HABILITATION À DIRIGER DES RECHERCHES

présentée devant

L'Université de Rennes 1
Institut des Sciences et Technologies
de l'Information et de la Communication (ISTIC)

par

François TAÏANI

Some Contributions to The Programming of Large-Scale Distributed Systems: Mechanisms, Abstractions, and Tools

soutenue le 17 Novembre 2011
devant le jury composé de :

Mme. Anne-Marie Kermarrec	Présidente
Mme. Laurence Duchien	Rapporteur
M. Pascal Felber	Rapporteur
M. Rick Schlichting	Rapporteur
M. Jean-Charles Fabre	Examinateur
M. Hugues Fauconnier	Examinateur
M. Rachid Guerraoui	Examinateur

À mes trois étoiles, qui se reconnaîtront

Acknowledgement

The work presented in this document would not have been possible without the support, friendship, and encouragement of many people, who are too many to name all. I would nevertheless like to thank Anne-Marie Kermarrec from IRISA/INRIA Rennes for her kind encouragement to engage in this task, for her hospitality, and for her much appreciated support and advice during my stay in her research group. I am also deeply indebted to Jean-Charles Fabre from LAAS/CNRS who first introduced me to middleware research, fault-tolerance, and software architectures, and whose ongoing guidance, perspicacity, and friendship have to a large extent shaped my approach to research. I am also very grateful to Michel Raynal, from IRISA, for our many discussions, and for his kind advice regarding the work presented in Chapter 3. Rick Schlichting and Matti Hiltunen from AT&T Labs Research deserve special thanks for their helpfulness, warm reception, and ongoing interest in my work. I would also like to thank Rick for accepting to report on this work. My Lancaster colleagues, Gordon Blair, and Geoff Coulson, have been a continuous source of inspiration, intellectual reflection (no pun intended), and encouragement: May they be thanked for their precious help.

I would like to thank warmly Laurence Duchien, from INRIA Lille Nord Europe and the University of Lille 1, and Pascal Felber, from the University of Neuchâtel, from accepting to serve as *rapporteurs* on my jury together with Rick Schlichting. I am also indebted to Rachid Guerraoui, from the École Polytechnique de Lausanne, and Hugues Fauconnier, from Université Paris 7 Denis-Diderot and LIAFA/CNRS, for kindly agreeing to serve as examiners with Jean-Charles Fabre on my defence panel.

I would like to express my gratitude to the Ph.D. students I have been privileged to work with at Lancaster. Barry Porter, Nathan Weston, Shen Lin and Rachel Burrows have all left their mark on my route, both professionally and personally.

I also wish to extend my heartfelt gratitude to the School of Computing and Communications of Lancaster for providing such a thriving and collegial academic environment. Special thanks are also due to Cécile Bouton, of INRIA, for her organisation skills, her continuing helpfulness, good humour, and patience.

Finally, this work would not exist without the love and every-day magic of Anne, Zoé and Sarah. To them goes my endless gratitude and love for being who they are.

Contents

Contents **7**

1 Foreword **11**

2 Introduction **17**
2.1 Large-scale decentralised computing 18
2.2 Components and component frameworks 19
2.3 Complexity in modern distributed software 20
2.4 Programming complex distributed platforms 21
2.5 Organisation of the document 22

3 Generic Repair in Peer-to-Peer Overlays **23**
3.1 The need for generic repair in overlays 23
3.2 Architecture of ECHO/SONAR 26
3.2.1 Main intuition 27
3.2.2 Repair mechanisms: key phases 27
3.3 ECHO : convergent agreement 28
3.3.1 The challenge: self-defining constituencies 29
3.3.2 System model and assumptions 30
3.3.3 Specification 31
3.3.4 Failure detector, multicast, region ranking 32
3.3.5 Algorithm 33
3.3.6 Proof of correctness 35
3.4 SONAR : Generic repair 39
3.4.1 Ongoing repairs and coordination in SONAR 40

3.4.2 Example repair strategies . . . 41
3.5 Evaluation . . . 42
3.5.1 Failure free runs . . . 42
3.5.2 On-going failures . . . 43
3.5.3 The impact of imperfect failure detection . . . 43
3.5.4 TBCP case study and PlatnetLab deployment . . . 44
3.6 Conclusion . . . 45

4 Programming Gossip Protocols: GossipKit and Whispers 47
4.1 Structure and behaviour in gossip protocols . . . 48
4.1.1 Gossip protocols . . . 48
4.1.2 Component frameworks . . . 49
4.1.3 High-level distributed programming . . . 49
4.1.4 Transparent componentisation: WhispersKit . . . 50
4.2 The GossipKit component framework . . . 51
4.2.1 GossipKit's common interaction pattern . . . 52
4.2.2 Rich and uniform event interactions . . . 53
4.2.3 Evaluation . . . 54
4.3 Transparent componentisation . . . 56
4.3.1 The Whispers language . . . 56
4.3.2 Synthesis and deployment . . . 58
4.3.3 Evaluation . . . 60
4.4 Conclusion . . . 63

5 Anomaly Diagnosis and Understanding in Complex Systems 65
5.1 Complexity and performance: Globus . . . 67
5.1.1 Between black and grey: an hybrid approach . . . 68
5.1.2 Results and analysis . . . 72
5.2 Structural contraction in performance graphs . . . 76
5.2.1 Approach . . . 77
5.2.2 Evaluation . . . 81
5.3 Conclusion . . . 85

6 Conclusion and outlook 87
6.1 Developing better distributed mechanisms . . . 87
6.2 Software abstractions for distribution . . . 88
6.3 Abstraction, transparency, and understanding . . . 89
6.4 Outlook . . . 90

List of Figures 93

CONTENTS

List of Tables **94**

Bibliography **95**

CONTENTS

CHAPTER 1

Foreword

This document summarises part of my research over the last six years. It focuses on the programmability of complex distributed systems, i.e. on approaches and tools to help developers design, develop and analyse large-scale and composite networked applications. The seeds of this research were planted ten years ago during my doctoral studies at LAAS-CNRS, in Toulouse, France. The works I present cover my postdoctoral stay at AT&T Labs, in Florham-Park, New Jersey, in 2004, and the following years at Lancaster University, in the United Kingdom, from 2005 to 2010.

Although the period this document covers has been incredibly rich, it would not have occurred without a spell in the German subsidiary of ILOG, an INRIA spin-off company (now a part of IBM) in the late nineties. ILOG would only recruit doctors in its development teams. This prompted my career turn. The bankruptcy of my initial Ph.D. sponsor, and a lucky encounter with Jean-Charles Fabre finally set me off into the fascinating field of distributed computer systems.

Since then, I have sought to better understand, from a developer's point of view, some of the challenges inherent to the development of modern distributed applications. Over the last six years since I left LAAS, this part of my research has been organised around two primary axes (Figure 1.1 on the following page):

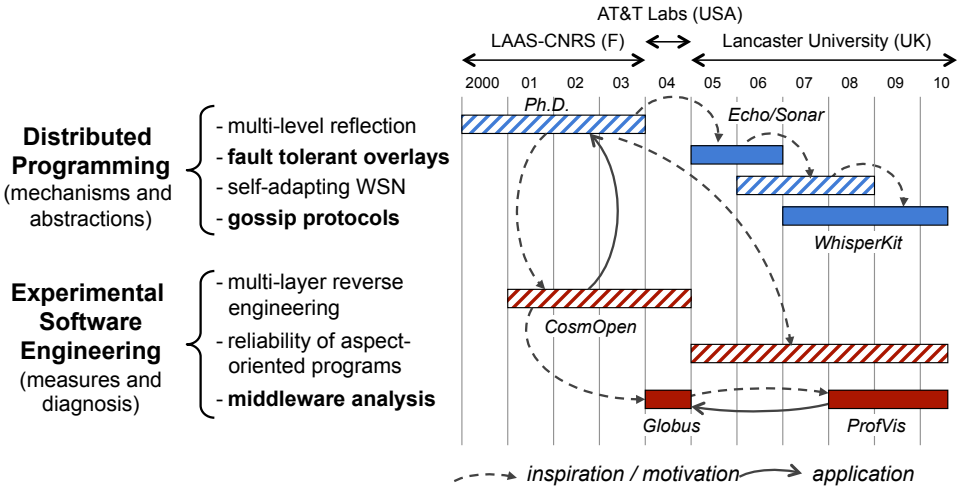


Figure 1.1: Research topics and their relationships over the period 2000-2010. Topics in bold (solid bars) are those presented in this document.

- **Axis 1: Distributed Programming.**
  In this first line of research, I have sought to identify reusable mechanisms and abstractions in large-scale decentralised systems, with a focus on peer-to-peer overlay networks, Wireless Sensor Networks (WSNs), and gossip protocols.
- **Axis 2: Experimental Software Engineering**
  In parallel, I have continued the work started in my Ph.D. on modern programming techniques, and on their effects on contemporary middleware. This line of research contains two strands:
  - I have pursued my analysis of industry-grade middleware to explore the challenges arising from high levels of reuse in modern distributed systems, in particular in terms of performance, analysis and comprehension.
  - As a continuation of my Ph.D. work on reflective architectures, I have also investigated the reliability of aspect-oriented programs, a technology closely related to reflection, first by looking at analysis approaches to detect undesirable aspect interactions, and then by developing measures to better assess the fragility of aspect-oriented mechanisms.

Although I only develop some of these topics in this document (marked in bold in Figure 1.1), these two axes have been closely related in my research, and, can be understood as representing different facets of the same problem:

Foreword

As distributed computer systems grow in size, complexity, and dependencies, they require higher levels of organisation and abstraction, prompting my research on reusable mechanisms and abstractions for distributed programming (first axis in Figure 1.1). Ultimately this line of research seeks to promote composite architectures made of a large number of reusable third-party constituents[1].

Reusable programming abstractions are not, however, without problems, which motivated my work in experimental software engineering (second axis in Figure 1.1): First, because reusable abstractions seek to make their implementation transparent, they make it harder for developers to analyse non-functional emergent properties, a particularly problematic situation in distributed systems. Second, advanced composition mechanisms (e.g. aspects and reflective architectures) that have been proposed to encapsulate reusable entities (in particular in distributed middleware and systems) can also make a system's structure more difficult to analyse and develop. I have used my research in these areas to better understand the implication of high reuse in modern middleware, and thus inform my work on abstractions.

In the following I provide a rapid overview of my research trajectory along these two research axes, before moving on to Chapter 2, which introduces the contributions included in this document.

In my Ph.D. [Taïani (2004)], performed at LAAS-CNRS in Toulouse between 2000 and 2004, I investigated the use of reflection to implement fault-tolerance mechanisms in complex multi-layer systems. Reflection is the ability of a computing system to act upon itself as part of its own computation [Smith (1984); Maes (1987)]. Reflection is primarily an architectural paradigm that guides a system's organisation into a set of principled interfaces and components, and seeks to promote modularity, composability, and reuse. As such, it is particularly well adapted to crosscutting non-functional behaviours, such as fault tolerance. My Ph.D. work focused more precisely on distributed systems made of numerous software components

[1] As often in Computer Science, *component* is an overloaded term. In in its broadest acceptation, *component* may refer to an independent piece of software (as in *Commercial-Off-the-Shelf* components, or COTS), usually developed by a third party for reuse into larger systems (e.g. an OS, a DBMS, a graphical library, a middleware). *Component* may however also refer to a software module with clearly defined dependencies [Szyperski (2002)], that usually follows some predefined binary format and is readily deployable within a component engine (e.g. as in *Component-Based Software-Engineering* (CBSE)). To avoid any confusion, we will reserve the word *component* for this latter meaning in this document, and will use interchangeably *constituent* and *element* to refer more generally to the reusable parts of a software system (which might in turn present themselves as components, or not).

(OS, libraries, virtual machines, middleware, etc.), often independently developed by third parties and organised in interdependent abstraction layers. I showed how the use of runtime information obtained from different layers of a complex architecture could help develop more efficient replication mechanisms. This work led me to highlight the importance of understanding and leveraging cross-layer interactions in complex platforms [Taïani et al. (2002, 2003); Taïani et al. (2005a)], and to propose tools to support the analysis of these interactions in large (more than 100,000 lines of code) and multi-level software [Taïani (2003); Taïani et al. (2009)].

After my Ph.D. I was privileged to join Rick Schlichting's research group at AT&T Labs on an INRIA International Post-Doctoral Scholarship. I used this visit to start considering larger-scale distributed platforms, specifically in the field of grid computing. In particular, I analysed the impact of the introduction Web Services in the grid middleware Globus [Taïani et al. (2005a)]. The version I considered was the first to feature a Service Oriented Architecture (SOA) based on Web Services. Because of the speed of this architectural transformation (a few months), Globus provided a very good case study for the potential limitation of current reuse practices in production middleware.

Since I joined Lancaster University in 2005, I have started to work on reusable abstractions and mechanisms for large-scale composite systems. This work has accompanied the concomitant evolution of distributed systems, and focused on large-scale fully decentralised systems such as peer-to-peer overlays [Porter et al. (2008, 2006b,a)], gossip protocols [Lin et al. (2011, 2007, 2009)], and wireless sensor networks [Grace et al. (2008); Greenwood et al. (2006); Hughes et al. (2006); Porter et al. (2010b,a)]. As part of this research, I have contributed to the design of the OpenCom V2 component engine [Coulson et al. (2008)], developed at Lancaster. OpenCom is marked by both its minimalism and reflective properties, allowing for a high versatility, both in terms of target platforms (Java, Linux, WSN nodes), and reconfiguration capability. The design of OpenCom has in turn influenced my own work, in particular on gossip protocols.

In parallel to this work on large-scale systems, I have further explored the research questions opened during my stay at AT&T on the analysis of middleware assembled from a large set of third-party constituents. This has led to a collaboration with the Psychology Department of Lancaster University, and the development of a novel analysis tool (ProfVis) to help developers navigate the performance traces of unfamiliar composite software.

Finally, I have studied how advanced composition mechanism such as aspect-orientation could be made less error-prone. Aspect-orientation [Kiczales

et al. (1997)] is a programming paradigm closely linked to computational reflection that allows developers to encapsulate crosscutting concerns (such a security, fault-tolerance, logging, caching) in well-identified and cohesive modules. Aspect-orientation comes however with its own challenges. I have shown, with colleagues from AT&T, how aspects are in fact closely related to event-based programming [Hiltunen et al. (2006)], and create similar difficulties. As part of my research, I have contributed to novel approaches to detect inter-aspect interactions [Weston et al. (2007, 2005)], and develop new software metrics to better understand the conditions that might lead to faults in aspect-oriented programs [Burrows et al. (2011, 2010a,b)].

The remainder of this document covers my work on reusable mechanisms and abstractions in overlays and gossip-based systems, as well as the performance analysis of complex composite middleware. Although my research on the OpenCom platform and on aspects played an important role in informing the works presented here, I have left it out of scope for brevity's sake.

Last, but not least, all the works presented here are the result of close collaborations with colleagues and students. I mention at the start of each chapter the publications which the chapter summarises, and the collaborators I worked with. All errors and lacks are however mine.

Chapter 1.

CHAPTER 2

Introduction

Large-scale distributed computer systems are progressively pervading most of our activities, from cloud services, and social and entertainment networks (Facebook, Twitter, the PlayStation Network), through to smart infrastructures for energy and water distribution. In contrast to earlier distributed systems, these systems are *large-scale*, typically involving millions of users and tens of thousands of nodes; they are *heterogeneous*, often combining large data centres, mobile appliances, and sensors, often managed by different organisations; and they are *composite*, integrating an increasing number of enabling technologies and services provided by third parties.

This explosion of size, diversity, and complexity has been accompanied by novel approaches in both *distributed computing*, and *software engineering*, two fields that are central to the works presented in this document. In distributed computing, new forms of distribution have emerged to accommodate the dynamicity and extra large scale of these new systems. These new forms of distribution include decentralised and self-adapting technologies, such as peer-to-peer systems, overlays, gossip protocols, and autonomous architectures. In software engineering, this has been mirrored by new forms of programming and composition, such as reflection, components, and aspect-oriented programming. These novel software engineering techniques allow developers to construct large and rich distributed systems by assembling reusable software elements provided by third parties. They thus facilitate the rise of *composite architectures* that integrate a large number of *reusable entities*, such as libraries, middleware, OSs, and standards.

Reusability, and its corollary *composite architectures*, form the main guiding thread that link together the works presented in this document. The first two following chapters (Chapter 3 on fault-tolerant overlays, and Chapter 4 on the programming of gossip protocols) seek to expose reusable mechanisms and abstractions in decentralised distributed technologies. Chapter 5 then considers the challenges posed by *composite architectures* in today's middleware platforms, with a focus on performance analysis and program understanding.

These three chapters cover a broad spectrum of topics. In the following, we briefly discuss the developments in distributed computing and software engineering that motivated each of these works, and conclude with an overview of the contributions we present in the remaining chapters.

2.1 Large-scale decentralised computing

Emerging distributed systems are increasingly large, multi-faceted, and composite. Social networking applications, for instance, illustrate how different providers can create an ecosystem of on-line services that build on each other to provide sophisticated functionalities to a very large number of users: companies such as twitter, Facebook, and Google now routinely provide advanced on-line Application Programming Interfaces (APIs) to exploit their services, providing a rich feeding ground for innovative start-ups such as foursquare or TweetDeck (now part of Twitter). The whole business model is in turn enabled by the cheap availability of elastic on-demand computing resources offered by cloud computing providers.

These services need to be highly scalable and evolvable, two goals that are difficult to attain in tightly integrated architectures. Among candidate technologies, *peer-to-peer overlays* [Milojicic et al. (2003); Doval and O'Mahony (2003)], and *gossip protocols* [Agrawal et al. (1997); Birman et al. (1999)], have emerged as particularly promising choices to address this challenge. Overlay networks provide application-level networking abilities that are not available or difficult to implement at OS or network level, such as streaming, multicast, or group communications. Gossip protocols, which can be use to implement overlay networks, take their inspiration from the spread of epidemics in large populations to provide a broad range of lightweight and highly resilient distributed mechanisms, from failure detection, and multicast, to clustering.

Both overlay networks and gossip protocols have attracted a considerable amount of attention over the last decade. Programming both kind of systems remains, however, as much an art as a science. Programming frameworks for

overlays, and large-scale distributed systems [Gong (2001); Babaoglu et al. (2002); Urbán et al. (2002); Li et al. (2004); Grace et al. (2005); Rodriguez et al. (2004); Leonini et al. (2008)] are often limited to functional concerns or low-level interactions, and rarely provide non-functional mechanisms that are both high-level and reusable, such as fault-tolerance. Similarly, gossip protocols lack highly reusable programming frameworks, in spite of early works in this direction [Jelasity and Babaoglu (2005); Kermarrec and van Steen (2007); Eugster et al. (2007)].

This lack of generic fault-tolerance for overlay networks has motivated our own research in the field, which we present in Chapter 3. Similarly, the lack of concrete programming frameworks for gossip protocols has led us to propose our WhispersKit technology, which combines component frameworks and macro-programming languages for gossip systems, which we present in Chapter 4.

2.2 Components and component frameworks

The solutions we present in Chapters 3 and 4 are strongly informed by early modular approaches to protocol design and implementations [Hiltunen and Schlichting (2000); van Renesse et al. (1998a); Bhatti et al. (1998); van Renesse et al. (1996)], and by Component-Based Software Engineering (CBSE). Components can be seen as an evolution of object orientation that seeks to avoid the hidden dependencies found in object-oriented systems [Szyperski (2002)]. A component explicitly exposes both provided and required interfaces, which greatly facilitates reconfiguration and system analysis. Components have been successfully applied both in the industry (c.f. EJB, CORBA Component Model, DCOM), and in middleware research, giving rise to lightweight component technologies [Coulson et al. (2004); Bruneton et al. (2006)], and their associated middleware frameworks [Grace et al. (2004); McKinley et al. (2001); Seinturier et al. (2011)].

Identifying reusable modular artefacts suitable for componentisation is however challenging for at least two reasons: First, generalising a set of mechanisms (such as repair approaches in overlays, or interaction patterns in gossip protocols) usually requires reaching a high-enough level of abstraction where commonalities become apparent. This search for commonalities is a theme we return to in Chapter 3 (in the context of fault-tolerance in overlay networks) and Chapter 4 (for gossip protocols). Second, finding an appropriate partitioning among emerging concepts demands that one walks a fine line between simplicity (avoiding too many components and interactions), reusability (avoiding too many specific implementations), and

flexibility (favouring loose and robust interactions). Chapter 3 considers distributed fault-tolerant interactions with a scalable consensus mechanism, while Chapter 4 touches on all three themes within the same component framework.

2.3 Complexity in modern distributed software

Component-based software engineering, and more generally third-party software parts, encourage reuse, and thus allow software developers to easily integrate a large number of concerns and technologies in their systems. This is particularly beneficial in large-scale distributed systems, which must face high levels of heterogeneity, must remain interoperable with other systems, and must cater for a large number of non-functional concerns (e.g. monitoring, maintenance, billing, security, fault-tolerance, scalability.)

This high level of reuse and the large number of entities involved can lead however to more complexity [Brooks (1987)], which is in many ways inherent to software and distribution. Like today's distributed systems, this complexity typically takes many forms (see for instance [Ranganathan and Campbell (2007)]). In my work, I have considered two examples of complexity, one involving component frameworks, and the other performance analysis in unfamiliar software. In both cases, I have tried to alleviate the cognitive complexity faced by developers, i.e. the difficulty for developers to make sense of the abstractions and systems they must deal with.

Because component frameworks tend to focus on structure rather than behaviour, they are limited in their ability to represent a system's detailed execution [Clements (1996)]. This in turn can make it difficult for developers who are unfamiliar with a particular framework to understand its logic [Edwards et al. (2004)]. These limitations explain in part the success of alternative high-level programming techniques for distributed systems, such as specification languages [Eijk and Diaz (1989); Amer and Çeçeli (1990)], Domain Specific Languages [Killian et al. (2007)], and macro-programming approaches [Newton et al. (2007); Madden et al. (2005); Gummadi et al. (2005)]. Macro-programming in particular seeks to present a distributed system (typically a wireless sensor network) as a single programmable entity, hiding distribution under a shared-memory or database metaphor. In Chapter 4, we argue that components and macro-programming are in fact complementary, delivering different benefits to different levels of the architecture, and propose an approach, *transparent componentisation*, that combines both technologies in the context of gossip protocols.

A second challenge pertains to the non-functional properties of reusable software entities, and of the composite systems they contribute to. Whereas the functions of components and libraries are usually reasonably well documented, their non-functional characteristics (performance, robustness, reliability) are much harder to gauge [Arlat et al. (2000)]. One reason is because these characteristics tend to be context dependent. They vary depending on the operational environment (workload, co-existing systems, underlying layers) a software element is exposed to. Another reason is linked to the increasing number of constituents and technologies found in modern distributed systems, multiplying the possible combinations and interactions they can give rise to. Analysing the non-functional characteristics of a large composite system thus becomes extremely challenging: Developers must analyse fine-grained interactions between libraries they have not developed, at layers they have not designed (or might even be unaware of) in implementations that might be undocumented and prone to change. We look at these challenges in Chapter 5, in the context of the Web Service core of the Grid middleware Globus, and present a novel abstraction-driven performance analysis tool to alleviate this problem.

2.4 Programming complex distributed platforms

In summary, the research presented in this document covers a broad spectrum of contributions, from fault-tolerant mechanisms (Chapter 3) and distributed programming (Chapter 4), through to performance analysis (Chapter 5). These contributions are however unified by the two goals they seek to advance: to better construct modern distributed systems, and to understand better the properties of the resulting software.

Better constructing advanced distributed systems. This document generally looks at how new forms of distribution can be modularised into generic and reusable entities. In the continuation of my Ph.D. work at LAAS-CNRS [Taïani et al. (2005a); Taïani et al. (2003, 2002)], Chapter 3 proposes a generic and reusable repair mechanism for overlay networks, and in particular a scalable coordination protocol to organise repair in large-scale overlays. Following this strategy of modularisation, Chapter 4 moves to epidemic protocols, and discusses the WhispersKit platform, a hybrid approach to the realisation of epidemic mechanisms that leverages both the power of components (GossipKit) and that of higher-level programming languages (Whispers).

Understanding high-reuse industrial middleware. The approaches we advocate to construct fault tolerance overlays and epidemic protocols generally seek to yield high-reuse architectures. We argue that, unfortunately, as reuse grows, so does complexity. With more reuse, today's developers have to integrate, expand, and correct software constituents that they have for the most part not developed. This growing reliance on "unfamiliar software" makes it much harder for developers to analyse and understand the systems they produce, in particular in terms of their non-functional characteristics. In my Ph.D., I have explored the problem of dynamic reverse engineering in multi-layer architectures [Taïani et al. (2009); Taïani (2003)] to implement generic replication mechanisms. Chapter 5 extends this line of research to quantitative properties, and looks at the difficulty of analysing the performance of a complex middleware platform (the WS-* core of Globus). Chapter 5 then presents a novel navigation approach of performance traces (ProfVis) to help developers negotiate the uncharted behaviours of unfamiliar software.

2.5 Organisation of the document

The remainder of this document is organised in four chapters. Chapter 3 presents Echo, a scalable and generic agreement protocol to coordinate repair actions in large-scale overlays, and Sonar, a generic repair for overlay protocols based on Echo. Chapter 4 describes WhispersKit, the combination of a component-based platform (GossipKit) and an associated high-level domain specific language (Whispers) for the modular development of epidemic algorithms. Chapter 5 moves on to the dynamic analysis of complex software in general, with a particular focus on middleware, first in the context of the Globus Grid Computing middleware, and then by proposing a novel navigation paradigm to explore dynamic performance data. Finally Chapter 6 concludes, and offers some long-term perspectives on the works presented in this document.

CHAPTER 3

Generic Repair in Peer-to-Peer Overlays

Overlays networks provide application-level networking capabilities that go beyond that typically offered by the underlying network (e.g. TCP/IP), for instance for large-scale broadcasting, multi-media streaming, or publish-subscribe services. Because overlays are designed to operate in large-scale and potentially failure-prone networks, they often include advanced self-repair mechanisms to maintain their structural integrity in the case of failures. In spite of commonalities, each self-repair mechanism is specialised to meet the specific needs of a particular overlay, and reuse and cross-pollination between systems have so far been difficult. This chapter summarises the approach I have developed, together with my colleagues Geoff Coulson and Barry Porter, to overcome this difficulty and to factor out a generic repair mechanism that is reusable and adaptable across a wide range of overlays. This chapter is based on two original publications, which can be found in [Porter et al. (2006b,a, 2008)]. An in-depth and detailed description of the same work is also available in [Porter (2007)].

3.1 The need for generic repair in overlays

Overlay networks [Doval and O'Mahony (2003)] are application-level services that offer specialised virtual network topologies (e.g. trees or rings), or application-specific functions (e.g. application-level multicast, DHT, ad-hoc routing) which are outside the scope of the underlying network. Their use

is increasingly common and the set of overlay proposed types is particularly diverse [Rowstron and Druschel (2001); Castro et al. (2002); Zhao et al. (2001); Chawathe et al. (2000); Clarke et al. (2001a)].

Most overlay networks provide some mechanism for *self-repair* so that the loss of nodes does not unduly affect overlay services. Such repair mechanisms are essential, as overlays typically operate in hostile environments in which nodes run on unstable machines that are subject to crash or to be switched off. One well-known example of a repair mechanism is that adopted by the Chord distributed hash-table (DHT) overlay [Stoica et al. (2001)] which redundantly stores data on multiple nodes, and ensures that requests for data on lost nodes are redirected to nodes holding replicas. As another example, the Overcast content-dissemination overlay [Jannotti et al. (2000)] maintains its structure in the face of failure using ancestor lists, which are used to locate and attach to a surviving ancestor when a parent's node fails.

Although these various repair strategies work, there are three main problems with this ad-hoc, per-overlay approach to repair.

Low reusability. The lack of generic, reusable repair mechanism causes overlay designers to repeatedly solve the same fundamental problems, and limits reuse across platforms and prototypes. This is particularly true in the DHT field: many DHTs adopt a similar approach to that of Chord (e.g. [Rowstron and Druschel (2001); Zhao et al. (2001); Ratnasamy et al. (2000)]), but modified to fit the precise operation of the overlay (such as CAN's n-dimensional coordinate space [Ratnasamy et al. (2000)]).

Tangling of concerns. Repair protocols are typically embedded in overlay's functional behaviour, with little separation of concerns. This lack of modularity makes overlays harder to design than they otherwise would.

Lack of flexibility. Because repair is just one of a much wider set of concerns, the repair approach adopted by overlay designers is often suboptimal or not as flexible as might be desired. For example, one approach to repairing the tree-based overlay of Figure 3.1-a on the facing page would be to restore copies of nodes 28, 29 and 68 on other hosts. But if such hosts cannot be found, an alternative approach would be to repair the tree without actually restoring the failed nodes (Figure 3.1-b and -c). As further examples, it may be useful in some installations to increase or decrease the degree of redundancy in a DHT by deciding whether or not to restore failed nodes; or, in Gnutella [Yang and Garcia-Molina (2003)], to proactively migrate the leaf nodes of a failed super-node to another super-node.

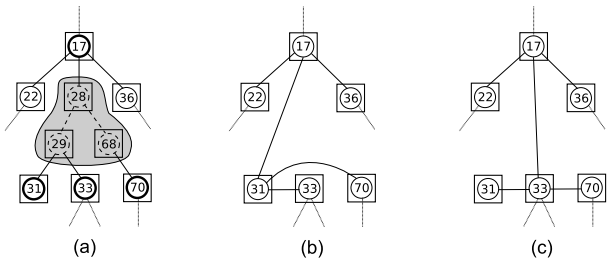


Figure 3.1: (a) An overlay with a failed region (nodes 28, 29 and 68); and (b) and (c), possible repairs of this failure. Squares are squares, and circles overlay nodes. (from [Porter et al. (2006b)])

The key barrier to addressing the above three problems, and achieving generic overlay repair is the imprecise and dynamic nature of the environment in which any repair mechanism must operate. In such an environment failures may stay undetected for a long time, nodes may hold inconsistent views of which other nodes have failed, and concurrent repair activities might conflict. Traditionally such problems have been addressed in three ways: (i) by imposing some form of global coordination (e.g. consensus, atomic broadcast); (ii) by relying on probabilistic approaches (e.g. gossip); or (iii) by employing pre-defined repair strategies based on application-specific knowledge (e.g. tree-specific repair). Unfortunately none of these approaches sit well with our goals. In particular, global coordination does not scale, probabilistic approaches do not lend themselves to consistency, and pre-defined repair strategies clearly do not meet the need for genericity.

We have therefore taken a fundamentally different tack: The core of our approach is a localized 'agreement protocol' called Echo (*Edge Consensus for High reliability Overlays*) that enables the set of nodes bordering a failed region of an overlay (i) to discover and agree on the extent of the failed region; (ii) to agree on a repair action to be taken; and (iii) to select a coordinator from among themselves to manage the repair. To succeed, Echo must however overcome a pernicious inter-dependency that arises between those who are *agreeing* (what we call the 'border set') and that which they are *agreeing to* (i.e. the extent of the failed region, which implies the constituency of the 'border set' itself). We refer to this phenomenon, which is one of the defining challenges addressed by Echo, as the 'self-defining constituency problem'.

Building on Echo, we have developed Sonar, a *generic* approach which offers reusable and flexible building blocks for overlay repair. Sonar illus-

trates the practical value of Echo for concrete systems, and delivers the following benefits:

Separation of generic and specific aspects of repair. Sonar/Echo separates overlay repair into two parts: a *generic* part in which the extent of a failure is detected and delineated (Echo); and a *repair specific* part in which a repair strategy is selected and enacted (Sonar). This separation allows us to support alternative repair strategies that make different tradeoffs depending on their deployment environment.

Localised repair. Sonar/Echo only involves nodes in the *locality* of a failed node or failed region. This locality is essential to guarantee the *scalability* of our approach. In very large overlays, such as Internet-scale P2P networks, it would be completely infeasible to involve centralised services or nodes beyond the failure locality.

Aggregated failure handling. Rather than restrict ourselves to treating *individual* overlay nodes as the unit of failure detection and repair, Sonar/Echo deals with *failed regions* of overlay. This is especially beneficial where the virtual structure of an overlay is related to the underlying physical topology of the network, and thus the simultaneous failure of adjacent overlay nodes is likely to be relatively common. Prominent examples are ad-hoc networks, or multicast trees organised in terms of IP domains.

The rest of the chapter is organised as follows. Section 3.2 first presents the overall architecture of the Echo/Sonar service. Section 3.3 proposes a formal specification of Echo (termed "convergent failure detection"), and sketches a proof of its Echo's correctness. Section 3.4 moves on to describe how Sonar builds upon Echo to offer a generic and adaptive repair mechanism for overlays. Section 3.5 provides some elements of evaluation, and Section 3.6 concludes the chapter.

3.2 Architecture of Echo/Sonar

Sonar/Echo is composed of three major services (Figure 3.2): a *distributed backup service* which stores backups of the local node on other hosts, a per-node *failure agreement protocol* (Echo) to detected and delineated the extent of a failure, and a *recovery service* (Sonar) that provide generic repair.

Sonar uses Echo to determine the extent of a failed overlay region, and decide on a repair strategy. Sonar and Echo use the backup service to obtain the state of failed nodes, including the neighbour links of those nodes, plus any additional overlay-specific data (e.g. DHT data), and repair-

specific information. Echo also relies on a failure detector (more on this on section 3.3) to learn about failed nodes

3.2.1 Main intuition

Achieving adaptive repair in arbitrary overlay networks ultimately aims to guarantee that *each failed node is repaired exactly once*; this is key to (safely) allowing fully generic adaptive repair, providing a platform from which many different *repair strategies* can be used at runtime. The challenge is to achieve this in a decentralised manner, and we solve it by leveraging distributed consensus [Chandra and Toueg (1996)] to have nodes collaborate and *agree* upon the course of action to take for each failure encountered.

While consensus protocols can achieve the necessary agreement, they do not however scale to the massive membership of many overlays—if *all* nodes in the overlay had to agree on a course of action for each failure, repairs would be unfeasibly slow. We therefore *scope* our consensus 'groups' to much smaller areas of overlay, such that participation of a node in a consensus group only occurs if it directly neighbours a given failure. This limits the number of nodes that must agree on how to deal with a failure, enabling scalability to very large systems. This ad-hoc consensus group formation and agreement is not straightforward, however, and give rise to a problem we have termed *self-defining constituencies*, to which we return in section 3.3.

3.2.2 Repair mechanisms: key phases

Sonar/Echo operates in *three* main phases, shared between Echo (Phases 1 & 2) and Sonar (Phase 3).

Phase 1 (Echo). When a node p detects a failed neighbour, p constructs a 'view' of the failure (which may be a 'failed region' of more than one node), and discovers the other live nodes surrounding the failure (the 'border nodes', including p). This phase operates by progressively discovering the extent of

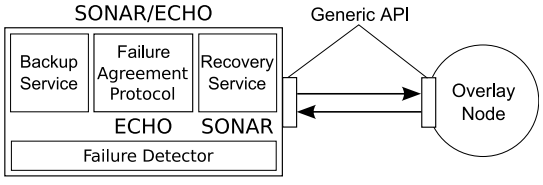
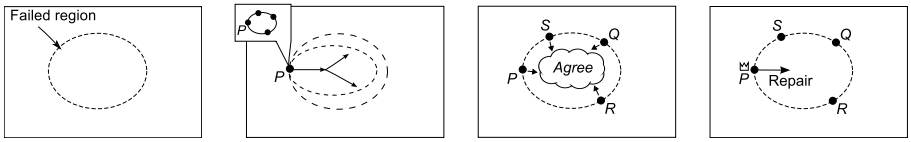


Figure 3.2: The architecture of Sonar/Echo



(a) (b) Phase 1 (c) Phase 2 (d) Phase 3

Figure 3.3: The three main phases of our repair algorithm (from [Porter et al. (2006b)])

a failed region, starting from a failed neighbour m. That neighbour's backup is acquired from the backup service and m's neighbours extracted. These 'next hop' neighbours are then queried with the failure detector to ascertain their liveness. Reportedly live nodes are added to the border node set, and failed nodes to the failed region. This process is repeated until no more failed nodes are found (Figure 3.3-b).

Phase 2 (Echo). Node p exchanges its phase 1 view with the other border nodes it has discovered, and those border nodes *negotiate* until they come to a single agreed view. During this process, border nodes may have their view *rejected* if their view conflicts with that of other nodes. It is at this point that we use consensus to reach agreement. Border nodes whose views are rejected at this stage return to phase 1 to re-consider their view, and may then re-enter phase 2. Repair strategy information is disseminated in phase 2, simply by piggy-backing it on agreement messages, and following agreement one of the border nodes is selected as the *repair coordinator* (Figure 3.3-c).

Phase 3 (Sonar). In phase 3 of the protocol, the chosen coordinator executes the selected repair strategy. The coordinator logs its intended repair in its local, per-node *repair log*, and while the repair is being carried out, the other border nodes wait until either they receive a repairNotification message from the coordinator, or they detect that the coordinator has failed. If the former happens, any local repair duties for non-coordinator nodes are carried out, completing the repair, and if the latter happens, the protocol loops back to the beginning.

3.3 Echo : convergent agreement

Echo allows the nodes bordering a failed overlay region to agree on the extent of this failed region, and decide on a common course of action. For scalability reasons, Echo limits communication to the failed region's border.

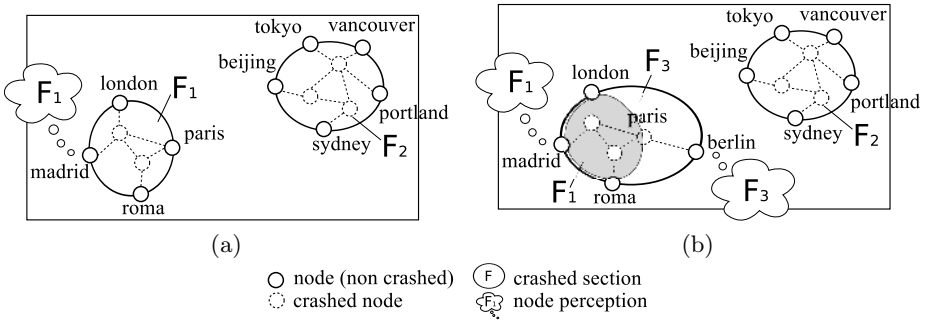


Figure 3.4: Protocol instances and conflicting views

This *scoping strategy* creates however a pernicious inter-dependency between the protocol's participants (the 'constituency') and what they are agreeing to: To start ECHO, a node needs to know with whom it should be agreeing (its fellow border nodes), but this set of nodes depends on the final outcome of the protocol (the failed region agreed upon).

In the following, we first illustrate this problem, which we have termed *self-defining constituencies* (Section 3.3.1). We then move on to define formally the properties of ECHO (Section 3.3.2); we present ECHO (Section 3.3.3 and 3.3.4; and we finally propose a proof of its correctness.

3.3.1 The challenge: self-defining constituencies

In the overlay of Figure 3.4-a, the nodes in region F_1 and F_2 have crashed. These crashes are being detected by the *border nodes* (i.e. the neighbouring nodes) of each failed region: *paris*, *london*, *madrid* and *roma* for F_1 and *tokyo*, *vancouver*, *portland*, *sydney*, and *beijing* for F_2. This detection occurs with the help of an appropriate failure detector and knowledge of the overlay's topology provided by the backup service.

Our scalability requirements impose that communications related to F_1 (resp. F_2) should be limited to nodes bordering F_1 (resp. F_2). For instance *vancouver* should not have to communicate with *madrid* to decide on a repair strategy for F_2. This excludes traditional consensus approaches that would involve the entire overlay in a protocol run.

Because of ongoing crashes, nodes bordering the same failed region might however possess divergent views regarding the extent of their region, and hence have diverging perceptions of who should get involved in a protocol run. In Figure 3.4-b, for instance, *paris* fails after *madrid* has detected F_1 as

crashed, but before an agreement on F_1 has been reached. The failed region F_1 thus grows into F_3, and a new node *berlin* (*paris*'s still non-crashed neighbour) becomes involved. *berlin* detects the entirety of F_3 as crashed.

madrid and *berlin* now have different, albeit overlapping views. If *madrid* is slow to detect *paris*' crash, it might try to agree on F_1 with *london* and *roma* alone, while *berlin* will try to involve all nodes bordering F_3 to decide on F_3. Each node's effort could possibly stall each other, or could lead to duplicated or inconsistent repairs. Echo prevents this and insures that any decisions pertaining to the same part of the network *converge* to a unified view.

3.3.2 System model and assumptions

We model an overlay network as a finite undirected graph $\mathcal{G} = (\Pi, \mathrm{E})$ of asynchronous message-passing nodes $\Pi = \{p_1, .., p_n\}$, where $\mathcal{G}$ represents the failure-free overlay topology. For space reasons, we only consider single execution runs in the remainder of this section and ignore repairs: we assume that $\mathcal{G}$ is static, i.e. that except for crashes, no nodes join or leave the overlay network while the algorithm executes. We return to this point in Section 3.4, where we explain how Echo can be adapted to work with Sonar in a context of ongoing repairs.

A node is *faulty* if it crashes at some point, *correct* if it does not crash during the execution of the algorithm. Any two nodes might exchange messages through asynchronous, reliable, and ordered (fifo) channels. We also assume that each node can query $\mathcal{G}$ on demand, either by directly contacting live nodes, or querying the backup service for failed nodes.

The *border* of a node p is the set of p's neighbours. By extension, the border of a set $S \subseteq \Pi$ of nodes are the nodes that have a neighbour in S but do not belong to S: $\mathsf{border}(S) = \{q \in \Pi \backslash S \mid \exists p \in S : (p, q) \in \mathrm{E}\}$. A *region* is a connected subgraph of $\mathcal{G}$. A *failed region* at a time t is a region in which all nodes have crashed.

To specify the liveliness of Echo, we need to define the three additional notions of *adjacency*, *faulty domain* and *faulty cluster*, which capture the maximum extent of failed regions during a run. More precisely, a *faulty domain* is a region in which all nodes are faulty, but whose border nodes are correct. By construction, two faulty domains can only be either equal or disjoint.

Two faulty domains F and H are *adjacent* (noted $F \parallel H$) if their borders intersect (e.g. $F_1 \parallel F_2$ in Figure 3.5). We say that two faulty domains F_0 and F_n are in the same *faulty cluster*, noted $\mathsf{clustered}(F_0, F_n)$, if they are

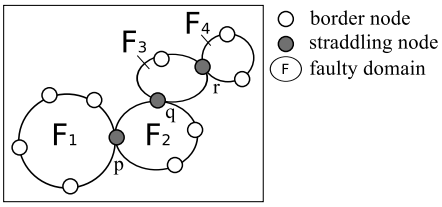


Figure 3.5: A cluster of adjacent faulty domains

transitively adjacent[1], i.e. if there is a sequence of faulty domains F_i so that $F_1 \parallel F_2 \ldots F_{n-1} \parallel F_n$. For instance, we have $\mathsf{clustered}(F_1, F_4)$ in Figure 3.5.

3.3.3 Specification

ECHO delivers a service we have termed *convergent detection of failed regions*. In the following, we use the event-based model to define this service formally in terms of operations and properties.

Operations

ECHO starts when a node detects one of its neighbours q as crashed ($\langle \mathbf{crash} \mid q \rangle$ event). It stops by raising a $\langle \mathbf{decideECHO} \mid V, d \rangle$ event, where V is the failed region decided by the local node, and d the decision taken respect to V (in SONAR, a repair plan). We call V the *view* of the deciding node.

Properties

The convergent detection of failed regions is characterised by the following properties:

CD1 (Integrity) No node decides twice on the same region.

CD2 (View Accuracy) If a node p decides (V, d), then $p \in \mathsf{border}(V)$, and V is a failed region.

CD3 (Locality) Communication is limited to faulty-domains and their borders, i.e. a node p only exchanges messages with a node q if there is a faulty domain S such that $\{p, q\} \subseteq S \cup \mathsf{border}(S)$.

CD4 (Border Termination) If p decides (V, d), then all correct nodes in $\mathsf{border}(V)$ eventually decide.

[1] More formally, $\mathsf{clustered}(.,.)$ is the transitive closure of the adjacency relation, and *faulty clusters* its equivalence classes.

CD5 (Uniform Border Agreement) If two nodes p and q decide, and p decides (V, d), and $q \in \mathsf{border}(V)$, then q decides (V, d).

CD6 (View Convergence) If two correct nodes decide V and W, $(V \cap W \neq \emptyset) \Rightarrow (V = W)$.

CD7 (Progress) In each faulty cluster, at least one correct node bordering a faulty domain in the cluster eventually decides: if $\mathcal{D}$ is the set of all faulty domains, $\forall V \in \mathcal{D} : \exists W \in \mathsf{clustered}(V, \cdot) : \exists p \in \mathsf{border}(W) : p$ decides.

CD1 (*Integrity*), CD5 (*Uniform Border Agreement*), and CD4 (*Border Termination*) are directly adapted from (uniform) consensus; CD2 (*View Accuracy*) is taken over from the strong accuracy of fault detectors; and CD7 (*Progress*) is a weak form of termination.

The problem's originality resides in the two remaining properties: CD6 (*View Convergence*) and CD3 (*Locality*). *View convergence* forbids conflicting agreements on overlapping failed regions (F_1 and F_3 in Figure 3.4). *Locality* provides scalability by limiting the system's reaction to the vicinity of failed regions. As a result, the protocol only depends on the amount of failures in the system, but not on the system's actual size. *Locality* also excludes the use of a system-wide consensus to fulfil the other properties.

3.3.4 Failure detector, multicast, region ranking

Our algorithm uses a perfect failure detector (we return in Section 3.5 on the practical implications of this choice), provided in the form of a *subscription-based* service: a node p subscribes to the crashes of a subset of nodes S by issuing the event $\langle \textbf{monitorCrash} \mid S \rangle$ to its local failure detector. Our failure detector is perfect and insures: *(i) Strong Accuracy*: if a node p receives a $\langle \textbf{crash} \mid q \rangle$, then q has crashed, and p did subscribe to be notified of q's crash; and *(ii) Strong Completeness*: if a node q has crashed, and p has subscribed to be notified of q's crash, then p will eventually receive a $\langle \textbf{crash} \mid q \rangle$ event.

For compactness, we use a basic multicast service, represented by the events $\langle \textbf{multicast} \mid R, [\text{m}] \rangle$ and $\langle \textbf{mDeliver} \mid p, [\text{m}] \rangle$. This service simply sends to each recipient the multicast message over the underlying point-to-point channels, in a plain loop. This service provides no guarantees beyond those of the underlying channels, and is essentially a shorthand to keep our code brief.

We also use a *ranking relation* between regions, noted $\succ$: $R \succ S$ iff either (i) R contains more nodes than S, or (ii) they contain the same number of

nodes but R's border contains more nodes than S's border, or (iii) R and S have the same size, and so do their respective borders, but R is greater than S according to some strict total order relation $\rhd$ on sets of nodes. The actual ordering relation $\rhd$ on node sets does not matter. One possibility is to use a lexicographic order on node IDs. By construction, $\succ$ is a strict total order on regions. For a set $\mathcal{C}$ of regions, $\mathsf{maxRankedRegion}(\mathcal{C})$ is the highest ranked region in $\mathcal{C}$.

Finally, for a subset S of nodes, $\mathsf{connectedComponents}(S)$ returns the set of the maximal regions of S, i.e., formally, the vertex sets of the connected components of the subgraph $\mathcal{G}[S]$ induced by S in $\mathcal{G}$.

3.3.5 Algorithm

The pseudo code of our algorithm is given in Algorithm 1. $\langle$**initECHO**$\rangle$ is executed by all nodes when the protocol starts. Each node then remains idle until one of its neighbours fails, as notified by a $\langle \mathbf{crash} \,|\, q \rangle$ event.

The bulk of the protocol is primarily a superposition of flooding uniform consensus instances [Chandra and Toueg (1996)] between the border nodes of proposed views. This superposition is complemented by an arbitrating mechanism to deal with overlapping but conflicting views (line 26). Because of this arbitration, all consensus instances must be tracked concurrently by ECHO, in the variables $\mathrm{op}^{\mathrm{ALL}}[\cdot][\cdot][\cdot]$ and waiting$[\cdot][\cdot]$, which are indexed by proposed views (in addition to rounds, and, for $\mathrm{op}^{\mathrm{ALL}}$, participants).

A node starts a consensus instance when it detects that one of its neighbours has crashed (line 17). The view it proposes has been incrementally built when receiving $\langle \mathbf{crash} \,|\, . \rangle$ events (line 5), and is the highest ranked failed region known to the node at this point. The view construction continues in the background as the consensus unfolds (lines 5-10), to be used if the attempt to reach an agreement fails.

The opinion vectors received from other nodes in a round are gathered at line 18. Because a node might be involved simultaneously in multiple conflicting consensus instances, messages related to conflicting views are also gathered and processed. The resulting opinion vectors, indexed by round and proposed view (line 24) are stored in $\mathrm{op}^{\mathrm{ALL}}[\cdot][\cdot][\cdot]$.

If a node becomes aware of a conflicting view with a lower rank (line 26), it sends a special $\mathrm{op}_{\mathsf{reject}}$ vector to this view's border nodes, and subsequently ignores any message related to this view (lines 28-31).

Rounds are completed at line 32 when all non-crashed border nodes of V_p have replied: if no more rounds are needed (line 34), and the node's final vector only contains accept values, a decision value is deterministically

Algorithm 1 ECHO: Convergent detection executed by node p

1: **upon event** $\langle$**initECHO**$\rangle$
2: decided $\leftarrow \perp$; proposed $\leftarrow \perp$; newCandidate $\leftarrow$ FALSE
3: $\text{crashed}_p, \text{candidateView}, V_p, \text{received}, \text{rejected} \leftarrow \emptyset$
4: **trigger** $\langle$**monitorCrash** $|$ $\mathsf{border}(p)\rangle$
5: **upon event** $\langle$**crash** $|$ $q\rangle$ ▷ View construction
6: $\text{crashed}_p \leftarrow \text{crashed}_p \cup \{q\}$
7: **trigger** $\langle$**monitorCrash** $|$ $\mathsf{border}(q) \backslash \text{crashed}_p\rangle$
8: $\mathcal{C} \leftarrow \mathsf{connectedComponents}(\text{crashed}_p)$
9: **if** candidateView $\prec \mathsf{maxRankedRegion}(\mathcal{C})$ **then**
10: candidateView = $\mathsf{maxRankedRegion}(\mathcal{C})$
11: newCandidate $\leftarrow$ TRUE
12: **upon event** proposed $= \perp \wedge$ newCandidate $=$ TRUE ▷ New consensus instance
13: $V_p \leftarrow$ candidateView ; newCandidate $\leftarrow$ FALSE
14: proposed $\leftarrow \mathsf{selectValueForView}(V_p)$
15: $\text{op}_{\mathsf{accept}}[p_k] \leftarrow \perp$ **for all** $p_k \in \mathsf{border}(V_p) \backslash \{p\}$
16: $\text{op}_{\mathsf{accept}}[p] \leftarrow (\mathsf{accept}, \text{proposed})$; $r \leftarrow 1$
17: **trigger** $\langle$**multicast** $|$ $\mathsf{border}(V_p), [1, V_p, \mathsf{border}(V_p), \text{op}_{\mathsf{accept}}]\rangle$
18: **upon event** $\langle$**mDeliver** $|$ $p_i, [r, V, B, \text{op}]\rangle \wedge V \notin$ rejected ▷ Updating opinions
19: **if** $V \notin$ received **then**
20: received $\leftarrow$ received $\cup \{V\}$ ▷ Initialise data structures for V
21: $\text{op}^{\text{ALL}}[V][r][p_k] \leftarrow \perp$ **for all** $p_k \in B \wedge 1 \leq r < |B|$;
22: waiting$[V][r] \leftarrow B$ **for all** $1 \leq r < |B|$
23: **for all** p_k **such that** $(\text{op}^{\text{ALL}}[V][r][p_k] = \perp \wedge \text{op}[p_k] \neq \perp)$ **do**
24: $\text{op}^{\text{ALL}}[V][r][p_k] \leftarrow \text{op}[p_k]$
25: waiting$[V][r] \leftarrow$ waiting$[V][r] \backslash (\{p_i\} \cup \{p_k | \text{op}[p_k] = \mathsf{reject}\})$
26: **upon event** $\exists L \in$ received $: L \prec V_p$ ▷ Rejecting a lower ranked view
27: **trigger** $\langle$**reject** $|$ $L\rangle$
28: **upon event** $\langle$**reject** $|$ $L\rangle$
29: $\text{op}_{\mathsf{reject}}[p_k] \leftarrow \perp$ **for all** $p_k \in \mathsf{border}(L) \backslash \{p\}$
30: $\text{op}_{\mathsf{reject}}[p] \leftarrow \mathsf{reject}$; received $\leftarrow$ received$\backslash\{L\}$; rejected $\leftarrow$ rejected $\cup \{L\}$
31: **trigger** $\langle$**multicast** $|$ $\mathsf{border}(L), [1, L, \mathsf{border}(L), \text{op}_{\mathsf{reject}}]\rangle$
32: **upon event** $V_p \in$ received $\wedge$ waiting$[V_p][r] \backslash \text{crashed}_p = \emptyset \wedge$ decided $= \perp$
33: **if** $r = |\mathsf{border}(V_p)| - 1$ **then** ▷ Consensus instance completed
34: **if** $\forall p_i \in \mathsf{border}(V_p) : \text{op}^{\text{ALL}}[V_p][r][p_i] = (\mathsf{accept}, v_{p_i})$ **then**
35: decided $\leftarrow \mathsf{deterministicPick}(\{v_{p_i}\}_{p_i \in \mathsf{border}(V_p)})$ ▷ decision
36: **trigger** $\langle$**decideECHO** $|$ V_p, decided$\rangle$
37: **else** proposed $\leftarrow \perp$ ▷ Consensus attempt failed, reset
38: **else** ▷ New round
39: $r \leftarrow r + 1$
40: **trigger** $\langle$**multicast** $|$ $\mathsf{border}(V_p), [r, V_p, \mathsf{border}(V_p), \text{op}^{\text{ALL}}[V_p][r-1]]\rangle$

selected for the proposed view (line 35), and the node decides[2]. Otherwise the whole process is reset, and restarts at line 12 as soon as a new crashed node is detected.

3.3.6 Proof of correctness

In the following, we use a subscript notation to distinguish between the same protocol variable at different nodes: e.g. v_p for variable V_p of p.

Theorem 3.3.1. ECHO *fulfils properties CD1 (Integrity), CD2 (View Accuracy), and CD3 (Locality).*

Proof. CD1 is fulfilled by construction. For CD2, connectedComponents() at line 8 and the strong accuracy of the failure detector insure that proposed views are failed regions. Using recursion on $\langle \textbf{crash} \,|\, . \rangle$ events, a node p can be shown to respect the two invariants *(i)* $p \in \mathsf{border}(\text{crashed}_p)$ and *(ii)* $\{p\} \cup \text{crashed}_p$ is connected, thus yielding that p is on the border of any view it proposes. CD3 follows from CD2, and the fact that two nodes only exchange messages when both border a region detected as failed by one of them. □

Our proof of the remaining four properties reuses elements of the proof of the consensus algorithm presented in [Chandra and Toueg (1996)] for strong failure detectors (S), of which the flooding uniform consensus is derived. The difficulty lies in that our protocol uses multiple overlapping consensus instances, each indexed by the view it proposes, with no prior agreement on either the set the consensus instances, their participants, or their sequence. In addition, our arbitrating mechanism means a node can first propose and then reject the same view, thus complicating the *uniform border agreement*, as we shall see.

Lemma 3.3.2. *At any execution point the vectors* $\text{op}_p^{\text{ALL}}[S][r][\cdot]$ *of* p *are such that* $\forall q \in \mathsf{border}(S)$:
1) $\text{op}_p^{\text{ALL}}[S][r][q] = \mathsf{reject} \Rightarrow q$ *rejected* S *earlier* $\wedge$
2) $\text{op}_p^{\text{ALL}}[S][r][q] = (\mathsf{accept}, \cdot) \Rightarrow q$ *accepted* S *earlier*

Proof. This lemma follows from a recursive data-flow argument on the values of opinions[S][r][·], the properties of the best-effort multicast and the transitivity of the happened-before relation. □

[2]For clarity's sake, the presented version is not optimised. A classical optimisation consists in terminating a consensus instance once a node sees that all nodes in its border set know everything (i.e. no $\perp$), i.e. after two rounds, in the best case.

Lemma 3.3.3. *A node proposes (resp. rejects) a given view S at most once. A node never proposes a view it has previously rejected.*

Proof. The uniqueness of rejection follows from the use of the rejected and received variables. The use of the strict ranking relation $\prec$ (line 9) means the series of values taken by V_p is strictly monotonic according to $\prec$, and by construction that this is also true of V_p, thus completing the lemma. □

Lemma 3.3.4. *If two nodes p and q complete a consensus instance on the same $v_{p|q} = S$ (line 34), they obtain the same opinion vector:*

$$\text{op}_p^{\text{ALL}}[S][N][\cdot] = \text{op}_q^{\text{ALL}}[S][N][\cdot] \text{ where } N = |\mathsf{border}(S)|$$

Proof. We prove this lemma by contradiction. Let's assume $\exists k \in \mathsf{border}(S)$: $\text{op}_p^{\text{ALL}}[S][N][k] \neq \text{op}_q^{\text{ALL}}[S][N][k]$. If one of the two values is $\perp$, we can use the well-known argument on cascading crashes, identifying $N-1$ distinct nodes in $\mathsf{border}(S)$ that did not complete the consensus instance, contradicting the fact that p and q completed it.

Let's now assume both values are non-$\perp$. The first sub-case is when both values are accept for k, with different decision values on p and q, i.e. $\text{opinions}_p[S][N][k] = (\mathsf{accept}, v_k^p)$ and $\text{opinions}_q[S][N][k] = (\mathsf{accept}, v_k^q)$ with $v_k^p \neq v_k^q$. Using lemma 3.3.3, we conclude that line 16 is executed only once by k for S, and that $v_k^p = v_k^q$, yielding the contradiction.

Finally, let's assume one value is accept, while another is reject, e.g. From lemma 3.3.2 we conclude that k has both proposed and rejected S. Let's call e^k_{accept} and e^k_{reject} the corresponding execution points. Because of lemma 3.3.3, e^k_{accept} and e^k_{reject} are unique, and e^k_{accept} happened before e^k_{reject}. Because the best-effort multicast is fifo, this means q received the message for e^k_{accept} before that of e^k_{reject}, and because line 24 only updates $\perp$ values, that $\text{op}_q^{\text{ALL}}[S][N][k] = (\mathsf{accept}, \cdot)$, yielding the contradiction. □

Theorem 3.3.5. Echo *fulfils properties CD5 (Uniform border agreement) and CD4 (Border termination).*

Proof. Let's assume p and q decide, p decides $(S, \text{decided}_p)$, and $q \in \mathsf{border}(S)$. If p decides on S, then p completed the corresponding consensus instance with only accept values, and since $q \in \mathsf{border}(S)$ we have $\text{op}_p^{\text{ALL}}[S][N][q] = (\mathsf{accept}, \cdot)$. By lemma 3.3.2, q proposed S. Since by construction a node *(i)* cannot propose any new view once it has decided on one, and *(ii)* cannot start a new consensus instance before completing the current one, q proposed S and completed the corresponding consensus instance before deciding. By lemma 3.3.4, q obtained the same vector op_q^{ALL} as p on S, and

hence decided $(S, \text{decided}_p)$ by determinism of deterministicPick (line 35), thus proving CD5.

CD4 follows the same line, with the observation that if a node p completes a consensus instance on a view S, then all other nodes in $\mathsf{border}(S)$ either took part in each round or crashed, implying that all correct nodes eventually complete the instance with the same opinion vector as p (by way of lemma 3.3.4). □

Theorem 3.3.6. ECHO *fulfils CD6 (View convergence).*

Proof. Let's consider two correct nodes p and q that decide on overlapping failed regions S_p and S_q: $S_p \cap S_q \neq \emptyset$. If one node is in the border of the other's region, e.g. $p \in \mathsf{border}(S_q)$, then *Uniform Border Agreement* (CD5) and *Integrity* (CD1) give us $S_p = S_q$.

Let's now assume $p \notin \mathsf{border}(S_q) \wedge q \notin \mathsf{border}(S_p)$, and use a proof by contradiction. Since $S_p \cap S_q \neq \emptyset$, there is a node $a \in S_p \cap S_q$ (Figure 3.6). S_p being a region bordered by p (CD2), there exists a path $(n_0 = p, n_1, ..., n_k = a)$ that links a to p through S_p: $\{n_1, ..., n_k, a\} \subseteq S_p$. Since $a \in S_q$, we can consider the point when this path "penetrates" for the first time into S_q, i.e. we can consider $n_{i_0} \in S_q$ and $\forall i < i_0 : n_i \notin S_q$. Since p is correct, $n_{i_0} \neq p$, i.e. $i_0 \geq 1$, and we can look at n_{i_0-1}, the node in the path just before n_{i_0}. Let's call this node r (Figure 3.6). Because n_{i_0} is the first node in the path to belong to S_q, we have $r \in \mathsf{border}(S_q)$, and since $p \notin \mathsf{border}(S_q)$, $r = n_{i_0-1}$ cannot be p ($i_0 > 1$). Because, with the exception of p, the path connecting p to a is embedded in S_p, this means that r is in fact located in p's failed region. This reasoning thus yields us a node (r) that is both on $\mathsf{border}(S_q)$ *and* in p's failed region: $r \in S_p \cap \mathsf{border}(S_q)$. Using an identical argument, we can find a node s such that $s \in S_q \cap \mathsf{border}(S_p)$ (Figure 3.6).

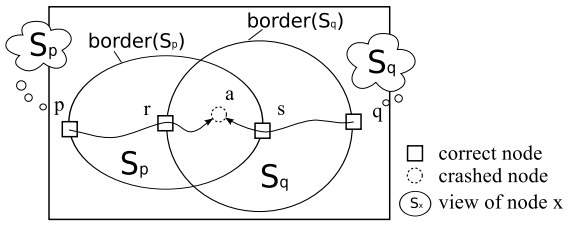


Figure 3.6: Convergence between overlapping views

To complete our proof, we now look at the happen-before relationships between events related to r and s. Let's first consider s. Since $s \in \mathsf{border}(S_p)$ and p decided on S_p, s itself did propose S_p (lemma 3.3.2). Since $r \in S_p$, s

did detect r as crashed as some point. By a similar reasoning, we conclude that r proposed S_q, and hence detected s as crashed as some point.

We thus end up with a set of 6 events that form a circular chain of happen-before events: *s_detects_r* $\rightarrow$ *s_proposes* $\rightarrow$ *s_crashes* $\rightarrow$ *r_detects_s* $\rightarrow$ *r_proposes* $\rightarrow$ *r_crashes* $\rightarrow$ *s_detects_r* ... This provides our contradiction. □

Theorem 3.3.7. Echo *fulfils properties CD7 (Progress).*

Proof. Again we use a contradiction: consider a cluster of adjacent faulty domains (Figure 3.5), and assume none of its correct border nodes ever decide. Since this situation lasts indefinitely, we can consider the case where all failed regions are maximal and all remaining nodes are correct.

Because the views proposed by a node are strictly monotonic according to $\prec$, and because $\mathcal{G}$ is finite, a node cannot propose an infinite sequence of views. A correct border node p that does not decide falls therefore into two cases: either (**C1**) p is blocked waiting for the reply of another node q (line 38); or (**C2**) the last view proposed by p failed (line 37), and p does not detect any new crashed node (line 5).

Case C1 : If p is waiting for the reply of some other node q, q must be correct (if it were not, q would eventually crash, thus unblocking p). Since there's a path of crashed nodes from p to q (since p is waiting for q), q is on the border of the same faulty domain as p, so q never decides (by assumption).

As for p, q falls in either case **C1** or **C2**. Let's first assume that the last view S_q^{max} proposed by q failed, and q does not detect any new crashed node (**C2**). Since we've assumed that all faulty nodes have crashed, by strong completeness of the failure detector, S_q^{max} is a faulty domain, and because of the use of maxRankedRegion (line 10) and the fact that $\prec$ subsumes set inclusion, S_q^{max} is higher ranked than any failed region bordered by q.

Since p is waiting for q, $S_p \neq S_q^{\mathsf{max}}$, and since q is on the border of both S_p and S_q^{max}, S_p is lower-ranked than S_q^{max}: $S_p \prec S_q^{\mathsf{max}}$. q has received a round-1 message proposing S_p (line 18), and should have rejected it (line 31), thus ending p's wait on q, which contradicts our assumption.

We therefore conclude that q cannot fall in case **C2**, and instead is blocked in a consensus round proposing a failed region S_q (case **C1**). q received p's proposal message, and did consider it for rejection (line 26). Because p is waiting for q, we know it did not receive any rejection message from q, and therefore, $S_p \succeq S_q$. Since p is waiting for q, q is not proposing the same view as p, yielding a strict ordering between the two views $S_p \succ S_q$.

This construction can be repeated recursively, first for q, and then for the node q is waiting on, etc, each time yielding an infinite number of pairwise distinct failed regions (via CD2) that are strictly ordered by the ranking relationship: $S_{p_1} \succ S_{p_2} \succ ... \succ S_{p_i} \succ ...$ This contradicts our assumption that each faulty cluster contains a finite number of faulty domains, each containing a finite number of nodes.

Case C2 : Let's now assume the last view S_p^{max} proposed by p failed, and p does not detect any new crashed node. As above S_p^{max} is a faulty domain, and all its border nodes are correct. Because the failure detector is strongly accurate, for p's proposal to fail, one node $q \in \mathsf{border}(S_p^{\mathsf{max}})$ must have rejected S_p^{max} because it was proposing a higher-ranked view. By assumption, q never decides, it must either fall in case **C1** or **C2**. If **C1**, we're back to the previous case. If **C2**, q's last view S_q^{max} is higher than any view q ever proposed, implying $S_p^{\mathsf{max}} \prec S_q^{\mathsf{max}}$.

By recursively applying this argument, we either come back to case **C1** at some point, or obtain an infinite sequence of strictly ordered faulty domains $S_{p_1}^{\mathsf{max}} \prec S_{p_2}^{\mathsf{max}} \prec S_{p_3}^{\mathsf{max}} \prec ...$, with the same kind of contradiction as in case **C1** above, which concludes our proof by contradiction. □

3.4 SONAR : Generic repair

SONAR (Algorithm 2 on the next page) builds on ECHO to encapsulate reusable repair strategies in a generic service. SONAR starts once a decision has been reached by ECHO, returning a repair plan for a failed region (line 3). This repair plan is selected by ECHO from the set of repair plans proposed by each border nodes (through the function selectValueForView at line 14 of Algorithm 1 on page 34). The chosen repair plan lays out the concrete repair strategy to be pursued (more on this in section 3.4.2)[3], including an indication of which node should coordinate the repair. This special node is known as the 'coordinator' and originally each participating node proposes itself as coordinator. While the repair unfolds, the other border nodes wait until either they receive a repairNotification message from the coordinator, or they detect that the coordinator has failed.

Algorithm 2 The Sonar repair protocol (executed by node p)

```
1:  upon event ⟨initSONAR⟩
2:      repairLog, logsSeen ← ∅
3:  upon event ⟨decideECHO | failedRegion, repairPlan⟩
4:      coord ← getCoordinator(repairPlan)
5:      if p = coord then                                  ▷ I am the coordinator
6:          repairLog ← repairLog ∪ {failedRegion}
7:          enactRepair(failedRegion, repairPlan)                          ▷ Phase3
8:          trigger ⟨send | border(failedRegion), [repairNotification, p, failedRegion]⟩
9:      else                                           ▷ I am not the coordinator
10:         wait until
11:             ⟨receive | coord, [repairNotification, coord, failedRegion]⟩ or
12:             ⟨crash | coord⟩
13: upon event ⟨receive | q, [repairNotification, s, reg]⟩                ▷ remote reset
14:     crashed_p ← crashed_p \ ⋃{x ∈ connectedComponents(crashed_p) | x ∩ reg ≠ ∅}
15:     if V_p ⊈ crashed_p then V_p ← ∅; proposed ← ⊥
16:     if candidateView ⊈ crashed_p then
17:         candidateView ← ∅ ; newCandidate ← FALSE
18:     logsSeen ← logsSeen ∪ {(s, reg)}
19:     trigger ⟨monitorCrash | border(p)\crashed_p⟩
20: upon event ∃v_i ∈ received, (s, ℓ) ∈ logsSeen : ℓ ∩ v_i ≠ ∅ ∧ (s, ℓ) ∉ v_i.logs
21:     trigger ⟨send | border(v_i), [repairNotification, s, ℓ]⟩
22:     received ← received\v_i
```

3.4.1 Ongoing repairs and coordination in Sonar

As in Echo, other nodes might attempt to repair a region overlapping with that of the coordinator: for instance, returning to Figure 3.4 on page 29, *madrid* might be selected as coordinator to repair region F_1 (Figure 3.4-a), but if *paris* fails, *berlin* will try to get an agreement on region F_3, which contains F_1 (Figure 3.4-a). Thanks to Echo's blocking mechanism, *berlin* is prevented to proceed. However, in Sonar, *berlin* should eventually be allowed to resume execution, either once the region F_1 has been successfully repaired by *madrid*, or if *madrid* fails halfway through its repair. In both cases, however, *berlin* should restart its view construction to take into account the new (possibly only partially) repaired state of the overlay.

To provide this property, the coordinator logs its intended repair in a permanent, per-node *repair log*, provided by the backup service (Figure 3.2).

[3] Formally only regions are repaired, not nodes. In particular, node IDs are not reused, so that a node that has failed is considered permanently so. As a shortcut, however, we say that a node p has been repaired if a failed region that contains p has been repaired.

Algorithm 3 Modified view construction of ECHO (executed by node p)

```
1:  upon event ⟨crash | q⟩                                          ▷ View construction
2:      if ∃reg ∈ q.repairLog : (q, reg) ∉ logsSeen then
3:          for all reg ∈ q.repairLog ∧ (q, reg) ∉ logsSeen do
4:              trigger ⟨send | p, [repairNotification, q, reg]⟩
5:          return
6:      if q ∉ border(crashed_p ∪ {p}) then return      ▷ crashed_p might now shrink
7:      crashed_p ← crashed_p ∪ {q}                  ▷ Back to original view construction
8:      trigger ⟨monitorCrash | border(q)\crashed_p⟩
9:      C ← connectedComponents(crashed_p)
10:     if candidateView ≺ maxRankedRegion(C) then
11:         candidateView ← maxRankedRegion(C)
```

Each node also keeps track of the repairs it has heard of in the variable 'logsSeen', which is appended to each proposed view (not shown). This mechanism has two usages. First, if a node p receives a view v_i that does not know of repairs that p knows of (line 20), all nodes holding this view are forced to reset their view (lines 13- 19).

Second, the view construction part of ECHO is modified to take into account the repair logs of crashed coordinators (algorithm 3): When a node encounters a failed node's backup for the first time, the logs are examined, and used to update the node's knowledge of the overlay's current state.

3.4.2 Example repair strategies

Each border node constructs a repair strategy with the function selectValueForView (line 14 of algorithm 1) and 'scores' it using a uniform grading system. This score is then used in deterministicPick (line 35 of algorithm 1) so that the best-graded repair is chosen for each failure, from all those suggested. Quite importantly, nodes can propose a wide range of repair strategies within the framework proposed by SONAR. In particular, each node can tune the repair it proposes based on run-time contextual information, thus providing maximum flexibility to the scheme.

For instance a node may choose to propose an **additive repair**, which *creates* or *restores* nodes as part of the repair, thereby adding new resources to the overlay (Figure 3.7 on the following page). This form of repair might be well adapted in Grid and Cloud-like environments, where additional resources may be available, and can be total (Figure 3.7-b), with an exact clone of the failed region recreated on new hosts, or partial (Figure 3.7-c), with only a few hubs (one in the figure) replacing the failed nodes.

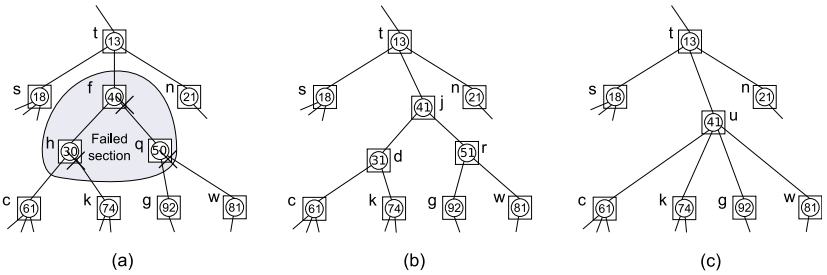


Figure 3.7: Tree overlay with a failed region (a), followed by a full (b) and partial (c) additive repair. Overlay nodes are shown with numbers, and physical hosts with letters.

Alternatively, a node may propose a **subtracting repair**, by re-distributing the responsibilities of the failed nodes among live overlay nodes to maintain functionality after failures. For instance, one simple option is to use a 'single hub' approach in which the coordinator takes on the responsibilities of the failed region (shown in Figure 3.1 on page 25). This kind of repair may be particularly suitable when resources beyond the scope of the overlay are limited—particularly in end-user-host deployments, or when the expected workload does not warrant the need for additional resources.

3.5 Evaluation

Echo and Sonar have been evaluated analytically, in simulations, and on a real deployment in PlanetLab. The following provides a high-level overview of these results, with details available in the relevant publications [Porter (2007); Porter et al. (2006b)].

3.5.1 Failure free runs

Echo is triggered when a region fails in an overlay. The most favourable runs, from a performance perspective, are when the nodes bordering the failed region all detect the full extent of this region at the start of the protocol, and no additional failure occurs while the protocol executes ('failure free runs'). In such runs, the per-node message complexity of Echo is proportional to $\mathcal{O}(b + f)$ (or $\mathcal{O}(b^2 + b \cdot f)$ overall), where b is the size of the border set, and f is the number of nodes in failed region. This complexity assumes the optimisation mentioned in note 2, and includes failure notifications, backup-accesses, and Echo messages.

Turning to Sonar, the number of messages involved is that of Echo, plus those of the repair itself, which are dependent on the selected repair strategy (Section 3.4.2). Generally, however, these costs are minor compared to those incurred by Echo. For example, node restoration typically incurs one extra message per restored node to instantiate and add state to that node, in addition to the cost of locating suitable hosts (not considered in this paper).

Overall, as the combined cost is a polynomial function of the size of the border set and the failed region, the overhead only becomes an issue with 'large' failed regions and highly connected overlays. Moreover, *this overhead is completely independent of the size of the overlay itself.*

To put this in context, any failure in the Chord [Stoica et al. (2001)] ring-based overlay would involve just *two* border nodes (excluding 'finger' nodes, which are refreshed continuously). This would yield a message count of 9 for a failed region of size 1, and 33 for a failed region of size 5, including FD probe usage, backup accesses[4], and agreement messages. Similarly, the failed region in Figure 3.1 would result in a border set of size 4 (yielding a repair message count of 66). In both cases the cost is independent of the size of the overlay.

3.5.2 On-going failures

Sonar/Echo accommodates ongoing failures, but assumes that failures will stop for long enough for the protocol to complete. The *worst* possible case occurs when (i) the round-reducing optimisation discussed in footnote 2 on page 35 cannot be used and (ii) the agreement concludes only for the view to be aborted. This happens when a border node fails at the very start of Echo without providing an opinion, and a full consensus instance is executed to no effect.

If this happens repeatedly as a failed section grows in size, the overall number of messages can become as large as $\mathcal{O}(b^3 \cdot f + b \cdot f^2)$, with b and f the final size of the border set and the failed section, respectively. This is to be compared against a count in $\mathcal{O}(b^2 + b \cdot f)$ in good runs. Although rather high, this worst-case overhead can be traced back to the strong robustness property provided, and should in practice remain relatively rare.

[4] In these examples we assume a backup service that incurs 2 messages to retrieve each backup.

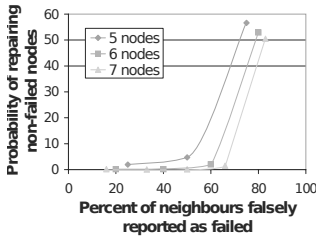


Figure 3.8: Effects of false positives in fully-connected mesh overlays

3.5.3 The impact of imperfect failure detection

Our demonstration of Echo in Section 3.3.5 assumed a perfect failure detector (class P). Such detectors are in practice extremely hard to approximate in asynchronous environments (and formally impossible to realise). Many practical fault-tolerance algorithms therefore either assume an eventually perfect ($\diamond P$) or eventually strong detector ($\diamond S$).

The reason behind Echo's reliance on a perfect failure detector is to be found in the semantic of the service: if the output of the failure detector cannot be trusted, the protocol can no longer be guaranteed to distinguish failed regions from non-failed ones, and the formal properties specified in Section 3.3.3 collapse.

Practice is however not so bleak: With an imperfect failure detector, Echo will in most cases still return valid failed regions, and guarantee the convergence of overlapping views. This is because in order to wrongly decide on a region that contains non-failed nodes, all the nodes bordering that region must receive the same incorrect suspicions from their local failure detector. (They must all be wrong in the same way.) With uncorrelated false positive, our simulations show that the probability of this happening is relatively low: a large percentage of per-node false positives is required for non-failed nodes to be wrongly repaired, and the probability of this happening decreases with the connectivity of the overlay (Figure 3.8).

3.5.4 TBCP case study and PlatnetLab deployment

To evaluate finally the viability of Echo/Sonar in a concrete overlay, we used it to replace the custom-build repair mechanism of TBCP, a well-known application-level multicast overlay [Mathy et al. (2001)]. In TBCP, nodes join at the tree root, and from there find a 'good' parent node to become a child of according to set preferences.

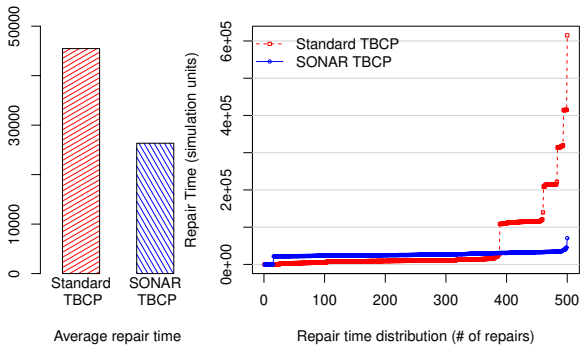


Figure 3.9: Standard TBCP & Sonar/TBCP repair times

In simulations, our own version of TBCP (termed Sonar/TBCP) was on average 1.8 times faster standard TBCP in repairing failures (Figure 3.9-a) in a 1000-node overlay. This is explained by the re-join-at-grand-parent/root strategy of the default TBCP repair mechanism. This strategy maintains a very good tree shape, but can take a significant time to 'stabilise' as displaced nodes find new positions in the tree. To put this in context, Figure 3.9-b shows the spread of repair times for the two versions—the fastest repairs were 1,868 time units for standard TBCP, and 21,444 for Sonar/TBCP; and their slowest repairs 615,144 and 70,533 respectively. On inspection, this is explained by the fact that Sonar/TBCP takes longer in 'easy' repairs where nodes fail close to the leaves of the tree, but performs much better near the root, giving it a lower average.

To test the practical feasibility of our approach, we also deployed Sonar/TBCP on 150 PlanetLab hosts. Figure 3.10 shows Sonar/TBCP's network usage at the root node on PlanetLab, giving a flavour of the real costs involved. Globally, Sonar/TBCP showed a low normal network usage, at around 5KB per second on average. During this experiment, the root node shown was involved in 4 repairs (marked with arrows), and these caused higher volumes of network traffic over brief periods.

3.6 Conclusion

This chapter has presented my work on Sonar/Echo, a generic and reusable service for repairing of arbitrary overlay networks. Sonar/Echo allows surviving nodes to dynamically select and enact a repair strategy in a fully

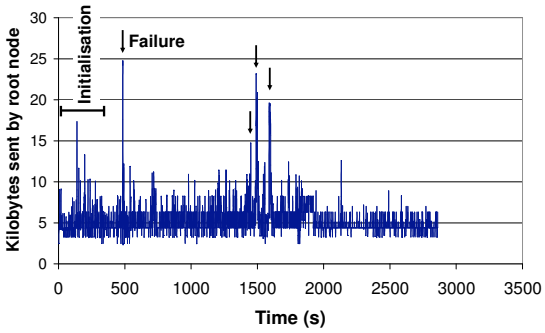


Figure 3.10: Sonar/TBCP network usage at the root node on PlanetLab.

decentralised and localised manner, and leverages distributed consensus with scoped communications (Echo) to create agreement on how to repair each failure. Building on this agreement service, Sonar provides a framework in which a range of repair strategies can safely be enacted. The approach as a whole simplifies the design and implementation of new overlays (because repair issues can be treated orthogonally to basic functionality), and can thus supports *tailorable* levels of dependability, by adapting the proposed repair strategies to runtime conditions.

The work presented bridges traditional small-scale fault-tolerance and large-scale decentralised systems. This is illustrated by the fact that although Echo is based on a traditional consensus algorithm, its message complexity is independent of the system's global size. Instead, Echo/Sonar only involve nodes bordering a failed region (*locality*), which inherently insures scalability in very large systems.

The results included also cover different forms of research: while we have proved formally the correctness of Echo under strictly defined conditions, our treatment of Sonar has been much more experimental and geared towards practical feasibility, using simulations and a prototype deployment on the PlanetLab platform.

CHAPTER 4

Programming Gossip Protocols: GossipKit and Whispers

Gossip protocols[1] have attracted a considerable amount of attention over the last decade. Their natural robustness, scalability, and self-stabilisation properties make them particularly well adapted to the needs of peer-to-peer social networks [Bertier et al. (2010)], wireless sensor networks [Haas et al. (2006)], and large-scale data-centres [Agrawal et al. (1997)]. Although the principles of gossip protocols are relatively easy to grasp, their variety, and the wide range of environments in which they can be deployed can make their design, implementation, and evaluation highly time consuming. Specifically, the lack of a unified programming model for gossip protocols means that developers can not easily reuse, compose, and adapt existing solutions to fit their needs, and limits opportunities for knowledge sharing and cross-pollination.

To address these challenges, I have looked in my research for approaches that maximise code reuse across gossip protocols, simplify the development of new and composite protocols, and lend themselves to dynamic evolution and re-deployment in a broad range of environments. This research has led me to envisage two complementary tracks, that of component frameworks, and that of distributed macro-programming languages, and to consider more particularly their synergies in terms of structure and behaviour. This chapter summarises the results of this line of research, which I per-

[1] 'Gossip' and 'epidemic' protocols are used interchangeably in the distributed system community.

formed together with my student Shen Lin and my colleague Gordon Blair from Lancaster, and with Marin Bertier and Anne-Marie Kermarrec from IRISA/INRIA Rennes. The original publications on which this chapter is based can be found in [Lin et al. (2007, 2008); Lin (2010); Lin et al. (2011)].

4.1 Structure and behaviour in gossip protocols

As pointed out by a number of authors [Jelasity and Babaoglu (2005); Kermarrec and van Steen (2007); Eugster et al. (2007)], Gossip protocols usually share a core set of common features, with key variation points determining their specific properties. An important effort of our research consisted in precisely identifying and documenting both these common features and variation points across a wide range of gossip protocols, and mapping these, first on an event-based component engine (GossipKit), and then an a specialised macro-programming language (Whispers). Our interest focused in particular on the synergies between these two technologies, through an approach we have termed *transparent componentisation*. This novel approach, which we detail in Section 4.3, provides the behavioural expressiveness of macro-programming, while preserving the reusability and composability of components.

4.1.1 Gossip protocols

In gossip protocols, individual nodes exchange data with some of their neighbours according to stochastic patterns, causing information to eventually spread through a distributed system like a "rumour" would. Gossip protocols offer three key advantages over more traditional systems: 1) they are particularly scalable; 2) they are naturally robust to node failures and message losses; and 3) they are efficient in terms of message exchanges and latencies. As a result, they have been applied to a wide range of problems such as peer sampling [Ganesh et al. (2003); Jelasity et al. (2004)], ad-hoc routing [Haas et al. (2006)], reliable multicast [Eugster et al. (2001); Birman et al. (1999)], database replication [Agrawal et al. (1997)], failure detection [van Renesse et al. (1998b)], and data aggregation [Gupta et al. (2001)].

To meet their requirements, individual protocols differ in their communication pattern (periodic or reactive); in the type of information they maintain (a measurement, a list, a dictionary); in the probability function that drives their message exchanges; and in the mechanisms they use to updates their state. Some protocols might also be composite: for instance one gossip protocol might rely on another to build and maintain its neighbourhood.

4.1.2 Component frameworks

Following on the work on configurable communication platforms [Bhatti et al. (1998); van Renesse et al. (1995)], and early proposals in the area of gossip protocols [Jelasity and Babaoglu (2005); Kermarrec and van Steen (2007); Eugster et al. (2007)], our first choice to systematise the development of gossip protocols was to consider component frameworks. Software components extend the notion of object-orientation by introducing explicit dependencies between provided and required interfaces [Szyperski (2002)]. Component frameworks add rules for composition, and structural constrains that capture the domain knowledge of a particular area. They thus encourage a compositional approach to system construction that fosters modularity, reuse, and configurability. They also facilitate the development of dynamically adaptive systems: knowledge about provided and required interfaces allows the reconfiguration logic to reason about dependencies, while dynamic bindings provide a simple mechanism to update a system at runtime. Finally, components foster knowledge sharing and reuse across systems, thus reducing both development effort and resource overhead.

These benefits make components a particularly popular approach to develop distributed platforms. They have been successfully applied both in the industry (c.f. EJB, CORBA Component Model, DCOM), and in middleware research, giving rise to lightweight component technologies (OpenCom [Coulson et al. (2004)], Fractal [Bruneton et al. (2004)]) and their associated middleware frameworks (GridKit [Grace et al. (2004)], RAPIDWare [McKinley et al. (2001)], FraSCAti [Seinturier et al. (2011)]).

4.1.3 High-level distributed programming

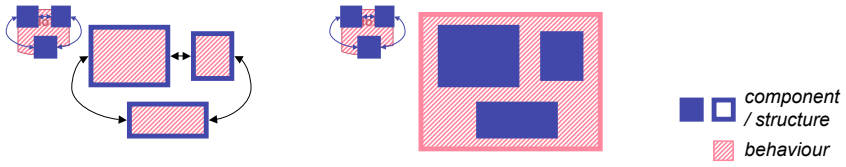
In spite of the many advantages of component-based approaches, components primarily cater for structural concerns. This can be expected to work well in situations in which each component's role is well understood, and the features of the emerging system can be easily derived from each component' characteristics. These conditions however often do not apply to distributed protocols and systems [Hiltunen et al. (2006)]. In particular, because component frameworks focus on structure rather than algorithms, they are limited in their ability to represent a system's detailed behaviour, such as execution flows and message exchanges [Clements (1996)]. This in turn makes it difficult for developers who are unfamiliar with a particular framework to understand its logic [Edwards et al. (2004)]. We have observed this effect in our own work. Our component platform GossipKit performs well in terms of configurability, reuse, and adaptability but remains radically differ-

ent from the pseudo code protocol designers typically employ when creating new gossip mechanisms.

This raised the question of whether a component framework could be augmented with alternative approaches more oriented towards the algorithmic logic of distributed systems. One inspiration for this part of our research came from the many high-level distributed languages that have been proposed over the years to specify, realise and verify distributed systems, such as Lotos [Eijk and Diaz (1989)], Estelle [Amer and Çeçeli (1990)], PLAN-P [Thibault et al. (1998)], Promela++ [Basu et al. (1997)], and Mace [Killian et al. (2007)]. A second strong inspiration was provided by the recent emergence of specialised languages for wireless sensor networks (WSNs), exemplified by systems such as Kairos, Regiment, Cougar and TinyDB [Gummadi et al. (2005); Madden et al. (2005); Newton et al. (2007); Fung et al. (2002)]. These languages, often referred to as *macro-programming* approaches, seek to hide the intricacies of distribution by abstracting a large-scale distributed system as a single programmable entity. To reach their goal, macro-programming approaches either use declarative abstractions that capture collections of nodes such as in TinyDB, or emulate a shared-memory programming model augmented with parallel constructs for distribution, such as in Kairos and Regiment. A compiler automatically translates a system-level program into deployable executable code, and shields the programmer from explicitly dealing with network messages used to access remote data.

4.1.4 Transparent componentisation: WhispersKit

While components cater for structure, macro-programming handles behaviour. Our goal was to bring these two technologies together, in order to benefit both from the advantages of components (reusability, maintainability, and adaptability), and that of macro-programming languages (expressiveness, understandability, proximity to practice of protocol designers). Structure and behaviour are two recurring concerns developers have to deal with. In some way or other, they both appear in most programming technologies. Two traditional strategies to bring these concerns together in component-based approaches consist in either (i) requiring developers to encapsulate behaviour within components (Figure 4.1a), or (ii) in extending the composition mechanism with programmatic capabilities, for instance by adding scripting capabilities to connectors and/or to the containing component engine (Figure 4.1b).



(a) encapsulation (b) orchestration

Figure 4.1: Structure and behaviour in traditional component-based approaches

Although both strategies have proved their values (encapsulation for instance is a hallmark of modular programming models, and of object-orientation in particular), they both require developers to explicitly handle the articulation of structure and behaviour in their code. This creates tensions: for instance the best granularity to leverage adaptation and reuse might not be the most appropriate to maximise expressiveness and understandability. Similarly, the orchestration features best adapted to self-organising systems might not be those that foster the most intuitive composition semantics.

These considerations have led us to explore a third way (Figure 4.2), which separates behavioural concerns from structural ones. Protocol designers, who wish to focus on a behavioural representation, can work at the level of a macro-programming language, with no or little structural constrains. The platform then takes care of converting this high-level representation into an intermediate component-based configuration. This component-based configuration can in turn draw on existing component libraries and can leverage reconfiguration approaches adapted to component-based systems. As a proof of concept of this approach, which we have termed *transparent componentisation*, and to assess its benefits, we have realised a prototype targeting gossip protocols, WhispersKit. WhispersKit is made of GossipKit, a generic and expressive component framework for gossip protocols, and Whispers, a macro-programming language based on GossipKit tailored for rapid protocol development. We detail each of them in the remainder of this chapter.

4.2 The GossipKit component framework

Identifying how best to partition a family of algorithms, here gossip protocols, into a set of generic and reusable entities is as much a craft as a science. We designed GossipKit with two principal aims: *generality* and *simplicity*.

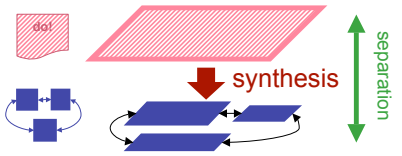


Figure 4.2: Transparent componentisation

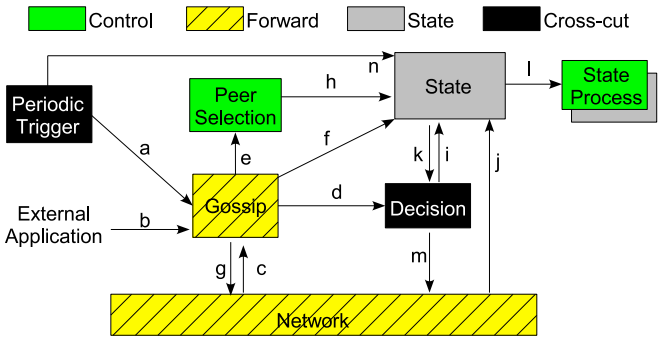


Figure 4.3: GossipKit's common architectural pattern (from [Lin (2010)])

These two traditional aims are generally at odds in component frameworks, and more generally in any approach targeting a high level of reuse. A very general framework might for instance require a large number of component types to cater for a broad range of underlying mechanisms. Alternatively, a simple framework, based on a limited number of component roles, might be applicable to only a limited family of mechanisms, or might require a large number of specific implementations for each supported protocol.

To achieve these two aims in GossipKit, we made two design choices: that of fine-grained components, to maximise the potential of one component implementation being reused in different protocols, and that of structured nested events, to simplify component interactions, while maintaining some structure in our handling of events.

4.2.1 GossipKit's common interaction pattern

The architecture of GossipKit is based on a *common interaction pattern* (Figure 4.3) that captures a recurring set of roles and interactions we have observed across a representative set of 30 gossip protocols. In Figure 4.3,

each rectangle represents a GossipKit component type (or role)[2], and each arrow a possible interactions between the connected components. As the APIs proposed in [Eugster et al. (2007)], this common interaction pattern combines the patterns identified in periodic gossip frameworks [Jelasity and Babaoglu (2005); Kermarrec and van Steen (2007)], with the reactive gossip patterns observed on gossip protocols such as [Ganesh et al. (2003)] and [Haas et al. (2006)]. In contrast to [Eugster et al. (2007)], this pattern also explicitly focuses on encapsulated and reusable entities (component types), rather than on procedural interfaces.

This pattern is both simple (in only encompasses seven component types), and highly generic: it is based on the analysis of 30 representative gossip protocols, which can all be mapped onto the pattern. As shown on the figure, it also conforms to the Control Forward State pattern generally observed in overlay protocols [Grace et al. (2004)], allowing it to readily fit into middleware that follows this architectural model [Grace et al. (2004, 2008)].

To increase opportunities for reuse, each of the roles shown in Figure 4.3 might be implemented as a composite component, made of smaller components. One particular case is when one role (e.g. the Peer Selection module) is implemented as a gossip protocol (e.g. the RPS protocol [Jelasity et al. (2007)]) that itself follows the same interaction model. Another example is the family of gossip-based ad-hoc routing protocols proposed in [Haas et al. (2006)], whose Decision role can be decomposed into a combination of three basic micro-components [Lin et al. (2008)].

GossipKit's architectural pattern covers all the steps of a typical gossip round in an gossip protocol: rounds might be triggered periodically (**a**) as in RPS [Jelasity et al. (2007)]), or started in reaction to an application event (**b**) or to an incoming gossip message (**c**), as in reactive routing protocols [Haas et al. (2006); Hou and Tipper (2004)]. The rest of the gossip round is then orchestrated by the Gossip component. First a decision is made (usually probabilistically) whether to gossip or not (**d**). If positive, a subset of neighbours is selected for communication (**e**), and finally the message is disseminated (**g**). On receiving a gossip message, the node might further update its internal data (e.g. merging neighbourhood lists) using the message's content (**j** and **l**). The state component plays a major role in this sequence, by storing the node's current neighbourhood; providing additional context for the gossiping decision and peer selection (e.g. as in [Ganesh et al. (2003)]); and storing any additional data maintained and disseminated by

[2]When the distinction is clear, we use *component* in the following as a shorthand for *component type*.

the node (e.g. a sensor value, routing tables). Finally, some of these steps are optional. For instance, in wireless sensor networks, gossip protocols typically broadcast to all neighbours within a node's radio range. In such a case, the `Peer Selection` component is not used, and the probabilistic nature of the protocol entirely relies on the `Decision` component.

4.2.2 Rich and uniform event interactions

Which interaction paradigm to use is a critical design decision of any component framework. In GossipKit we have opted for event-based interactions, following in that respect the design of earlier configurable communication platforms [Hiltunen and Schlichting (2000); van Renesse et al. (1998a); Bhatti et al. (1998)], and high-level APIs proposed for gossip-based systems [Eugster et al. (2007)]. Events are well adapted to the asynchronous interactions found in gossip protocols. They also minimise explicit coupling between modules [Bhatti et al. (1998)], which allow our framework to be easily extended by plugging in new micro-modules (i.e. event handlers) and reconfiguring event bindings to support new interaction patterns.

GossipKit uses rich events that carry a number of contextual parameters needed by each component (e.g. protocol ID, event source, data payload). GossipKit events also feature two innovations: first the same event mechanism is used for both local and remote interactions, i.e. whether the involved components reside within the same address space, or on different machines. This allows for a uniform interaction model, that naturally captures the distributed nature of gossip protocols. Second, events can be nested into compound events, to express complex event sequences at different levels of abstractions.

4.2.3 Evaluation

Our evaluation of GossipKit is based on a prototype implementation in Java, and builds upon the OpenCom component engine developed at Lancaster [Clarke et al. (2001b)], a lightweight and reflective component engine. The prototype is available on-line[3], and relies on events (rather than threads) for an efficient handling of concurrency. By design, the GossipKit prototype naturally supports the execution of composite and co-existing gossip protocols, a feature we have used for instance to investigate synergies in overlay networks [Lin et al. (2009)] (not detailed in this document). A GossipKit configuration is declared as an XML file that is parsed by the

[3] `ftaiani.ouvaton.org/GossipKit/`

Protocol	Reused (LoC)	Specific (LoC)	Reuse Rate
Gossip1 [Haas et al. (2006)]	626	134	82.3%
Gossip2 [Haas et al. (2006)]	626	138	81.9%
SCAMP [Ganesh et al. (2003)]	888	120	88.1%
RPS [Jelasity et al. (2007)]	1221	0	100%
Anti Entropy [Demers et al. (1987)]	1349	56	96.0%
Averaging [Jelasity et al. (2005)]	1102	152	87.9%
Ordered Slicing [Jelasity and Kermarrec (2006)]	1102	178	86.1%
T-Man [Jelasity and Babaoglu (2005)]	1144	309	78.7%
Average	**1007**	**136**	**88.1%**

Table 4.1: Reused achieved by GossipKit (from [Lin (2010)])

GossipKit engine and specifies which component to instantiate, and which event interactions should take place between these components.

Evaluating a component framework's suitability to a particular domain (here gossip protocols) is always a somewhat disputable task. What constitute an appropriate measure of generality? How should one assess simplicity? Our first step was to refine our two original aims of generality and simplicity (Section 4.2) into more concrete properties: size and complexity—for simplicity, and configurability, reusability, reconfigurability, and expressiveness for generality. We also implemented a representative set of eight diverse gossip protocols, with GossipKit and directly in Java, to serve as a reference point for comparison. In the following we focus on reuse (as a quantitative measure of GossipKit's ability to factorise common code, and thus capitalise on development efforts), and size (as a surrogate for simplicity). The full detail of our evaluations can be found in the relevant publications [Lin (2010); Lin et al. (2008, 2007)].

Table 4.1 compares for each target protocols the amount of component code shared with at least another protocol against component code unique to this particular protocol. Although the study only covers 8 protocols, the reuse rate (defined as the proportion of reused component code) ranges from 100% (RPS) to 78.7% (T-Man), and remains particularly high, with a weighted average of 88.1%. Furthermore, reuse would likely increase if additional protocols were added, as more commonalities would show up within a larger protocol population.

As a surrogate for simplicity, we also compared for each protocol the size of the GossipKit configuration file against that of a direct Java implementation of the same protocol (Table 4.2). If one assumes, as is reasonable to believe here for XML and Java, that programming efforts are roughly pro-

Protocol	GossipKit (XML LoC)	Java (LoC)	Ratio	XML +Specific	Ratio
Gossip1	39	277	14.1%	173	62.5%
Gossip2	39	279	14.0%	177	63.4%
SCAMP	88	463	19.0%	208	44.9%
RPS	81	439	18.5%	81	18.5%
Anti Entropy	100	544	18.4%	156	28.7%
Averaging	85	466	18.2%	237	50.9%
Ordered Slicing	85	471	18.0%	263	55.8%
T-Man	93	491	18.9%	402	81.9%
Average	**76.3**	**424**	**18.0%**	**212.3**	**50.1%**

Table 4.2: GossipKit's implementation effort (in LoC) compared to Java

portional to code size, GossipKit's declarative approach allows for a much more direct construction of a protocol (code about five times smaller) than a Java implementation. Even when one adds the part of the component from Table 4.1 that is specific to each protocol, the effort requires by GossipKit remains about half that of Java (last column). This represents however an extreme case, as the specific parts would be likely to decrease with more protocols added, as discussed above. We return to simplicity and expressiveness (the ability to express advanced behaviour in an easily comprehensible manner) when we will discuss Whispers' evaluation in Section 4.3.3 on page 60.

In addition to reuse and compactness, GossipKit also brings the traditional advantages associated with component frameworks, such as the ability to easily reason about configurations, and the basic mechanisms to reconfigure a running deployment. We come back in particular to dynamic adaptation in the next section, where we discuss the synergies between GossipKit and our macro-programming solution Whispers.

4.3 Transparent componentisation

In spite of its benefits, GossipKit's style of programming remains quite different from the approach favoured by designers of new epidemic protocols, as documented in the literature. Configuring GossipKit is mainly a declarative task that focuses on components and event-based connections, while protocol designers usually prefer a more algorithmic approach, based on pseudo-code.

To leverage the advantages of GossipKit's components (reuse, structural reasoning, dynamic adaptation), we have therefore explored how a component-based development framework (GossipKit) could be combined

productively with a higher-level representation, here a macro-programming language tailored for epidemic protocols (Whispers). The resulting platform (WhispersKit) illustrates a more general design strategy we have termed *transparent componentisation* (Section 4.1.4), that seeks to uncouple as much as possible behavioural concerns from structural ones.

4.3.1 The Whispers language

Whispers is a macro-programming language, strongly inspired from solutions such as Kairos [Gummadi et al. (2005)] that presents developers with a shared-memory metaphor of a peer-to-peer systems, and provides a number of useful constructs to program gossip protocols.

As an illustration of Whispers' capabilities, Figure 4.4 shows the Whispers source code of the RPS protocol [Jelasity et al. (2007)], Figure 4.5 that of the wireless gossip1 protocol [Haas et al. (2006)], and Figure 4.6 lists Whispers' most important programming constructs.

We designed Whispers based on the understanding of gossip protocols gained through GossipKit, with transparent componentisation in mind (Section 4.1.4 on page 50). The main challenge in this context consisted in translating the core concepts of GossipKit into a different programming paradigm (an imperative language), while maintaining an explicit reverse mapping to allow any Whispers program to be translated into a GossipKit configuration. The *controlled mapping* we have used encompasses three cases:

Direct Exposition. Some GossipKit component types, whose function is well identified (`Peer Selection` and `State Process`), are directly exposed in Whispers as method calls on `Node` objects. That's the case of `RandomPeerSelection` and `RandomStateCompress` in Figure 4.4, which realise the component types `State Process` and `Peer Selection`, respectively.

Blocks. The execution sequences and control flows involved in a gossip protocol are captured as traditional programmatic constructs in Whispers, to increase the programmability and the understandability of the language. This is the case for instance of the `every` (Figure 4.4) and `wait` blocks (Figure 4.5), which capture the effect of the `Periodic Trigger` component, and of external events, respectively. This also applies to conditional `if` statements, which are automatically synthesised into `Decision` components.

```
RPS {
  State sample = new State[Node:PeerID][Size=5];
  Node n, i;
  every (5000) { // do the following every 5000 ms
    foreach (n in AllNodes) { // for each node n
      i=n.RandomPeerSelection(n.sample)[Size=1];
      n.sample.add([n]);
      i.RandomStateCompress(i.sample,n.sample)[Size=5];
      n.RandomStateCompress(i.sample,n.sample)[Size=5];
    } // end of foreach
  } // end of every
} // end of RPS protocol block
```

Figure 4.4: Whispers program of the RPS protocol [Jelasity et al. (2007)]

```
Gossip1 {
  State state = new State[Event:Packet][Size=10];
  wait(Event evt type Forward) {
    if(!this.state.contains(evt.ID) &
       (Math.random()<0.6 | evt.HopCount<4))  {
      evt.HopCount++;
      this.broadcast(evt);
    } // end if
    if (!this.state.contains(evt.ID)) {
      this.state.add([evt]);
    } // end if
  } // end of wait block
} // end of Gossip1
```

Figure 4.5: Whispers program of the Gossip1 protocol [Haas et al. (2006)]

```
SomeProtocol {..}           // protocol block
every (time) {..}           // periodic behaviours
wait (Event e type T) {..} // reactive behaviours
foreach(n in nodeSet)       // distribution
synchronised {..}           // pairwise data exchange
State state = new State[fields][size] ; // state decl.
state.field ;               // get a column of data
state.add([fields])         // add
state.remove(row_ID)        // remove
i.RandomStateCompress(...) // library call
```

Figure 4.6: The language primitives of Whispers

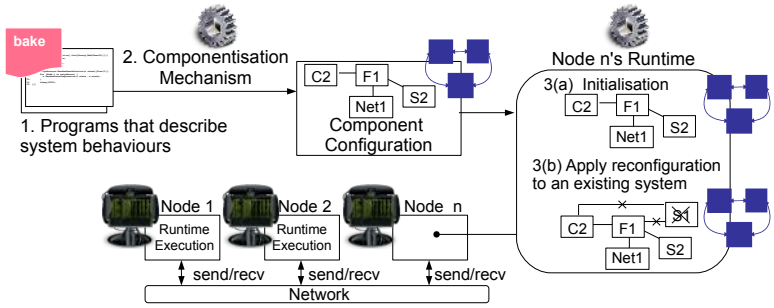


Figure 4.7: Deployment Process in WhispersKit

Types. Finally, some Whispers concepts are exposed as specialised types, such as `State` or `Node`. State declaration directly maps onto `State` components, while `Node` variables describe a locus of execution and as such might involve the `Network` component (in case of a remote invocation as in Figure 4.4), or other components directly exposed as libraries.

Finally, because a gossip system typically involves multiple collaborative protocols, Whispers provides high-level expressions to describe interactions between gossip protocols. These expressions can then be automatically translated into a combination of coexisting gossip protocols in GossipKit.

4.3.2 Synthesis and deployment

The Whispers language forms the start of a compiling tool chain, WhispersKit, that transforms and deploys Whispers programs as GossipKit configurations (Figure 4.7). Whispers programs (Step **1**) are compiled by the componentisation mechanism (Step **2**) (based on JavaCC[4]) into a GossipKit configuration. The componentisation mechanism also generates any protocol-specific components that might be required (essentially `Decision` components) by the original Whispers program. The resulting configuration file and generated components are disseminated to each node's runtime, where it gets deployed (Steps **3a** and **3b**).

The componentisation mechanism. The Whispers compiler uses a two-step process that first transforms a Whispers program into a set of direct acyclic graphs representing the behaviour of individual nodes (shown

[4] `http://javacc.java.net/`

in Figure 4.8 for the RPS protocol of Figure 4.4). This intermediate representation exposes low-level programming details such as threading, message handling, data synchronisation, remote data access, and network interface management for sending or receiving messages. It is used in a second step to generate the final GossipKit configuration (Figure 4.9), and generated components (if any).

Deployment and dynamic reconfiguration. The deployment of the initial GossipKit configuration generated by WhispersKit is assumed to occur at installation time, when each node is set up (Step **3a** in Figure 4.7). Once an initial configuration is running, WhispersKit provides a distributed reconfiguration service (Step **3b**), which piggybacks reconfiguration requests on the messages of the currently running gossip protocol(s). When receiving such a request, each node compares its current architecture with the target one, and computes an optimised set of local transformations (parameter change, component instantiations and bindings) to realise the new configuration.

This mechanism directly benefits from the self-stabilisation of gossip protocols, which allows the system to tolerate transient states in which nodes do not all execute the same configuration. It also clearly demonstrates the advantages of transparent componentisation: while developers work at the level of Whispers programs, and do not need to worry about the scope and granularity of GossipKit components, the platform is able to leverage the component structure of GossipKit to reason about change and evolution.

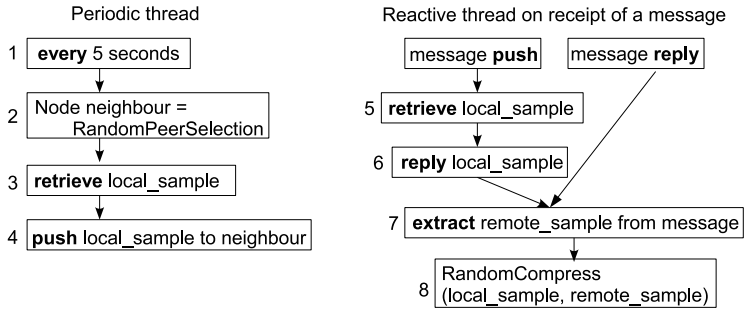


Figure 4.8: Per-node program of RPS [Lin et al. (2011)]

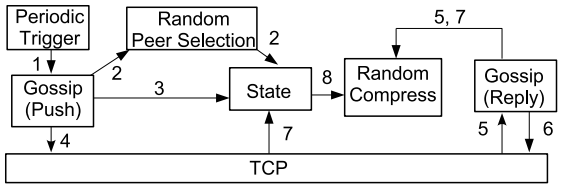
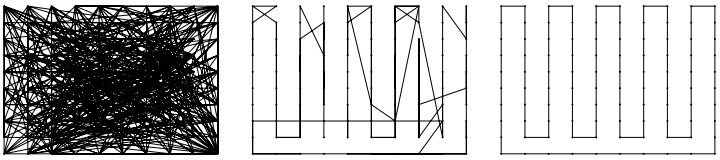
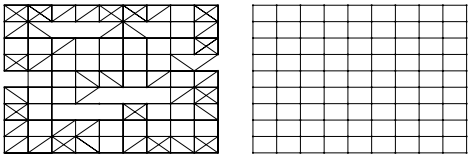


Figure 4.9: Component realisation of RPS [Lin et al. (2011)]



(a) RPS alone (b) RPS+TMan (ring), 5^{th} round (c) RPS+TMan (ring), 11^{th} round



(d) RPS+TMan (grid), 20^{th} round (e) RPS+TMan (grid), 29^{th} round

Figure 4.10: Dynamic reconfigurations with WhispersKit [Lin et al. (2011)]

4.3.3 Evaluation

The following discusses a reconfiguration scenario with WhispersKit, and then focuses in more detail on a quantitative assessment of the simplicity of the Whispers language. We have deliberately left performance measurements out of scope, which can be found in [Lin et al. (2011)].

Dynamic reconfiguration. Figure 4.10 demonstrates on a simple scenario how WhispersKit is able to reconfigure a set of distributed nodes executing gossip protocols. The scenario involves 100 nodes simulated on the Jist/SWANS network simulator [Barr et al. (2005)]. Each node is initiated with the RPS protocol (Figure 4.10a). The first reconfiguration consists in launching an implementation of T-Man [Jelasity and Babaoglu (2005)] to

construct a *ring* topology (Figure 4.10b). Because T-Man relies on RPS to sample peers, the WhispersKit compiler includes RPS in the generated configuration, and each node's runtime then takes care of instantiating T-Man on top of the already running instance of RPS. Once the ring topology has converged (Figure 4.10c), a second reconfiguration is triggered that uses a modification of T-Man to build a *grid* topology (Figures 4.10d and 4.10e).

The two reconfigurations occur at two levels of granularity, both transparently supported by WhispersKit. The first reconfiguration is coarse-grained: it deploys an entirely new protocol (i.e. T-Man) atop RPS, injecting 8 new components and 10 new bindings. By contrast, the second reconfiguration is fine-grained, and only involves the `State Process` component of T-Man and two bindings. Thanks to the Transparent Componentisation provided by WhispersKit, developers do not need to worry how coarse- or fine-grained a reconfiguration is in practice, or which architectural dependencies should be taken care of. These concerns are automatically addressed by the underlying tool chain, on the sole basis of the provided Whispers programs.

Simplicity. Table 4.3 shows the length of Whispers programs for the eight protocols we used for our evaluation, together with the size of the corresponding GossipKit configuration, and the size of a direct java implementation. Whispers programs, thanks to their imperative approach and high-level distribution constructs, are substantially shorter than GossipKit configurations (20.2% of the size on average), and much shorter than their Java counterpart (3.6%).

Protocol	Whispers	GossipKit	**Ratio**	**Java**	**Ratio**
Gossip1	14	39	35.9%	277	5.1%
Gossip2	14	39	35.9%	279	5.0%
Anti Entropy	16	100	16.0%	544	2.9%
Averaging	14	85	16.5%	466	3.0%
Ordered Slicing	14	85	16.5%	471	3.0%
RPS	12	81	14.8%	439	2.7%
SCAMP	19	88	21.6%	463	4.1%
T-Man	20	93	21.5%	491	4.1%
Average	**15.4**	**76.3**	**20.2%**	**424**	**3.6%**

Table 4.3: Whispers program sizes vs. GossipKit and java (LoC)

Comparing heterogeneous programming technologies (here a macro-programming languages, a component framework, and a general purpose language) simply in term of code size can however be misleading. Both Java and

Protocol	Whispers (Cyclomatic)	Java Comp.)	GossipKit Comp.	Param.	Bindings
Gossip1	2	11	5	6	7
Gossip2	2	11	5	6	7
Anti Entropy	3	10	9	15	13
Averaging	3	6	8	12	11
Ordered Slicing	3	11	8	12	11
RPS	2	12	7	15	10
SCAMP	3	20	8	10	12
T-Man	3	11	8	15	12
Average	**2.6**	**11.5**	**7.3**	**11.4**	**10.4**

Table 4.4: Whispers programs vs. Java and GossipKit [Lin (2010)]

XML are known for instance for their verbosity, but more lines (in particular if they are short and only involve simple operations) do not necessarily make a program harder to understand. One could even argue that a too concise program, in particular if it relies on dense and unintelligible code, might be harder to read and understand than a more expanded version. Simply comparing size becomes even more problematic when it involves different programming paradigms: while Whispers and Java, being imperative, share a number of commonalities, GossipKit configuration files, being declarative, differ markedly.

To address these limitations of raw program size, Table 4.4 presents alternative measures of the same protocol implementations: cyclomatic complexity [McCabe (1976)] for Whispers and Java, and the number of components, parameters and bindings for GossipKit (Table 4.4). If one considers a cycle to represent a basic unit of understanding, somewhat comparable to a components or binding, it appears that Whispers, with an average cyclomatic complexity of 2.6, maintains here also its advantage over Java (11.5) and GossipKit (7.3 components and 10.4 bindings on average).

4.4 Conclusion

GossipKit has demonstrated that gossip protocols could greatly benefit from a component-based development approach. Our results open the path in particular for the application to gossip systems of higher-level development processes, such as software product lines [Paul Clements (2001)]. This research also sheds light on the inherent structures and interaction patterns of gossip protocols, offering a unified conceptual and implementation frame-

work in which to design, implement, and analyse gossip protocols. We have started to explore this area ourselves, by using the common interaction pattern provides by GossipKit to investigate synergies in co-existing overlays [Lin et al. (2009)].

The combination of Whispers and GossipKit shows how two heterogeneous programming paradigms, components and macro-programming languages, can be brought together to harness the strengths of each approach. Our work with the Whispers language more generally raises the issue of the role of structure and behaviour in programs, and how they might be addressed at different levels of the compilation and deployment tool-chain to provide an optimal combination of understandability, expressiveness, and flexibility to developers in a particular domain.

CHAPTER 5

Anomaly Diagnosis and Understanding in Complex Systems

The works presented in Chapters 3 and 4 both attempt to propose a *unified view* of distributed systems, either by proposing a generic mechanism for fault-tolerant overlays (Chapter 3), or by developing a reusable component framework for gossip protocols (Chapter 4). Many modern distributed systems are however far from being unified. Instead, they commonly involve parts (middleware, libraries, OS) developed by distinct teams in independent organisations. They are continuously expanded and corrected; and for many end up resembling a living organism, constantly evolving in a loosely controlled manner.

This type of system would not be possible without appropriate software technologies that provide the flexibility, interoperability, and separation of concerns they require. In terms of distributed interactions, these technologies range from distributed object technologies such as Java RMI or CORBA [OMG (2008)], proposed in mid 80's and 90's, to web services [Chinnici et al. (2007)], and REST architectures [Fielding and Taylor (2002)] proposed over the last 15 years. Many of these technologies are available as reusable software elements (libraries, tools, frameworks), some of which rely on advanced composition technologies (e.g. aspects and reflective architectures) to interface with the rest of the system [Fleury and Reverbel (2003); Colyer and Clement (2004)].

These technologies allow for a loosely coupled, evolving, and distributed development, where reuse and abstraction (in the form of APIs) play a critical role. Unfortunately, this constant evolution and somewhat organically

grown structures can also lead to a 'Frankenstein' effect, in which no single person thoroughly understands all finer details of a large system. When this happens, undesirable emergent behaviours (unexpected performance drops, unstable behaviours leading to crashes, faulty synchronisation between different parts of the program leading to deadlocks) become particularly hard to trace, and often need to be masked or hastily dealt with on an ad-hoc basis.

The reason abnormal behaviours are particularly difficult to diagnose in high-reuse distributed systems is, we argue, that they typically involve both synergistic interactions between a system's many parts, and microscopic interferences within the system's lowest levels. System architects, who possess an overview of the fundamental structure of the system, lack the necessary understanding of the system's finer details. Conversely specialist developers, who have an in-depth understanding of individual components, do not have the 'architectural' overview needed to identify the stream of interactions that produces these undesirable behaviours.

One strand of research to address this problem involves the dynamic reverse engineering of a system's behaviour based on its observation at runtime [Systä et al. (2001); Jerding et al. (1997); Richner and Ducasse (1999); Taïani et al. (2009)]. Another trend consists in automatically localising faults, using data obtained from a program's execution (system logs, dumps, network latencies) [Xu et al. (2009); Chen et al. (2009); Zamfir and Candea (2010)] to identify patterns and executions that lead to failures. Unfortunately, dynamic reverse-engineering techniques often involve complex data manipulation languages [Systä et al. (2001); Reiss and Renieris (2003)] that can represent a steep learning curve, and do not take into account quantitative measurements such as resource usage, or exception density. Similarly, approaches for automatic fault location can considerably assist diagnosis, but—except in simple or small-case examples—usually assume some good knowledge of the underlying platform to interpret and analyse the output they produce.

In this context, we have sought to better understand the problems faced by developers who must diagnose anomalies in high-reuse unfamiliar software. We have started this work with an in-depth analysis of the impact of web-service technologies [Gudgin et al. (2007); Chinnici et al. (2007)] on the performance of the grid middleware Globus [Foster et al. (2005)]. Building on this concrete experience, we have then developed an interactive tool for performance diagnosis, ProfVis, that combine architectural reverse engineering with performance analysis. ProfVis is the result an interdisciplinary collaboration with psychologists from the Department of Psychology

of Lancaster, and provides a simple visual interface to navigate the performance traces of large unfamiliar software. Its evaluation is based on a user study to observe how developers react to our technology, which we think is an essential step for this type of research.

This chapter summarises the results of this research, which I performed in collaboration with my colleagues Rick Schlichting and Matti Hiltunen from AT&T, and Shen Lin, Tom Ormerod and Linden Ball from Lancaster. The original publications on which this chapter is based can be found in [Lin et al. (2010); Taïani et al. (2005b)].

In the following, Section 5.1 presents our work on web-services and Globus, which motivated the rest of the research. Section 5.2 then moves on to our interactive performance analysis tool, ProfVis, and Section 5.3 concludes the chapter.

5.1 Complexity and performance: Globus

Globus is one of the reference implementations for grid computing. While Globus originally started as a set of distributed tools implemented in C and then Java, it experienced a drastic architectural turn in 2004/2005 when its Java implementation was recast into a pure service-oriented architecture (SAO) based on the Web-Service stack (WS-*) of the World-Wide-Web consortium (W3C) [Foster et al. (2005)]. The first version (3.9.x) to feature this new SOA structure was unveiled within a few months of the announcement of the new direction of the project, a substantial development feat considering the Java code base at the time already contained more than 100,000 lines of code.

This swift transition towards Web-Services was in large part enabled by the availability of reusable software elements (libraries, tools) implementing the essential mechanisms needed to transform Globus into a set of networked services. These libraries, mainly from the Apache foundation, provided the Globus development team with the ability to rapidly integrate technologies such as XML, SOAP, WSDL (the web service description language), and WSRF (the web service resource framework [Graham et al. (2004)], which provides stateful operations to web-services) into Globus under a short period. Their use perfectly illustrates the benefit of reuse and appropriate encapsulation of standard functionalities.

Because of the speed of this architectural transformation (a few months), Globus also provides a very good case study of the challenges created by current reuse practices in production middleware. While the functional APIs of mature third party libraries are usually reasonably well documented, their

non-functional characteristics (e.g. performance, robustness, reliability) are much harder to gauge on paper. In turned out, for instance, that the SOA version of Globus exhibited extremely long start-up times, without there being any clear explanation of why this was the case. Anecdotal discussions with Globus users (themselves most of the time developers) suggest that they too were at a loss and had little spare resources to dive into the depth of the platform and understand the source of its poor performance.

Analysing the performance impact of Web Services in Globus brought two contributions:

- At a technological level, this work illustrates the challenges of introducing a novel middleware technology into a production-level distributed platform. In particular, this demonstrates in very concrete terms the potential side effects that appear when realising a substantial piece of software from third party elements, under quite a short time.

- At a methodological level, this study also highlights the need developers have of analysing dynamic properties in unfamiliar software. It thus lays the foundation for our follow-up research on the navigability of performance traces, presented in Section 5.2.

In the following, we first give an overview of the experimental methodology we used to analyse the performance of Globus (Section 5.1.1), and then provide a summary of our results (Section 5.1.2). A more complete description of these results can be found in [Taïani et al. (2005b)].

5.1.1 Between black and grey: an hybrid approach

Analysing the performance of a complex software platform requires observing it at runtime. Unfortunately, a detailed and fined-grained observation will tend to disturb the target system, sometimes to the point of rendering the results meaningless. To overcome this inherent tension between observation and interference, we adopted a progressive strategy combining a black and grey box approach to explore the link between the architecture of Globus and its observable performance. Black box profiling is particularly basic, and incurs no interference with the server part of the platform. The execution times and latencies obtained with this approach are thus fairly representative of what one would get in a production environment, assuming a similar workload and hardware set-up.

Black box profiling is however limited, in that it does not provide any insights on the *causes* of observed execution times. In order to better analyse how individual technologies were impacting the performance of Globus, we

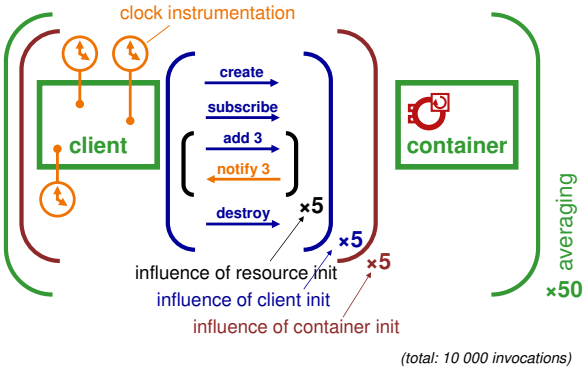


Figure 5.1: Black box profiling: experimental set-up

therefore used a complementary technique, known as sample-based profiling [Zhang et al. (1997); Anderson et al. (1997)]. A sampled-based profiling tool periodically interrupts an application and captures the state of the currently active thread. This state may be limited to the currently executing function, or might include additional context information, from the direct caller (as in gprof [Graham et al. (1983)]), up to the full call path from the thread's starting point (e.g. hprof [O'Hair (2004)], which we have used, or STAT from the Paradyn project [Arnold et al. (2007)]). The granularity of the obtained data is usually coarser than with approaches based on an exhaustive instrumentation of the source code, but sampling-based approaches are much less invasive and impose far less overhead.

Black-box profiling: tracking lazy initialisation

The main challenge in designing our black-box profiling campaign consisted in capturing all cases of *lazy initialisation* present in Globus. Lazy initialisation is directly linked to the extreme flexibility of the web service framework used by Globus. Service implementations are loaded on demand using numerous customisation files (e.g., deployment files, service description files) that are written in various XML dialects, such as the *Web Service Definition Language*, the *Web Service Deployment Descriptor language*, and the *Resource Specification Language*. Because of this flexibility, many initialisation steps do not occur until they are actually required. As a consequence, the first execution of an operation usually takes much more time than its successors. This effect needs to be precisely understood, in particularly for dynamic execution scenarios, such as elastic computing provisioning, in

which containers or resources might be regularly launched and destroyed according to demand.

The resulting experimental set-up we used to analyse this effect is shown in Figure 5.1. It involves a set of fundamental operations on a Globus entity known as a *container* (essentially a server), to create, update, and destroy a simple counter (implemented as WSRF resource). In addition, the client subscribes to *notifications* by the server on the counter just created, so that any change on the counter causes the server to call back the client ('notify' in Figure 5.1). Simple timer probes measure the end-to-end latency of each individual operation, from the client's point of view. All measurements probes are located on the client, so as to avoid any interference with the server. Sequences of interactions (e.g. creating, manipulating, and destroying a counter) are repeated in a set of nested loops to precisely analyse the influence of lazy initialisation steps on Globus' performance.

Sample-based profiling: the challenges of representation

Our sample-based profiling campaign used a similar workload to that of Figure 5.1, albeit with one single client and container to limit data explosion (shown in Figure 5.2). The primary challenge when using sample-based profiling is however dealing with the volume of data. This problem is further compounded when working on an unfamiliar code, as was Globus for us. Our experiment, although relatively limited, returned more than 55550 method invocations, distributed over 32 threads, and involving up to 1861 methods, 724 classes, and 182 different Java packages.

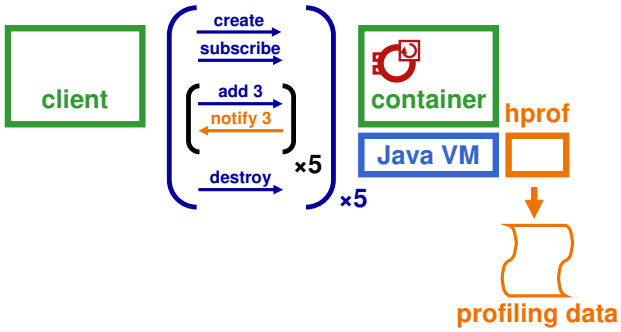


Figure 5.2: Workload of the sample-based profiling campaign

Our approach to deal with this mass of unfamiliar data was to adapt techniques proposed for software visualisation [Pauw et al. (2002, 1993); Reiss and Renieris (2000)] to our context.

The resulting strategy is best explained on a small toy example, such as that Figure 5.3, which shows in a graph form the data returned by a typical sample-based profiling tool. In this example, an hypothetical biology simulation program has been sampled six times, and three different stacks have been captured (the stacks are shown in a call graph for clarity). In 3 out of the 6 samples, the active thread of the program was executing the first stack; in one sample, it was executing the second; and in two samples, the third. As can be seen, using stacks to capture the program's execution state indicates which methods the program is actually executing (e.g. `lib3.Signal.travel` in 5 out of 6 samples). Most importantly, it also indicates *on behalf of whom* these methods are being executed. On the example of Figure 5.3, the use of `lib3.Signal.travel` entirely results from two methods of the class `lib2.Muscle`: `contract` and `stop`.

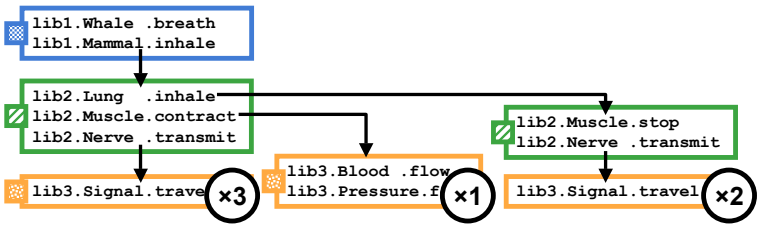


Figure 5.3: An example sampling result

To represent the data of Figure 5.3 in a more compact form, and raise its level of abstraction, we took inspiration from Jinsight [Pauw et al. (2002)], where an execution trace is represented on a *stack-depth* × *time-line* diagram in which stack frames are colour-coded according to their class. More precisely we adapted this representation in two major ways:

Package-Level Granularity. Due to the large number of classes present in our traces (724), our representation uses package-based rather than class-based colour codes. This is quite flexible since nested packages might be grouped together with their parent depending on the level of granularity required.

Depth × Weight Projection. Since sample-based profiling does not provide ordering information on the observed stack traces, a time-line representation is not appropriate. Instead, to make the internal layering

of the Globus container apparent, we represent each Globus package according to the frequency and the stack depths at which it occurs in the sampled traces.

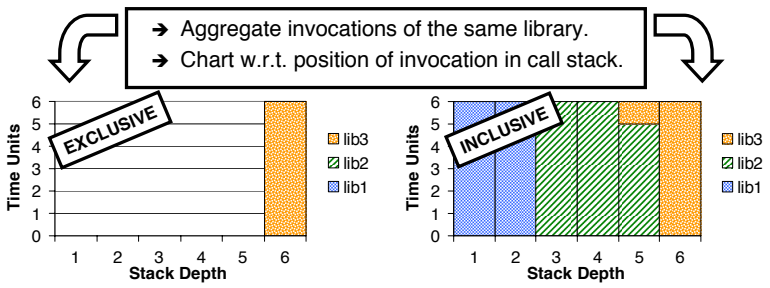


Figure 5.4: Graphical projection of figure 5.3

Figure 5.4 illustrates our presentation technique based on the toy example of figure 5.3. The diagrams indicate how often a given package was encountered in the samples at a given stack depth. Two diagrams are shown. The first represents exclusive weights, i.e., only the last invocation of each stack is considered. Here, only the library `lib3` appears, since in all six samples of figure 5.3, the active thread is executing inside `lib3` at a call depth of 6. This diagram fails however to capture *on behalf of which higher-level packages* `lib3` is executed. The right hand diagram does just that. It takes into consideration all the frames of all sampled stacks, and presents the packages according to their depth and to the weight of the individual stacks. In this second diagram, the surface of each library in the diagram is proportional to the time spent by the processor in the library or in code called by this library at execution time.

5.1.2 Results and analysis

For brevity's sake, we only provide here a high-level summary of the results obtained with the experimental approach we have just described. A detailed account can be found in [Taïani et al. (2005b)].

Measuring the influence of lazy initialisation

Our black box measurements clearly demonstrated that execution times in Globus were heavily influenced by the initialisation state of the different entities involved. With our settings, the first notification sent by a container

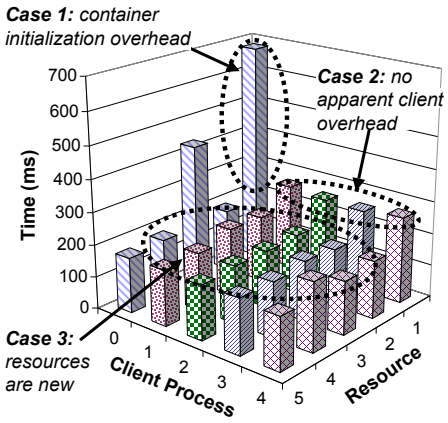


Figure 5.5: First add requests

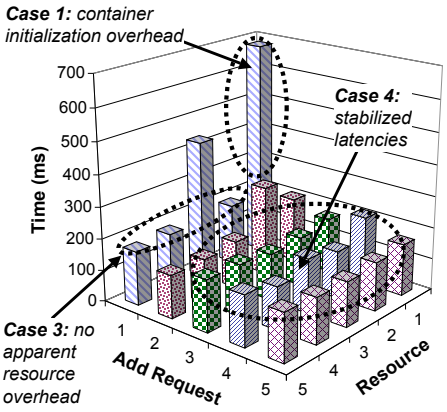


Figure 5.6: Add requests of first client

experienced a 1700% overhead (3029 ms) over the stabilised latencies for the same operation (176 ms). Similarly, the creation of a resource took 8700% longer the first time it was invoked compared to when the system had stabilised (33425 ms compared to 384 ms). All latencies were measured after the container and client had completed their initial set up, indicating that the overhead must be caused by this lazy initialisation activity. In addition, the experiments showed that even the stabilised latencies were quite high: 159 ms for a round-trip `add` request and 176 ms for a notification message. These figures do not include any network delays since both the client and the service were on the same machine.

As a representative case, Figures 5.5 and 5.6 show measured latencies for `add` operations on a WSRF resource (a sort of distributed object). Figure 5.5 shows the latency of the first `add` request on each of the 25 resources created (i.e., 5 resources for each of the 5 clients). Figure 5.6 shows the latencies of the 25 `add` requests made by the *first* client only (i.e., 5 requests on each of the 5 resources the client creates). (Note that the data in the figures overlap: the Client 0 row of the Case 3 area in figure 5.5 is duplicated in figure 5.6.)

By comparing the latency of Case 1 (new container, new client, new resource) with that of Case 4 (stabilised latencies in Figure 5.6), we can infer from figure 5.5 that some lazy initialisation is triggered by the container when an `add` request is executed for the first time, the effect being roughly 421 ms. Similarly the two figures shows that, for add requests, neither the client nor the resources seem to cause any significant lazy initialisation.

Table 5.1 summarises these latencies for `add` and `notifications` requests. The last three columns of the table evaluate how much lazy initialisation is triggered by each of the entities involved (Container, Client, and Resource). These numbers clearly illustrate the lazy-initialisation effect related to resources for change notifications. Specifically, while the average stabilised latency for notification is 176 ms (Case 4), the first notifications for Resources 2 to 5 (Case 3) take an average of 1110 ms. Hence, the overhead is approximately 934 ms. As this is just under that caused by client initialisation (1487 ms, column before last), this hints that much of the initialisation work such as deployment and hot-plugging occurs when a resource is actually instantiated, and is repeated regardless of the use of prior resources of the same type.

	New Cont. New Client New Res.	Old Cont. New Client New Res.	Old Cont. Old Client New Res.	Old Cont. Old Client Old Res.	**Overhead**		
	Case 1	**Case 2**	**Case 3**	**Stable**	Cont.	Client	Res.
Add latency	682	261	194	159	**421**	**66**	**36**
Notification	3029	2597	1110	176	**432**	**1487**	**934**

Table 5.1: Lazy initialisation caused by the container, the clients, and the resources for add and notification operations

Analysing the impact of individual WS-* technologies

Figure 5.7 shows the results of a sample-based profiling of a Globus container with the workload of Figure 5.2 (with samples corresponding to blocked server sockets removed for clarity's sake, since they lay outside the execution path for a client request).

The diagram reveals three areas of interest (marked 1 to 3 in the Figure). None of them can be fully understood by solely looking at the diagram, but the compact presentation it offers provides a critical orientation map to direct further analysis. The first area (marked 1) corresponds to stacks stopping at depth 13 in the `java.net` package. Further analysis reveals that these stacks belong to the handling of notifications by the server, when the server waits for a acknowledgement by the client that the notification has been received.

The second area (marked 2) shows a clearly layered structure, with most packages spanning less that a few depth levels. The different layers mir-

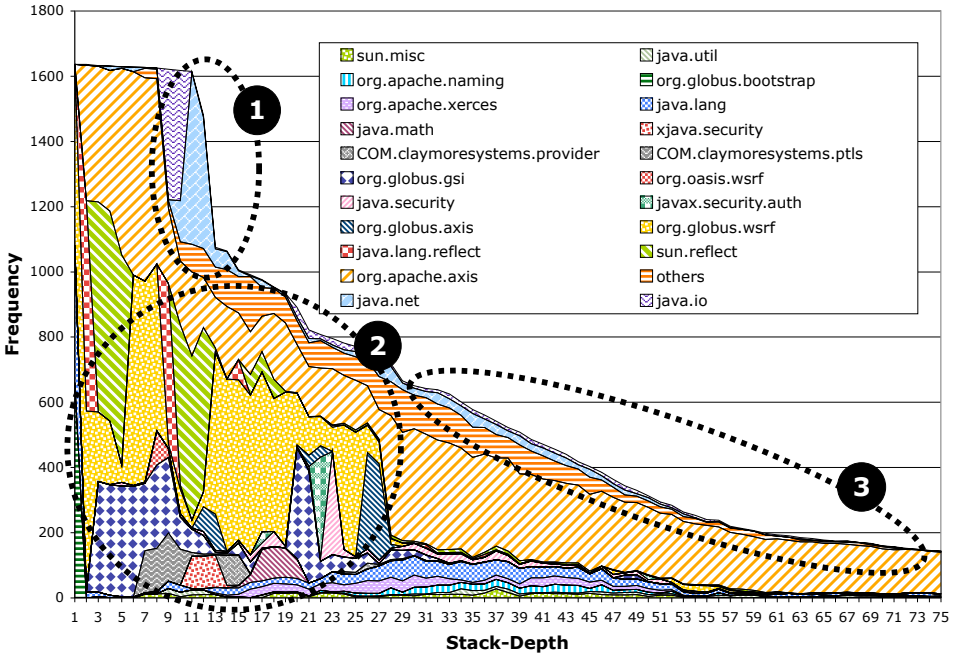


Figure 5.7: Performance sampling of the Globus container (inclusive)

ror the internal organisation of the Globus middleware, and highlight the weight of various libraries in the execution of the platform. It clearly shows the important role of played by reflection (`sun.reflect`) (used by Globus to locate appropriate implementations), the Globus Security Infrastructure (`org.globus.gsi`), and Axis[1] (`org.apache`.axis), a core library from the Apache foundation to provide a bridge between Java and web-service RPCs (known as SOAP-RPC [Gudgin et al. (2007)]).

Most interesting, however, is the area marked 3, from depth 28 onward. Any layered structure disappears in this part of the graph, with `org.apache.axis` representing most of the activity at all stack depths. The recorded stacks go as deep as 108 nested invocations, which is quite noteworthy, even for a reasonably complex piece of middleware. The relative weight of each package remains quite regular, and as the decreasing slope of the total number of samples for each depth suggests, stacks have been sampled regularly at all lengths from 28 to 75 and beyond. Further analysis reveals that activity in the package `org.apache.axis.wsdl.symbolTable`

[1] `http://ws.apache.org/axis/`

Package	count	%
org.apache.axis.wsdl	231	21%
org.apache.axis.encoding	66	6%
org.apache.axis (others)	113	10%
org.globus.gsi	249	23%
org.globus.wsrf	49	4%
cryptix.provider.rsa	82	7%
org.apache.xerces	78	7%
org.bouncycastle.asn1	57	5%
others	180	16%
Total	**1105**	**100%**

Table 5.2: Breakdown of samples

makes up most of the `org.apache.axis` package in this part of the graph, and that this abnormally long tail corresponds to recursive invocations of the method `symbolTable.SymbolTable.setTypeReferences`. This hints at some algorithmic issues in WSDL symbol management.

A high level summary of this analysis is provided by table 5.2, which gives the package breakdown of profiling samples on non-java packages, once the effect of notifications has been removed (i.e. Area 1 is ignored). This table confirms that `org.apache.axis` (and in particular its `wsdl` component) takes a significant amount of execution time (37%). It also shows that SOAP and XML processing—`org.apache.axis` as a whole and `org.apache.xerces`—have a strong impact on performance (44%). Finally, enforcing security has a non-trivial cost, since the Globus Security Infrastructure (GSI) together with the encryption library `rsa` together account for 30% of the samples.

5.2 Structural contraction in performance graphs

The conditions in which we analysed Globus are quite typical of the situation of many developers of distributed systems today. The architectural layers present in current systems continue to grow, contributing to their complexity, and to the difficulty of their analysis. Simultaneously, the time available to developers to analyse the software elements contained in these layers (OS, libraries, middleware, frameworks, development tools) tends to decrease (both because of productivity constraints, and because of the growing number of elements to consider). This is particularly true for the non-functional properties of these elements (reliability, performance, energy efficiency), which

are often much less documented (if at all) than functional concerns. As a first step to attack this problem, our work of Globus highlighted how the structural information present in execution traces (in our case package names in Java) could potentially help analyse such properties (in this case performance) in complex and unfamiliar software.

In our study of Globus, the structural information contained in profiling traces provided the necessary filter to *raise* the level of abstraction of our performance data, and helped guide our analysis in terms of technological impact (Section 5.1.2 on page 74). In the Globus study, however, our level of abstraction was fixed. Figure 5.7 on page 74, for instance, shows the activity of the package `org.apache.axis` (the Axis library), but does not provide any insight on the activity of sub-packages within axis. Deciding on an optimal abstraction level to represent behavioural data is not an easy task: the final criterion of success is the understanding gained by developers, which is influenced both by the characteristics of the targeted system, and the preferences of each individual.

This research context led us to initiate a collaboration with the Department of Psychology at Lancaster to investigate how developers could determine themselves the right level of abstraction to represent performance data in unfamiliar software. More precisely, we used our experience with Globus to proposed a method that allows developers to *selectively* raise or lower the *local* abstraction level of profiling call trees, and thus reduce the amount of information presented to them while retaining a systemic overview of a system's behaviour.

5.2.1 Approach

Intuition. A set of profiling traces, such as that of Figure 5.3 on page 71, can be represented as a weighted call tree (e.g. Figure 5.8 on the next page) in which the weight of each node represents the execution time taken by the corresponding invocation. The same information can be represented at different levels of abstraction: Figure 5.9a only shows first level packages (similarly to Figure 5.4), while Figure 5.9b presents an intermediary situation, in which different nodes use different levels of compaction. The same package might in particular be represented at different levels of abstraction in different parts of the graph: In Figure 5.9b, `lib3` is expanded on the right, while it is left compacted on the left.

Interestingly, this intuition is in line with other approaches for interactive structural compaction (e.g. Creole[2], DA4Java [Pinzger et al. (2008)] or

[2] http://www.thechiselgroup.org/creole

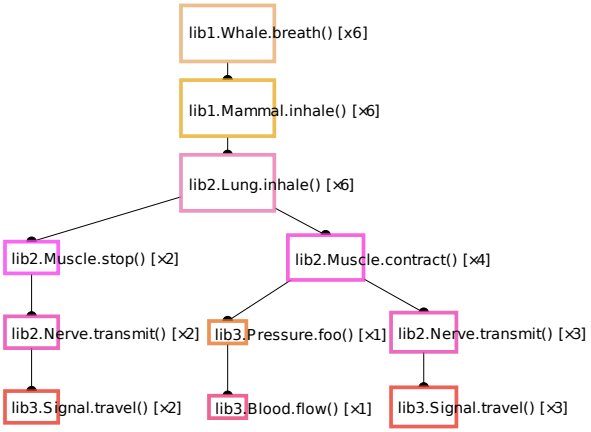
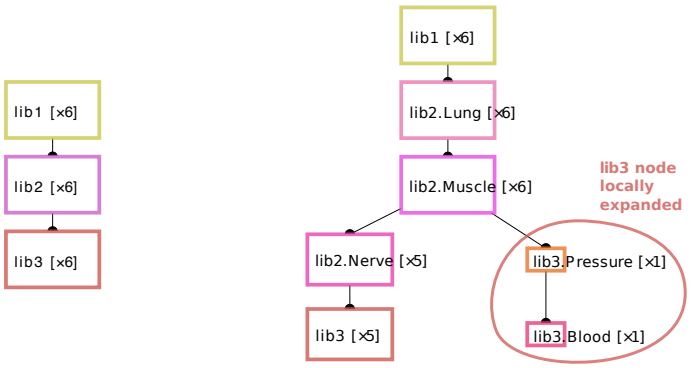


Figure 5.8: The weighted profiling tree corresponding to Figure 5.3



(a) Full compaction: only showing packages

(b) Intermediate expansion, with a localised expansion of `lib3`

Figure 5.9: Different degrees of structural compaction

Bundle Views [Cornelissen et al. (2008)]) proposed in the field of reverse engineering (rather than performance analysis). These previous approaches, however, do not allow for *localised* levels of abstraction, which accounts for the main complexity of our mechanism. This localised ability, we argue, is a critical ability for diagnosis tasks, which are highly context-dependent, and is one of the key differentiating factors of our contribution.

Formalisation. Our approach allows developers to seemingly navigate between the various representations exemplified in Figures 5.8 and 5.9 by using

two simple operations on nodes: *structural compaction* and *extension*. These two operations are based on two elements, both illustrated in Figure 5.10 on the facing page:

- The ability to specify the *local compaction level* that should apply for a particular package in a particular area of the profiling tree.
- A *localised merging mechanism* that captures the interplay of both structural and behavioural closeness to determine the final abstraction level of each program execution points.

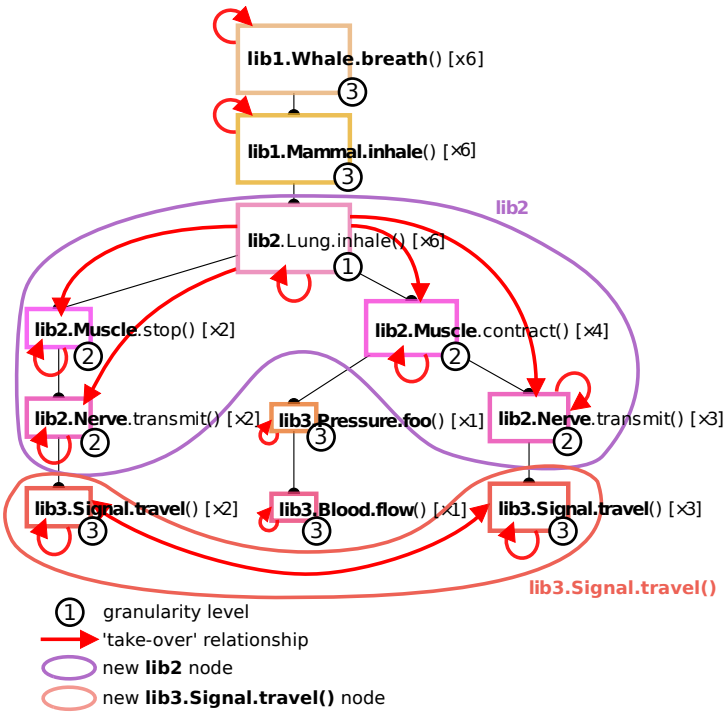


Figure 5.10: Local granularity levels and compaction process

To address the first point, we associate each node with a *granularity level*, an integer that represents how much of the node's full name should be represented in the rendered tree. For instance, Figure 5.10 shows the value of granularity levels (in circles) leading to the compaction of `lib2` in a single node (Figure 5.11). The granularity level of a node determines its *compacted name* (essentially a prefix of its full name), by indicating how many elements of the node's name should be retained in the final graph. For instance in

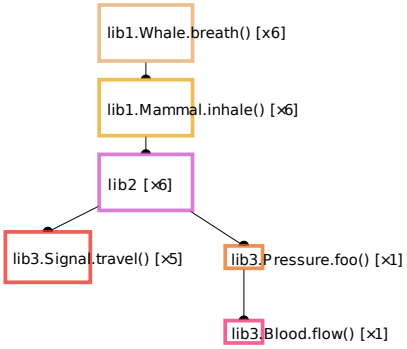


Figure 5.11: Partial compaction of lib2 resulting from Fig. 5.10

Figure 5.10, node `lib2.Lung.inhale()` has a granularity level of 1, meaning that it should be merged with nodes in its vicinity (essentially descendants or siblings) that also belong to `lib2` (represented by the 'lib2' set of nodes on Figure 5.10). The resulting compacted node will then be represented by its top-level package `lib2` (in bold) in the compacted tree (Figure 5.11). All nodes outside `lib2` have a granularity level of '3', meaning they should be represented with 3 name elements (in this case package, class and method).

Compacted names create a 'take-over' relationship between nodes (shown with red arrows on Figure 5.10) that indicates how nodes should be merged in the resulting graph. The goal of this relationship is to capture the interplay of both *structural* and *behavioural* closeness to implement a *localised* merging mechanism. *Structural* because only nodes that belong to a common enclosing package (e.g. `lib2` in Figure 5.10) should be merged together. *Behavioural* because this merging should only happen between nodes that lay in each other's *vicinity* in the call-tree.

The concept of *vicinity* is meant to encompass children and siblings, but needs to be defined somewhat more broadly to capture the situation where two nodes are brought close together because their parents have merged. For instance in Figure 5.10, two leaf nodes refer to the method `lib3.Signal.travel()`. In the fully expanded tree of Figure 5.10, these two nodes are neither siblings, nor descendants of one another, and are therefore represented as independent nodes. However, once the nodes belonging to `lib2` are merged into one compacted node (upper enclosing shape in Figure 5.10), both `lib3.Signal.travel()` nodes become 'siblings' referring to the same program element and should therefore also be merged (with an appropriately updated weight, as explained below).

The nodes of the resulting *compacted tree* are the connected components (in the sense of graph theory) of the take-over relationship (represented as free-form shapes in Figure 5.10). The weight of each compacted node is that of the highest node being merged in the original tree, if there is only one such node (e.g. `lib2.lung.inhale()` in Figure 5.10), or the sums of the weights of the highest nodes if there are several (such as the two leaf nodes `lib3.Signal.travel()` in the same example).

The two operations of localised compaction and extension can then be simply implemented by manipulating each node's granularity level. Essentially, a compacting operation on a compacted node will lower the granularity level of all the nodes in the original profiling tree that correspond to the selected compacted node. An expansion is the reverse: the granularity is raised. In both cases a new merged tree is computed, and a dynamic animation is used to highlight how nodes either merge or separate.

5.2.2 Evaluation

Design. We based our evaluation on an explorative user study with an evaluation prototype implementing our compaction mechanism. Figure 5.12 shows a screenshot of this prototype (named ProfVis[3]) on a sampled-down version of the Globus profiling data of Section 5.1.2. Figure 5.13 shows the layout of the fully compacted profiling tree for the same data. This fully compacted tree only contains 89 nodes, which compares favourably against the 1341 nodes in the original profiling tree.

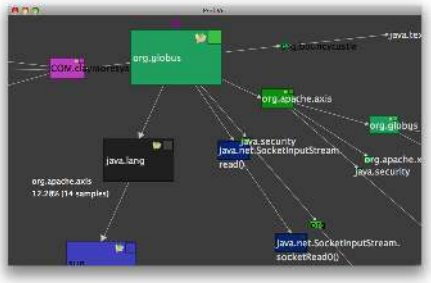


Figure 5.12: The prototype applied to a Globus trace

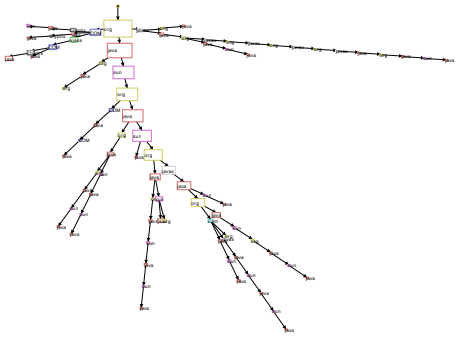


Figure 5.13: The fully compacted graph produced by our tool for the Globus trace (89 nodes)

[3]Available at `http://ftaiani.ouvaton.org/7-software/profvis.html`.

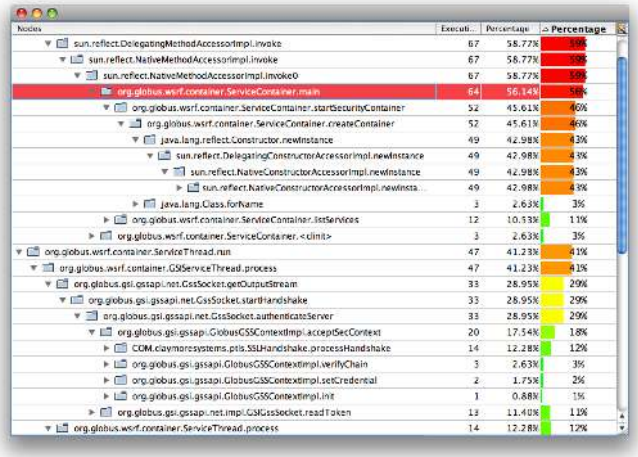


Figure 5.14: Comparison baseline: a semi-textual navigation tool based on branch collapsing

As a comparison baseline for our user study, we selected a semi-textual navigation tool (shown in Figure 5.14 on the next page) that uses the typical branch-based collapsing found in tree widgets. This baseline is representative of the tools commonly available in the industry to navigate performance data [HPj (2009); Ecl (2010)]. We then asked four users to analyse the performance of four different programs (two small and two larger ones, see Table 5.3 on the following page) with ProfVis, and with the baseline (called TreeTable). We asked each user to verbalise their activities (an approach commonly used in Psychology and User Studies in general [Van Someren et al. (1994)]), and recorded each session, both as a video, and a stream of interaction events (node expansion, contraction, etc.). We also asked users to self-assess the quality of their understanding of performance issues in each target program, and assessed off-line this same understanding based on our recordings. We performed 16 sessions in total: four per users, 8 per tool.

Our analysis of the resulting data focuses of two aspects: a comparison of the understanding gained by users with the two tools, and the identification of any arising interaction patterns. This analysis is of course constrained by the small size and nature of our user study: rather than a full-fledged controlled experiment, our goal was to highlight potential issues and trends in the use of localised structural compaction for performance analysis.

	LoC	classes	methods	prof. tree
BubbleSort	59	2	7	100
Simulation	140	5	16	71
OPSBrowser	13624	172	1002	1059
Globus (ws-core-3.9.4)	42477	432	2550	1341

Table 5.3: Some statistics regarding the target programs of our study

In the following, we summarise the main findings from this study, with a more detailed description of our experimental set-up and results available in [Lin et al. (2010)].

Understanding. In our experiment, users were generally able to provide better performance analysis with ProfVis than with the baseline (Figure 5.15). The difference is slight, but promising, in particular when considering that most users felt more comfortable with the baseline tool, which might hint as both familiarity and usability factors that played against our prototype.

Quite interestingly, users had generally a wrong assessment of their own understanding (Figure 5.16). As Figure 5.16 shows, the two measures are largely unrelated: some users thought they did well in some tasks, while missing most of the key points and thus scoring low on the assessed measure, while others did the reverse. Some patterns do seem to appear though: Users are best aligned with their assessed performance when analysing small programs with the baseline (TreeTable, hollow rhombus); they tend to underestimate their understanding of large programs with ProfVis (solid squares); and tend to overestimate their understanding of both small programs with ProfVis (hollow squares) and large programs with TreeTable (solid rhombus).

One possible explanation is to observe that the branch navigation used by the baseline only displays as many nodes as the user has expanded. As a result users may easily perceive a large trace graph as smaller than it really is with the baseline tool, and from there reach *a false sense of confidence* when they have missed some key parts of a program's execution. By contrast, ProfVis forces users to confront a program's full call-tree from the onset, even if in a highly compacted form. For instance, the fully compacted version of the Globus traces contains 89 nodes when ProfVis starts (Figure 5.12), while TreeTable only shows two lines for the same trace file. This might lead users to a sense of helplessness with ProfVis, explaining

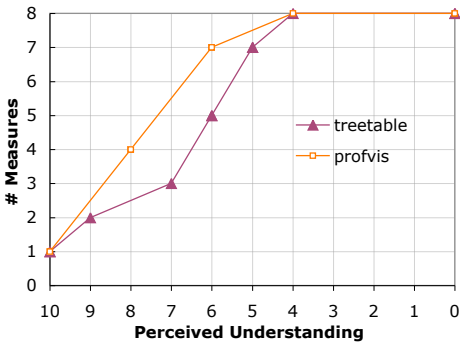


Figure 5.15: Cumulative distributions of understanding

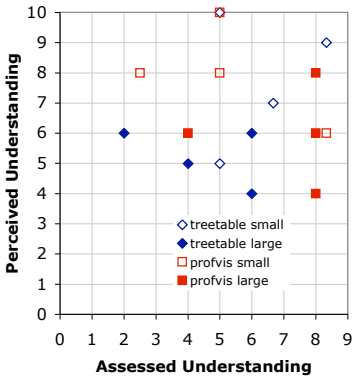


Figure 5.16: Contrasting perceived and assessed understanding

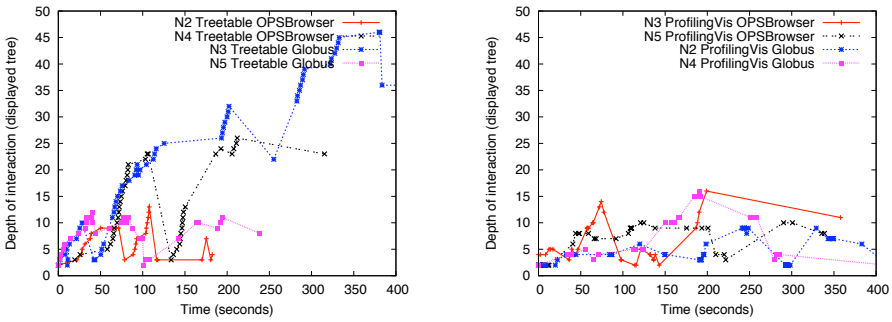
their underestimation of their performance, and part of the discomfort they expressed with our prototype (while performing well with it).

Interaction patterns and strategies. Our evaluation shows an even stronger contrast between the two tools in the strategies taken by the users to explore the profiling data. With the baseline tool (TreeTable) most users adopted a *depth-first* strategy with, rapidly moving deep into the call tree along a single execution branch (generally that of the most weighted child), and only occasionally backtracking through large jumps back to the top of the tree (Figure 5.17a). With ProfVis, by contrast, users went far less deep, and tended (for the majority at least) to keep interacting at the same depth over long periods of time (appearing as 'plateaus' on Figure 5.17b).

This pattern hints at the key importance of presentation, in particular layout, in influencing user choices in their exploration of a program's execution. The layout of TreeTable naturally encourages users to go deep first: the next child with the highest share of CPU usage is always the closest and lies in a predictable position. By contrast, the relative location of nodes in ProfVis evolves in a two-dimensional plane with each new interaction. As a result node positions are far less predictable, possibly deterring users from rapidly moving away from their current position.

5.3 Conclusion

The introduction of new technologies in existing distributed systems is always a challenging endeavour. Our work on Globus has shown how the intro-



(a) TreeTable (baseline) (b) ProfVis

Figure 5.17: Interaction patterns on large programs

duction of a generic web-service layer, essentially provided by the reusable Apache Axis library, could have a critical impact on the general performance of a platform. Beside the inherent interest of this research for WS-* architectures, and grid computing, our study of Globus also demonstrates the difficulty in diagnosing a complex composite software made of a large number of third party elements. In the profiling results we obtained for Globus, Globus packages only account for a small part of the container execution time (Figure 5.7 and Table 5.2), while reused WS-* libraries explain most of the latencies observed.

The tool ProfVis we have presented addresses this problem, and leverages the structural information present in performance traces to offer developers with a simple and interactive technique to navigate profiling trees. Contrary to typical performance analysis tool that hide or show part of the execution, our approach continuously offers a complete and holistic view of a program's execution, but allows developers to vary the level of abstraction at which the data is presented. This level of abstraction can be fine-tuned depending on the context of execution, contrary to related reverse engineering tools, and more generally demonstrates how structure and behaviour can be combined to analyse quantitative runtime data.

Interestingly, because our technique essentially produces an alternative and more compact tree, it can be combined with almost any additional tree navigation and visualisation approach, such a branch-base collapsing, or advanced layout and panning techniques.

Finally, although our study has focused on execution time, a large range of runtime properties can be represented using the same approach. ProfVis is essentially applicable to any weighted set of stack traces, which might cor-

respond to object allocation sites (for memory usage analysis), exception stack traces from fault injection campaigns (to analyse error detection capabilities), or error messages from logs (to identify problematic behaviour in deployed software).

CHAPTER 6

Conclusion and outlook

The works presented in this document cover a large range of research topics, from distributed algorithms in large-scale systems to performance analysis in complex middleware architectures. Although each work belongs to a distinct research area, they remain linked to each other, at least in terms of motivation: They all aim to facilitate the development and analysis of large-scale distributed systems, with better *mechanisms*, better *programming abstractions*, and better *tools*.

In the following we revisit each of these three facets in the light of the works presented in the previous chapters, and conclude with some discussion of the future work we envisage for our research.

6.1 Developing better distributed mechanisms

Today's distributed protocols can roughly be divided into two categories: a first category of protocols, such as those for consensus, or state machine replication [Lamport (1998); Chandra and Toueg (1996); Castro and Liskov (2002)], provide strong (and proved) guarantees. Some of these protocols (in particular Paxos) can be found at the core of many cloud infrastructures such as that of Google and Yahoo! where they provide critical coordination services [Burrows (2006); Cooper et al. (2009)]. These protocols are however only moderately scalable, in particular on wide area networks: recent experiments have reported consensus implementation on PlanetLab with a few

hundreds of nodes [Maia et al. (2011)], but ranges of hundreds of thousands to hundreds of millions of nodes appear much more problematic.

An alternative to these strong-guarantees protocols are opportunistic mechanisms, such as gossip protocols [Agrawal et al. (1997); van Renesse et al. (1998b); Gupta et al. (2006)], and other mechanisms found in peer-to-peer overlays [Stoica et al. (2001); Rowstron and Druschel (2001)]. These protocols are extremely scalable, and target highly dynamic systems (where peers join and leave continuously), at the cost of weaker guarantees: There is typically some probability (even minute) that some properties will not hold (e.g. that message will not be distributed to all participants, or that some inconsistencies will be returned by the system), and decision problems (such as consensus) are typically not supported.

Our own work on fault-tolerant overlays represents a middle ground between these two extremes. We have shown how traditional consensus could be used in a scalable manner in networks of arbitrary size to coordinate repair. Our approach consists in only involving those nodes present in the vicinity of a failed region. The key challenge with this strategy is in deciding within the consensus which nodes should be involved (what we have termed *self-defining constituencies*). The approach we propose (Echo) consists in coordinating multiple instances of competing consensus, and arbitrating between overlapping consensus attempts. Our mechanism allows nodes that hold conflicting views of the system's state (and hence of whom should be participating in a consensus instance) to converge onto a unified perception of the overlay's condition. As a proof concept we have developed a generic repair framework (Sonar) based on Echo. Our evaluations are positive, and show that this type of generic repair is generally robust, even when our strict assumptions are not met, and that our framework can even outperform existing tailored approaches.

6.2 Software abstractions for distribution

To become accessible to developers, distributed mechanisms must be presented within appropriate programming abstractions. We have shown with GossipKit, that components represent a particularly promising technology to organise and systematise the development of gossip protocols. In spite of their broad range, and the many services they provide (fault-detection, partitioning, multicast, data aggregation, search, routing), GossipKit demonstrates how gossip protocols can be decomposed in a set of limited and highly reusable elements applicable across a large range of representative protocols, thus greatly simplifying the development of gossip-based systems.

Components are however not the only approach available to develop gossip protocols: Annotations [Princehouse and Birman (2010)] and object frameworks[1] have for instance been used. In our own work, we have shown how a high-level distributed language (Whispers), inspired from macro-programming approaches, could be coupled to a component framework to present developers with a familiar set of constructs and abstractions. This synergy between two programming paradigms (components providing structure and (re)configurability, while imperative constructs provide expressiveness and programmability) is reminiscent of dual architectures, in which one level mirrors the other in a different representation. This is the case of reflective architectures, but also of recent work on models at runtime [Floch et al. (2006)]. These multi-paradigm approaches provide teams of developers with different perspectives on the same concrete system, and certainly represent a powerful trend to master the conflicting concerns (reliability, adaptability, interoperability) of today's large-scale distributed systems.

6.3 Abstraction, transparency, and understanding

Our work on fault-tolerant overlays (Chapter 3) and gossip protocols (Chapter 4) has sought to identify and encapsulate generic patterns into reusable abstractions. Ultimately, this philosophy, which is generally shared by middleware researchers and developers, aims to propose modular and multi-levels architectures, in which different developer teams (possibly from different organisations) can manipulate the underlying system using their own point of view. The use of abstractions is however a double-edge sword: Most abstractions aim to make their implementation at least partly *transparent* to the developers who use them. Unfortunately this transparency is often imperfect, and cannot hide many undesirable side effects, in particular in distributed systems. This fact, combined with the growing size and complexity of today's systems, makes it particularly difficult for developers to track non-functional anomalies across abstraction boundaries. One reason is that non-functional properties are closely linked to a particular implementation of an abstraction, rather than to an abstraction itself, and often result from emerging interactions between independently developed software parts. Another reason pertains to the layered organisation that is naturally promoted by abstractions: Developers end up relying on libraries or components (e.g. Axis in our case) that lay below the layer they normally work at, and represent unfamiliar code.

[1] http://gossiplib.gforge.inria.fr/

In my research, I have explored the concrete impact of this phenomenon in the web-service version of Globus, one of today's production-level middleware for grid computing. Web-services provide extreme levels of genericity. For instance both the typing system and the on-wire representation of data can be specified independently of a web-service interface, and, at least in theory, can be discovered on the fly by a web service endpoint. This genericity has implications, both in terms of software organisation, and performance behaviour, as illustrated by the general use of lazy initialisation in Globus, and the role played by internal (non-Globus) libraries in the observed latencies.

Analysing the performance of multi-layer software such as Globus represents a combination of reverse engineering and non-functional analysis, for which current tools and approaches are not well adapted. Inspired from our work on Globus, and our previous research in reverse engineering [Taïani et al. (2009)], we have therefore proposed ProfVis, a simple and interactive tool that combines the structural information found in execution traces with the quantitative information of performance profiling. Our user study based on ProfVis has shed light on some interesting strategies taken by developers to explore runtime data of unfamiliar code. In particular, our work suggests that presenting too many details at the expense of an overall perspective can lure developers into a fall sense of mastery, and lead to mis-diagnosis.

6.4 Outlook

The works presented in this document raise a number of exciting opportunities for future research, in particular at the boundaries between the research areas involved. In the short term, GossipKit and Whispers raise the possibility of extending them beyond gossip-based systems (for instance to structured overlays), and for the application of higher-level development processes to gossip protocols, such as software product lines [Paul Clements (2001)]. The work on ProfVis calls for further studies of developers' perception of unfamiliar software, and their strategies to explore performance data.

In the medium term, GossipKit and Whispers also open interesting avenues for the development of large-scale smart systems, for which I am about to start an industrial collaboration at Lancaster. Smart systems are being proposed for industries with a low technological awareness, such as water distribution companies. These industries are today looking for approachable and easily programmable systems that can provide aggregation

and regulation mechanisms over large areas (several 10,000 km^2), in adverse and dynamic communication conditions. Gossip protocols, and high-level languages provide a promising mix to address these goals. On a different note, GossipKit and Whispers also offer an attractive framework to reason about interference and synergies in co-existing decentralised systems. This is a path I have started to explore conceptually [Lin et al. (2009)], and that I would like to pursue, in particular to automate the discovery and automatic activation of synergies in such multi-tenant multi-purpose systems.

In the longer term, our application of component frameworks for gossip-based systems in GossipKit raises the reciprocal question whether components should not be adapted to the kind of interactions promoted by gossip protocols. Some programming paradigms such as fragmented components [Makpangou et al. (1994)] and chemical and membrane computing [Banâtre et al. (2005); Paun (2003)] seem to offer particularly promising avenues to organise the structure and computation of gossip-based systems. Similarly, families of gossip-based protocols are now emerging for specific areas, such as social networks [Bertier et al. (2010)], and peer-to-peer search [Bai et al. (2010)]. Extending the strategy of Whispers, there seem to be some strong potential for exposing the ingredients that form these systems as high-level programmable constructs, and thus simplify their development.

Finally, gossip protocols can be considered as a special case of a broader class of decentralised opportunistic systems, hinted at for instance by works on morphogenetic engineering [Doursat (2008)]. If one assumes that these decentralised opportunistic systems will form an important part tomorrow's digital infrastructure, this also raises the question of the analysis, diagnosis, and understanding of the runtime behaviour of these systems. Current diagnosis approaches, including the one we have proposed for Globus, are unlikely to translate unchanged to these emerging programming paradigms. An open question is therefore how the behaviour of decentralised opportunistic systems should be monitored, processed and represented to make sense for developers.

Chapter 6.

List of Figures

1.1 Research topics and their relationships over the period 2000-2010 . . 12

3.1 An overlay with a failed region and possible repairs 25
3.2 The architecture of Sonar/Echo 27
3.3 The three main phases of our repair algorithm 28
3.4 Protocol instances and conflicting views 29
3.5 A cluster of adjacent faulty domains 31
3.6 Convergence between overlapping views 37
3.7 Tree overlay with a failed region, a full, and a partial additive repair 42
3.8 Effects of false positives 44
3.9 Standard TBCP & Sonar/TBCP repair times 45
3.10 Sonar/TBCP network usage at the root node on PlanetLab. 46

4.1 Structure and behaviour in traditional component-based approaches 51
4.2 Transparent componentisation 51
4.3 GossipKit's common architectural pattern 52
4.4 Whispers program of the RPS protocol 57
4.5 Whispers program of the Gossip1 protocol 57
4.6 The language primitives of Whispers 57
4.7 Deployment Process in WhispersKit 59
4.8 Per-node program of RPS 60
4.9 Component realisation of RPS 60
4.10 Dynamic reconfigurations with WhispersKit 61

5.1 Black box profiling: experimental set-up 69
5.2 Workload of the sample-based profiling campaign 70
5.3 An example sampling result 71
5.4 Graphical projection of figure 5.3 72

LIST OF FIGURES

5.5 First add requests . . . 73
5.6 Add requests of first client . . . 73
5.7 Performance sampling of the Globus container (inclusive) . . . 74
5.8 The weighted profiling tree corresponding to Figure 5.3 . . . 78
5.9 Different degrees of structural compaction . . . 78
5.10 Local granularity levels and compaction process . . . 79
5.11 Partial compaction of lib2 resulting from Fig. 5.10 . . . 80
5.12 The prototype applied to a Globus trace . . . 81
5.13 Fully compacted graph of the Globus trace . . . 81
5.14 Comparison baseline: a semi-textual navigation tool . . . 82
5.15 Cumulative distributions of understanding . . . 84
5.16 Contrasting perceived and assessed understanding . . . 84
5.17 Interaction patterns on large programs . . . 84

List of Tables

4.1 Reused achieved by GossipKit 55
4.2 GossipKit's implementation effort (in LoC) compared to Java . . . 56
4.3 Whispers program sizes vs. GossipKit and java (LoC) 62
4.4 Whispers programs vs. Java and GossipKit 63

5.1 Lazy initialisation for add and notification operations 74
5.2 Breakdown of samples 75
5.3 Some statistics regarding the target programs of our study 82

LIST OF TABLES

Bibliography

(2009). Hpjmeter 4.0 user's guide. Technical Report HP Part Number: 5992-5899, Hewlett-Packard Development Company.

(2010). Eclipse test & performance tools platform project. http://www.eclipse.org/-tptp/index.php (accessed 8 April 2010).

Agrawal, D., El Abbadi, A., and Steinke, R. C. (1997). Epidemic algorithms in replicated databases (extended abstract). In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '97, pages 161–172, New York, NY, USA. ACM.

Amer, P. D. and Çeçeli, F. (1990). Estelle formal specification of iso virtual terminal. *Comput. Stand. Interfaces*, 9:87–104.

Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S.-T. A., Sites, R. L., Vandevoorde, M. T., Waldspurger, C. A., and Weihl., W. E. (1997). Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390.

Arlat, J., Blanquart, J.-P., Boyer, T., Crouzet, Y., Durand, M.-H., Fabre, J.-C., Founau, M., Kaâniche, M., Kanoun, K., Meur, P. L., Mazet, C., Powell, D., Scheerens, F., Thévenod-Fosse, P., and Waeselynck, H. (2000). *Composants logiciels et sûreté de fonctionnement: intégration de COTS*. Hermes Science.

Arnold, D., Ahn, D. H., de Supinski, B. R., Lee, G., Miller, B. P., and Schulz, M. (2007). Stack trace analysis for large scale debugging. In *21^{st} Int. Parallel and Dist. Processing Symp. (IPDPS 2007) Long Beach, CA*.

Babaoglu, O., Meling, H., and Montresor, A. (2002). Anthill: a framework for the development of agent-based peer-to-peer systems. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 15 – 22.

BIBLIOGRAPHY

Bai, X., Bertier, M., Guerraoui, R., Kermarrec, A.-M., and Leroy, V. (2010). Gossiping personalized queries. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 87–98, New York, NY, USA. ACM.

Banâtre, J.-P., Fradet, P., and Radenac, Y. (2005). Principles of chemical programming. In Abdennadher, S. and Ringeissen, C., editors, *Proceedings of the 5th International Workshop on Rule-Based Programming*, volume 124(1) of *ENTCS*, pages 133–147. Elsevier.

Barr, R., Haas, Z. J., and van Renesse, R. (2005). JiST: an efficient approach to simulation using virtual machines: Research articles. *Softw. Pract. Exper.*, 35:539–576.

Basu, A., Hayden, M., Morrisett, G., and Eicken, T. (1997). A language-based approach to protocol construction. In *In Proc. of the ACM SIGPLAN Workshop on Domain Specific Languages (WDSL*, pages 87–99.

Bertier, M., Frey, D., Guerraoui, R., Kermarrec, A.-M., and Leroy, V. (2010). The gossple anonymous social network. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware '10, pages 191–211, Berlin, Heidelberg. Springer-Verlag.

Bhatti, N. T., Hiltunen, M. A., Schlichting, R. D., and Chiu, W. (1998). Coyote: a system for constructing fine-grain configurable communication services. *ACM Trans. Comput. Syst.*, 16:321–366.

Birman, K. P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., and Minsky, Y. (1999). Bimodal multicast. *ACM Trans. Comput. Syst.*, 17:41–88.

Brooks, Jr., F. P. (1987). No silver bullet essence and accidents of software engineering. *Computer, IEEE*, 20:10–19.

Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., and Stefani, J.-B. (2004). An Open Component Model and Its Support in Java. In Crnkovic, I., Stafford, J. A., Schmidt, H. W., and Wallnau, K., editors, *Component-Based Software Engineering*, volume 3054 of *Lecture Notes in Computer Science*, chapter 3, pages 7–22. Springer Berlin / Heidelberg, Berlin, Heidelberg.

Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. (2006). The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36:1257–1284.

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

BIBLIOGRAPHY

Burrows, R., Ferrari, F. C., Garcia, A., and Taïani, F. (2010a). An empirical evaluation of coupling metrics on aspect-oriented programs. In *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, WETSoM '10, pages 53–58, New York, NY, USA. ACM.

Burrows, R., Ferrari, F. C., Lemos, O. A. L., Garcia, A., and Taïani, F. (2010b). The impact of coupling on the fault-proneness of aspect-oriented programs: An empirical study. In *IEEE 21st International Symposium on Software Reliability Engineering*, pages 329–338, San Jose, CA, USA. IEEE Computer Society.

Burrows, R., Taïani, F., Garcia, A., and Ferrari, F. C. (2011). Reasoning about faults in aspect-oriented programs: A metrics-based evaluation. In *Proceedings of the 19th IEEE International Conference on Program Comprehension (ICPC'2011)*, pages 131–140.

Castro, M., Druschel, P., Kermarrec, A.-M., and Rowstron, A. (2002). SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE J. on Selected Areas in communications (JSAC)*.

Castro, M. and Liskov, B. (2002). Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20:398–461.

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal for the Association for Computing Machinery (JACM)*, 43(2):225–267.

Chawathe, Y., McCanne, S., and Brewer, E. A. (2000). RMX: reliable multicast for heterogeneous networks. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, pages 795 –804 vol.2.

Chen, S., Joshi, K., Hiltunen, M., Sanders, W., and Schlichting, R. (2009). Link gradients: Predicting the impact of network latency on multitier applications. In *INFOCOM 2009, IEEE*, pages 2258 –2266.

Chinnici, R., Moreau, J.-J., Ryman, A., and Weerawarana, S. (2007). *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. W3C, version 2.0 edition. `http://www.w3.org/TR/2007/REC-wsdl20-20070626`.

Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W. (2001a). Freenet: a distributed anonymous information storage and retrieval system. In *Int. Wkshp. on Designing Privacy Enhancing Tech.*, pages 46–66.

Clarke, M., Blair, G. S., Coulson, G., and Parlavantzas, N. (2001b). An efficient component model for the construction of adaptive middleware. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 160–178, London, UK. Springer-Verlag.

BIBLIOGRAPHY

Clements, P. C. (1996). A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, IWSSD '96, pages 16–, Washington, DC, USA. IEEE Computer Society.

Colyer, A. and Clement, A. (2004). Large-scale aosd for middleware. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD '04, pages 56–65, New York, NY, USA. ACM.

Cooper, B. F., Baldeschwieler, E., Fonseca, R., Kistler, J. J., Narayan, P. P. S., Neerdaels, C., Negrin, T., Ramakrishnan, R., Silberstein, A., Srivastava, U., and Stata, R. (2009). Building a cloud for yahoo! *IEEE Data Eng. Bull.*, 32(1):36–43.

Cornelissen, B., Zaidman, A., Holten, D., Moonen, L., van Deursen, A., and van Wijk, J. J. (2008). Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*.

Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., and Ueyama, J. (2004). A component model for building systems software. In *In Proc. of the IASTED Conference on Software Engineering and Applications (SEA'04)*.

Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., and Sivaharan, T. (2008). A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26:1:1–1:42.

Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D. (1987). Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 1–12, New York, NY, USA. ACM.

Doursat, R. (2008). Programmable architectures that are complex and self-organized: from morphogenesis to engineering. In *11th International Conference on the Simulation and Synthesis of Living Systemes (ALIFE XI)*, pages 181–188. MIT Press.

Doval, D. and O'Mahony, D. (2003). Overlay networks: A scalable alternative for p2p. *IEEE Internet Comp.*, 7(4):79–82.

Edwards, G., Deng, G., Schmidt, D. C., Gokhale, A., and Natarajan, B. (2004). Model-driven configuration and deployment of component middleware publisher/subscriber services. In *Proc. of the 3rd ACM International Conference on Generative Programming and Component Engineering*, pages 337–360.

Eijk, P. V. and Diaz, M., editors (1989). *Formal Description Technique Lotos: Results of the Esprit Sedos Project*. Elsevier Science Inc., New York, NY, USA.

Eugster, P., Felber, P., and Le Fessant, F. (2007). The "art" of programming gossip-based systems. *SIGOPS Oper. Syst. Rev.*, 41:37–42.

BIBLIOGRAPHY

Eugster, P. T., Guerraoui, R., Handurukande, S. B., Kouznetsov, P., and Kermarrec, A.-M. (2001). Lightweight probabilistic broadcast. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, DSN '01, pages 443–452, Washington, DC, USA. IEEE Computer Society.

Fielding, R. T. and Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2:115–150.

Fleury, M. and Reverbel, F. (2003). The JBoss extensible server. In *ACM/IFIP/USENIX Int. Middleware Conference (Middleware'03)*, pages 344–373.

Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., and Gjorven, E. (2006). Using architecture models for runtime adaptability. *Software, IEEE*, 23(2):62 – 70.

Foster, I., Kishimoto, H., Savva, A., Berry, D., Djaoui, A., Grimshaw, A., Horni, B., Macieli, F., Siebenlist, F., Subramaniam, R., Treadwell, J., and Reich, J. V. (2005). The open grid services architecture, version 1.0. http://www.ggf.org/documents/GWD-I-E/GFD-I.030.pdf.

Fung, W., Sun, D., and J.Gehrke (2002). Cougar: the network is the database. In *ACM SIGMOD International Conference on Management of Data*.

Ganesh, A. J., Kermarrec, A.-M., and Massoulié, L. (2003). Peer-to-peer membership management for gossip-based protocols. *IEEE Trans. Comput.*, 52:139–149.

Gong, L. (2001). JXTA: a network programming environment. *Internet Computing, IEEE*, 5(3):88 –95.

Grace, P., Coulson, G., Blair, G. S., Mathy, L., Yeung, W. K., Cai, W., Duce, D. A., and Cooper, C. S. (2004). Gridkit: Pluggable overlay networks for grid computing. In Meersman, R. and Tari, Z., editors, *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE, OTM Confederated International Conferences, Agia Napa, Cyprus, October 25-29, 2004, Proceedings, Part II*, volume 3291 of *Lecture Notes in Computer Science*, pages 1463–1481. Springer.

Grace, P., Coulson, G., Blair, G. S., and Porter, B. (2005). Deep middleware for the divergent grid. In *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, Middleware '05, pages 334–353, New York, NY, USA. Springer-Verlag New York, Inc.

Grace, P., Hughes, D., Porter, B., Blair, G. S., Coulson, G., and Taiani, F. (2008). Experiences with open overlays: a middleware approach to network heterogeneity. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys'08, pages 123–136, New York, NY, USA. ACM.

BIBLIOGRAPHY

Graham, S., Karmarkar, A., Mischinsky, J., Robinson, I., and Sedukhin, I. (2004). Web services resource 1.2 (ws-resource). http://docs.oasis-open.org/wsrf/2004/11/wsrf-WS-Resources-1.2-draft-02.pdf. Document Identifier: wsrf-WS-Resource-1.2-draft-02.

Graham, S., Kessler, P., and McKusick, M. (1983). Execution profiler for modular programs. *Software - Practice and Experience*, 13:671–685.

Greenwood, P., Hughes, D., Porter, B., Grace, P., Coulson, G., Blair, G., Taiani, F., Pappenberger, F., Smith, P., and Beven, K. (2006). Using a grid-enabled wireless sensor network for flood management. In *the demonstration supplement of the 8th International Conference on Ubiquitous Computing (Ubicomp'06)*, Orange County, CA, USA.

Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., Nielsen, H. F., Karmarkar, A., and Lafon, Y. (2007). *SOAP Version 1.2 Part 2: Adjuncts*. W3C, second edition edition. http://www.w3.org/TR/2007/REC-soap12-part2-20070427/.

Gummadi, R., Gnawali, O., and Govindan, R. (2005). Macro-programming wireless sensor networks using kairos. In *Distributed computing in sensor systems*.

Gupta, I., Kermarrec, A.-M., and Ganesh, A. J. (2006). Efficient and adaptive epidemic-style protocols for reliable and scalable multicast. *IEEE Trans. Parallel Distrib. Syst.*, 17:593–605.

Gupta, I., Renesse, R. v., and Birman, K. P. (2001). Scalable fault-tolerant aggregation in large process groups. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, DSN '01, pages 433–442, Washington, DC, USA. IEEE Computer Society.

Haas, Z. J., Halpern, J. Y., and Li, L. (2006). Gossip-based ad hoc routing. *IEEE/ACM Trans. Netw.*, 14:479–491.

Hiltunen, M., Taïani, F., and Schlichting, R. (2006). Reflections on aspects and configurable protocols. In *Proceedings of the 5th international conference on Aspect-oriented software development*, AOSD '06, pages 87–98, New York, NY, USA. ACM.

Hiltunen, M. A. and Schlichting, R. D. (2000). The cactus approach to building configurable middleware. In *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*.

Hou, X. and Tipper, D. (2004). Gossip-based sleep protocol (gsp) for energy efficient routing in wireless ad hoc networks. In *The 2004 IEEE Wireless Communications and Networking Conference (WCNC 2004)*, volume 3, pages 1305–1310 Vol.3.

Hughes, D., Greenwood, P., Porter, B., Grace, P., Coulson, G., Blair, G., Taiani, F., Pappenberger, F., Smith, P., and Beven, K. (2006). Using grid technologies

to optimise a wireless sensor network for flood management. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, pages 389–390, Boulder, Colorado, USA.

Jannotti, J., Gifford, D. K., Johnson, K. L., Kaashoek, M. F., and O'Toole, Jr., J. W. (2000). Overcast: Reliable multicasting with an overlay network. In *Proc. of the Fourth Symp. on Operating Sys. Design and Impl. (OSDI)*, pages 197–212.

Jelasity, M. and Babaoglu, O. (2005). T-man: Gossip-based overlay topology management. In *Proc. of the 3rd Int. Workshop on Engineering Self-Organising Applications*, pages 1–15.

Jelasity, M., Guerraoui, R., Kermarrec, A.-M., and van Steen, M. (2004). The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, Middleware '04, pages 79–98, New York, NY, USA. Springer-Verlag New York, Inc.

Jelasity, M. and Kermarrec, A.-M. (2006). Ordered slicing of very large-scale overlay networks. In *Proceedings of the Sixth IEEE International Conference on Peer-to-Peer Computing*, pages 117–124, Cambridge, United Kingdom. IEEE Computer Society.

Jelasity, M., Montresor, A., and Babaoglu, O. (2005). Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23:219–252.

Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.-M., and van Steen, M. (2007). Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25.

Jerding, D. F., Stasko, J. T., and Ball, T. (1997). Visualizing interactions in program executions. In *19th Int. Conf. on Soft. Engineering (ICSE '97)*, pages 360–370.

Kermarrec, A.-M. and van Steen, M. (2007). Gossiping in distributed systems. *SIGOPS Oper. Syst. Rev.*, 41:2–7.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Springer, editor, *European Conference on Object-Oriented Programming (ECOOP'97)*, volume LNCS 1241, pages 220–242, Jyväskylä, Finland.

Killian, C. E., Anderson, J. W., Braud, R., Jhala, R., and Vahdat, A. M. (2007). Mace: language support for building distributed systems. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 179–188, New York, NY, USA. ACM.

Lamport, L. (1998). The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169.

BIBLIOGRAPHY

Leonini, L., Riviere, E., and Felber, P. (2008). P2P experimentations with SPLAY: from idea to deployment results in 30 min. In *Eighth International Conference on Peer-to-Peer Computing (P2P'08)*, pages 189 –190.

Li, B., Guo, J., and Wang, M. (2004). iOverlay: A lightweight middleware infrastructure for overlay application implementations. In *Proc. of IFIP/ACM/USENIX Middleware*, Toronto, Canada.

Lin, S. (2010). *Transparent Componentisation: A hybrid approach to support the development of contemporary distributed systems*. PhD thesis, Lancaster University, UK.

Lin, S., Taïani, F., Bertier, M., Blair, G., and Kermarrec, A.-M. (2011). Transparent componentisation: high-level (re)configurable programming for evolving distributed systems. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 203–208, New York, NY, USA. ACM.

Lin, S., Taïani, F., and Blair, G. (2009). Exploiting synergies between coexisting overlays. In *Proceedings of the 9th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, DAIS '09, pages 1–15, Berlin, Heidelberg. Springer-Verlag.

Lin, S., Taïani, F., and Blair, G. S. (2008). Facilitating gossip programming with the gossipkit framework. In *Proceedings of the 8th IFIP WG 6.1 international conference on Distributed applications and interoperable systems*, DAIS'08, pages 238–252, Oslo, Norway. Springer-Verlag.

Lin, S., Taïani, F., Ormerod, T. C., and Ball, L. J. (2010). Towards anomaly comprehension: using structural compression to navigate profiling call-trees. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 103–112, New York, NY, USA. ACM.

Lin, S., Taïani, F., and Blair, G. S. (2007). Gossipkit: A framework of gossip protocol family. In *Proceedings of the 5th MiNEMA Workshop (Middleware for Network Eccentric and Mobile Applications)*, pages 26–30, Magdeburg, Germany.

Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. (2005). Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173.

Maes, P. (1987). Concepts and experiments in computational reflection. In *Proceedings of the 1987 ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 147–155, Orlando, Florida, United States.

Maia, F., Matos, M., Pereira, J., and Oliveira, R. (2011). Worldwide consensus. In Felber, P. and Rouvoy, R., editors, *Distributed Applications and Interoperable Systems*, volume 6723 of *Lecture Notes in Computer Science*, pages 257–269. Springer Berlin / Heidelberg.

BIBLIOGRAPHY

Makpangou, M., Gourhant, Y., Le Narzul, J.-P., and Shapiro, M. (1994). Fragmented objects for distributed abstractions. In Casavant, T. L. and Singhal, M., editors, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press.

Mathy, L., Canonico, R., and Hutchison, D. (2001). An overlay tree building control protocol. In *Proceedings of the Third International COST264 Workshop on Networked Group Communication*, NGC '01, pages 76–87, London, UK, UK. Springer-Verlag.

McCabe, T. J. (1976). A complexity measure. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA. IEEE Computer Society Press.

McKinley, P. K., Padmanabhan, U. I., and Ancha, N. (2001). Experiments in composing proxy audio services for mobile users. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 99–120, London, UK. Springer-Verlag.

Milojicic, D. S., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S., and Xu, Z. (2003). Peer-to-peer computing. Technical Report HPL-2002-57R1, HP Labs, Palo Alto, CA.

Newton, R., Morrisett, G., and Welsh, M. (2007). The regiment macroprogramming system. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 489–498, Cambridge, Massachusetts, USA. ACM.

O'Hair, K. (2004). Use hprof to tune performance. http://java.sun.com/developer-/technicalArticles/Programming/HPROF.html (accessed 8 April 2010).

OMG (2008). Common object request broker architecture (corba/iiop) (3.1). `http://www.omg.org/spec/CORBA/3.1/`.

Paul Clements, L. N. (2001). *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Paun, G. (2003). Membrane computing. In *Fundamentals of computation theory*, pages 177–220. Springer.

Pauw, W. D., Helm, R., Kimelman, D., and Vlissides, J. (1993). Visualizing the behavior of object-oriented systems. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications (OOPSLA'93)*, pages 326–337. ACM Press.

Pauw, W. D., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J. M., and Yang, J. (2002). Visualizing the execution of java programs. In *Software Visualization: International Seminar, Dagstuhl*, volume 2269 of *LNCS*, pages 151–162. Springer-Verlag.

BIBLIOGRAPHY

Pinzger, M., Graefenhain, K., Knab, P., and Gall, H. C. (2008). A tool for visual understanding of source code dependencies. In *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 254–259.

Porter, B. (2007). *An Approach to Generalising the Self-Repair of Overlay Networks*. Phd thesis, Lancaster University.

Porter, B., Coulson, G., and Taïani, F. (2006a). A generic self-repair approach for overlays. In Meersman, R., Tari, Z., and Herrero, P., editors, *Proc. of the Int. Workshop on Reliability in Decentralized Distributed systems (RDDS 2006), On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, volume 4278 of *Lecture Notes in Computer Science*, pages 1490–1499. Springer Berlin / Heidelberg. 10.1007/11915072_54.

Porter, B., Roedig, U., Taiani, F., and Coulson, G. (2010a). A comparison of static and dynamic component models for wireless sensor networks. In *Proceedings of the The First International Workshop on Networks of Cooperating Objects (CONET2010), Stockholm, Sweden*.

Porter, B., Roedig, U., Taïani, F., and Coulson, G. (2010b). The lorien dynamic component based os. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 355–356, Zürich, Switzerland. ACM.

Porter, B., Taiani, F., and Coulson, G. (2006b). Generalised repair for overlay networks. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS 2006)*, pages 132–142, Leeds, UK. IEEE Computer Society.

Porter, B., Taïani, F., and Coulson, G. (2008). On the convergent detection of crashed regions in overlay networks. Technical Report COMP-010-2008, Computing Department, Lancaster University (UK). 10 pages.

Princehouse, L. and Birman, K. (2010). Code-partitioning gossip. *SIGOPS Oper. Syst. Rev.*, 43:40–44.

Ranganathan, A. and Campbell, R. H. (2007). What is the complexity of a distributed computing system?: Research articles. *Complex.*, 12:37–45.

Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. (2000). A scalable content addressable network. Technical Report TR-00-010, UC Berkeley.

Reiss, S. P. and Renieris, M. (2000). Generating java trace data. In *Java Grande Conference (ACM 2000 conference on Java Grande)*, pages 71–77, San Francisco, CA, USA. ACM.

Reiss, S. P. and Renieris, M. (2003). *Software Visualization – From Theory to Practice*, chapter The BLOOM Software Visualization System. MIT Press.

BIBLIOGRAPHY

Richner, T. and Ducasse, S. (1999). Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 13–, Washington, DC, USA. IEEE Computer Society.

Rodriguez, A., Killian, C., Bhat, S., Kostic, D., and Vahdat, A. (2004). Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proc. of the USENIX/ACM Symp. on Networked Sys. Design and Implementation (NSDI 2004)*, San Francisco, California, USA.

Rowstron, A. I. T. and Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proc. of the IFIP/ACM Int. Conf. on Dist. Sys. Platforms*, pages 329–350.

Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., and Stefani, J.-B. (2011). A component-based middleware platform for reconfigurable service-oriented architectures. *Software: Practice and Experience*, pages n/a–n/a.

Smith, B. C. (1984). Reflection and semantics in lisp. In *Eleventh Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 23–35, Salt Lake City, Utah. ACM.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 149–160, San Diego, California, United States. ACM.

Systä, T., Koskimies, K., and Müller, H. (2001). Shimba—an environment for reverse engineering java software systems. *Softw. Pract. Exper.*, 31(4):371–394.

Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

Taïani, F. (2003). CosmOpen: A reverse-engineering tool for complex open-source architectures. In *DSN-03 supplemental volume, Student Forum of DSN'03, The International Conference on Dependable Systems and Networks*, pages A49–A51, San Francisco, CA. IEEE Computer Society.

Taïani, F., Fabre, J.-C., and Killijian, M.-O. (2002). Principles of multi-level reflection for fault-tolerant architectures. In *2002 Pacific Rim International Symposium on Dependable Computing (PRDC 2002)*, pages 59–66, Tsukuba (Japan).

Taïani, F., Fabre, J.-C., and Killijian, M.-O. (2003). Towards implementing multi-layer reflection for fault-tolerance. In *The International Conference on Dependable Systems and Networks (DSN-2003)*, San Francisco, CA. IEEE Computer Society.

BIBLIOGRAPHY

Taïani, F., Killijian, M.-O., and Fabre, J.-C. (2009). CosmOpen: dynamic reverse engineering on a budget. *Softw. Pract. Exper.*, 39(18):1467–1514.

Taïani, F. (2004). *La Réflexivité dans les architectures multi-niveaux : application aux systèmes tolérant les fautes.* Thèse de doctorat, Université Paul Sabatier (Toulouse 3), Toulouse, France.

Taïani, F., Fabre, J.-C., and Killijian, M.-O. (2005a). A multi-level meta-object protocol for fault-tolerance in complex architectures. *Dependable Systems and Networks, International Conference on*, 0:270–279.

Taïani, F., Hiltunen, M., and Schlichting, R. (2005b). The impact of web service integration on grid performance. In *The 14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, pages 14–23, Research Triangle Park, NC, USA.

Thibault, S., Consel, C., and Muller, G. (1998). Safe and efficient active network programming. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, SRDS '98, pages 135–, Washington, DC, USA. IEEE Computer Society.

Urbán, P., Défago, X., and Schiper, A. (2002). Neko: A single environment to simulate and prototype distributed algorithms. *Journal of Information Science and Engineering*, 18(6):981–997.

van Renesse, R., Birman, K., Hayden, M., Vaysburd, A., and Karr, D. (1998a). Building adaptive systems using ensemble. *Software Practice and Experience*, 28(9):963–979.

van Renesse, R., Birman, K. P., Friedman, R., Hayden, M., and Karr, D. A. (1995). A framework for protocol composition in horus. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 80–89. ACM.

van Renesse, R., Birman, K. P., and Maffeis, S. (1996). Horus: A flexible group communication system. *Communications of the ACM (CACM)*, 39(4):76–83.

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

Van Someren, M., Barnard, Y., and Sandberg, J. (1994). *The think aloud method: A practical guide to modelling cognitive processes.* Academic Press, London. ISBN 0-12-714270-3.

Weston, N., Taïani, F., and Rashid, A. (2005). Modular aspect verification for safer aspect-based evolution. In Cazzola, W., Chiba, S., Saake, G., and Tourwé, T., editors, *Proceedings of the 2nd ECOOP Workshop on Reflection, AOP and*

Meta-Data for Software Evolution (RAM-SE'05), pages 17–28, Glasgow, Scotland. Fakultät für Informatik, Universität Magdeburg.

Weston, N., Taïani, F., and Rashid, A. (2007). Interaction analysis for fault-tolerance in aspect-oriented programming. In *Proceedings of the Workshop on Methods, Models and Tools for Fault Tolerance (MeMoT), held in conjunction with iFM 2007: integrated Formal Methods*, pages 95–102. Technical report CS-TR-1032, Newcastle University, UK.

Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. I. (2009). Detecting large-scale system problems by mining console logs. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132, Big Sky, Montana, USA. ACM.

Yang, B. and Garcia-Molina, H. (2003). Designing a super-peer network. In *Proc. of the 19th Int. Conf. on Data Eng.*

Zamfir, C. and Candea, G. (2010). Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 321–334, New York, NY, USA. ACM.

Zhang, C. X., Wang, Z., Gloy, N. C., Chen, J. B., and Smith, M. D. (1997). System support for automated profiling and optimization. In *Symposium on Operating Systems Principles*, pages 15–26.

Zhao, B. Y., Kubiatowicz, J. D., and Joseph, A. D. (2001). Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley.