondavalli, 2005). The parameter β represents the importance of the error in the last estimation.

$$\hat{rtt}_{t+1} = \hat{rtt}_t + \beta(rtt_t - \hat{rtt}_t)$$

The considered approaches to compute dynamic safety margins were proposed in Nunes & Jansch-Porto (2004), and also used in Falai & Bondavalli (2005). They are:

- Cib(n,γ): Assumes that the predictor correctly models the communication delays, and the estimator is a linear function. In the equation, γ corresponds to the confidence level in the standard Normal distribution function, $\hat{\sigma}$ is the estimator of the standard deviation and $\bar{rtt}$ is the mean of the last n observed delays.

$$Cib_{t+1} = \gamma\hat{\sigma}\sqrt{1 + \frac{1}{n} + \frac{(rtt_t - \bar{rtt})^2}{\sum_{i=t-n}^{t}(rtt_i - \bar{rtt})^2}}$$

- Jac(ϕ,α): Based on the Jacobson estimation method (Jacobson, 1988), considers the error on the last estimation to adapt its value. α is a smoothing constant which defines the speed of the reaction to error variation (Nunes & Jansch-Porto, 2004), and ϕ is a multiplier factor that defines how conservative the margin will be.

$$Jac_{t+1} = \phi(Jac_t + \alpha(|rtt_t - \hat{rtt}_t| - Jac_t))$$

6.3 *Adaptare-FD*

In this section we introduce *Adaptare-FD*, a dependability-oriented adaptive failure detector. *Adaptare-FD* is based on a novel approach for specifying the required QoS for failure detection, and is built upon *Adaptare*, which is used as an underlying service for timeout computation.

The architecture of *Adaptare-FD* is illustrated in Figure 6.2. The fundamental idea behind our approach is to achieve a failure detector that dynamically adapts to changing network conditions and is easily configured, by specifying an upper bound on the detection time (T_D^U, as with Chen's FD) and the minimum coverage (C^L) for failure detection (i.e., a lower bound on the probability that the failure detector output - *trust*

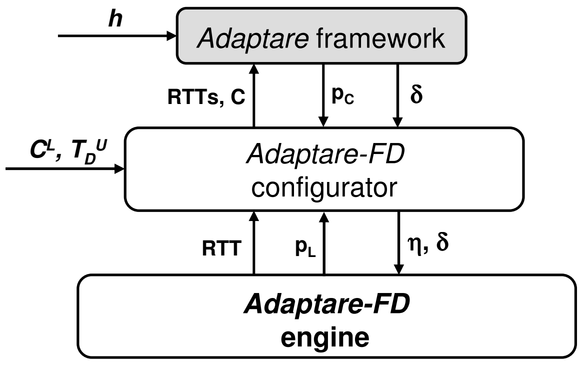


Figure 6.2: *Adaptare-FD* architecture.

or *suspect* - will be correct). By specifying the required confidence on a fundamental assumption we achieve a *dependable approach for failure detection*. Moreover, the definition of those two parameters at the failure detector interface allows controlling both speed and accuracy of *Adaptare-FD*.

The timeout δ_i is adapted in runtime, according to *Adaptare*'s output. The random variable to be monitored is the elapsed time from sending each query message mq_i (*"are you alive?"*) to receiving the corresponding response m_i (*"I'm alive"*), i.e., the round trip delay (RTT). Thus, on the reception of a response message, *Adaptare-FD* measures the RTT and provides this information to *Adaptare*.

The coverage value specified at *Adaptare-FD*'s interface (C^L) is the coverage perceived by end users (applications that use the failure detection service). Thus, it defines a threshold for the total number of mistakes (false suspicions) that the failure detector is allowed to make. Clearly, if we want to limit the number of mistakes, we need to understand the conditions that lead to them. Based on that, it will be possible to define how *Adaptare* can be used to deal with those mistakes, and in particular what is the coverage value that must be requested to *Adaptare*. We distinguish the following three causes of failure detection mistakes:

- Message losses: considering the failure detection algorithm presented in Algorithm 8 and that the timeout is typically lower than the interrogation period (as we discuss ahead), every lost message will cause a failure detection mistake: if the message is lost, the timeout will expire and *Adaptare-FD* will suspect that a correct process is crashed until the reception of the next response message. This happens with probability p_L.

- Incorrect timeout: as explained in Chapter 3, the bounds provided by *Adaptare* when the environment starts to change (initiating a transient phase) are not safe until this change is detected (usually after a sample from the transient phase is measured and a new bound is computed). In this case, unsafe timeouts might not be large enough, possibly causing a false suspicion: *Adaptare-FD* will suspect a correct process between the timeout expiration and the reception of the response message. This occurs with probability p_C, which is the "environment change" probability.

- Late message: even when timeouts computed by *Adaptare* are correct (given the required coverage), some messages might arrive after the timeout. In these cases, *Adaptare-FD* will be in a mistaken state between the timeout expiration and the reception of the response message. This happens with probability $1 - C$, where C is the coverage requested to *Adaptare*.

Given all the above, it follows that in order to ensure the coverage for failure detection requested by the user, we need to deal with p_L and p_C. In a network subject to losses and stability changes, the final observed coverage of failure detection (C_{FD}) will be:

$$C_{FD} = (1 - p_L) \times (1 - p_C) \times C,$$

The intuition is the following: the failure detector will not make mistakes when there are no message losses, the environment is stable and message delays are within the considered timeouts. Therefore, the probability of not making mistakes (observed coverage of failure detection, C_{FD}) is the product of $1 - p_L$ (no message loss), $1 - p_C$ (probability that the environment is stable) and C (probability that message delays are within selected timeouts).

Consequently, in order to bound the observed coverage C_{FD} according to the requested coverage C^L, provided at the failure detector interface, the coverage to be requested to *Adaptare* is:

$$C = \frac{C^L}{(1 - p_L) \times (1 - p_C)}$$

It remains to explain how p_L and p_C are obtained. We could simply postulate arbitrary values for these probabilities. However, a better strategy is to calculate them dynamically, which can be done based on the monitoring data. Therefore, in our solution we have *Adaptare-FD* estimating p_L and p_C in runtime. Since each message has a sequence number, losses are easily detected. p_L is estimated as the number of missing responses, divided by the highest sequence number received so far. This method is suggested in Chen *et al.* (2002), and it was also used in our adaptive implementation of Chen's FD.

Regarding p_C, *Adaptare-FD* uses a counter to keep track of the number of times the environment changes from stable to transient, which is an information obtained from *Adaptare*. Then, p_C is estimated by the number of observed changes divided by the number of computed timeouts.

As with Chen's FD, there are conditions under which the requested QoS cannot be secured by *Adaptare-FD*. Namely, if the coverage C derived from the equation above is higher than 1, this means that the environment is too unstable to allow satisfying the required coverage (C^L). In other words, p_L or p_C (or both) are too high, possibly leading to more mistakes than accepted by the client application. In this case, in a best effort attempt, *Adaptare-FD* requires a very high coverage to *Adaptare* in order to get a high timeout and avoid extra mistakes, but still the minimum coverage may not be achieved.

The interrogation period η is computed by the configurator module, according to the timeout δ received from *Adaptare* and the requested upper bound on the detection time T_D^U. As shown in Figure 6.3, the detection time is bounded by:

$$T_D^U = \eta + \delta$$

Thus, in order to secure the required T_D^U, whenever a new timeout value is provided by *Adaptare*, the interrogation period is adjusted to:

$$\eta_i = T_D^U - \delta_i$$

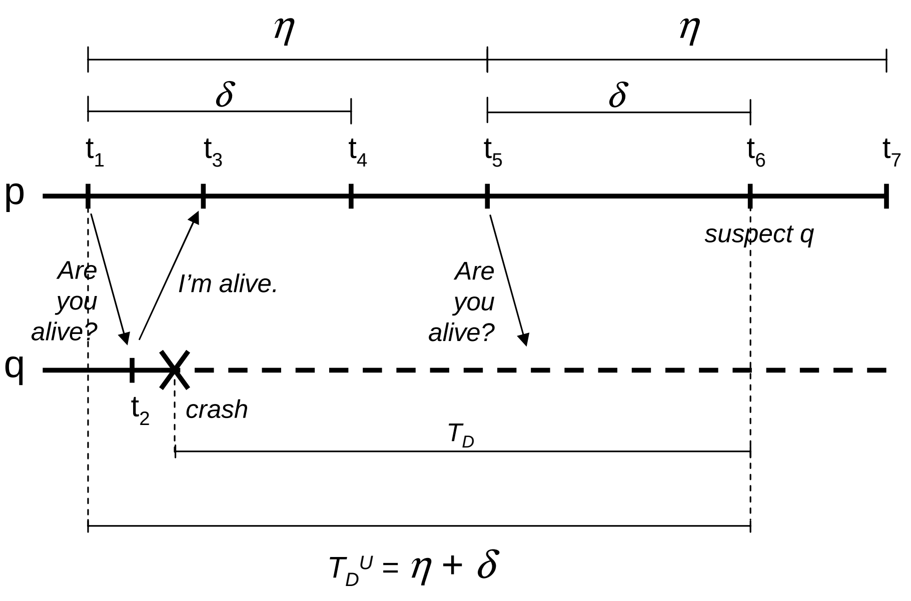


Figure 6.3: Upper bound on the detection time T_D^U.

Note that the interrogation period cannot be made too small, to ensure that the overhead of failure detection does not become relevant. In particular, it is physically impossible to satisfy T_D^U if $T_D^U \leq \delta_i$. In practice, we consider that it is reasonable to assume that T_D^U will be at least twice the timeout, therefore ensuring that the interrogation period will be larger than the timeout (which is the reason why a lost message will always lead to a timing failure).

6.4 Why using *Adaptare-FD*?

6.4.1 *Adaptare-FD* vs. Chen's failure detector

In asynchronous systems with process crashes any timeout-based failure detector implementation is unreliable. Mistakes will eventually occur from time to time, when assumptions about communication delays, which are used for setting timeouts, are violated. When specifying the QoS of a failure detector it is thus necessary to limit these mistakes in some manner.

This is achieved with Chen's FD by defining a lower bound on the mistake recurrence time (T_{MR}^L). However, this single (safety) condition is not sufficient to specify a useful failure detector – a trivial implementation would just produce an output every T^L_{MR} time units. At least one liveness condition is necessary in addition. Thus, the authors propose the specification of upper bounds on the detection time (T^U_D) and on the average mistake duration (T^U_M). Then, the failure detector configurator module computes the necessary interrogation period to secure these bounds, and thus liveness is achieved.

Adaptare-FD limits its number of mistakes by requiring a lower bound on the probability of providing a correct output (C^L), which together with other parameters (namely the message loss and environment change probabilities) will define the minimum coverage for the assumed round trip transmission delays. *Adaptare-FD* also requires an upper bound on the detection time (T^U_D), like Chen's FD. The interrogation period is derived from the timeouts computed by *Adaptare*, and the specified T^U_D.

By taking the coverage of the failure detector output as a QoS parameter, *Adaptare-FD* is a good approach when the designer knows *how dependable* it wants the failure detector to be. For instance, if a high coverage, close to one, is specified (e.g., $C^L = 0.999$), this implies that *Adaptare-FD* will tend to use high timeouts to avoid false positives and achieve the specified probability. Additionally, the interrogation period is limited by the given upper bound on the detection time T^U_D, securing detection speed and leading to a very accurate failure detector. On the other hand, a coverage close to zero will lead to very small timeout values and hence more mistakes, which means poor overall accuracy. The maximum achievable coverage is, however, limited by p_L and p_C, as explained earlier.

In functional terms, with *Adaptare-FD* the timeout is dynamically adjusted to always be the lowest possible, given the environment conditions and the required coverage. Moreover, the interrogation period is the highest possible, given the required T^U_D and the computed timeout. Consequently, the average detection time and the network utilization are optimized.

One relevant difference between *Adaptare-FD* and Chen's FD is related to the way in which they deal with the message loss probability p_L and the environment change probability p_C. Since in *Adaptare-FD* timeouts are set to the lowest possible values, they are in general lower than the interrogation period, as already mentioned. Because of that, every message loss will result in a mistake. Furthermore, when the environment changes, the current timeout may not hold to the new conditions, which can also be the cause of mistakes. The approach followed by *Adaptare-FD* to address this problem is to increase the coverage requested to *Adaptare* in order to minimize the occurrence of mistakes in the “normal” conditions, in the necessary measure to compensate the false suspicions generated by losses or environment changes. However, if the required coverage for failure detection is already high, this procedure may fail. In the case of Chen’s FD, message loss is compensated by reducing the interrogation period in such a way that several heartbeats can be sent within one single timeout duration. However, this may increase the overhead of failure detection significantly, making it impractical. Additionally, the approach followed by Chen to deal with p_L requires some extra assumptions to be made (e.g., they assume that no consecutive message losses occur), and these assumptions may not always hold. In these cases, the QoS provided by the failure detector may not satisfy the requirements. In particular, the T_M requirement may not be satisfied. Our approach is flexible in this respect, since we do not impose a restriction on T_M, and thus we do not need to make any additional assumption.

Regarding dependability, Chen’s approach requires the specification of upper bounds on T_{MR} and T_M, which is an alternative way of achieving a dependable failure detector. This has implications on the configuration process. While *Adaptare-FD* optimizes timeouts, and then sets the necessary interrogation period to satisfy T_D^U, Chen’s approach first computes the interrogation period, then configures the timeout ($\delta = T_D^U - \eta$). Since in Chen’s FD interrogation periods usually end up being small to deal with message loss and to achieve the required QoS, timeouts tend to be higher than the interrogation periods. Therefore, the price to pay for dealing with message loss is that no effort is made to select lower timeouts, implying that the average detection time is not optimized.

Interestingly, we note that uncertainty, when higher than expected, causes problems to both approaches, for different reasons. In the case of *Adaptare-FD*, it affects p_C (more environment changes) and p_L (more message losses), preventing a high coverage to be achieved. With Chen’s FD, it affects p_L (and the probability that consecutive losses occur), leading to reduced periods and to possible violations of the T_M^U requirement.

6.4.2 *Adaptare-FD* vs. other timeout-based adaptive failure detectors

Except for Chen's FD, all timeout-based adaptive failure detectors considered in our evaluation combine some estimator with a safety margin in order to compute timeouts.

Comparing to them, a striking advantage of using *Adaptare-FD* is the possibility of specifying the desired coverage as a dependability requirement, which serves to automatically and dynamically configure the failure detector. In contrast, with other solutions the required QoS level can only be achieved if some underlying parameters are carefully adjusted. This can only be done based on observed execution results, following a typical trial and error approach in order to tune the failure detector parameters for that specific environment and QoS demand. Whenever the network changes, or when the client application requires a new dependability level, it will be necessary to fine tune the failure detector again, for the new conditions.

Another distinguishing feature is that *Adaptare-FD* optimizes network utilization by adjusting the interrogation period to the highest possible value with respect to the required T_D^U and the current timeout. The considered failure detectors based on estimators and safety margins adopt a fixed interrogation period, which must be manually configured by the system designer.

The remaining differences between *Adaptare-FD* and the other timeout-based adaptive failure detectors follow from the mechanisms that are used to compute the timeout values. In the next section we provide experimental evaluation results that allow comparing the different approaches, based on typical metrics for the QoS of failure detectors.

6.5 Experimental evaluation

In this section we describe the performed experiments and discuss the evaluation results in terms of achieved QoS, comparing *Adaptare-FD* with other timeout-based adaptive failure detectors. We give special emphasis to the comparison with Chen's FD, which is more similar to *Adaptare-FD* in the sense that it takes QoS input parameters, and adapts both timeout and interrogation period.

6.5.1 Environment setup

The experiments were performed in the PlanetLab environment (PlanetLab Consortium, 2004), which is a platform widely used by the research community as a test bed for distributed applications. We selected six geographically distributed nodes from PlanetLab, grouped in three pairs of linked nodes with different characteristics, in order to simulate distributed failure detection services operating in distinct networks.

The first link is composed by two nodes in a local area network (LAN), located in the same institution in Portugal. The second link contains two nodes in the USA which are 900km apart, representing a low-delay wide area network (low-delay WAN). Finally, the third link represents an intercontinental high-delay wide area network (high-delay WAN), with one node in the USA, and the other node in Australia. Interestingly, the intercontinental WAN presented a more stable behavior (in terms of delays variation) than the low-delay WAN composed by nodes in the USA. Table 6.1 shows RTT statistics for the selected links.

Link	AVG	STD	MIN	MAX
LAN	0.41	0.97	0	295
Low-delay WAN	23.30	52.50	17	7279
High-delay WAN	226.75	6.62	226	1994

Table 6.1: Links RTT statistics (ms).

Figure 6.4 illustrates the operation of nodes in our experiments. Each link connects two nodes. The first node executes two monitoring processes: one of them runs *Adaptare-FD*'s algorithm, and the other runs Chen's algorithm. The second node executes two monitored processes: one replies to query messages from *Adaptare-FD*, and the other replies to query messages from Chen's FD.

The remaining adaptive failure detectors, which use static interrogation periods, were evaluated offline using the RTT traces collected by *Adaptare-FD* during the execution of the experiments. Thus, we simulate their execution as if they used the same interrogation periods of *Adaptare-FD*. We believe that this methodology is valid for the evaluation that we perform, given that the interrogation period has no influence on the achieved coverage, and consequently on the dependability of the solution. We realize that in the cases in which the failure detectors based on estimators and safety margins

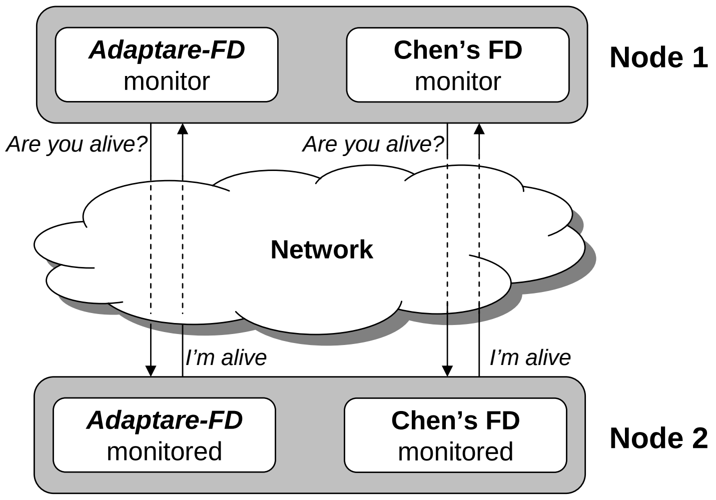


Figure 6.4: Simultaneous execution of *Adaptare-FD* and Chen's FD.

compute higher timeouts than *Adaptare-FD*, using the same interrogation periods that were computed by *Adaptare-FD*, the required T_D^U will not be satisfied. However, as we see later, higher timeouts are computed by only two of the twelve evaluated failure detectors. Note that if these failure detectors used smaller periods, they would at most secure T_D^U (as *Adaptare-FD* does), but increasing the network load, which means that no improvement would be achieved in comparison with *Adaptare-FD*. As we will see, in practice there is no impact of using non-fixed interrogation periods, at least in a way that could influence the conclusions in benefit of *Adaptare*. Our evaluation is focused on demonstrating that the main limitation of these solutions lies on their inability of automatically adjusting operational parameters according to the network conditions and required QoS, which is independent of specific interrogation periods.

6.5.2 Failure detectors configuration

To compare *Adaptare-FD* and Chen's FD, we set three experiments with varying levels of QoS requirements, as follows.

- Experiment $E1$ represents a failure detection service operating in a system requiring high availability. Failures must be detected within at most 10 seconds

($T_D^U = 10s$), and in order to have such a fast detection, the system accepts a false positives rate of 10% ($C^L = 0.9$).

- Experiment $E2$ represents a system in which accuracy is the primary concern, instead of availability. Mistakes must be avoided, for example due to high recovery costs. Thus, the failure detector may present no more than 1% of false positives ($C^L = 0.99$), and in case of failures, a detection time of up to 40 seconds is acceptable ($T_D^U = 40s$).
- Experiment $E3$ allows to analyze how the evaluated failure detectors would perform in a system requiring even higher accuracy ($C^L = 0.999$), and also a reasonable fast detection time ($T_D^U = 25s$).

Table 6.2 presents the QoS parameters for both *Adaptare-FD* and Chen's FD, in each experiment. Chen's parameters were set according to *Adaptare-FD* parameters. *Adaptare-FD*'s timeouts will be as tight as possible, in the order of milliseconds. Thus, the interrogation period, which is computed as $\eta = T_D^U - \delta$, will be close to T_D^U. Take experiment $E1$ as an example: a coverage of 90% means that the system accepts up to one mistake every ten observations. Since the interrogation period will be close to 10 seconds, *Adaptare-FD* may present approximately one mistake per 100 seconds. Thus, we set the upper bound for the average mistake recurrence time T_{MR}^U required by Chen's FD to 100 seconds. We followed this same reasoning to set the T_{MR}^U for experiments $E2$ and $E3$.

QoS	C^L	T_D^U	T_{MR}^L	T_M^U
Experiment $E1$	0.9	$10s$	$100s$	$10s$
Experiment $E2$	0.99	$40s$	$1h$	$40s$
Experiment $E3$	0.999	$25s$	$6h$	$25s$

Table 6.2: QoS parameters for FDs configuration.

Chen's FD also requires an upper bound for the mistake duration, which we set to be the same as T_D^U. Our reasoning is that in the worst case, a mistake will be caused by a message loss, and it will be corrected upon the reception of the response for the next query. Thus, a very conservative upper bound for the mistake duration would be the interrogation period plus the timeout, which is the definition of T_D^U.

Usually, Chen's FD is very conservative on setting interrogation periods. The configuration procedure presented in Section 6.2.2 allows a minimum period of 1 millisecond. In practice, this value is too small, and it has two negative impacts: the network is overloaded, and processes receive more messages than they can process, exhausting the system memory. In situations like this, PlanetLab automatically kills the running experiments. In fact, we had this problem in our first experiments. Because of that, we set a minimum interrogation period of 500 milliseconds, which was sufficient to execute our experiments in PlanetLab. If the configuration procedure of Chen's FD would yield a value lower than 500 milliseconds, we would use this minimum instead. Clearly, this can have an impact on the results, but the limitation would be felt in a real system anyway.

Finally, *Adaptare* was configured to use a history size $h = 30$, as suggested by the results presented in Chapter 4.

The remaining failure detectors in this evaluation compute timeouts using a simple estimator, plus some safety margin, as presented in Section 6.4.2. By combining the WinMean ($n = 30$) and Lpf ($\beta = 0.125$) predictors with three different levels of Cib and Jacobson's safety margins (*high*, *med* and *low*), we defined 12 different timeout-based adaptive failure detectors. The safety margin parameters were configured as proposed in Falai & Bondavalli (2005): 3.31, 2.586, and 1.648 for Cib's γ parameter; and 4, 2, and 1 for Jac's ϕ parameter.

6.5.3 Evaluation results

In this section we analyze the QoS achieved by the evaluated adaptive failure detectors, focusing on the results in which the differences between them are more evident. The analysis of these results shows that the characteristics and limitations of each solution, discussed in Section 6.4, were verified in practice. The complete set of results is presented in Appendix B.

Securing T_D^U. Figure 6.5 shows the average timeout values of each implemented failure detector in experiment $E1$. As previously explained, for the pull-style failure detector algorithm considered in this work, the detection time is bounded by the sum of the timeout δ and the interrogation period η. Observing these results, we classified

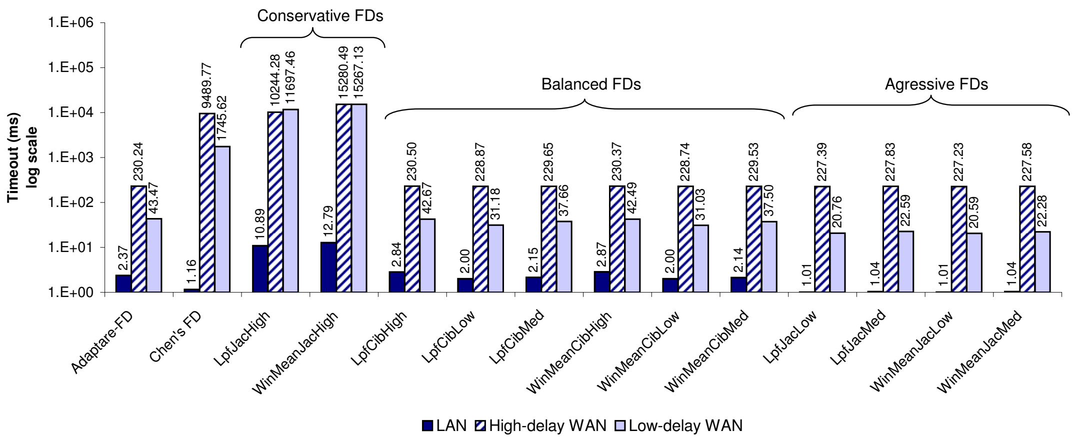


Figure 6.5: Average timeouts in experiment E1 ($T_D^U = 10$s).

the failure detectors based on estimators and safety margins into three groups: the *conservative* group comprises two failure detectors that had timeouts much higher (at least one order of magnitude) than the others; the *aggressive* group includes the four failure detectors which presented the lowest average timeouts; and the remaining six failure detectors constitute the so-called *balanced* group.

Both *Adaptare-FD* and Chen's FD secure the required T_D^U by construction, ensuring that $\eta + \delta = T_D^U$. As the remaining failure detectors used the same interrogation periods as *Adaptare-FD*, all failure detectors in the balanced and aggressive groups secured the required T_D^U. In contrast, the conservative failure detectors exceeded the specified T_D^U in all experiments, due to their higher timeouts.

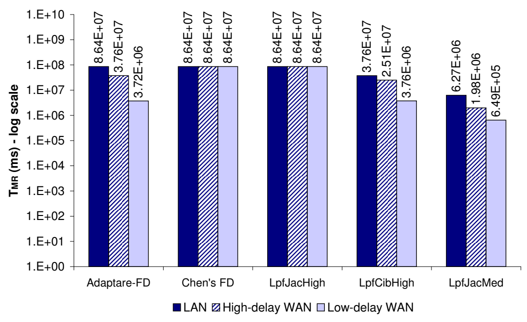
Despite of ensuring that $\eta + \delta = T_D^U$, *Adaptare-FD* and Chen's FD differ on how these parameters are configured. In Section 6.4.1 we discussed that while *Adaptare-FD* computes the lowest possible timeouts, and then set the interrogation periods accordingly, Chen's FD computes small interrogation periods to cope with message losses, and then set timeouts considering the required T_D^U and the computed interrogation periods. Because of that, Chen's FD may produce too small interrogation periods, overloading the network. For example, in experiment $E1$, Chen's FD presented an average timeout of 9.489 seconds in the high-delay WAN test scenario (Figure 6.5). Given that $T_D^U = 10s$, the average interrogation period was of 511 milliseconds, which suggests that the minimum interrogation period (500ms) was used several times. In fact, we analyzed Chen's FD results for this experiment, and approximately 70% of the interrogation periods were set to the minimum value. In order to verify what would be the values computed by Chen's FD if we did not stipulate a minimum value, we applied the Chen's algorithm to compute the interrogation period in the collected traces, offline. The result was an interrogation period of 191 milliseconds on average, which we consider to be too small to be used in a practical failure detector. We realize that this offline analysis does not represent a real execution, since more frequent messages could possibly interfer in the message loss probability and in the measured RTTs. Anyway, the message loss probability would likely increase, which would lead to even lower periods, thus increasing the problem. We believe that this clearly illustrates the limitation of Chen's approach, which we discussed in Section 6.4.1.

Additionally, it is important to observe that T_D^U specifies an upper bound on the detection time, but lower timeouts will produce better detection times on average. Also, lower timeouts imply higher interrogation periods (for *Adaptare-FD* and Chen's FD), which is a desirable feature as it minimizes the network load imposed by the failure detection service.

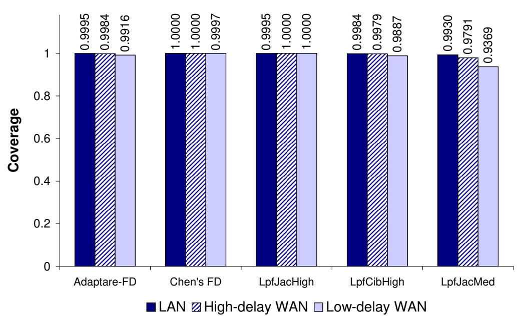
The graphs presented in the remaining of this section show the results of five failure detectors: *Adaptare-FD*, Chen's FD, and the best failure detector of each group (conservative, aggressive, balanced). This approach allows cleaner but still representative graphics, since that failure detectors in the same group achieved very similar results. We recall that the full set of results is presented Appendix B.

Securing T_{MR}^L and C^L. The average mistake recurrence time (T_{MR}) and coverage (C) results follow the expected trend: failure detectors with higher timeouts make less mistakes, thus they achieve better T_{MR} and C values. In fact, all failure detectors secured the required T_{MR}^L and C^L in experiment $E1$ for all the considered scenarios. In experiment $E2$, the balanced failure detectors did not secure the required coverage in the low-delay WAN environment tests, and the aggressive failure detectors failed in securing both T_{MR}^L and C^L, in the low-delay WAN and high-delay WAN scenarios, as shown in Figure 6.6.

Nevertheless, the differences between the evaluated failure detectors are more evident when a higher QoS level is required, as in experiment $E3$ (Figure 6.7): only Chen's FD and the conservative *LpfJacHigh* failure detector (the two failure detectors with higher average timeouts) could secure the expected T_{MR}^L and C^L in all networks.

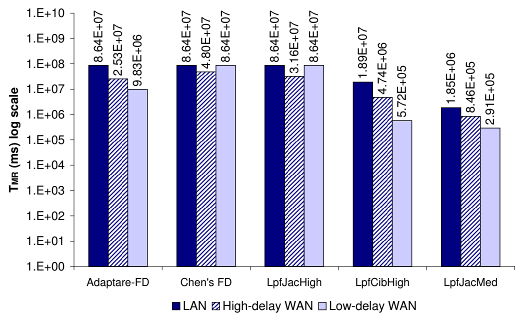


(a) T_{MR} results ($T^L_{MR} = 1\text{h} = 3.6E + 06\text{ms}$).

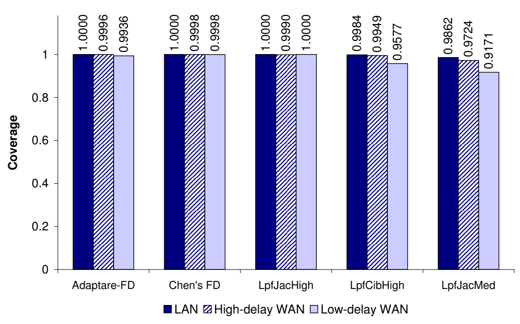


(b) C results ($C^L = 0.99$).

Figure 6.6: Average mistake recurrence time and average coverage in experiment $E2$.



(a) T_{MR} results ($T^L_{MR} = 6\text{h} = 2.16E + 07\text{ms}$).



(b) C results ($C^L = 0.999$).

Figure 6.7: Average mistake recurrence time and average coverage in experiment $E3$.

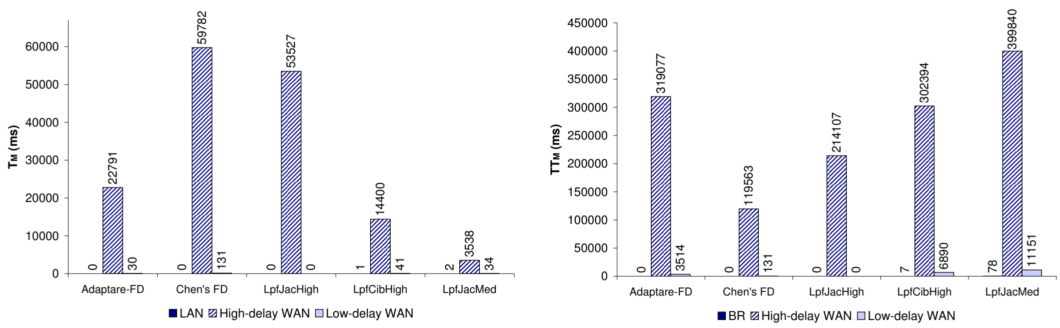
As discussed in Section 6.4, the approach followed by *Adaptare-FD* tries to counterbalance mistakes from losses and environment changes by requesting higher accuracy requirements (coverage) to *Adaptare*, in order to decrease the amount of mistakes. If accuracy requirements are already high (like in experiment $E3$), there is no slack for applying this compensation: mathematically, the computed coverage to be required to *Adaptare* will be higher than 1. In this situation, *Adaptare-FD* requests *Adaptare* a timeout with very high coverage (in our experiments, $C = 0.9999$), in an effort to avoid more mistakes. However, the required coverage, as high as it is, cannot be guaranteed.

This was observed in the results of experiment $E3$ in the low-delay WAN: *Adaptare-FD* did not achieve the expected coverage. This low-delay WAN link was the one exhibiting higher variations in the network delays, leading to a computed p_C that could not be compensated, considering the high required coverage $C^L = 0.999$. Note that Chen's approach is based on the conservative one-sided inequality to compute its parameters, thus it should not be affected by environment changes as *Adaptare-FD*. This is also reflected in the average timeouts for this experiment, which are lower in the case of *Adaptare-FD* (1.126 seconds), comparing with Chen's FD (6.831 seconds). In this particular case, the general strategy of choosing small timeouts becomes too aggressive, since the environment is not sufficiently stable. Note that in a real deployment, and since *Adaptare-FD* is able to detect that it cannot guarantee the requested coverage (C^L is too high for the observed p_C), it would be possible to provide an indication to the client application, letting it know that the requirements could possibly not be met.

Concerning the failure detectors based on estimators and safety margins, none of them achieved the expected T^L_{MR} and C^L in experiment $E3$, whatever the considered network environment (with the exception of the conservative *LpfJacHigh* failure detector, as already mentioned). This result illustrates the main weakness of these failure detectors: because they are statically configured for a given environment, they cannot automatically adapt to variations in the environment conditions. Given a specific configuration, they may work fine for some set of QoS requirements and some environment. However, to achieve different QoS levels or the same QoS in a different environment, they must be reconfigured, and the mapping from the desired QoS into their operational parameters values is not obvious, requiring adjustments through a fine tuning process.

Securing T^U_M. Although not considered in the configuration of *Adaptare-FD*, the upper bound on the average mistake duration (T^U_M) is one of Chen's FD input parameters, and represents an important metric for evaluating the performance of failure detectors.

We observed in our experiments that the longest mistake durations are due to consecutive message losses, corresponding to extreme situations. All failure detectors do these mistakes. However, the failure detectors with lower timeouts also make mistakes due to late messages (messages that arrive after the timeout), which have relatively small durations, since they are corrected as soon as these messages arrive. Because of that, the average mistake duration is reduced. Failure detectors with higher timeouts do not produce false positives due to incorrect timeouts, their mistakes are almost



(a) T_M results ($T_M^U = 25$s).

(b) TT_M results.

Figure 6.8: Average mistake duration and total mistake duration in experiment $E3$.

exclusively derived from message losses. Consequently, they present higher average mistake durations. This effect becomes evident when we analyze the total mistake duration (TT_M), which represents the sum of the durations of all mistakes. For example, this can be observed in the T_M and TT_M results for experiment $E3$, shown in Figure 6.8: as expected, higher average timeouts result in higher T_M and lower TT_M.

As discussed in Section 6.4.1, Chen's FD considers the message loss probability in the configuration of its parameters in order to satisfy the required T_M^U, but it assumes that no consecutive messages are lost. When this assumption fails, the observed T_M may be higher than the specified upper bound. This occurred in the high-delay WAN results of experiment E3 shown in Figure 6.8(a), where the assumption of no consecutive losses was violated twice. In fact, both Chen's FD and *LpfJacHigh* failure detector did not achieve the required T_M^U in this experiment. However, for the *LpfJacHigh* failure detector, this result is a consequence of using the same interrogation periods as *Adaptare-FD*. If smaller interrogation periods were used, it would be possible to secure T_M^U.

Performance limits. *Adaptare-FD* performance is limited, in particular, by the smallest implementable interrogation period. In practice, the limit is imposed by the *Adaptare* framework for the time it takes to recompute the timeout on every new input sample. Since the evaluation presented in Chapter 4 indicates that in a standard Pentium 4 PC the framework takes 1 millisecond on average to perform this job (for a set of 30 samples), the approach seems practically realizable.

Discussion. The results presented in this section illustrate the advantages of using QoS-driven failure detectors such as *Adaptare-FD* and Chen's FD, rather than other simpler adaptive approaches. While QoS-driven solutions automatically adjust their fundamental parameters according to variations in the observed network conditions, the remaining failure detectors require the reconfiguration of operational parameters when they are used in different networks and/or need to satisfy different QoS demands.

Nevertheless, the conclusion to derive from our experimental results is that there is no perfect solution. Choosing between *Adaptare-FD* and Chen's FD depends on the requirements of applications that will use the failure detector service. Comparing the results obtained by these two failure detectors in the evaluated scenarios, we can observe that:

- *Adaptare-FD* computed lower timeouts and higher interrogation periods, which results in less network utilization and best average detection times;
- The conservative interrogation periods produced by Chen's FD may be prohibitive in practice, since they cause a too high network consumption. This situation was observed in experiment $E1$, for the high-delay WAN scenario;
- Due to its higher timeouts, Chen's FD secured the required T_{MR}^{L} and C^{L} in all experiments and environments;
- *Adaptare-FD* did not achieve the required T_{MR}^{L} and C^{L} when the QoS requirements were too high to accommodate the mistakes from environment changes. This was observed in the low-delay WAN environment, for experiment $E3$;
- Due to consecutive losses which are not considered in its operation, Chen's FD did not secure the required T_{M}^{U} in the high-delay WAN environment, for experiment $E3$.

We believe that because of its lower timeouts, *Adaptare-FD* is more suitable to systems with high availability requirements, where failures should be detected as soon as possible. The tradeoff is that more mistakes will be made. Nevertheless, our results shown that even using timeouts much lower than Chen's, there was only one scenario in which *Adaptare* could not deliver the required failure detection accuracy.

6.6 Summary

In this chapter we presented *Adaptare-FD*, a dependability-oriented adaptive failure detector built upon *Adaptare*. In our approach, the QoS for failure detection is specified in terms of minimum required coverage (accuracy parameter) and maximum detection time (speed parameter).

We compared *Adaptare-FD* with the well-known approach of Chen *et al.* (2002), emphasizing the expected impacts of the different configuration approaches on the QoS metrics. We discussed their strengths and weaknesses, contributing to a better understanding of the appropriate uses for both failure detectors.

Our practical evaluation focused in comparing *Adaptare-FD* with Chen's FD and a set of different timeout-based adaptive failure detectors using the same pull-style algorithm for failure detection. Our results highlight the tradeoffs in choosing *Adaptare-FD* over Chen's FD. Moreover, the results from the comparison with other adaptive failure detectors indicate the main advantage of our solution: it is fully adaptive, even when the stochastic behavior of the environment or the QoS requirements change. While *Adaptare-FD* is like a plug-and-play solution, the remaining approaches require some *a priori* configuration, where static parameters are defined according to the expected performance. They are thus unable to dependably adapt to significant changes on the QoS demand or on the network behavior.

Notes

In our first approach for designing an adaptive failure detector using *Adaptare*, the failure detector QoS was specified in a different way. *Adaptare-FD* received as input two accuracy related parameters: the lower bound on the observed coverage C^L and the lower bound on the average mistake recurrence time T^L_{MR}. The interrogation period, fixed for the entire execution, was derived from those values: $\eta = T^L_{MR}(1 - C^L)$. Timeouts were adjusted in runtime according to *Adaptare*'s output. We compared that failure detector with the same timeout-based adaptive failure detectors considered in this chapter, except Chen's FD, since we were focusing on approaches with fixed interrogation periods. The results of this previous work appeared in "Adaptare-FD: A dependability-oriented adaptive failure detector", Dixit and Casimiro, "Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems", Delhi, India, November 2010 (Dixit & Casimiro, 2010).

Despite the promising results presented in that work, we decided to make the definition of liveness requirements more explicit, through T_D^U, and develop an improved version of *Adaptare-FD*, which was presented in this chapter. In fact, we improved the previous results in the following four directions:

- In this new specification, *Adaptare-FD* takes one accuracy-related and one speed-related input parameter, allowing the control of both characteristics of failure detection, while this was not explicit in the previous version.
- In this improved version of *Adaptare-FD* both timeout and interrogation period are adaptive, while in the previous version the period was fixed. Using adaptive periods allows to bound the speed of failure detection according to the requested QoS.
- The evaluation presented in Dixit & Casimiro (2010) revealed that the occurrence of message losses or significant variations in the network conditions compromised the accuracy of *Adaptare-FD*. Therefore, in this new specification *Adaptare-FD* takes into account the occurrence of those phenomena in the computation of the coverage required to *Adaptare*, in order to provide improved accuracy.
- We performed a new set of practical experiments comparing *Adaptare-FD* to Chen's failure detector (Chen *et al.*, 2002), which is the most closely related work in the literature, and was not considered in Dixit & Casimiro (2010).

Currently, we are preparing a journal paper based on the contents of this chapter to be submitted to the IEEE Transactions on Parallel and Distributed Systems.

Chapter 7

Conclusions and Future Research Directions

7.1 Conclusions

This thesis investigated the problem of supporting adaptive systems and applications operating in stochastic environments, from a dependability perspective. The goal was to provide the adequate support to develop dependable adaptive applications whose assumed bounds for temporal variables are adapted in order to ensure that these bounds are secured with a desired and known probability, or coverage.

The model considered in this thesis assumed that the environment behaves stochastically and can be characterized by known probabilistic distributions, which remain stable for sufficiently long periods of time. Furthermore, it was assumed that a system alternates stable periods, during which the environment can be probabilistically characterized, and transition periods, during which a variation of the environment conditions occurs. The proposed approach for dependable adaptation allowed to dynamically determine improved bounds for temporal variables when a stable phase is detected, while it provided conservative but still dependable bounds during transient phases.

The thesis introduced a framework for supporting automatic and dependable adaptation, called *Adaptare*. The implemented framework is composed by two phase detection mechanisms based on statistical tests, which try to characterize the monitored environment using five different probability distributions. However, *Adaptare* is a generic and open framework that may be extended by adding other mechanisms and/or considering additional distributions, depending on the needs.

This thesis is fundamentally concerned with dependability objectives. Therefore, *Adaptare* was developed to take dependability-related criteria (the required coverage of an assumption) as input and provide information to support adaptation (the concrete bound that must be assumed). Moreover, the benefits of the approach were also evaluated from a dependability perspective, measuring the effectively achieved coverage in addition to the improvements in terms of bounds produced by the framework.

The methodology used to evaluate *Adaptare* was based on experiments with synthetic data traces generated from specific distributions, and real data traces available in the Internet, which were collected by different applications, operating in different environments. The experiments with synthetic traces were useful to validate the correctness of the implemented phase detection mechanisms, while the results using real data traces proved that with *Adaptare* it is possible to compute bounds in the order of 10% to 25% lower than the bounds produced by the baseline stochastic (and conservative) approach originally proposed in Casimiro & Verissimo (2001), securing the required coverage in all cases. *Adaptare* was also compared with simpler solutions for adaptation, in order to evaluate the relative performance benefits. The costs incurred by *Adaptare* to provide these benefits were also addressed in the thesis. A complexity analysis and measurements of the execution overhead of *Adaptare* were performed, allowing to conclude that the incurred costs present an acceptable tradeoff for systems and applications that require some level of dependability assurance.

These results are very relevant in practical systems, in the measure that achieving a dependable behavior will become an increasing requirement of autonomous and adaptive applications. Without adequate support, these applications may end up relying on mechanisms that use mere heuristics, sometimes compromising application correctness. In order to illustrate the usefulness of our approach with concrete examples, the thesis presented two cases in which we successfully applied *Adaptare*. The problems of consensus and failure detection were investigated, due to their major importance for the design of distributed and reliable systems.

First, the thesis presented an adaptive version of a randomized consensus algorithm for wireless ad hoc networks, using *Adaptare*. In the studied algorithm the timeout was used to decide whether, and when, a broadcast message must be retransmitted. The original static version of the protocol used a fixed timeout, which had to be set at deployment time, requiring some previous knowledge, or measurements, of the operational environment. In the adaptive consensus this timeout was adjusted in runtime, based on *Adaptare*'s indications.

The concrete performance improvements were measured in terms of the consensus execution time (latency) and network utilization (load). In particular, the results highlighted the impact of a varying number of participating processes on the observed performance. While the static protocol version exhibits significant performance degradation even with a small increase in the number of processes, in the dynamic version the network load generated by each process remains essentially constant, and the execution time only increases linearly, instead of exponentially.

Finally, in the second application studied in this thesis, *Adaptare* was used for failure detection purposes. The thesis introduced *Adaptare-FD*, an adaptive failure detector that was built over *Adaptare*. A novel approach was proposed for configuring the required QoS for the failure detector, in which applications specify the minimum required coverage for failure detection, and an upper bound on the detection time. *Adaptare-FD* follows this specification, taking into account the message loss probability, the environment change probability, and the required QoS in order to adapt timeouts and interrogation periods.

The performed evaluation compared *Adaptare-FD* with an adaptive version of the well-known approach proposed in Chen *et al.* (2002). The obtained results allowed to identify the tradeoffs of using each solution. While *Adaptare-FD* favours the speed of detection and can thus be more appropriate for systems with high availability requirements, Chen's FD tends to use higher timeouts, which favours accuracy metrics. The down sides of both approaches are revealed when the environment is too unstable. With *Adaptare-FD* the achievable coverage becomes more limited, while Chen's FD tends to increase the network load significantly (thus worsening instability) and may not be able to secure the lower limits of the average mistake duration metric. Regarding the comparison with other timeout-based adaptive failure detectors, *Adaptare* has the advantage of automatically adapting to variations in the expected QoS level, while the remaining failure detectors depend on the configuration of operational parameters to be used in different networks and/or for different QoS demands.

These results illustrated the benefits of using more elaborated adaptation solutions, as the approach for dependable adaptation proposed in this thesis, in order to improve system's dependability.

Given the above, we believe that the objectives of the thesis, stated in Chapter 1, were successfully accomplished. In Chapter 3 we defined our set of assumptions, the considered system model, and methodology for dependable adaptation, as required by our first objective. The second objective was met in Chapter 4, with the definition and validation of probabilistic mechanisms for supporting adaptive applications, as well as a complete evaluation of the implemented framework, considering different scenarios and networks. Chapters 5 and 6 presented practical applications of *Adaptare*, reaching our third and final objective. Thus, we conclude that the thesis definitely contributes to supporting automatic adaptation of time-sensitive applications operating in stochastic environments, whilst safeguarding correctness.

7.2 Future research directions

The research work presented in this thesis can certainly be improved and extended in several different ways. Here we present some open issues that could be addressed in the near future.

- In Chapter 4 we analyzed the overhead of *Adaptare*, in comparison with simpler approaches for adaptation. One interesting research direction is to investigate the possibility of using alternative mechanisms from the probability theory, in order to improve *Adaptare*'s overhead in terms of complexity and execution time.

- The approach for dependable adaptation proposed in this thesis, as well as the current implementation of *Adaptare*, are based on the "sufficient stability" and "sufficient activity" assumptions. This means that environment changes are not monitored in order to verify if the required coverage can be secured. This issue was externally handled by *Adaptare-FD*. Despite of the impossibility of predicting the exact instants in which environment changes may occur, it is possible to assume that the environment change probability in a given instant can be estimated from past observed probabilities of change, as it is done in *Adaptare-FD*.

If this probability is known, we can adjust the overall required coverage to circumvent the timing faults generated by these changes, and/or indicate to applications when the required dependability level cannot be secured. We believe this is a useful feature to be added to *Adaptare*.

- Currently, mapping application requirements into *Adaptare*'s parameters (namely, the minimum coverage) is a task performed by the system/application designer. In some cases, this mapping can be quite simple, like it is in the consensus protocol addressed in this thesis. In other cases, deriving the appropriate coverage to meet applications requirements can be extremely challenging. It is possible that the coverage is not directly related to some application requirement but, instead, results from the balance of several dependent factors that have impact on performance. For instance, if the objective is to achieve a good performance in terms of execution time, and if the protocol has an associated recovery time whenever a timeout expires, then there must be a balance between the time the protocol is delayed until it takes some appropriate action (triggered by a timeout, when a fault occurs) and the time it wastes performing useless recovery actions (when no fault occurs and the timeout expires). In this case, the coverage requirement that will lead to an appropriate timeout must be derived from a cost/benefit function that quantifies the value of changing the current coverage/timeout to new values. Such cost/benefit function, defined by the system designer, is essential to express whether changing the coverage/timeout is favorable to the protocol. We believe that addressing this issue would be an important added value to *Adaptare*. In principle, this is an optimization problem: the application would specify its cost/benefit function, and *Adaptare* would search for the most appropriate pair of required coverage and resulting timeout, which maximizes the achievable gains.

- With respect to failure detection, it would be interesting to verify the possibility of using the best out of Chen's approach and *Adaptare-FD* through a configuration method that more explicitly takes into account the message loss probability, and still does not lead to too small interrogation periods.

- Finally, *Adaptare-FD*'s evaluation could be enriched by comparing it with more recent approaches for autonomic failure detection, namely the solution based on the feedback control theory proposed in de Sá & de Araújo Macêdo (2010).

Appendix A

Critical values for the Goodness-of-Fit tests

A.1 Anderson-Darling critical values

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.10	0.599
10	0.05	0.712
10	0.01	0.976
30	0.10	0.615
30	0.05	0.73
30	0.01	1.001
50	0.10	0.619
50	0.05	0.736
50	0.01	1.009

Table A.1: AD critical values for Weibull distribution.

History size h	Significance level α	Critical value $s_{n;\alpha}$
any	0.15	0.922
any	0.10	1.078
any	0.05	1.341
any	0.01	1.957

Table A.2: AD critical values for Exponential and Shifted Exponential distributions.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.15	0.875
10	0.10	1.031
10	0.05	1.311
10	0.01	1.872
30	0.15	0.756
30	0.10	0.862
30	0.05	1.07
30	0.01	1.631
50	0.15	0.62397
50	0.10	0.70945
50	0.05	0.8455
50	0.01	1.91943

Table A.3: AD critical values for Pareto distribution with $\alpha = 0.5$.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.15	0.646
10	0.10	0.726
10	0.05	0.886
10	0.01	1.303
30	0.15	0.665
30	0.10	0.756
30	0.05	0.913
30	0.01	0.1337
50	0.15	0.62397
50	0.10	0.70945
50	0.05	0.8455
50	0.01	1.91943

Table A.4: AD critical values for Pareto distribution with $\alpha = 1.0$.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.15	0.594
10	0.10	0.675
10	0.05	0.808
10	0.01	1.102
30	0.15	0.679
30	0.10	0.777
30	0.05	0.952
30	0.01	1.361
50	0.15	0.62397
50	0.10	0.70945
50	0.05	0.8455
50	0.01	1.91943

Table A.5: AD critical values for Pareto distribution with $\alpha = 1.5$.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.15	0.589
10	0.10	0.655
10	0.05	0.783
10	0.01	1.113
30	0.15	0.665
30	0.10	0.768
30	0.05	0.947
30	0.01	1.368
50	0.15	0.62397
50	0.10	0.70945
50	0.05	0.8455
50	0.01	1.91943

Table A.6: AD critical values for Pareto distribution with $\alpha = 2.0$.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.15	0.588
10	0.10	0.654
10	0.05	0.788
10	0.01	1.1
30	0.15	0.678
30	0.10	0.774
30	0.05	0.96
30	0.01	1.401
50	0.15	0.62397
50	0.10	0.70945
50	0.05	0.8455
50	0.01	1.91943

Table A.7: AD critical values for Pareto distribution with $\alpha = 2.5$.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.15	0.601
10	0.10	0.678
10	0.05	0.805
10	0.01	1.147
30	0.15	0.688
30	0.10	0.776
30	0.05	0.937
30	0.01	1.413
50	0.15	0.62397
50	0.10	0.70945
50	0.05	0.8455
50	0.01	1.91943

Table A.8: AD critical values for Pareto distribution with $\alpha = 3.0$.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.15	0.597
10	0.10	0.677
10	0.05	0.818
10	0.01	1.169
30	0.15	0.708
30	0.10	0.822
30	0.05	0.999
30	0.01	1.5
50	0.15	0.62397
50	0.10	0.70945
50	0.05	0.8455
50	0.01	1.91943

Table A.9: AD critical values for Pareto distribution with $\alpha = 3.5$.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.15	0.61
10	0.10	0.691
10	0.05	0.835
10	0.01	1.2
30	0.15	0.69
30	0.10	0.791
30	0.05	0.99
30	0.01	1.475
50	0.15	0.62397
50	0.10	0.70945
50	0.05	0.8455
50	0.01	1.91943

Table A.10: AD critical values for Pareto distribution with $\alpha = 4.0$.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.25	12.419
10	0.15	16.277
10	0.10	19.518
10	0.05	25.121
10	0.025	30.990
10	0.01	39.083
30	0.25	12.457
30	0.15	16.210
30	0.10	19.313
30	0.05	25.130
30	0.025	31.111
30	0.01	39.673
50	0.25	1.2425
50	0.15	1.6163
50	0.10	1.9277
50	0.05	2.4941
50	0.025	3.0933
50	0.01	3.9200
100	0.25	1.2399
100	0.15	1.6235
100	0.10	1.9325
100	0.05	2.4901
100	0.025	3.0655
100	0.01	3.8319
> 100	0.25	1.2480
> 100	0.15	1.6100
> 100	0.10	1.9330
> 100	0.05	2.4920
> 100	0.025	3.0700
> 100	0.01	3.8800

Table A.11: AD critical values for Uniform distribution.

A.2 Kolmogorov-Smirnov critical values

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.20	0.3226
10	0.10	0.3687
10	0.05	0.4093
10	0.01	0.4889
30	0.20	0.1903
30	0.10	0.2176
30	0.05	0.2417
30	0.01	0.2899
50	0.20	0.149
50	0.10	0.17
50	0.05	0.189
50	0.01	0.226
100	0.20	0.106
100	0.10	0.121
100	0.05	0.134
100	0.01	0.161
140	0.20	0.089
140	0.10	0.102
140	0.05	0.114
140	0.01	0.136
200	0.20	0.075
200	0.10	0.086
200	0.05	0.095
200	0.01	0.114

Table A.12: Standard KS critical values.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.15	0.721
10	0.10	0.759
10	0.05	0.822
10	0.01	0.949
30	0.15	0.745
30	0.10	0.789
30	0.05	0.854
30	0.01	0.98

Table A.13: KS critical values for Weibull distribution.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.15	0.277
10	0.10	0.295
10	0.05	0.325
10	0.01	0.38
30	0.15	0.164
30	0.10	0.174
30	0.05	0.192
30	0.01	0.226

Table A.14: KS critical values for Exponential and Shifted Exponential distributions.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.15	0.268
10	0.10	0.284
10	0.05	0.308
10	0.01	0.348
30	0.15	0.149
30	0.10	0.159
30	0.05	0.173
30	0.01	0.204

Table A.15: KS critical values for Pareto distribution with $\alpha = 0.5$.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.15	0.23
10	0.10	0.241
10	0.05	0.257
10	0.01	0.297
30	0.15	0.138
30	0.10	0.145
30	0.05	0.156
30	0.01	0.18

Table A.16: KS critical values for Pareto distribution with $\alpha = 1.0$.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.15	0.226
10	0.10	0.236
10	0.05	0.254
10	0.01	0.29
30	0.15	0.142
30	0.10	0.149
30	0.05	0.162
30	0.01	0.187

Table A.17: KS critical values for Pareto distribution with $\alpha = 1.5$.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.15	0.228
10	0.10	0.239
10	0.05	0.258
10	0.01	0.3
30	0.15	0.142
30	0.10	0.151
30	0.05	0.165
30	0.01	0.189

Table A.18: KS critical values for Pareto distribution with $\alpha = 2.0$.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.15	0.232
10	0.10	0.245
10	0.05	0.265
10	0.01	0.308
30	0.15	0.145
30	0.10	0.153
30	0.05	0.167
30	0.01	0.196

Table A.19: KS critical values for Pareto distribution with $\alpha = 2.5$.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.15	0.236
10	0.10	0.251
10	0.05	0.272
10	0.01	0.314
30	0.15	0.146
30	0.10	0.155
30	0.05	0.169
30	0.01	0.199

Table A.20: KS critical values for Pareto distribution with $\alpha = 3.0$.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.15	0.239
10	0.10	0.253
10	0.05	0.277
10	0.01	0.322
30	0.15	0.15
30	0.10	0.161
30	0.05	0.175
30	0.01	0.207

Table A.21: KS critical values for Pareto distribution with $\alpha = 3.5$.

History size h	Significance level α	Critical value $s_{n;\alpha}$
10	0.15	0.242
10	0.10	0.258
10	0.05	0.282
10	0.01	0.239
30	0.15	0.149
30	0.10	0.159
30	0.05	0.174
30	0.01	0.207

Table A.22: KS critical values for Pareto distribution with $\alpha = 4.0$.

Appendix B

Quality of Service of Adaptive Failure Detectors

Failure detector	δ (ms)	C (%)	T_{MR} (ms)	T_M (ms)	TT_M (ms)
Adaptare-FD	2.37	99.74	4.08E+06	1.41	31
Chen's FD	1.16	99.20	1.28E+06	1.37	93
LpfJacHigh	10.89	99.97	8.56E+07	1.00	2
WinMeanJacHigh	12.79	99.96	4.28E+07	1.34	4
LpfCibHigh	2.84	99.85	7.14E+06	1.16	15
LpfCibLow	2.00	99.97	3.17E+06	1.39	39
LpfCibMed	2.15	99.69	3.43E+06	1.38	36
WinMeanCibHigh	2.87	99.85	7.17E+06	1.15	15
WinMeanCibLow	2.00	99.66	3.06E+06	1.34	39
WinMeanCibMed	2.14	99.70	3.57E+06	1.40	35
LpfJacLow	1.01	99.05	1.07E+06	1.47	119
LpfJacMed	1.04	99.07	1.09E+06	1.47	116
WinMeanJacLow	1.00	99.04	1.05E+06	1.48	121
WinMeanJacMed	1.04	99.07	1.09E+06	1.49	118

Table B.1: QoS results for experiment $E1$ in a LAN.

Failure detector	δ (ms)	C (%)	T_{MR} (ms)	T_M (ms)	TT_M (ms)
Adaptare-FD	9.39	99.94	8.64E+07	98.00	98
Chen's FD	16.39	100.00	8.64E+07	0.00	0
LpfJacHigh	157.03	99.94	8.64E+07	98.00	98
WinMeanJacHigh	156.41	99.94	8.64E+07	99.00	99
LpfCibHigh	4.00	99.84	3.76E+07	35.00	105
LpfCibLow	2.56	99.67	1.50E+07	18.33	110
LpfCibMed	2.99	99.67	1.50E+07	18.33	110
WinMeanCibHigh	4.03	99.84	3.76E+07	35.00	105
WinMeanCibLow	2.56	99.67	1.50E+07	18.33	110
WinMeanCibMed	3.01	99.67	1.50E+07	18.33	110
LpfJacLow	1.10	99.19	5.37E+06	8.33	125
LpfJacMed	1.22	99.29	6.26E+06	9.46	123
WinMeanJacLow	1.10	99.19	5.37E+06	8.33	125
WinMeanJacMed	1.24	99.24	5.78E+06	8.86	124

Table B.2: QoS results for experiment $E2$ in a LAN.

Failure detector	δ (ms)	C (%)	T_{MR} (ms)	T_M (ms)	TT_M (ms)
Adaptare-FD	18.52	100.00	8.64E+07	0.00	0
Chen's FD	14.76	100.00	8.64E+07	0.00	0
LpfJacHigh	5.35	100.00	8.64E+07	0.00	0
WinMeanJacHigh	5.96	100.00	8.64E+07	0.00	0
LpfCibHigh	2.95	99.84	1.89E+07	1.16	7
LpfCibLow	2.02	99.44	4.73E+06	1.24	26
LpfCibMed	2.26	99.55	5.91E+06	1.23	21
WinMeanCibHigh	2.97	99.87	2.37E+07	1.20	6
WinMeanCibLow	2.01	99.44	4.73E+06	1.24	26
WinMeanCibMed	2.26	99.60	6.76E+06	1.24	19
LpfJacLow	1.02	98.54	1.75E+06	1.47	81
LpfJacMed	1.07	98.61	1.85E+06	1.50	78
WinMeanJacLow	1.01	98.53	1.75E+06	1.47	81
WinMeanJacMed	1.08	98.64	1.89E+06	1.51	77

Table B.3: QoS results for experiment $E3$ in a LAN.

Failure detector	δ (ms)	C (%)	T_{MR} (ms)	T_M (ms)	TT_M (ms)
Adaptare-FD	230.24	99.80	5.36E+06	34.88	593
Chen's FD	9489.76	100.00	8.64E+07	0.00	0
LpfJacHigh	10244.28	100.00	8.64E+07	0.00	0
WinMeanJacHigh	15280.49	100.00	8.64E+07	0.00	0
LpfCibHigh	230.50	99.80	5.36E+06	34.70	590
LpfCibLow	228.87	99.45	1.82E+06	13.33	640
LpfCibMed	229.65	99.67	3.06E+06	21.03	610
WinMeanCibHigh	230.37	99.80	5.36E+06	34.70	590
WinMeanCibLow	228.73	99.43	1.75E+06	12.92	646
WinMeanCibMed	229.53	99.67	3.06E+06	21.03	610
LpfJacLow	227.39	97.67	4.23E+05	4.18	853
LpfJacMed	227.83	98.01	4.96E+05	4.64	808
WinMeanJacLow	227.23	97.64	4.18E+05	4.15	855
WinMeanJacMed	227.58	97.98	4.87E+05	4.61	816

Table B.4: QoS results for experiment $E1$ in a high-delay WAN.

Failure detector	δ (ms)	C (%)	T_{MR} (ms)	T_M (ms)	TT_M (ms)
Adaptare-FD	235.32	99.84	3.76E+07	30.33	91
Chen's FD	37365.22	100.00	8.64E+07	0.00	0
LpfJacHigh	5125.48	100.00	8.64E+07	0.00	0
WinMeanJacHigh	6368.58	100.00	8.64E+07	0.00	0
LpfCibHigh	230.06	99.78	2.51E+07	27.75	111
LpfCibLow	228.87	99.46	8.36E+06	12.00	120
LpfCibMed	229.41	99.52	9.40E+06	13.11	118
WinMeanCibHigh	229.46	99.78	2.51E+07	27.75	111
WinMeanCibLow	228.25	99.46	8.36E+06	12.00	120
WinMeanCibMed	228.80	99.57	1.07E+07	14.62	117
LpfJacLow	227.99	97.64	1.75E+06	3.70	163
LpfJacMed	228.72	97.90	1.98E+06	4.05	158
WinMeanJacLow	227.19	97.47	1.64E+06	3.53	166
WinMeanJacMed	227.53	97.74	1.83E+06	3.81	160

Table B.5: QoS results for experiment $E2$ in a high-delay WAN.

Failure detector	δ (ms)	C (%)	T_{MR} (ms)	T_M (ms)	TT_M (ms)
Adaptare-FD	3454.96	99.96	2.53E+07	22791.21	319077
Chen's FD	21044.46	99.98	4.80E+07	59781.50	119563
LpfJacHigh	48523.84	99.90	3.16E+07	53526.75	214107
WinMeanJacHigh	82683.13	99.78	1.18E+07	25348.22	228134
LpfCibHigh	2005.14	99.49	4.74E+06	14399.71	302394
LpfCibLow	1264.12	98.97	2.31E+06	7937.57	333378
LpfCibMed	1682.26	99.24	3.16E+06	9948.74	308401
WinMeanCibHigh	1917.18	99.46	4.51E+06	13761.45	302752
WinMeanCibLow	1176.16	98.76	1.89E+06	7020.51	358046
WinMeanCibMed	1594.31	99.22	3.06E+06	9818.87	314204
LpfJacLow	633.79	97.00	7.76E+05	3473.39	427228
LpfJacMed	722.38	97.24	8.45E+05	3538.41	399840
WinMeanJacLow	567.36	95.97	5.66E+05	3079.24	508074
WinMeanJacMed	686.63	96.34	6.37E+05	3102.53	465380

Table B.6: QoS results for experiment $E3$ in a high-delay WAN.

Failure detector	δ (ms)	C (%)	T_{MR} (ms)	T_M (ms)	TT_M (ms)
Adaptare-FD	43.47	98.52	6.83E+05	76.42	9706
Chen's FD	1745.62	99.99	8.64E+07	913.00	913
LpfJacHigh	11697.46	100.00	8.64E+07	0.00	0
WinMeanJacHigh	15267.13	100.00	8.64E+07	0.00	0
LpfCibHigh	42.67	98.52	6.83E+05	75.73	9620
LpfCibLow	31.18	97.03	3.39E+05	39.32	10000
LpfCibMed	37.66	98.09	5.28E+05	59.58	9770
WinMeanCibHigh	42.49	98.49	6.67E+05	74.18	9640
WinMeanCibLow	31.03	97.01	3.36E+05	39.17	10100
WinMeanCibMed	37.50	98.08	5.25E+05	59.36	9800
LpfJacLow	20.76	93.32	1.50E+05	18.80	10800
LpfJacMed	22.59	94.25	1.74E+05	21.40	10600
WinMeanJacLow	20.59	93.49	1.54E+05	19.30	10800
WinMeanJacMed	22.28	94.36	1.77E+05	21.76	10600

Table B.7: QoS results for experiment $E1$ in a low-delay WAN.

Failure detector	δ (ms)	C (%)	T_{MR} (ms)	T_M (ms)	TT_M (ms)
Adaptare-FD	46.11	99.16	3.72E+06	30.48	701
Chen's FD	5036.28	99.97	8.64E+07	45.00	45
LpfJacHigh	6582.20	100.00	8.64E+07	0.00	0
WinMeanJacHigh	6291.85	100.00	8.64E+07	0.00	0
LpfCibHigh	29.07	98.87	3.76E+06	33.67	707
LpfCibLow	24.41	97.19	1.48E+06	14.83	771
LpfCibMed	27.04	98.22	2.35E+06	22.12	730
WinMeanCibHigh	28.22	98.76	3.42E+06	30.87	710
WinMeanCibLow	23.57	96.98	1.37E+06	13.86	776
WinMeanCibMed	26.20	98.16	2.28E+06	21.59	734
LpfJacLow	20.47	93.25	6.07E+05	7.54	943
LpfJacMed	21.80	93.69	6.49E+05	7.75	907
WinMeanJacLow	19.43	93.20	6.02E+05	7.47	941
WinMeanJacMed	20.18	93.79	6.60E+05	7.91	910

Table B.8: QoS results for experiment $E2$ in a low-delay WAN.

Failure detector	δ (ms)	C (%)	T_{MR} (ms)	T_M (ms)	TT_M (ms)
Adaptare-FD	1126.96	99.36	9.83E+06	30.29	3514
Chen's FD	6831.27	99.98	8.64E+07	131.00	131
LpfJacHigh	127711.83	100.00	8.64E+07	0.00	0
WinMeanJacHigh	222502.17	100.00	8.64E+07	0.00	0
LpfCibHigh	66.41	95.77	5.72E+05	41.26	6890
LpfCibLow	44.23	94.19	4.16E+05	38.70	8862
LpfCibMed	56.75	95.34	5.19E+05	41.42	7621
WinMeanCibHigh	66.03	95.69	5.62E+05	40.63	6907
WinMeanCibLow	46.86	94.02	4.04E+05	37.78	8917
WinMeanCibMed	56.35	95.18	5.02E+05	40.26	7649
LpfJacLow	24.43	90.45	2.53E+05	31.40	11837
LpfJacMed	28.41	91.71	2.91E+05	34.10	11151
WinMeanJacLow	23.94	90.93	2.66E+05	33.05	11833
WinMeanJacMed	27.64	92.12	3.06E+05	35.77	11123

Table B.9: QoS results for experiment $E3$ in a low-delay WAN.

References

ACHARYA, P., SHARMA, A., BELDING, E., ALMEROTH, K. & PAPAGIANNAKI, K. (2008). Congestion-aware rate adaptation in wireless networks: A measurement-driven approach. In *Proceedings of the 5th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, 1 –9. 81

AGUILERA, M.K., DELPORTE-GALLET, C., FAUCONNIER, H. & TOUEG, S. (2001). Stable leader election. In *Proceedings of the 15th International Conference on Distributed Computing*, 108–122. 80

AGUILERA, M.K., DELPORTE-GALLET, C., FAUCONNIER, H. & TOUEG, S. (2004). Communication-efficient leader election and consensus with limited link synchrony. In *Proceedings of the 23rd Annual ACM symposium on Principles of Distributed Computing*, 328–337. 26

ALLEN, A.O. (1990). *Probability, statistics, and queueing theory with computer science applications*. Academic Press Professional, Inc., San Diego, CA, USA. 37, 54

ANDERSEN, D., BALAKRISHNAN, H., KAASHOEK, F. & MORRIS, R. (2001). Resilient overlay networks. *SIGOPS Operating Systems Review*, 35(5):131–145. 63

AURRECOECHEA, C., CAMPBELL, A.T. & HAUW, L. (1998). A survey of QoS architectures. *Multimedia Systems*, 6(3):138–151. 21, 22

BABAOGLU, Ö., JELASITY, M., MONTRESOR, A., FETZER, C., LEONARDI, S., VAN MOORSEL, A.P.A. & VAN STEEN, M., eds. (2005). *Self-star Properties in Complex Information Systems, Conceptual and Practical Foundations*, vol. 3460 of *Lecture Notes in Computer Science*, Springer. 24

BALAKRISHNAN, N. & BASU, A.P. (1995). *The exponential distribution: Theory, methods and applications*. CRC Press, USA. 53

BERTIER, M., MARIN, O. & SENS, P. (2002). Implementation and performance evaluation of an adaptable failure detector. In *Proceedings of the 32nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 354–363. 14, 28, 101, 106

BHATTI, S.N. & KNIGHT, G. (1999). Enabling QoS adaptation decisions for internet applications. *Computer Networks*, 31(7):669–692. 22

BOLOT, J.C. (1993). Characterizing end-to-end packet delay and loss in the internet. *Journal of High Speed Networks*, 2:305–323. 51

BORRAN, F., PRAKASH, R. & SCHIPER, A. (2008). Extending Paxos/LastVoting with an adequate communication layer for wireless ad hoc networks. In *Proceedings of the 27th IEEE Symposium on Reliable Distributed Systems*, 227–236. 26

BOWERMAN, B.L. & O'CONNEL, R.T. (1993). *Forecasting and Time Series: an Applied Approach*. Duxbury Press, Belmont, CA, USA. 18

CASIMIRO, A. & DIXIT, M. (2011). From static to dynamic protocols: adapting timeouts for improved performance. In *Proceedings of the 1st Workshop on Autonomic Distributed Systems*, 17–20. 98

CASIMIRO, A. & VERISSIMO, P. (2001). Using the Timely Computing Base for dependable QoS adaptation. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, 208–217. vi, vii, 2, 3, 6, 17, 18, 20, 24, 35, 37, 39, 59, 66, 128

CASIMIRO, A., LOLLINI, P., DIXIT, M., BONDAVALLI, A. & VERÍSSIMO, P. (2008). A framework for dependable QoS adaptation in probabilistic environments. In *Proceedings of the 23rd ACM symposium on Applied Computing*, 2192–2196. 45

CHANDRA, T. & TOUEG, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267. 12, 13, 26, 100

CHANDRA, T.D., HADZILACOS, V. & TOUEG, S. (1996). The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722. 13, 25

CHEN, K.T., JIANG, J.W., HUANG, P., CHU, H.H., LEI, C.L. & CHEN, W.C. (2006). Identifying MMORPG bots: a traffic analysis approach. In *Proceedings of the 3rd ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*. 20

CHEN, W., TOUEG, S. & AGUILERA, M.K. (2002). On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(1):13–32. 7, 14, 27, 28, 29, 30, 101, 104, 105, 106, 110, 125, 126, 129

CRISTIAN, F. & FETZER, C. (1999). The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657. 15

DE SÁ, A.S. & DE ARAÚJO MACÊDO, R.J. (2010). QoS self-configuring failure detectors for distributed systems. In *Proceedings of the 10th IFIP International Conference on Distributed Applications and Interoperable Systems*, 126–140. 30, 101, 132

DIXIT, M. & CASIMIRO, A. (2010). Adaptare-FD: A dependability-oriented adaptive failure detector. In *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems*, 141–147. 126

DIXIT, M., CASIMIRO, A., LARANJEIRO, N. & VIEIRA, M. (2008). Using experimental measurements to assess dependable adaptation support mechanisms for timed transactions. In *Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems, with the 27th IEEE Symposium on Reliable Distributed Systems*. 77

DIXIT, M., CASIMIRO, A., LOLLINI, P., BONDAVALLI, A. & VERISSIMO, P. (2009). A probabilistic framework for automatic and dependable adaptation in dynamic environments. Tech Report TR-09-19, Department of Informatics, University of Lisboa. 77

DIXIT, M., MONIZ, H. & CASIMIRO, A. (2010). Timeout adaptive consensus: Improving performance through adaptation. Tech Report TR-2010-06, Department of Informatics, University of Lisboa. 97

DIXIT, M., CASIMIRO, A., LOLLINI, P., BONDAVALLI, A. & VERISSIMO, P. (2011). Adaptare: Supporting automatic and dependable adaptation in dynamic environments. *ACM Transactions on Autonomous and Adaptive Systems*, (to appear). 45, 77

DIXIT, M., MONIZ, H. & CASIMIRO, A. (2012). Timeout-based adaptive consensus: Improving performance through adaptation. In *Proceedings of the 27rd ACM Symposium on Applied Computing (to appear)*. 98

DOWNEY, A.B. (2001). Evidence for long-tailed distributions in the internet. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, 229–241. 51

DWORK, C., LYNCH, N. & STOCKMEYER, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323. 14, 17

ELTETO, T. & MOLNAR, S. (1999). On the distribution of round-trip delays in TCP/IP networks. In *Proceedings of the 24th Annual IEEE Conference on Local Computer Networks*, 172–181. 19, 20

EVANS, J.W., JOHNSON, R.A. & GREEN, D.W. (1989). Two- and three-parameter weibull goodness-of-fit tests. FPL-RP-493, Forest Products Laboratory Research Paper. 52

FALAI, L. & BONDAVALLI, A. (2005). Experimental evaluation of the QoS of failure detectors on wide area network. In *Proceedings of the 35th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 624–633. 14, 28, 30, 72, 101, 106, 107, 118

FETZER, C. (2003). Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computers*, 52(2):99–112. 15

FETZER, C. & CRISTIAN, F. (1995). On the possibility of consensus in asynchronous systems. In *Proceedings of the 1st Pacific Rim International Symposium on Fault-Tolerant Systems*, 86–91. 15

FETZER, C., RAYNAL, M. & TRONEL, F. (2001). An adaptive failure detection protocol. In *Proceedings of the 7th Pacific Rim International Symposium on Dependable Computing*, 146–153. 29, 101

FISCHER, M.J., LYNCH, N.A. & PATERSON, M.S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382. 11, 80, 100

FREILING, F.C. & VÖLZER, H. (2006). Illustrating the impossibility of crash-tolerant consensus in asynchronous systems. *SIGOPS Operating Systems Review*, 40(2):105–109. 26

FRIEDMAN, R., MOSTEFAOUI, A. & RAYNAL, M. (2004). A weakest failure detector-based asynchronous consensus protocol for f < n. *Information Processing Letter*, 90(1):39–46. 14

GAMMA, E., HELM, R., JOHNSON, R. & VLISSIDES, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston MA 02116 USA. 56

GASS, R., SCOTT, J. & DIOT, C. (2005). CRAWDAD trace set cambridge/inmotion/tcp (v. 2005-10-01). http://crawdad.cs.dartmouth.edu/cambridge/inmotion/tcp. 62

GASS, R., SCOTT, J. & DIOT, C. (2006). Measurements of in-motion 802.11 networking. In *Proceedings of the 7th IEEE Workshop on Mobile Computing Systems & Applications*, 69–74. 62

GORENDER, S., MACEDO, R.J.D.A. & RAYNAL, M. (2007). An adaptive programming model for fault-tolerant distributed computing. *IEEE Transactions on Dependable and Secure Computing*, 4(1):18–31. 23

HAYASHIBARA, N., DÉFAGO, X., YARED, R. & KATAYAMA, T. (2004). The F accrual failure detector. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, 66–78. 101

HENDERSON, T., KOTZ, D. & ABYZOV, I. (2004). The changing usage of a mature campus-wide wireless network. In *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*, 187–201. 62

HERNANDEZ, J. & PHILLIPS, I. (2006). Weibull mixture model to characterise end-to-end internet delay at coarse time-scales. *IEE Proceedings Communications*, 153(2):295–304. 19, 35, 51

JACOBSON, V. (1988). Congestion avoidance and control. *SIGCOMM Computers Communication Review*, 18(4):314–329. 28, 71, 72, 107

JAIN, R. (1991). *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, New York, USA. 52

JARDOSH, A.P., RAMACHANDRAN, K.N., ALMEROTH, K.C. & BELDING-ROYER, E.M. (2005). Understanding congestion in IEEE 802.11b wireless networks. In *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement*, 25–25. 81

JIFENG, M.Y., YANG, M., RU, J., RONG, X., HUIMIN, L. & BASHI, C.A. (2004). Predicting internet end-to-end delay: A multiple-model approach. In *Proceedings of 36th IEEE Southeastern Symposium on Systems Theory*, 210–214. 18

KOLIVER, C., NAHRSTEDT, K., FARINES, J.M., FRAGA, J.D.S. & SANDRI, S.A. (2002). Specification, mapping and control for QoS adaptation. *Real-Time Systems*, 23(1):143–174. 23

KOTZ, D., HENDERSON, T. & ABYZOV, I. (2004). CRAWDAD trace set dartmouth/-campus/tcpdump (v. 2004-11-09). http://crawdad.cs.dartmouth.edu/dartmouth/campus/tcpdump. 62

KRISHNAMURTHY, S., SANDERS, W.H. & CUKIER, M. (2001). A dynamic replica selection algorithm for tolerating timing faults. In *Proceedings of the 31st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 107–116. 25

LAMPORT, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169. 26, 80

LAMPORT, L. & LYNCH, N. (1990). Handbook of theoretical computer science (vol. b). chap. Distributed computing: models and methods, 1157–1199, MIT Press, Cambridge, MA, USA. 10

MARKOPOULOU, A., TOBAGI, F.A. & KARAM, M.J. (2006). Loss and delay measurements of internet backbones. *Computer Communications*, 29(10):1590–1604. 19, 51

MENTH, M., MILBRANDT, J. & JUNKER, J. (2006). Time-exponentially weighted moving histograms (TEWMH) for application in adaptive systems. In *Proceedings of the 49th Global Telecommunications Conference*, 1–6. 20

MONIZ, H., NEVES, N.F., CORREIA, M. & VERISSIMO, P. (2009). Randomization can be a healer: consensus with dynamic omission failures. In *Proceedings of the 23rd International Conference on Distributed Computing*, 63–77. 6, 82, 84, 85, 93

MOSTEFAOUI, A. & RAYNAL, M. (2000). Consensus based on failure detectors with a perpetual accuracy property. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, 514–519. 13

MUKHERJEE, A. (1992). On the dynamics and significance of low frequency components of internet load. *Internetworking: Research and Experience*, 5:163–205. 19

NUNES, R.C. & JANSCH-PÔRTO, I. (2002). Modeling communication delays in distributed systems using time series. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, 268–273. 18

NUNES, R.C. & JANSCH-PORTO, I. (2004). QoS of timeout-based self-tuned failure detectors: The effects of the communication delay predictor and the safety margin. In *Proceedings of the 34th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 753–761. 14, 28, 30, 72, 101, 106, 107

PAPAGIANNAKI, K., MOON, S., FRALEIGH, C., THIRAN, P. & DIOT, C. (2003). Measurement and analysis of single-hop delay on an IP backbone network. *IEEE Journal on Selected Areas in Communications. Special Issue on Internet and WWW Measurement, Mapping, and Modeling*, 21(6):908–921. 35, 51

PAXSON, V., PANG, R., ALLMAN, M., BENNETT, M., LEE, J. & TIERNEY, B. (2007). lbl-internal.20041004-1303.port001.dump.anon (package). http://imdc.datcat.org/package/1-507R-8=lbl-internal.20041004-1303.port001.dump.anon. 63

PIRATLA, N., JAYASUMANA, A. & SMITH, H. (2004). Overcoming the effects of correlation in packet delay measurements using inter-packet gaps. In *Proceedings of the 12th IEEE International Conference on Networks*, 233–238. 19, 35

PLANETLAB CONSORTIUM (2004). PlanetLab). Web page: http://www.planet-lab.org. 115

PORTER, I., J.E., COLEMAN, J. & MOORE, A. (1992). Modified KS, AD, and C-vM tests for the Pareto distribution with unknown location and scale parameters. *IEEE Transactions on Reliability*, 41(1):112–117. 52

POWELL, D. (1992). Failure mode assumptions and assumption coverage. In *Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing*, 386–395. 20

RAHMAN, M., PEARSON, L.M. & HEIEN, H.C. (2006). A modified Anderson-Darling test for uniformity. *Bulletin of the Malaysian Mathematical Sciences Society*, 29(1):11–16. 53

RAYNAL, M. (2002). Consensus in synchronous systems: A concise guided tour. In *Proceedings of the 8th Pacific Rim International Symposium on Dependable Computing*, 221–228. 10

RELIASOFT (2006). Using rank regression on Y to calculate the parameters of the Weibull distribution - ReliaSoft Corporation. http://www.weibull.com/LifeDataWeb/estimation_of_the_weibull_parameter.htm. 54

RON, M. (2001). RON - Resilient Overlay Networks. http://nms.csail.mit.edu/ron. 63

SAMPAIO, L. & BRASILEIRO, F. (2005). Adaptive indulgent consensus. In *Proceedings of the 35th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 422–431. 26, 27

SAMPAIO, L., BRASILEIRO, F., CIRNE, W. & FIGUEIREDO, J. (2003). How bad are wrong suspicions? Towards adaptive distributed protocols. In *Proceedings of the 33rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 551–560. 26

SAMPAIO, L.A.V., BRASILEIRO, F., NUNES, R.C. & JANSCH-PÃ´RTO, I. (2005). Efficient and robust adaptive consensus services based on oracles. *Journal of the Brazilian Computer Society*, 10:33–43. 26, 27

SANTORO, N. & WIDMAYER, P. (1989). Time is not a healer. In *Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science*, 304–313. 82

SANTORO, N. & WIDMAYER, P. (2007). Agreement in synchronous networks with ubiquitous faults. *Theoretical Computer Science*, 384(2):232–249. 82

SATZGER, B., PIETZOWSKI, A., TRUMLER, W. & UNGERER, T. (2007). A new adaptive accrual failure detector for dependable distributed systems. In *Proceedings of the 22nd ACM Symposium on Applied Computing*, 551–555. 101

SCHIPER, A. (1997). Early consensus in an asynchronous system with a weak failure detector. *Journal of Distributed Computing*, 10(3):149–157. 13

SERGENT, N., DÉFAGO, X. & SCHIPER, A. (2001). Impact of a failure detection mechanism on the performance of consensus. In *Proceedings of the 7th Pacific Rim International Symposium on Dependable Computing*, 137–145. 100

STEPHENS, M.A. (1974). Edf statistics for goodness of fit and some comparisons. *Journal of the American Statistical Association*, 69(347):730–737. 52

STEPHENS, M.A. (1976). Asymptotic results for goodness-of-fit statistics with unknown parameters. *Annals of Statistics*, 4:357–369. 52

TICKOO, O. & SIKDAR, B. (2004). Queueing analysis and delay mitigation in IEEE 802.11 random access MAC based wireless networks. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 2, 1404–1413. 51

TRIVEDI, K.S. (2002). *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley and Sons, New York, USA. 50, 52, 53

TZAGKARAKIS, G., PAPADOPOULI, M. & TSAKALIDES, P. (2009). Trend forecasting based on singular spectrum analysis of traffic workload in a large-scale wireless LAN. *Performance Evaluation*, 66(3):173–190. 24

UMASS TRACE REPOSITORY (2006). UPRM wireless traces. http://traces.cs.umass.edu/index.php/Network. 62

VERISSIMO, P. (2006). Travelling through wormholes: a new look at distributed systems models. *SIGACT News*, 37(1):66–81. 16

VERISSIMO, P. & CASIMIRO, A. (2002). The Timely Computing Base model and architecture. *Transactions on Computers - Special Issue on Asynchronous Real-Time Systems*, 51(8):916–930. vi, 2, 34

VERISSIMO, P. & RODRIGUES, L. (2001). *Distributed Systems for System Architects*. Kluwer Academic Publishers, Norwell, MA, USA. 10

VERISSIMO, P., CASIMIRO, A. & FETZER, C. (2000). The Timely Computing Base: Timely actions in the presence of uncertain timeliness. In *Proceedings of the 30th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 533–542. 16

WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C. & JOGLEKAR, A. (2002). An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 255–270, Boston, MA, USA. 84, 93

YANG, M., LI, X.R., CHEN, H. & RAO, N.S.V. (2004). Predicting internet end-to-end delay: an overview. In *Proceedings of the 36th IEEE Southeastern Symposium on Systems Theory*, 210–214. 18

ZHANG, W. & HE, J. (2007). Statistical modeling and correlation analysis of end-to-end delay in wide area networks. In *Proceedings of the 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, vol. 3, 968–973. 19