An Approach to Generalising the Self-Repair of Overlay Networks

A thesis submitted for the degree of Doctor of Philosophy

Barry Francis Porter
Computer Science BSc. Hons., Lancaster University, 2004

October 2007

Computing Department

An Approach to Generalising the Self-Repair of Overlay Networks

Barry Francis Porter
Computer Science BSc. Hons., Lancaster University, 2004

A thesis submitted for the degree of Doctor of Philosophy
Computing Department, Lancaster University

October 2007

Abstract

Overlay networks are emerging as the technology of choice for deploying new large-scale distributed services and applications. Despite being such a key technology, overlays have no coherent strategy for failure recovery, employing a plethora of ad-hoc solutions.

This thesis surveys the state-of-the-art in overlay network and fault-tolerance research, examining the shortcomings of present approaches. It then proposes SONAR, a highly configurable and adaptive distributed service, to provide generalised fault-tolerance to a wide range of overlay networks. The design of SONAR is discussed, and its unique API is presented in detail—an API enabling genericity of SONAR while permitting expressiveness of an overlay's specific traits.

The heart of the proposal is then presented and discussed; a new kind of *distributed repair protocol*, which leverages distributed consensus to provide a strong base from which multiple kinds of repair can be dynamically selected and safely enacted at runtime. This in turn enables *adaptive* repair, which can intelligently decide on the best course of action for each failure.

The proposed service is then evaluated in its ability to support several major and varied overlays, showing that the approach taken is viable despite its genericity, and demonstrating the powerful capabilities of an adaptive repair mechanism.

The outcome of this proposal is a re-usable service which can be employed by overlay developers to support repair in their overlay, successfully achieving a clean separation of concerns, and leaving the developer to focus on the functional aspects of their design.

Declaration

This thesis is entirely my own work, and has not been submitted in any form for the award of a higher degree elsewhere. No chapters have been published elsewhere, though some of the concepts described have been—these are clearly noted where applicable.

The prototype implementation of the proposed system was written entirely by myself, but the Gridkit middleware implementation was written largely by Dr. Paul Grace. The overlay network implementations with which the proposed system interoperates were written in various proportions by Dr. Grace and myself.

Acknowledgements

A number of people have been instrumental in the completion of this thesis. My supervisor, Professor Geoff Coulson, has been of constant support, continually asserting his belief in my work. His advice and editing skills have been invaluable, and I cannot imagine a better supervisor to have taken this journey with.

Dr. François Taïani adopted an interest in my work at a crucial time, and was simply irreplaceable in his assistance with the development of the protocols underpinning this work. His quick mind, expansive knowledge, and oft-used whiteboard have my deepest thanks.

Dr. Paul Grace served as the research associate on the project to which I belonged, and the huge volumes of work he performed for the project left me with almost all of my time to focus on my personal areas of interest—this is not always the case, and I am very grateful to his efforts.

I must also thank my parents for their ever-present support of my chosen path, whatever that might be, and for their love and care. Finally, this work was funded by the UK's EPSRC, without whom it would not have come to be.

Thank you all,

Barry

Contents

1 Introduction **12**

1.1 Overlays and fault-tolerance 13

1.2 Goals 14

2 Background **16**

2.1 Overlay networks 16

2.1.1 Content dissemination overlays 16

2.1.2 Distributed Hash Table overlays 18

2.1.3 Semi-structured overlays 22

2.1.4 Unstructured and randomised overlays 23

2.1.5 Summary 26

2.2 Overlay Deployment Environments 27

2.2.1 The Open Overlays Framework 27

2.2.2 ODIN-S 28

2.2.3 JXTA 29

2.2.4 iOverlay 30

2.2.5 Specification-language frameworks 31

2.3 Dependability Services 32

2.3.1 Checkpointing 32

2.3.2 Replication 34

2.3.3 Failure detection 36

2.3.4 Fault Tolerant Software Architectures 38

2.3.5 Alternative approaches 41

2.4 Overall analysis 43

3 Services for Overlay Network Adaptive Resilience **45**

3.1 Approach 45

3.1.1 Backup service 46

3.1.2 Failure detection service 47

3.1.3 Recovery service . . . 48
3.1.4 Other services . . . 50

4 SONAR APIs **52**

4.1 The API . . . 53
4.1.1 A simple, extensible API . . . 54
4.1.2 An overlay model . . . 55
4.1.3 Repair actions . . . 58
4.2 Case studies . . . 58
4.2.1 Chord . . . 58
4.2.2 TBCP . . . 62
4.3 Summary . . . 65

5 A SONAR Repair Protocol **66**

5.1 Motivation and contributions . . . 66
5.2 The Repair Protocol . . . 68
5.2.1 Assumptions . . . 68
5.2.2 An overview of the protocol . . . 69
5.2.3 Phase 1: Discovery of the extent of the failed section . . . 71
5.2.4 Phase 2: Failed section agreement and repair coordinator selection . . . 72
5.2.5 Phase 3: Enacting the repair . . . 74
5.3 Repair strategies . . . 75
5.3.1 Additive repairs . . . 75
5.3.2 Subtractive repairs . . . 78
5.4 Correctness, determinism and probability . . . 80
5.4.1 Phase 0 . . . 81
5.4.2 Phase 1 . . . 81
5.4.3 Phase 2 . . . 91
5.4.4 Phase 3 . . . 93
5.5 Concluding remarks . . . 99

6 Implementation **100**

6.1 Architecture . . . 100

6.2 Failure detection service . . . 101

6.3 Backup service . . . 102

6.4 Recovery service . . . 102

6.5 Summary . . . 103

7 Evaluation **104**

7.1 Repair protocol evaluation . . . 104

7.1.1 Evaluation under normal conditions . . . 104

7.1.2 Evaluation of key complicating factors . . . 107

7.2 Comparative evaluation . . . 110

7.2.1 Chord . . . 111

7.2.2 TBCP . . . 115

7.2.3 Gnutella Super-Peer overlay . . . 120

7.3 Genericity evaluation . . . 125

7.3.1 Overlay Model & API . . . 125

7.3.2 Service implementations . . . 127

7.4 Summary . . . 128

8 Future directions **130**

9 Conclusions **132**

A Full round-optimised repair protocol pseudo-code **143**

B Avoiding deadlock caused by false-positives **146**

List of Figures

1 (a) An overlay deployed on selected hosts across different networks, in which data propagates from node 'R' to the other members of the overlay; (b) The architecture within a single member host. To avoid clutter, physical network links are not shown. 13

2 (a) A Chord ring (physical hosts and links are not shown), and (b) The topology data and (key, value) pairs stored at node 9 19

3 (a) Chord nodes copy their locally stored (key, value) pairs to n nodes clockwise around the ring and (b) The failure of node 9 causes its keys to be re-located to node 14, successor and predecessor links of nodes 7 and 14 updated, and node 7's (key, value) pairs now copied to nodes 14 and 22 instead of 9 and 14 21

4 The topology of a typical super-peer overlay like Gnutella 0.6; leaf-peers cluster around a super-peer, and super-peers handle search queries within their own mesh topology 22

5 The architecture of a Gridkit node, expanding on the open overlays layer, which is hosting a Pastry and Scribe node, with the Scribe node re-using Pastry's forwarder. 28

6 Coordinated and uncoordinated checkpointing approaches. Horizontal lines are process timelines, squares are checkpoints and arrows are messages sent from one process to another. 33

7 Active, passive and gossip-based replication schemes, showing client-server and server-server interaction for a single request from the client 35

8 The main architectural elements of Chameleon. Additionally, each host shown runs a daemon, plus a number of application modules and associated ARMORs. Lines represent 'usage', such that the left SM (sub-manager) is using 4 hosts to execute an application. To avoid clutter, the diagram does not show the fact that the FTM holds references to every host in the system. 39

9 SONAR, horizontally composed with an overlay node 45

10 A single interface encapsulates the three sub-services for self-repair, and specific interfaces for specific sub-services can be optionally acquired via the unified interface if necessary 53

11 A two-way API assists with genericity and performance 54

12 The interfaces used in the architecture: (a) SONAR's main interfaces, and (b) Interfaces to be implemented by overlay nodes. Methods return 'void' unless otherwise noted. All parameters are 'in' only. 54

13 API interactions: (a) An overlay node exposing its accessinfos and nodestates; and (b) SONAR inspecting and adapting these (e.g. at repair time). 55
14 Neighbours exposed by the overlay are used by the service to send data to other service instances 57
15 An accessinfo implementation for Chord 59
16 The implementation of the *ISONARNode* and *IRepairableNode* interfaces for Chord 60
17 The modified implementation of the stabilize method in Chord 61
18 The implementation of *setNodeID()* for Chord 61
19 An accessinfo implementation for TBCP 63
20 The implementation of the *ISONARNode* and *IRepairableNode* interfaces for TBCP 64
21 The modified implementation of the joinTree method in TBCP 64
22 Pseudo-code of the repair protocol when executed by node p 69
23 The three main phases of the repair protocol 70
24 The pseudo-code of AgreeOnView when executed by node p 72
25 An example scenario using view rejection—in this case avoiding deadlock 73
26 (a) Part of a tree overlay (b) A failure develops (c) A repair suggests restoring the failed nodes on alternative hosts. Overlay nodes are identified with numbers, and physical hosts with letters. 76
27 A partial restoration of a failed tree section 76
28 (a) Part of a tree overlay, with a failed region; (b) A single-hub subtractive repair with node 31, and (c) node 33 as the repair coordinator 79
29 (a) A failed region of overlay FR_1, and the live nodes surrounding it LN_1, (b) A node p with a view containing some of FR_1, due to the belief that node c is alive (and thus a border node) due to failure detector latency 82
30 (a) Node p has a view FS_p which is smaller than the full failed section, but includes node g as a border node; g itself discovered the entire failed section FS_g during phase 1, so their views conflict in phase 2, (b) A node p holds the view FS_p, which contains only itself as a border node, and node g has the larger view FS_g, and awaits node p's opinion on that view 83

31 Views constructed by nodes c_1 (a) and c_2 (b); either the nodes in BN_d (c) or BN_x (d) must have failed before being able to provide opinions to c_1 or c_2, respectively 84
32 Phase 1 is executed by node g, and walks 'through' a failed border node of a repair in progress by node p 87
33 Nodes about to enter phase 1 in attempt to discover a failed section 88
34 The coordinator of a repair (a), fails mid-way through its repair, having restored failed nodes k and w (b). The nodes it was not able to repair become part of a future agreement attempt and repair (c). 95
35 Different nodes observe different, contradictory structures of the overlay due to lost repair logs following the failure of coordinator node p mid-repair 97
36 The implementation architecture of SONAR, showing relationships between services 100
37 Per-border-node overhead during repair 105
38 Combined overhead at border nodes during repair 105
39 Effects of false positives 109
40 Standard Chord (left) and SONAR-Chord (right) network usage at node 8 . . . 112
41 Standard Chord (left) and SONAR-Chord (right) average memory usage 113
42 Standard Chord (left) and SONAR-Chord (right) average workload 114
43 Chord-SONAR average workload with additive repairs 115
44 Standard TBCP (left) and SONAR-supported TBCP (right) repair times distribution. 116
45 Standard TBCP (left) and SONAR-TBCP (right) network usage at the root node in simulations. 117
46 SONAR-supported TBCP network usage at the root node on PlanetLab. 118
47 TBCP-SONAR memory usage 119
48 Standard Gnutella (left) and SONAR-supported Gnutella (right) repair times distribution. 121
49 Standard Gnutella (left) and SONAR-Gnutella (right) network usage at node 4 . 122
50 Standard Gnutella (left) and SONAR-Gnutella (right) average memory usage . . 123
51 Standard Gnutella (left) and SONAR-Gnutella (right) average resource list sizes 124

52 Gnutella-SONAR (additive) resource list sizes 124
53 Pseudo-code of the repair protocol when executed by node p 143
54 Pseudo-code of phase 1 when executed by node p 144
55 Pseudo-code of ConstructFailedSection when executed by node p 144
56 The pseudo-code of ReceiveViewMessage 144
57 The pseudo-code of AgreeOnView when executed by node p 145
58 The pseudo-code of ProcessRound when executed by p 145
59 A node Q proposes a view including false positives (of nodes G, H and L), in which other border nodes have no interest 146

List of Tables

1 Summary of overlays 26
2 Summary of overlays' fault-tolerance 26

1 Introduction

This thesis considers *fault tolerance* in *overlay networks*; it examines the current state of the art in overlay network mechanisms to respond to crash-failures [28], the problems with these approaches, and ways of improving upon them. In pursuit of finding improvements, existing approaches to providing fault-tolerance services to general distributed systems are also examined.

Overlay networks [23]—commonly termed "overlays"—are virtual, software-implemented networks which abstract above another network. They provide specific functionality or scoping beyond that available in the underlying network. As such, they exist between the underlying network (commonly the Internet) and the application making use of the overlay's targeted functionality.

Example specialised functionalities provided by overlays are content dissemination [41], key-based routing [76], and string-based resource location [83]. An application wishing to provide a distributed service can then select an appropriate overlay network and deploy it to the relevant hosts, then use that overlay to underpin the application's needs.

To illustrate this, consider the need for a live media-streaming application to disseminate content to a large number of hosts spread across the Internet. While IP-multicast may be able to cope with this in a limited network, it is often not enabled, and is certainly not possible today over the Internet. To solve this problem, such an application can incorporate a content dissemination overlay in its deployment, and use this overlay to connect end-user hosts to the media-streaming session in a highly scalable manner. Such a deployment is shown in figure 1.

As can be seen, a special piece of software ('node') exists on each host wishing to be part of the overlay (i.e. media-streaming session), and those 'nodes' form logical links (e.g. unicast TCP connections) with selected other nodes in the overlay. Data can then be forwarded from the root node through to the leaves of the tree, by using only the standard capabilities of end-user hosts. The logical links maintained by the overlay network form its *topology*—the overlay shown in figure 1 has a 'tree' topology.

Overlays in recent years have rapidly evolved in complexity and the service they provide, and overlays themselves can be configured in an overlay *stack*, providing composite services [13, 30].

As well as end-system multicast, some examples of the diverse uses of overlays today are for the

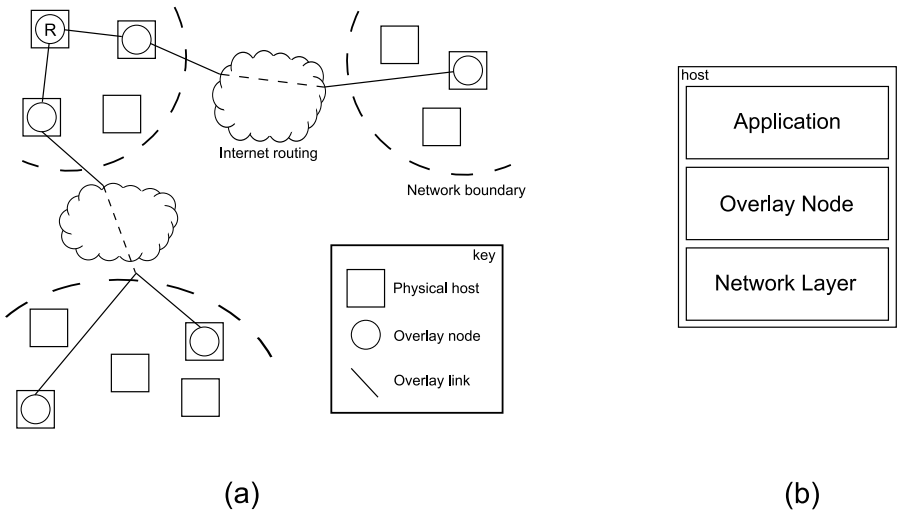


Figure 1: (a) An overlay deployed on selected hosts across different networks, in which data propagates from node 'R' to the other members of the overlay; (b) The architecture within a single member host. To avoid clutter, physical network links are not shown.

provision of quality of service [77], massive distributed data storage [50], resilient communication services [5], support of online gaming communities [40] and in support of voice-over-IP telephony and conferencing [33]. All of this clearly points to the architecture illustrated in figure 1 being the emerging general technique for deploying and supporting new services and applications, by deploying virtual networks on-demand, with topologies and functionality designed specifically to support a given service (or service 'class', like key-based routing [76]).

An important emerging technology in this regard is the overlay-building toolkit / deployment environment [82, 30], which provides just this kind of capability. The overlay deployment environment has at its disposal a wide range of different overlay components, and in response to an application request, is able to mix and match overlay components to build, and sometimes deploy and maintain, tailorable overlay platforms for different applications. Such toolkits are therefore considered as an important part of this work, and a domain in which re-usable fault-tolerance components should operate.

1.1 Overlays and fault-tolerance

A crash failure (termed simply 'failure' throughout this thesis) in an overlay network is said to occur when one of the overlay's member nodes suddenly departs the system without warning. Due to the nature of distributed systems, and particularly the Internet—with hosts appearing

and leaving with high frequency—having a failure response mechanism is almost mandatory for a new overlay design to be taken seriously. While such a mechanism is sometimes natural to an overlay's design, in many cases it is presented as an 'added extra', such as the successor list and replication in Chord (see section 2.1.2) or the added-on ancestor list in Overcast (see section 2.1.1).

This practice has three major shortcomings:

- It frequently results in duplication of effort by designers working on similar kinds of overlays such as multicast trees, who implement their own specific fault-tolerance mechanism which is ultimately very similar to others that have been proposed for similar overlays.

- The approaches taken to fault tolerance by overlay designers—not usually being the focus of their design—are typically specific to a single deployment environment and repair strategy. With the phenomenal rise in the popularity of overlays as a re-usable system component, however, the ways in which they are being deployed are growing well beyond the single environment of 'minimal-infrastructure, all-end-user hosts over the Internet'. As this happens, the included failure response and repair mechanisms show shortcomings by not taking their deployment environment into account to make more appropriate fault-tolerance decisions.

- The application/system designer considering which overlay(s) to use must understand the nuances of the fault-tolerance approach of each potential overlay, and the fault-tolerance approach may need to be manually tailored to specifically suit its deployment environment and the criticality of the system.

1.2 Goals

The motivation of this work is to solve the three problems listed above. It proposes to do so by separating the concern of *self-repair* from the core functionality of overlay networks. This has the clear benefit of *re-usability*, speeding the development of new overlays, and reducing duplication of effort by overlay designers.

Such an approach, focusing solely on fault-tolerance and not on how the overlay achieves its particular functions, can more easily examine the wider field in which overlays exist, providing

a fully rounded solution capable of operating in many different deployment environments, able to harness the particular strengths and weaknesses of each of those environments.

And finally, a common, well-understood and documented *method* of repair is provided, which promotes greater *understanding* for those wishing to *use* overlay networks in support of their applications or services. Instead of needing to understand and evaluate the various self-repair mechanisms of a large number of functionally similar overlays, for example, a developer can choose an overlay based solely on the particular way it provides its functionality, and then configure the general self-repair service supporting the overlay with simple options like 'amount of redundancy'.

Achieving this would present the benefits of a re-usable, well-tested and configurable service for the increasingly numerous developers involved in overlay-based work. The rest of this thesis is an account of this work and its results.

This thesis attempts to i) provide a deep understanding of the problem space in chapter 2, ii) propose a new approach to overlay network fault-tolerance with a generalised and adaptive repair service in chapter 3, iii) explain how overlay networks can be sufficiently generalised for management by an external service while preserving their functional diversity, in chapter 4, and iv) provide a particular approach to generically repairing node failures in overlays, in chapter 5. This work is then evaluated, and concluding remarks are offered.

2 Background

This chapter discusses in detail i) overlay networks, ii) overlay-building toolkits / deployment environments, and iii) dependability / repair services, demonstrating the context in which this work was done and the problem space which exists. The chapter concludes with a brief analysis and the resulting requirements of a generalised repair service.

2.1 Overlay networks

To demonstrate the diverse range of overlays available today, this section is split into four sub-sections representing the major overlay 'classes' that exist today. Content-dissemination overlays are first discussed in detail, followed by distributed hash table (DHT) overlays, then semi-structured super-peer overlays, and finally unstructured overlays. These sections are further divided into 'functionality' and 'fault-tolerance' sections, describing these two separate attributes of each overlay. The list of overlays presented is far from exhaustive (there are simply too many to discuss), but a balanced view of the different *kinds* of overlays that exist is represented as best as is possible.

2.1.1 Content dissemination overlays

Functionality Content dissemination overlays [58, 84, 60] deliver streaming content to multiple users in a scalable manner (as was illustrated in figure 1). They are typically organised as a tree with the sender at the root. Each non-root node receives data from its parent, and forwards it to each of its children using a point-to-point link. If IP-level multicast is deployed in some sub-domains of the physical network, this may be leveraged where possible [16].

TBCP [58] is a good representative of this class of overlay. TBCP builds a single-rooted tree, and new nodes join the tree by first contacting the root node on a published or well-known address. The root node decides if it wants to accommodate the joining node as one of its direct children; if not, it forwards the join request to its most suitable child. This process recurses until the joining node finds a place in the tree. For performance reasons, TBCP attempts to build a tree that reflects the structure of the underlying IP network—i.e. the nodes contained in each sub-tree should tend to share IP-level locality. Decisions about whether to accept a node as a

direct child or to pass it on are therefore made on this basis.

Another approach to maintaining a close correspondence between an overlay multicast tree and the underlying IP topology is to employ a network metric such as round trip time between nodes [60]. Other multicast overlays prefer to use the bandwidth of nodes to influence the structure of the tree [86], so that higher bandwidth nodes appear closer to the root, with the lowest bandwidth nodes as leaf nodes. Whichever metric is used during tree construction, the obvious property desired following repair of a failure is for the post-failure tree structure to be optimal with respect to those metrics.

There are several other multicast overlays worth discussing, which use notably different tree building approaches or designs. Overcast [41] is an example of a server-oriented multicast overlay (see also RMX [16]), where every node in the tree is a *server* managed or leased by the owner of the multicast session—a suggested application domain being TV streaming. End-user clients connect to a nearby server in the multicast tree to get content. The advantage of using managed servers to form the tree is that additional services can be practically offered; servers can cache content so that clients can request play-back from a specified time, and servers can stream content to each client at different data and compression rates.

Narada [37] builds trees in two steps; it first builds a mesh of all participants in the multicast session, then builds a tree over this mesh, rooted at the sender of the multicast data. The mesh approach has the advantages of providing a more strongly connected graph, with redundant links in case of failure, and of being able to sometimes make additional optimisations by forwarding data for a multicast session through a mesh node not part of that session. It does, however, maintain a full group membership list at each node in the mesh, which limits its scalability.

SplitStream [14] also uses two stages of tree construction. For a single multicast session, it splits the data to be sent into n streams. For each stream, a separate multicast tree is built. Of all the nodes in the overlay that wish to receive the multicast data, each node is ideally an 'internal node' of *one* stream tree, and a *leaf node* of *every* stream. The forwarding load of the tree is then better spread across all participants, instead of the situation in other trees where the leaves of the tree contribute nothing to the overlay.

Fault-tolerance There are a number of 'standard' approaches in multicast trees to defend against node failures [84]. One approach is for all the nodes below the failure point to re-join via the root of the tree. The drawback of this is that the resultant bottleneck can cause traffic to be significantly disrupted during the (possibly extensive) re-building phase.

An alternative is to have each node record a "backup parent", or several, and re-join the tree at one of these in case of failure. This carries the assumption, however, that the backup parent chosen has enough spare capacity to accommodate the additional node(s).

Overcast nodes (i.e. servers) keep a full 'ancestor list' at each node to survive multiple simultaneous failures in the same branch of the tree. A node which discovers its parent has failed then examines this list to find a suitable live node to become its new parent. Due to the expected limited size of the Overcast server tree, the scalability issues inherent in this approach are deemed unimportant.

All of these approaches can be seen as 'added extras', as they contribute nothing to the ability of the overlay to perform its functions. The diversity of repairs which can be used in a tree is interesting—virtually *any* repair of a tree-structured overlay is valid as long as it remains a tree (i.e. a graph with no cycles)—repairs therefore should aim to be as good as possible with regard to *performance*, both in terms of short-term disruption and long-term throughput.

2.1.2 Distributed Hash Table overlays

Functionality The general purpose of this class of overlay is to provide an efficient and scalable "key-based routing" facility in which a message can be routed in O(log N) hops (where N is the total number of nodes in the overlay) to a target node that is designated by a given 'key'. In Chord [76], for example, nodes are organised as a logical ring, as shown in figure 2. Each overlay node is assigned an ID by applying a uniform hashing function to its IP address, and keys are generated using that same hashing function (normally by applying the function to a *value* to be stored in the overlay). This hashing function implicitly creates a 'key space', and overlay nodes are responsible for storing and serving the keys (and associated values) that map to their ID.

To cater for situations in which there are not enough nodes in the overlay to provide a one-to-one mapping from keys to nodes within the key space, keys are stored at the node they are numerically equal to or less than (and closest to). For example, the overlay shown in figure 2 (a)

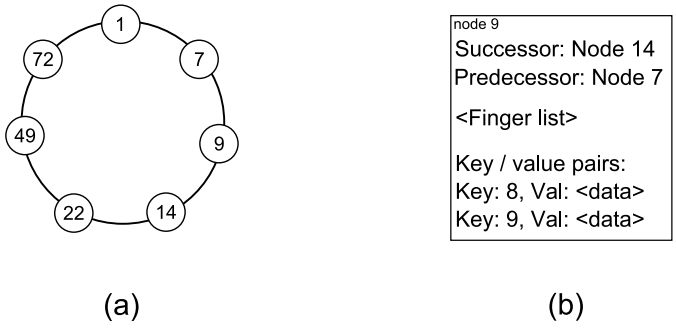


Figure 2: (a) A Chord ring (physical hosts and links are not shown), and (b) The topology data and (key, value) pairs stored at node 9

would store the key '8' at node 9, the key '9' at node 9, the key '11' at node 14, etc.

When a request is made to locate a given key, a Chord node receiving that request is tasked simply with forwarding it as close as possible to its destination (unless it is itself that destination, in which case the key location has been found and a response can be made). To facilitate this, all Chord nodes have a 'successor' and 'predecessor' neighbour, representing the next highest and lowest ID'd nodes in the overlay, thereby forming a basic ring structure, and allowing O(N) progress of routing a key to a node.

Chord nodes also each maintain a so-called *finger list*—a list of nodes that are exponentially distant around the ring. This is used for O(log N) routing towards a target node, in the best case (i.e. assuming the finger list is both fully populated and up to date). When routing a key to a destination, a node examines its successor and finger list, selects the node that will forward the key closest to its final destination, then passes the routing request on to that node, which performs the same operation recursively. One use of Chord is as a distributed data repository; in such an application, a data item which is submitted for storage in the repository is simply stored at the node whose ID is closest to a hash of the data.

DHTs like Chord are attractive because they scale very well—each node maintains a constant number of neighbours as the overlay grows, and routing to a resource takes a known maximum number of overlay hops. They also aim to *guarantee* that an existing key can be retrieved from the overlay, in contrast to the less strongly structured overlays that are discussed in sections 2.1.3 and 2.1.4. There are various other works on DHTs using different overlay structures and identifier spaces. Pastry [72] builds a similar ring to Chord, but takes network locality into

account so that a route taken from one node to another in response to a query more closely matches the physical network infrastructure (i.e. there may be less physical network hops per query).

Tapestry [85] nodes are organised as a mesh, while CAN [68] uses an n-dimensional identifier space as its routing structure; for example a 2-dimensional identifier space would be laid out as a grid. A CAN node takes a 'zone of responsibility' within this identifier space, storing all resources that map to that zone. Zones can be split and merged as nodes join and leave, transferring resources between nodes as appropriate.

SkipNet [35] is somewhat different; whereas the above overlays do not give any control over where values are placed, SkipNet has the design goals of being able to store data within the administrative network domain that it belongs to, while still trying to ensure that data can be located from any node in O(log N) overlay hops, independent of the size of the overlay (with high probability). A SkipNet is a doubly-linked ring, and nodes are identified by string names instead of hashed keys. This means that a user can identify their data using a prefix from a known node, and the data will be stored at that node. Data can also be stored at a collection of nodes that all share the same prefix to enable a degree of load balancing in the overlay. To achieve O(log N) routing in this string-based namespace, nodes are arranged in alphabetical order of their names, and nodes maintain the equivalent of a finger list (but for both directions round the ring) so that large jumps round the ring can be made to get closer to the target name more quickly.

Besides SkipNet's basic abilities to localise data to particular nodes, locality in DHTs is generally seen as a problem. The IDs given to nodes are normally assigned in a uniform random way, and as such a node's neighbours may be physically distant. As a message is routed through a DHT to its destination, this problem accumulates, and could result in the message taking a much longer route than the direct path from source node to destination. While some attempts have been made to reduce this overhead, it remains a topic of open research.

Fault-tolerance Despite their differences, all the above-mentioned overlays have a similar approach to dependability. In particular, when used as a data repository, they increase availability of the data by storing copies of data items on the n nodes whose ID is "closest" to the hash of a stored piece of data—in figure 3, n is 2, so node 9 for example copies its keys to nodes 14 and

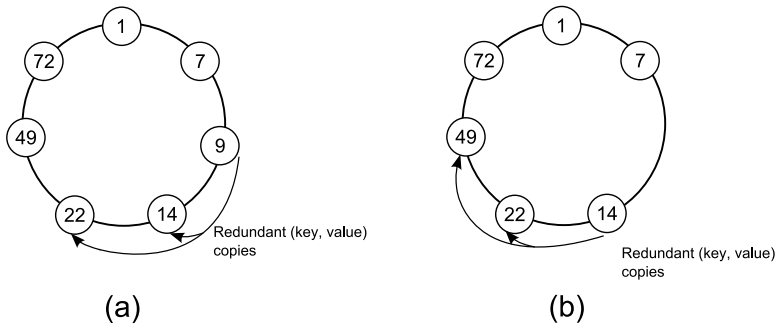


Figure 3: (a) Chord nodes copy their locally stored (key, value) pairs to n nodes clockwise around the ring and (b) The failure of node 9 causes its keys to be re-located to node 14, successor and predecessor links of nodes 7 and 14 updated, and node 7's (key, value) pairs now copied to nodes 14 and 22 instead of 9 and 14

22. The response to a node failure is to update the links in the routing tables of the affected nodes to reflect the change, and also to restore the number of copies of data items stored at the failed node by copying them to further nodes.

The general disadvantage of this approach is that the self-repair algorithm permanently increases the load on the surviving nodes and reduces the total amount of redundancy in the overlay.

The phenomenon of *churn* is a well-studied problem in DHTs [70], and overlay networks in general. Churn is the process of nodes continuously joining and leaving the overlay, and is particularly problematic in DHTs because of the shared key-space of member nodes; a node which leaves a DHT must have its keys transferred to a suitable remaining member, and a node which joins may also cause transfer of keys from another node that now lie within the new node's ID space. The problem itself is unavoidable, as users are always free to do as they wish, but the effects are an often-studied property for new overlays, and it remains an open area of research.

Again, it can be seen that fault-tolerance mechanisms in this class of overlay are deliberate additions to the overlay's core design, and again are not necessary to achieving its functional aims. Finally, unlike tree-based networks, DHTs are strongly reliant on a very particular topology in order to function correctly, leaving fewer repair-time options—either the repair maintains the correct, exact topology of an ordered ring / namespace, or it does not.

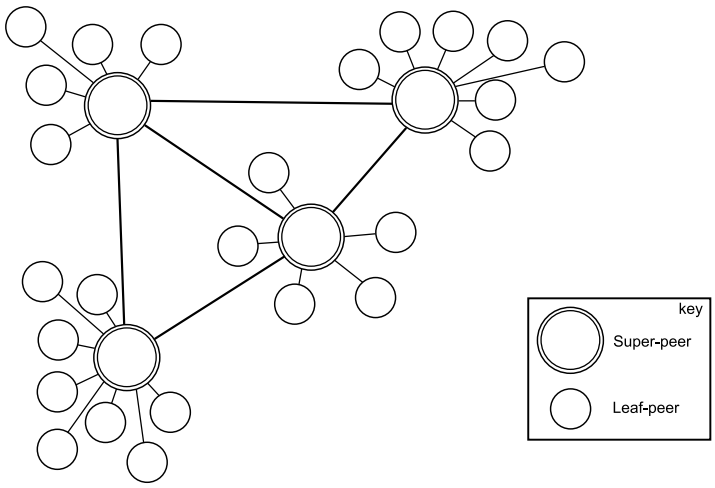


Figure 4: The topology of a typical super-peer overlay like Gnutella 0.6; leaf-peers cluster around a super-peer, and super-peers handle search queries within their own mesh topology

2.1.3 Semi-structured overlays

Functionality This section discusses a class of 'semi-structured' super-peer based overlay networks, used for storing and locating resources. DHTs provide this functionality, but unlike DHTs, semi-structured super-peer style overlays do *not* guarantee that a resource that exists in the overlay will always be found. Balancing this deficiency, however, is the ability to perform partial and *wildcard*-based searches of resources, rather than the situation in DHTs where the exact key must be known for the resource to be found.

An example of this kind of overlay is Gnutella 0.6, depicted in figure 4. This topology is an evolution of a more simple, pure peer-to-peer resource sharing protocol, created due to scalability issues of the latter (discussed shortly in section 2.1.4).

In Gnutella 0.6, a subset of the overlay's member nodes are selected as 'super-peers', either by advertising this capability to the overlay or being promoted to this status by the overlay protocol. Super-peers support a collection of leaf-peers, and store an index of the resources that those leaf-peers are sharing. Super-peers also form their own *super-peer network*, and when a leaf node submits a search for a resource, its super-peer examines its own resource index, and also forwards the search to the other super-peers it is connected to. Such a search is flooded through the super-peer network for a given number of logical (overlay) hops, and all results are returned to the originating leaf-peer.

Leaf-peers may then communicate directly with each other to actually transfer resources. In real-world systems, unstructured or semi-structured peer-to-peer networks like Gnutella are more popular for resource sharing, mainly because of their ability to perform partial / nearest-match string-based searches for resources. However, they have other advantages, including an inherent load balancing—popular resources are downloaded and shared at many peers, which spread the load of other peers downloading those resources. In Chord, however, a single node would hold a popular resource, servicing all requests for it. Further, in Gnutella, a peer decides for itself which resources it will store and share, whereas in Chord resources are assigned to a node by the overlay, which is perhaps not an attractive policy for end-users. The FastTrack/Kazaa [52] protocol is similar to Gnutella 0.6, and has seen wide acceptance and use within the Internet community.

Fault-tolerance In terms of dependability, the clear weak points in this kind of overlay are the super-peers [83], which enable the functioning of the overlay. The failure of a super-peer requires the leaf-peers it was supporting to re-locate to another active super-peer. As the number of super-peers drops, the load on the remaining ones clearly increases, which tends toward the emergence of bottlenecks in the overlay.

The locations of alternate super-peers can be provided to leaf-peers from the initial super-peer they were connected to, or can be listed for example on a website, showing well-known super-peer sites, for manual end-user lookup. Unfortunately, there is little published work on fault-tolerance approaches for super-peer systems at present—and again, they represent an added-extra beyond the core functionality of the overlay.

2.1.4 Unstructured and randomised overlays

Functionality The kinds of overlays discussed in this section have no defined structure, and all nodes have equal levels of responsibility. Gnutella 0.4, the precursor to the super-peer architecture discussed in section 2.1.3, was one of the first of this kind of overlay.

A node 'joins' a Gnutella 0.4 overlay by locating another active node (or peer), and making itself a neighbour of that node. It may then acquire a neighbour list from that node, and store a number of neighbours as its own. The new node will then send queries for resources by *flooding*

a query to its neighbours. Once a query reaches those neighbours, it is recursively forwarded through the Gnutella network until the hop limit of the query, or the edge of the network, is reached. This creates a 'search horizon' for queries, such that they reach only a subset of the total nodes in the Gnutella network—this means that one node has no guarantees of finding a resource that exists on another node.

Freenet [17] is a hybrid Gnutella/Chord like system, where resources are stored by hashing them to promote anonymity, and resources are stored on peers chosen by the overlay itself. Peers are connected in an unstructured way, and queries are performed as an exhaustive search through the network (as long as the hop limit of the query is not exceeded), though performance of this searching improves over time for popular items as correct routes to their owners are remembered by other nodes that have been used as part of successful routes.

Although very popular among end-users, it was eventually accepted that Gnutella-style flooding overlays put enormous strain on the physical network [81], resulting in the super-peer architectures discussed in section 2.1.3. Such super-peer architectures also significantly increase the 'search horizon' of queries, due to the amount of results that can be acquired from a single super peer (so that less search hops may attain many more results).

Both unstructured and super-peer resource sharing overlays are plagued by issues of 'free-riding' [39]—the situation in which very few users share resources, with the majority of users downloading resources without sharing any. Various schemes have been examined to try to discourage this behaviour [59, 4], but like the issue of churn in DHTs, it remains an open research question.

Moving away from resource sharing overlays, other completely unstructured overlays have been proposed for a variety of roles, generally related to (probabilistic) information dissemination, which come under the umbrella term of 'gossip' overlays.

In gossip-based overlays, each overlay node has a partial, randomly-generated view of the entire set of nodes in the system [46]. It is often impractical to provide 'perfect' random views of this kind, as this requires each node to have an accurate view of every other node in the system from which to select its smaller random view. As a substitute, partial, 'uniform-random' views are often used by selecting nodes from a partial list of the entire membership of the system—thus

approximating a random sample taken from the entire membership. The size of the selection is normally configurable, and using different view sizes alter the properties of the system (i.e. the probabilities of being able to disseminate information to all nodes). The issues surrounding this bootstrapping and maintenance of partial views are covered in detail in [43], where the authors propose encapsulating the solutions in a general 'peer sampling service'.

Assuming such partial views can be generated and maintained, a number of higher-level services have been proposed to build on this (e.g. [42, 27, 44]). They generally disseminate information by forwarding it to a random selection of the nodes in their view; it has been shown that with high probability information forwarded in this way will reach all nodes in the system [27].

Recent work in this area has proposed an architecture in which overlay nodes may have a large *scale* of different roles between, and including, that of super-peer and leaf-peer [73]. In this overlay, the highest 'utility' peers form the 'core' of the overlay, with lower utility peers gradually further away from the core. Peers maintain a set of random links to other peers (from a sampling protocol such as that described above), and a set of links to 'similar' peers, creating the clustering of the 'core' and the 'gradient' of responsibilities to the edges of the overlay.

Fault-tolerance Early flooding overlays such as Gnutella v0.4 [80] made no provision at all for fault-tolerance because it was assumed that resources would naturally be replicated over multiple nodes, and that flooding would likely locate a suitable copy. Additionally, the number of peers a node connected to was a simple way of improving its resilience to those peers crashing or otherwise leaving the network.

This approach applies equally to gossip-based overlays—due to the random links that are maintained and continuously updated in this kind of overlay, they are inherently resilient to node failures, and unlike all of the other overlays discussed in previous sections, have no 'added on' resilience approach which clearly operates in addition to the basic overlay protocol. However, it should be noted that protocol-based schemes for ensuring persistence of *data* at nodes in this kind of overlay are less well developed than those for ensuring communication resilience.

Table 1: Summary of overlays

	Core topology	Extended topology	Multiple valid structures	Multiple node types	Workload	Per-node view
TBCP	Tree	None	Yes	No	Child nodes	Partial
Overcast, RMX	Tree	Star(s)	Yes	Yes	Child nodes, client nodes	Partial
Narada	Mesh	Tree	Yes	No	Child nodes, forwarding support	Full
SplitStream	Tree (multiple)	None	Yes	No	Child nodes (per stripe)	Partial
Chord, SkipNet, Pastry	Ring	Fingers	No	No	Key storage, routing	Partial
Tapestry	Mesh	None	No	No	Key storage, routing	Partial
CAN	Hypercube	None	No	No	Key storage, routing	Partial
Gnutella 0.6	Mesh	Star(s)	Yes	Yes	Index storage, query handling	Partial
Gnutella 0.4	Mesh	None	Yes	No	Routing, serving resources	Partial
FreeNet	Mesh	None	Yes	No	Routing, serving resources	Partial
Gossip overlays	Mesh	None	Yes	No	Routing	Partial

Table 2: Summary of overlays' fault-tolerance

	Approach	Adaptivity to environment	Configurable	Re-usability
TBCP	None	N/A	No	N/A
Overcast, RMX	Ancestor list	None	No	Other trees
Narada	Alternate paths (mesh)	Continuous	No	None
SplitStream	DHT-layer support	None	No	N/A
Chord, SkipNet, Pastry, CAN	Successor list + replication	None	Yes	Other ring / hypercube DHTs
Tapestry	Replication	None	Yes	Other mesh-based DHTs
Gnutella 0.6	Alternate super-peer site location	None	No	Other super-peer overlays
Gnutella 0.4	None	N/A	No	None
FreeNet	None	N/A	No	None
Gossip overlays	Inherent (for topology)	N/A	Yes	Other gossip overlays

2.1.5 Summary

The overlay networks surveyed in this section are presented with their major properties in table 1, with their fault-tolerance properties in table 2, based on their published literature. The chosen properties represent the major characteristics of overlays in terms of topology, service provision and scaling, as well as the re-usability and generic applicability of their fault-tolerance approaches, which are the major factors of interest in this work.

In any practical deployment, any overlay network *must* have mechanisms in place to deal with situations in which nodes fail—a representative selection of approaches has been presented in this section for various different overlay networks, demonstrating the duplication of effort currently involved in this area, the lack of re-usability and easy configuration options, and the lack of adaptivity to their environment. This illustrates the benefits to the community that a general, re-usable and adaptive solution could have.

2.2 Overlay Deployment Environments

'Overlay deployment environments' are systems which use overlays networks as components, often transparently, to provide higher-level services. They are of interest due to their extensive use and composition of overlays, and their generalisation of overlay functionality to achieve this—an environment with such a rich collection of overlay networks would benefit greatly from a common, easily configurable fault-tolerance approach across all of the overlays, able to be automatically configured as suitable for each deployed overlay.

They are also of interest because in many cases they provide a more resource-rich setting for the overlay to operate in—different from the purely end-user host deployments normally considered by overlay designers. In this setting, hosts exist in the deployment environment beyond the purview of a single overlay, and these can potentially be harnessed if appropriate.

2.2.1 The Open Overlays Framework

The Open Overlays framework is a sub-layer of the Gridkit [31] architecture—a middleware for building configurable large-scale communication infrastructure over diverse physical networks. The overlays framework is responsible for configuring and deploying overlay networks at the direction of higher layers, and as with the rest of Gridkit, is based upon the OpenCOM component model [18, 20], allowing runtime reflection and reconfiguration.

Overlay nodes are constructed within this component runtime, and communicate with nodes in runtimes on other hosts using proprietary methods (e.g. TCP sockets). Overlay nodes within the framework have a well-defined structure; each overlay node has a *control*, *state* and *forwarder* component. The control component manages the topology of the overlay, handling 'join' and 'leave' requests. The forwarder deals with routing messages through the overlay, while the state component encapsulates any state required by the overlay. It is believed that this structure is a good fit for all overlays, and the defined structure allows Gridkit to compose multiple overlays in arbitrary ways to offer a wide range of configurable communication abstractions. An example configuration of a complete Gridkit node is shown in figure 5.

To give a concrete example, an application may request from the interaction framework (provided by the services layer) a publish-subscribe session over a given collection of hosts,

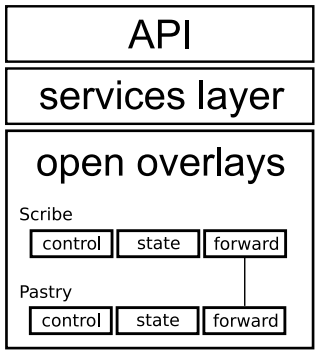


Figure 5: The architecture of a Gridkit node, expanding on the open overlays layer, which is hosting a Pastry and Scribe node, with the Scribe node re-using Pastry's forwarder.

spread across an Ethernet and ad-hoc wireless network. In an Ethernet infrastructure, the framework may already have deployed a Pastry Distributed Hash Table (DHT) to satisfy a previous application's requirements. For the hosts within this network, the framework may then re-use the Pastry overlay (possibly adding some nodes as required) and layer a Scribe overlay on top of it. For those hosts in an 802.11 ad-hoc network, there may be no overlays yet deployed, and framework will select an application-level multicast overlay such as Narada to support the publish-subscribe session in the ad-hoc network. Finally, a bridging point will be selected (i.e. a host with dual network interfaces) between the Ethernet and 802.11 networks, and configure a node at this point to deliver messages between Scribe and Narada (with appropriate header conversion).

Residing within Gridkit, the overlays framework has access to the environment's higher-level services such as resource discovery, and is therefore potentially able to leverage a very wide base of resources to perform its tasks—resources that are both within the scope of a specific overlay (i.e. the member hosts), and external to that overlay, but known to Gridkit.

2.2.2 ODIN-S

Like the open overlays framework, ODIN-S [19] specifies a particular way of constructing overlay nodes. Its design goal is to permit resource management between multiple overlay networks in which a host is participating, and it is proposed that this is achieved using message filters (incoming and outgoing).

A host using ODIN installs all of its overlay nodes within the ODIN framework, and all of

those nodes use a common 'communication manager' layer through which to send and receive messages. Overlay nodes themselves are architecturally separated into *routing logic* and *topology management* components, which are analogous to Gridkit's *forwarder* and *control* components for overlays.

Using a common communication layer allows transparent management of some key resources used by overlay networks through attached filters; the outgoing bandwidth used by each overlay can be tuned by having an outgoing filter buffer messages that exceed the overlay's upload quota, and the CPU load of an overlay can often be tuned by having an incoming filter similarly buffer messages (so that if an overlay is handling less messages per second, its CPU load should be less).

This, in turn, allows different overlay networks to be prioritised (as entire distributed systems), so that a mission-critical overlay is always given priority of bandwidth and CPU, and a less important overlay receives a more best-effort level of service. Various schemes are suggested by the authors to achieve this while preventing starvation of lower-priority overlays.

While ODIN brings an interesting approach to inter-overlay resource management, it does not offer anything else beyond the open overlays framework in terms of modularisation and reusability of components / overlays. Fault-tolerance (in terms of node failure) is not addressed as a general concern.

2.2.3 JXTA

JXTA [82] (pronounced "juxta", derived from abbreviating from the word 'juxtapose') is a general toolkit for building peer-to-peer systems, and offers the developer a layered and modular architecture with which to develop distributed systems, choosing which modules and protocols to use and how to best configure them for the particular application.

JXTA is comparatively well-established, though again lacks any coherent approach to fault-tolerance in its architecture. It also lacks the self-configuration of Gridkit, and the multiple-overlay resource arbitration capabilities of ODIN. The work described in this thesis in terms of a generalised, configurable and adaptive self-repair service for overlay networks, could be used equally well in any of these deployment environments.

2.2.4 iOverlay

iOverlay [53] is an evaluation environment for overlays, and focuses on a minimal interaction point (API) between the iOverlay middleware and the overlays that are hosted within it. Overlays need only use the *send* function of the middleware to communicate with other nodes. This simplicity is intended to make the development of overlays for the iOverlay framework as low-cost as possible.

iOverlay proposes an architecture with three layers; the message switching engine, the application-specific algorithm (overlay protocol), and the application. It also provides a centralised user interface so that algorithms can be visualised, monitored and debugged.

Overlays are written as C++ classes, implementing message processing functions to be called from iOverlay, and iOverlay provides all low-level communication handling (e.g. socket programming). Because iOverlay handles this, it can easily instrument all communications without the overlay algorithm needing to know it is being done. A major focus of iOverlay is on an efficient design for the message switching engine, as this supports all overlays in the environment; to this end, messages between nodes are sent over the same connection for the lifetime of the application, messages processed by the engine are never deep-copied (so only references are passed around), and the engine has a small in-memory footprint.

Physical hosts can support multiple iOverlay nodes, and each node and link can be given "virtual" bandwidth limits, so that experimentation options are not completely tied to the characteristics of the hosts used. Nodes can be terminated at will by the user to observe the effects of failures.

iOverlay (and similar work like X-Bone [79]) is a framework for user-driven experimentation and evaluation of overlays, where Gridkit is a self-configuring middleware layer driven by real application needs, automatically deploying/re-using overlays to meet a requested interaction paradigm.

Again, iOverlay does not offer standard fault-tolerance mechanisms to the overlays deployed within it.

2.2.5 Specification-language frameworks

There is some work in specifying overlays using a specifically created *language* or style; two examples of this are now summarised.

MACEDON [71] (**M**ethodology for **A**utomatically **C**reating, **E**valuating, and **D**esigning **O**verlay **N**etworks), and the follow-up MACE [47] systems, represent a common way to implement overlays. A developer writes their overlay for the MACEDON framework as a finite state machine (using a special script language) in which events trigger state transitions. MACEDON generates code from this specification, deploying nodes along with the MACEDON engine library which provides transport and event invocation. Overlays are specified in a sufficiently abstract way that the transport engine can use various real network protocols, or can operate over the network simulator ns-2 [11]. Overlays can then be evaluated against one another using a common programming model and underlying engine for each overlay, which hopes to promote fairer comparisons between overlays, eliminating factors such as specific programming language (and even programming style) performance issues.

Overlays can be instrumented to various levels of detail by the framework for evaluation, and overlay composition is supported in a similar way to Gridkit, using similar standard APIs for overlays to enable this composition. If an overlay specifies a need for failure detection, MACEDON will detect failures on its behalf and report to the overlay when a node fails. The overlay is then expected to handle the failure appropriately.

MACEDON, like iOverlay, is a framework for experimenting with and evaluating overlays, rather than a middleware service supporting real applications.

The work in [8] proposes the use of database concepts to specify overlay networks (as does P2 [54]). In [8], an overlay node is maintained as a *database* containing knowledge of a subset of other nodes from the global topology. The application part of the node (i.e. topology creation) is written in a specially developed language called SLOSL (SQL-Like Overlay Specification Language), which operates on the local database to define the overlay.

For example, part of a Chord overlay specification will define a *view* from the local database to represent its finger list. Populating the local database with nodes to fill these views is therefore a critical part of the approach; methods of acquiring this information are encapsulated in a

'harvester'. Harvesters are interchangeable and can use various gossip, broadcast or centralised lookup protocols as desired. Remote nodes within a view can be 'ranked' by arbitrary attributes such as latency or bandwidth to modify the performance of the overlay by selecting neighbours according to an attribute.

This is perhaps the highest level framework in which to develop overlays, as everything but the structural rules of the overlay is abstracted by the framework. Tools to move from this high-level specification to actual implementation are yet to be fully developed, however.

2.3 Dependability Services

The ultimate goal of this work is to find a solution to generalising fault-tolerance for overlays—this section surveys existing work on 'dependability services' to explore their possible applicability to overlays. There are a wide range of different techniques which offer general failure handling capabilities; this discussion focuses on those aimed at supporting crash-failures only, and ultimately concludes that most are fundamentally unsuitable for overlays.

A number of techniques are often combined to provide fault-tolerance, and these are discussed individually first, including checkpointing, replication and failure detection. Services that provide all of these together are then discussed.

2.3.1 Checkpointing

Checkpointing [25] is widely used in dependable computing to record a 'good' state of a process, and restore it to that state should it fail. Checkpoints are generally intended to be used for *deterministic* processes; those which will eventually arrive at the state they were in when they failed even when reverted to an earlier state. The aim with checkpointing then is to minimise the amount of work that a process has to re-do due to a failure, while impacting the failure-free performance of a system as little as possible.

A survey of checkpointing approaches is presented in [25], which is now summarised. Most checkpointing concerns are around the problem of consistency—in a distributed system, if a process fails and is restored to a previous good state, other processes that were interacting with it may need to roll back to a consistent point in *their* execution history. Following the failure of a single process, then, the *entire system* is required to be in a mutually consistent state when

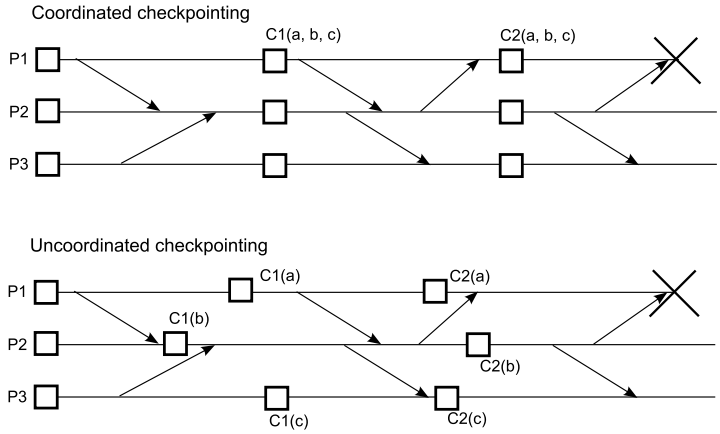


Figure 6: Coordinated and uncoordinated checkpointing approaches. Horizontal lines are process timelines, squares are checkpoints and arrows are messages sent from one process to another.

recovery is completed, so that no processes record a message having been received when their senders don't record them having been sent and vice-versa. Such a state is known as 'globally consistent', and the job of the checkpointing approach is to record globally consistent states which are as recent as possible.

To achieve this, processes can either communicate to determine a globally consistent state from which they can be quickly restored (*coordinated checkpointing*), or processes can independently take checkpoints and try to establish a globally consistent collection of them at the time of recovery (*uncoordinated checkpointing*). [25] surmises that the former approach has become the most suitable one; although it requires additional control overhead to organise the taking of a checkpoint, in modern systems this cost is negligible compared to the cost of actually saving state, such that the overhead of both approaches during failure-free operation is almost the same. It is argued that the very minor performance gain in uncoordinated checkpointing does not then justify the problems of trying to locate a globally consistent point from them when a failure occurs. Both approaches are shown in figure 6.

From the figure, it can be seen that if a failure occurs in process P1 (the large X shown), in the coordinated approach the set of checkpoints C2(a, b, c) can be used to roll back all processes to a consistent state. In the uncoordinated approach, the set of checkpoints C2(a, b, c) is inconsistent because process P2 has recorded sending a message which process P1 does not record having received. The same is true of the set of checkpoints C1(a, b, c) between the processes P3 and P2. In this case, then, the three processes are rolled back all the way

to their first checkpoints, which represent the only globally consistent state that was saved. While coordinated checkpointing is generally therefore preferable, in a mobile environment it may actually be more appropriate (or even necessary) to use the uncoordinated approach if devices have limited or sporadic communication capabilities.

This same reasoning applies to overlays—taking a globally consistent coordinated checkpoint of an entire overlay is simply infeasible, and usually unnecessary due to the way overlays work. Consistent checkpointing is most suitable for long-running scientific applications (checkpointing is successfully used in Condor [78], for example) rather than interactive, dynamic systems. Overlays in particular offer a system model in which *any* node can normally perform the services of the overlay, and it is simply the essence of that service which must be maintained—the members in a tree to whom data must be forwarded, the key-value pairs in a DHT, or the resource indices in a super-peer network.

As long as there are redundant copies of this information, the responsibility for delivering these services can be assigned to other nodes without fear of inconsistency or corruption of the system. Overlays can therefore be seen as often employing a kind of uncoordinated checkpointing of highly specific state, without consideration of consistency between nodes.

2.3.2 Replication

Replication is an alternative to checkpointing to create resilient systems, though the two approaches are somewhat related. Using replication, multiple *copies* of a process are kept running at the same time, so that if one of the processes fails then a replica can take over its responsibilities. Checkpointing could therefore be seen as a kind of "offline" replication, keeping 'inactive' copies of a process, and as with checkpointing, the major issue with replication is consistency; ideally all replicas of a process should be equally up to date at all times, but this can cause significant overhead.

It is useful to consider replication in terms of a *server* which provides some service to clients. This server is then replicated to improve its availability. *Clients* have two broad types of operation that they can perform on servers; a *write* operation, which changes the state of the server, and a *read* operation, which returns some value from the server without causing a state change. There are three major classic replication schemes, shown in figure 7; active, passive, and gossip based.

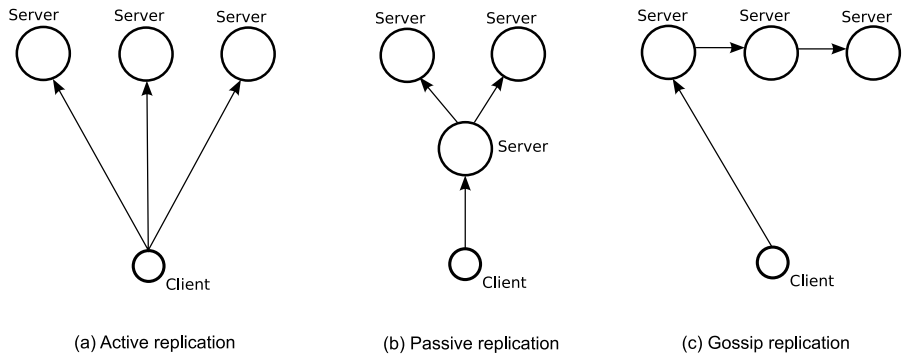


Figure 7: Active, passive and gossip-based replication schemes, showing client-server and server-server interaction for a single request from the client

Active replication [26] (see figure 7 (a)) is targeted largely at providing fault tolerance, where a group of server replicas is kept and client write operations are passed concurrently (often using a multicast protocol) to each server in the replica group. If every server is executing exactly the same requests in the same order, and those servers are deterministic, then they will always have the same internal state. If a replica fails, the others in the group can continue as normal. Client read operations can be handled by a single server in the replica group, thus offering some load-balancing. Active replication is also useful for tolerating *processing faults*, since every replica is asked to perform the same tasks—the majority result of an operation can be taken as the accurate one and any others discarded (though, in this setting, it may be necessary for clients to also multicast *read* operations to all servers).

Passive replication [66], shown in figure 7 (b), has the notion of a *primary replica* which clients interact with, and backup replicas which are kept up to date by the primary but which never interact with clients. The general approach is that a request arrives at the primary, which handles it appropriately, and then makes the request known to the backup replicas. If the primary fails, one of the backups can quickly take its place.

Gossip based schemes such as [51] offer relaxed consistency, using replication to improve the general throughput of the system rather than purely for fault tolerance. The consistency conditions are that the *client* must observe a consistent service from the replica group, as if there was only one copy of the server to interact with. Provided that this holds, the processes within the replica group can be in any state of consistency, with only the guarantee that if there were no more client requests, the replicas would *eventually* become consistent through the exchange

of gossip messages. This allows different clients to use different replicas in parallel for both read and write requests, hence increasing the throughput of the server group. The failure of a replica in this case could be problematic as some recent updates at that replica could be permanently lost before they were gossiped to other replicas, but the degree of consistency between replicas can be scaled based on the gossip interval chosen.

As with checkpointing, it is clear that classic replication is simply inappropriate for overlays—largely due to the paradigm shift from 'client-server' to 'peer-to-peer', where every node may serve both client and server roles. In particular, the model of running replicas of other peers is not only unattractive to end users, but the resource intensity of it in terms of running the additional processes and keeping them consistent is generally unsuitable.

2.3.3 Failure detection

Failure detection is at the heart of dependable computing, being simply the ability to know whether or not an entity is still alive. Unfortunately, it is very difficult to know this about a remote entity with any certainty—a node is typically probed with 'are you alive?' messages, but the receiver may be too busy to respond to such a message quickly enough, or network congestion could drop the probe or response packets, causing a node to be falsely declared as failed (a *false positive*).

Failure detection research has focused on limiting the number of false positives while detecting genuine failures as quickly as possible. A study of various approaches in [87] outlines some general principles for designing a failure detection system, which are summarised in the following.

There are many types of failure detectors, the most simple approach being for a node to probe its neighbours periodically and decide on its own whether any of them have failed (i.e. there is no response to probes). This can lead to a large amount of false positives for the reasons outlined earlier, and improvements can be made by having nodes *share* information about the liveness of other nodes—in sharing schemes, all the neighbours of a node C may share information with each other about the liveness of C. If a majority of C's neighbours believe it has failed, it is less likely to have been unable to respond to all of those neighbours, and all of the links to its neighbours are less likely to have been congested, therefore there is a smaller probability of false positive.

The above scheme requires that the neighbours of C maintain links to each other, however, which introduces additional control overhead. In some overlays this can be avoided where it is appropriate for nodes to simply share information with their normal neighbours about the liveness of another node. This approach requires no extra state, but relies on a sufficient number of node pairs sharing a common neighbour, so that the pair is able to share information about that neighbour.

Nodes can share positive information (i.e. *I think this node is alive*), negative information (i.e. *I think this node is dead*), or a mixture of both. Generally, sharing positive information increases control overhead in failure-free systems, but reduces the number of false positives. It is also important to take into account the number of neighbours that the node C has when deciding on a failure detection probe interval—ideally C should receive a constant rate (i.e. a constant N every minute) of probes irrespective of how many neighbours it has, else it could easily become overloaded with handling many probes. Finally, the interval of failure detection probes is a consideration—if they are sent more frequently, a failed node may be detected more quickly, but a higher control overhead is incurred.

Significant work on general failure detection methods includes [69], in which nodes gossip probabilistically with each other to share liveness information instead of actively probing. Each node maintains a list of other nodes that it knows about, and a counter for each, as well as the last time the counter was incremented. Every gossip interval, a node will select one other node at random from this list and send its own list to it with an incremented heartbeat counter for itself. This basically serves as an 'I am alive' message from a node. This list is merged at the receiver with the local list, using the highest heartbeat counter value for each member on the lists. A node is declared dead by another node if it has not received a new heartbeat within a given time. Due to the random selection of a neighbour, the protocol's accuracy is based on probabilities, and it is expected that every member knows about all others for these probabilities to hold.

Taking the active probing approach, [34] has a node send a failure detection probe to a neighbour (call it A), and if no acknowledgement is received from A, sends a probe request to a random set of other neighbours, asking them to probe A and report their findings—this

out-sourcing reduces the probability of false positives as already discussed. The approach can guarantee an equal expected load on all participants, and has configurable speed and accuracy parameters. As with the gossip approach, the view of participants maintained at each node is important in the performance of the protocol.

Little is published about the failure detection approaches used in current overlays, but a number of existing approaches are sufficiently well suited for general use, and are discussed as such in chapter 6.

2.3.4 Fault Tolerant Software Architectures

Architectures which provide complete dependability services to other systems are now discussed—they typically take checkpoints, detect failures, and restore failed modules from their last checkpoint. They differ mainly in how these elements are executed and integrated into a complete service.

Chameleon [6] is one such system, which is now described in some detail. Chameleon uses objects termed ARMORs (Adaptive Reconfigurable Mobile Objects for Reliability) to provide fault tolerance to hosted applications; these are essentially deployable services, where different ARMORs will provide different kinds of services within the fault tolerant environment. All Chameleon elements (i.e. belonging to the middleware) are ARMORs, and are "rigorously tested modules", such that ARMOR failures are expected to be the result of external problems.

The overall Chameleon execution environment is managed by a Fault Tolerance Manager (FTM), installed manually on an arbitrary computer by a domain administrator. This (primary) FTM creates a backup FTM, which monitors the primary and takes over if it fails. The FTM manages membership of the Chameleon environment (i.e. the available hosts), accepts requests to execute applications from users with attached availability requirements, and deploys appropriate sub-managers and other ARMORs to control and monitor those application modules. The FTM also maintains a history of execution for member hosts so that it can decide how reliable an application will be if it is deployed on a given host, based on past experience with that host.

A daemon runs on each host that is a member of the Chameleon environment, which handles the deployment and monitoring of ARMORs on that host, acts as a gateway for communication between managers and all ARMORs on the host, and responds to heartbeat requests. The basics

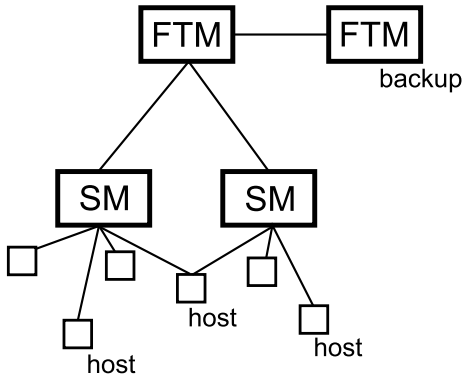


Figure 8: The main architectural elements of Chameleon. Additionally, each host shown runs a daemon, plus a number of application modules and associated ARMORs. Lines represent 'usage', such that the left SM (sub-manager) is using 4 hosts to execute an application. To avoid clutter, the diagram does not show the fact that the FTM holds references to every host in the system.

of this architecture are shown in figure 8.

During failure free operation of an application, various ARMORs monitor it; a heartbeat ARMOR is used to determine the liveness of hosts, an execution ARMOR deploys application code and monitors its progress (reporting the result to the sub-manager), and a checkpointing ARMOR periodically checkpoints the running application module.

When a failure occurs, it is handled according to which type of failure it is. Host failures are dealt with by the FTM, which maintains a membership list of daemons available in the environment. If a heartbeat times out, a host is presumed failed, and the FTM informs the sub-managers owning the application modules and ARMORs that were on that host. Sub-managers then restore application modules from checkpoints on alternative hosts.

Application module failures are 'detected' by execution ARMORs (the detection mechanism is through the application signalling the ARMOR that something is wrong). In this case, the application is re-started from its last checkpoint, or from the beginning of its execution. If the same failure occurs multiple times, the application is moved to an alternative host and re-started as before.

ARMOR failures are detected by daemons and flagged to a manager, which may either re-install the affected ARMOR on the same host, or move it to another host. Chameleon is one of several job-submission style environments where applications are registered as tasks to be completed in a fault-tolerant manner.

Phoenix [48] is aimed at Grid environments, and uses 'failure agents' placed with various

systems, which are responsible for discovering and classifying failures, and a failure manager to make decisions on an appropriate response to a failure. The failure manager has access to a knowledge base of past failures and responses so that it may be able to tune its future responses based on the quality of past ones. Phoenix uses application-specific knowledge to try to track the cause of an error—for example, error tracking for the failure to locate an URL may involve checking if the DNS server is alive, checking if the DNS server has an entry for the URL, checking the relevant networks and hosts are alive, ensuring the necessary protocols are available, and so on.

The Meta system [57] is a framework for the management of distributed applications, which advocates the separation of the functionality of an application from its management aspects—notably performance and resilience to failures. A 'control layer' performs all management of an application, and is basically a set of condition-action rules which define a policy. Applications and their environments are instrumented with sensors and actuators which tie in to the policy rules of the control layer. Example sensors are the measure of CPU load and the detection of failures, and example actuators the ability to promote a process' runtime priority, re-distribute work following a failure or recover a process from a checkpoint. The policy language and sensor/actuator integration are the major contributions of the work, and it uses the Isis [10] toolkit to provide fault tolerance, using its message logging facilities to take checkpoints for critical system components.

DARX [56, 55] is another fault tolerance framework, aimed at providing dependability to distributed multi-agent systems (e.g. [24]), which are composed of different types of agents cooperating to solve a large and/or complex problem. DARX uses adaptive replication to offer fault tolerance, instead of checkpointing as used by the systems previously described in this section. A DARX server runs on every host containing an agent of the system to be made fault tolerant. This server maintains a replica group of its agent (i.e. a group containing replicas only of that agent), applying a replication scheme and size of replica group according to how critical the agent in question is deemed to be. The scheme (e.g. active or passive) can be changed dynamically as the 'criticality' and environment of an agent changes. A failure detection service maintains a list of all agents and DARX servers, and each replica in a group has an associated degree of consistency indicating how up to date it is likely to be. When a failure occurs, the

replica with the highest degree of consistency is chosen as the new group leader.

In terms of overlay networks, all of these services have one major problem—they employ specialised management elements which require their own servers on which to operate. This is clearly in conflict with the peer-to-peer nature of overlays, which are designed specifically to avoid reliance on any managed infrastructure. Any generalised service supporting overlays must therefore itself avoid such a use of managed infrastructure, adopting a peer-to-peer architecture to achieve its aims within the same constraints of the overlay itself.

However, the clear separation between *redundancy*, *failure detection* and *recovery* is a common theme, and suitable for transferral to an overlay setting, in which these three concerns are individually addressed and tailored according to their deployment environment.

2.3.5 Alternative approaches

Two other noteworthy approaches to dependability are now briefly discussed in order to provide a glimpse beyond the more classical work in the area.

Graceful degradation Shelton [75] writes that "the term graceful degradation means that a system tolerates failures by reducing functionality or performance". Graceful degradation research focuses on this ability of a system to tolerate subsystem failures until those failed systems are repaired, and typically does not concern itself in how failures *are* eventually repaired. A design framework is proposed in [75] for gracefully degradable systems, which is now summarised.

First, it is important that systems are built in a 'gracefully degradable' way; they should be able to perform some of their functions even when parts of the system are missing. This requires some careful analysis of the design of the system to elicit the configurations in which it can provide some level of service. This analysis phase can make graceful degradation unattractive for large and complex systems (the approach does not 'scale' from a design point of view), so a different, non-exhaustive approach is proposed, where a system is divided into a set of subsystems with an 'input' and 'output'. The internals of a subsystem are then unimportant (and may be a further collection of subsystems), the only concern is that the subsystem is capable to taking some input and producing some output.

Every subsystem is allowed to communicate only through system variables, so the number

and type of faults possible is reduced to errors surrounding these system variables; a variable could fail to be updated (written to) in a timely manner, or a variable could be corrupted to either an invalid or a valid but incorrect state. These conditions are often detectable, and the response is to either make do without the value expected from the variable, or attempt to acquire compensatory data from another subsystem.

The proposal presented in [22] is an example of using adaptive systems at the software component level to gracefully degrade as failures occur. A system is seen as a set of "black-boxes" which are interconnected through defined interfaces, and it is assumed that the exact services provided by the block-boxes cannot be easily located elsewhere should they fail (e.g. if tied to a particular piece of hardware). In this environment, it is proposed that the fault management system operates on the *connections* between components instead of making the components themselves resilient. If a connection is not satisfied, an alternative (but similar) service is sought from those still running, and if necessary, connectors are inserted which adapt the service provided to the one expected. Clearly a knowledge of what a "similar service" might be is required at design time, as well as possible adaptation strategies, though these are not covered in detail.

Software rejuvenation Rejuvenation has existed as a technique for many years, and is "the concept of gracefully terminating an application and immediately restarting it at a clean internal state" [38] while the system is in a failure-free state. The theory behind this is that the longer a system is left running, the more likely it is to fail, so if it is periodically re-booted in a clean state at an opportune (i.e. lightly loaded) time before any failures have occurred, the number of failures at peak load can be reduced. Recent work in this area includes *recursive microreboots* [12], which is now explained in some detail.

The concept of rejuvenation exists because most complex software contains programming errors, which are a major source of downtime (and therefore a major source of profit loss in businesses that directly rely on software for revenue). There are two main aspects of microre-boots: 1) Building a fault propagation graph for the system, calculating the mean time to failure (MTTF) of each component, and re-booting those components before their MTTF expires, and 2) Responding to failures at runtime, when a component becomes unresponsive or otherwise

unstable; in this case the component is re-booted and the system re-assessed. If the system is still unstable, possible fault propagation paths from the original component are examined, and components at the other ends of those paths are rebooted. This expanding reboot continues until the system is returned to a working condition, or the entire system has been rebooted and still has problems (at which point user action is required).

This approach requires some strict rules for the applications it can support, which must be built from so-called "crash-only" components. Such components can be forcibly shut down and restarted with no warning, requiring no invocation of a 'shutdown' procedure, without losing any data when this happens. Further, components must be able to tolerate momentary unavailability of their connected components when they are being rebooted in this manner. This requires that components have all of their persistent state on dedicated storage, which itself is built in a crash-only way.

2.4 Overall analysis

A wide range of overlay networks have been discussed, along with their approaches to dependability, which have been developed over time in an ad-hoc, per-overlay manner. Even with this ad-hoc design of fault-tolerance measures, some strong similarities have emerged within particular overlay classes, evidenced most strongly in the tree-based and DHT overlay classes.

These two facts point to an opportunity to return to the issue of overlay fault tolerance in a more carefully considered way, drawing on the similarities in some overlay repair schemes, and on the work of the wider fault-tolerance community.

Examining overlay deployment environments has shown another perspective on the usage of overlays, beyond the single case of a purely end-user host deployment. It has also shown that overlays themselves are beginning to be generalised, and *componentised*, such that different overlay types, and indeed different overlay components, can be re-used and composed to suit a given application and deployment setting. What these overlay deployment environments do *not* consider, however, is the componentisation, re-usability and composability of fault-tolerance mechanisms alongside the overlays that they support.

This seems like an odd omission, given the long-established theme in the dependability com-

munity of separating the concerns of an application's functionality from its fault tolerance, either via middleware, toolkits, or management frameworks. The clear benefits of this would be that, as well as choosing how to compose an overlay's functionality with the choice of components and layerings, the user of such a deployment environment could just as easily then choose how the resilience of the overlay to failures should operate, by choosing and configuring appropriate resilience components.

Relevant fault-tolerance research has been discussed, demonstrating the state-of-the-art in common approaches to responding to node failure in distributed systems. Classical approaches to fault tolerance have been shown to be largely unsuitable to the scaling and dynamicity requirements of overlay networks, with the exception of failure detection approaches. A sample of more recent work has been presented, and shown to be too specific to particular kinds of application, constructed in specific ways.

All of this leads to the following set of requirements:

1. Any fault-tolerance mechanism supporting an overlay network must *scale* seamlessly with that overlay.

2. Mechanisms should be *re-usable* and *composable* with a wide range of overlay networks—generic components must be devised to support this, along with a generic API.

3. Mechanisms should take their deployment environment into account, considering resources, failure models, and system criticality in their decisions—and further, should be able to adapt dynamically as factors change.

4. Mechanisms should be *generic* across deployment environments as well as the overlays they support—a scheme that functions in a purely end-user host environment should be fundamentally compatible with more resource-rich environments, but should also take advantage of the latter when possible.

5. The approaches taken should be suitable for past and present overlay network designs, and should as far as possible be forward-compatible with likely future designs.

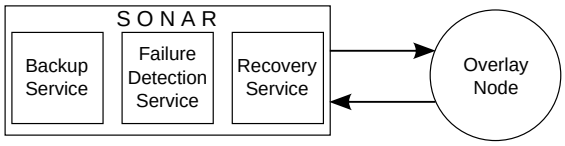


Figure 9: SONAR, horizontally composed with an overlay node

3 Services for Overlay Network Adaptive Resilience

This chapter proposes a solution to generalised fault tolerance in overlays, named SONAR. The overall vision of SONAR is presented, as well as how *runtime adaptivity* of any dependability mechanism is a feature of growing importance for near-future distributed systems.

3.1 Approach

SONAR consists of a framework of services—the overarching framework can load and configure sub-services which address a particular area of fault-tolerance. An instance of SONAR resides alongside *each* overlay node, as shown in figure 9.

This establishes the peer-to-peer nature of SONAR, which operates within the same infrastructure of the overlay itself, filling requirement 1 from the previous chapter. Every sub-service is required to use only 'soft state' (i.e. state that can be re-built from instantiation simply by existing in the environment), so that services are themselves inherently self-repairing, and each service must be able to operate in an entirely decentralised manner, matching the model of many overlays.

SONAR uses three major sub-services, following the model of the classic dependability services discussed in section 2.3.4. These are i) a *backup* service, ii) a *failure detection* service, and iii) a *recovery* service; these three are required in order to provide all aspects of self-repair for overlays.

This section discusses these three services abstractly, in terms of their general functionality, configurability and adaptivity, and the service model they must provide (i.e. their 'contract').

3.1.1 Backup service

The backup service is used to redundantly store the vital information that constitutes an overlay node—the details of this information are ultimately determined by the overlay (explained in chapter 4), but encompass everything that might be needed to fully restore a node should it fail. This typically involves the topological position of a node, expressed through its neighbour links, and any 'additional state' required, such as DHT data in Chord. No inter-node coordination is needed, with nodes simply making backups of their own state as appropriate. To meet the goal of *decentralisation*, this backup data is normally stored on one or more appropriate 'backup hosts' in the overlay (other than the host of the origin overlay node), though other storage locations may be used in different deployment environments, such as dedicated regional backup storage servers.

Configuration and adaptivity The major *configuration* options for the backup service are the *amount* of redundancy to use, and the *locations* to store that backup data. The manner in which backup data is stored can also be a detail of the service's implementation, for example storing complete backups of nodes, or fragmenting their backups into smaller parts and storing them separately.

The *runtime adaptivity* of the backup service involves the continuous self-configuration of the amount of redundancy used and the locations used to store that redundancy, according to the dynamics of the environment in which the overlay is deployed. The former would be a result of the observed stability of the overlay, with a minimum level of redundancy specified by the overlay deployer to suit their particular application.

The locations at which backup data is stored would depend upon two major factors: i) the speed with which the data can be retrieved again if needed, and ii) the 'safety' of the location; for example storing a node's backup at a host outside that node's physical network domain may be wise in case the entire domain becomes disconnected from the rest of the network. In this case, surviving overlay nodes would still have access to the relevant backup data in order to make repairs.

Another aspect of safety is the observed *stability* of a host on which backup data may be stored, and its projected time to failure based on knowledge accrued over time (i.e. the 'availability'

of the host). Several studies have analysed the behaviour of end-user hosts participating in an overlay network [74, 9, 65], and such behavioural data could be used as a basis for judging the availability of hosts in the overlay, alongside data collected at runtime.

Finally, the *capabilities* of hosts used to store backup data may be taken into account where appropriate—in a diverse overlay deployment involving hosts of greatly differing capacities, those with lower free storage capacity, for example, should obviously be asked to store less backup data.

Such adaptation of the backup service is essential in diverse systems where hosts of different capabilities share the same overlay network—today's systems are seeing increasing integration and collaboration of powerful computers with smaller devices [36, 32].

In addition, with the enormous scale of many overlay networks, it is inevitable that different *regions* of some overlays will experience different levels of failure and host availability—which may further evolve over time, with many overlays intended to run almost indefinitely [50]—and thus adapting the amount of redundancy both regionally in the overlay, and over time as conditions change, is essential to permit the practical use of such large scale systems. Constant human administration of such fine details in such a vast system is almost impossible, and the system must therefore perform adaptation and configuration autonomously.

Service contract Generally, the 'contractual obligations' of the backup service come from the needs of the recovery service. The backup service must store the essential data of each overlay node (defined as the data necessary to fully restore each node), and be able to retrieve such data when requested by the recovery service. Where the data is stored is an implementation detail, but it is assumed that a host that was supporting a failed node will be inaccessible following the failure, so other hosts should be considered for backup storage.

The specific recovery service implementation discussed later in this thesis has some particular assumptions about the capabilities of the backup service, and these are discussed in section 5.2.1.

3.1.2 Failure detection service

The failure detection service is used simply to detect node failures in a decentralised manner.

Configuration and adaptivity The abstract configuration options of a failure detector are 'speed' and 'accuracy', where one can be traded for the other. These options normally map to the probe interval (whether this be for 'active probes', heartbeats or gossip messages), and the time after which a node is declared as having failed, due to not having responded within that period. Of course, the specific *implementation* of the failure detector, such as those discussed in section 2.3.3, can also be part of the 'configuration', as appropriate to the deployment.

The *runtime adaptivity* of failure detectors relates to the continuous self-configuration and adjustment of these configuration options—if a host is known to be slow, or the physical network paths to that node slow or heavily congested, the amount of time the host is given before being declared as failed can be increased. In addition, the *amount* of probes sent to such a node, if an active probing approach is taken, may be *reduced* to alleviate some of the host's workload, increasing its chance of being able to answer one of the less frequent probes.

Such adaptation of the failure detector is essential because in any but the most small and localised distributed systems, hosts and networks of varying speed and available processing capacity *will* be encountered, and a homogenous standard of failure detection will cause degraded accuracy and performance as a result.

Service contract The contractual obligations of the failure detection service again come from the needs of the recovery service. The failure detector must i) accept 'registrations' of nodes that need to be monitored (i.e. the neighbours of an overlay node), ii) inform the recovery service if one of these registered nodes is detected as failed, and iii) accept 'active' requests from the recovery service to enquire if an arbitrary node is believed to be failed. The latter functionality is used during repair operations, as will be discussed later.

3.1.3 Recovery service

The general behaviour of a recovery service component is, on learning of the failure of a node from the failure detection service, to create some kind of strategy to repair the overlay. A detailed discussion of the protocol developed with this work is deferred to chapter 5, but a brief overview is provided here.

In order to allow *diverse* repair strategies, the repair protocol developed as part of this work

centres around *agreement*. If all the nodes that depended on a failed node must *agree* about i) the fact that it has failed, and ii) the action to take to repair the failure, then different repair strategies can *safely* be chosen and enacted dynamically at runtime.

The approach taken also handles entire failed *regions* of overlay network as well as single failed nodes—a necessity for true genericity. Finally, it is entirely *decentralised*, as can be the previous two services, in order to both scale with the overlay network and remove any requirement of managed infrastructure to use the service. Example repair strategies are that of *restoring* failed nodes on alternative hosts, in a resource-rich deployment environment, or that of simply *adapting the structure* of the overlay, and the distribution of responsibilities within the overlay, so that surviving nodes take over the roles of failed ones, if operating in more resource-constrained deployment environments.

If less varied capabilities of the recovery service implementation are sufficient to a particular deployment, other implementations may be used for this component instead, such as gossip-based topology management [42].

Configuration and adaptivity The *configuration* options of the recovery service are more subtle than those of the previous two services, and centre on repair strategies—the choice of which ones should be considered for use by the service, and which metrics are important to the deployment. When adapting the overlay's structure, for example, and re-distributing the responsibilities of failed nodes among selected survivors, which survivors should be chosen for the added responsibilities? Should it be those with higher bandwidth, lower latency, or higher spare CPU?

In terms of *runtime adaptivity*, the recovery service's aim is to make repairs such that the short and long-term performance of the system is as optimal as possible, according to the needs of that system, by selecting repair strategies in an intelligent way according to the dynamics of the overlay's deployment environment at the time the repair is required, and the metrics of importance to the overlay, as mentioned above.

The recovery service has no contractual obligations to other services.

3.1.4 Other services

The three services discussed above are necessary to provide all aspects of self-repair to overlay networks. There are, however, other concerns that may be separated and generalised in a similar way, to further 'componentise' the construction of overlay networks.

A fourth service considered in this work was a 'cloning service', proposed to continuously *optimise* overlay networks, in as generic a way as possible. The general premise behind this service is that it would examine the overlay's behaviour, and the capabilities of its supporting hosts, and re-distribute workload as appropriate. It is based around the observation that surprisingly few overlay networks are suitable for deployment in heterogeneous environments, and expect that each host is equally capable—many distributed hash tables behave in this way [76, 68, 72], and indeed some attempts have been made to distribute workload more suitably in these overlays, without changing the basic behaviour of the overlay itself [45, 67].

Many overlays supporting application-level multicast services also lack provisions for hosts of greatly varying capabilities, leaving a large area of work unsuitable for diverse physical networks. The so-called 'cloning' service was therefore designed to work in a Grid-like environment, and was to *create* overlay nodes and re-distribute workload to them as appropriate. It used an abstract 'workload' element, defined in detail by the overlay network itself, such that overlay nodes would advertise to their supporting service instance their current workload units.

The cloning service was then permitted, through special API methods *addWorkloadUnit()* and *removeWorkloadUnit()*, to re-distribute this advertised workload as it saw fit. In multicast trees, for example, new tree nodes would be created on suitable hosts, and units of workload would be expressed by the overlay as child links—these child links would be re-distributed to new nodes created on available hosts, and moved away from low-capacity hosts in the overlay, to improve performance.

Another general service of interest may be one that disseminates information about the deployment environment, such as host capabilities and current loading, host availabilities as observed over time, and similar data about physical network paths. The other services could use such data to determine where best to store backup data, how the failure detector would behave towards different hosts, and which repair strategies to use. An overlay network, for its part,

could use this data to determine node status and responsibilities (such as 'super peer' or 'leaf peer').

While this thesis does not examine such further separation of concerns in any more detail, it is interesting to imagine other areas which could be generalised into re-usable services in the same way that self-repair / resilience is generalised in this work. The concepts described in this chapter, along with a preliminary evaluation, were originally published in [61].

4 SONAR **APIs**

This chapter presents the API that was developed for SONAR—how the service is able to interact with and support various different kinds of overlay network. Designing an API for a standard service presents three major challenges:

- It must be *easy to use*, so that application developers do not have to spend long understanding its use. While "ease of use" can be difficult to quantify, it is often analogous to "simple" and "natural"—an API should have a simple method set, and those methods should behave in a natural way to the target user.

- It must be *expressive*—while simplicity is desirable to allow quick use of a service, the developer wishing to do more complex operations (either to support a more complex application, or to use the service in a way more 'precisely' tailored to their application) should be catered for.

- It must of course be *generic*—fulfil the above, while remaining applicable to as wide a range of different applications as possible. Unfortunately there is often a conflict between *genericity* and *performance*. A tailor-made solution to a problem almost always outperforms a generic solution—the challenge is to make the generic solution performant *enough* that the tailor-made solution is in most cases more effort to create than the performance gain it will provide.

Because the API is the conduit through which SONAR sees the applications it is supporting, this 'view' must therefore be sufficiently generic as to describe a large number of applications, but sufficiently *detailed* to allow *efficient* management of the application.

To give an example, the detail of 'successor', and its semantics—the node in the overlay with the next highest ID—is generic to the Chord and Pastry DHTs, but rules out many other overlays if this detail is a *necessary* part of the standard API. Removing this detail from the API allows other overlays like Overcast to use the service, but reduces the effectiveness of the service when applied to Chord, as it cannot know the meaning of 'successor' when making repairs to the overlay (unless some kind of 'Chord plug-in' is used, which, while maybe useful as an *option*, defeats the purpose of a truly generic service if it is a *requirement*).

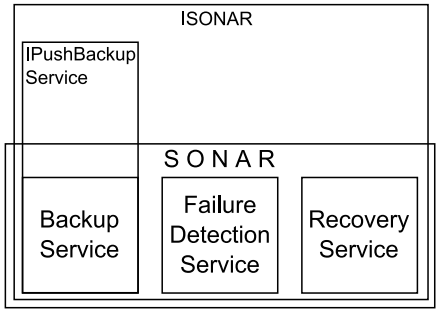


Figure 10: A single interface encapsulates the three sub-services for self-repair, and specific interfaces for specific sub-services can be optionally acquired via the unified interface if necessary

4.1 The API

The first challenge—ease of use—is approached in a number of ways, but primarily with a *simple* overarching API. SONAR exposes a single interface, *ISONAR*, through which the overlay can communicate with it.

The second challenge—expressiveness of the API—is achieved through 'progressive disclosure'; while the *ISONAR* interface can achieve many common tasks, it can also provide the overlay with access to more focused interfaces if desired. This is illustrated in figure 10, where the *ISONAR* interface is the general accessor for using the service, and the specific *IPushBackupService*[1] can be used by requesting it through the general interface.

Finally, the *genericity* of the API is achieved by specifying a *model* which overlays must conform to in order to be repairable by SONAR. This genericity is made sufficiently *performant* by identifying the *key state* of an overlay, and by using a *two-way* API. At one side is an API belonging to SONAR which permits exposition and guidance by the overlay with regard to its *key state* elements, and at the other side is an API allowing management of the overlay by the service, as illustrated in figure 11.

This enables the overlay to *inform* SONAR of important events with respect to its key state, allowing SONAR to provide a focused service with respect to this state, rather than having to meticulously 'checkpoint' an entire process, for example. Unlike some transparent checkpointing approaches, SONAR does *not* therefore attempt to be fully transparent to the overlay. Rather, overlay nodes and SONAR instances cooperate using explicit two-way interaction, whereby an

[1]The 'push' terminology is derived from the way that the overlay 'pushes' data to the service when a change occurs in an overlay node's state.

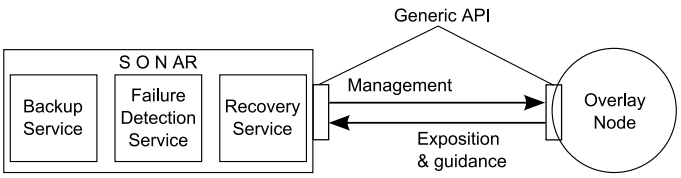


Figure 11: A two-way API assists with genericity and performance

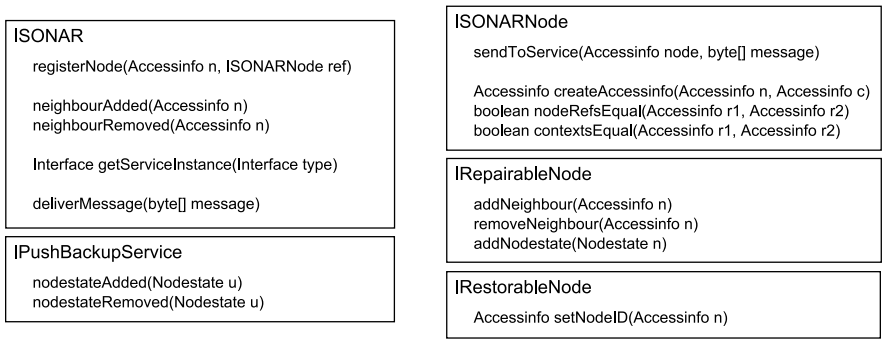


Figure 12: The interfaces used in the architecture: (a) SONAR's main interfaces, and (b) Interfaces to be implemented by overlay nodes. Methods return 'void' unless otherwise noted. All parameters are 'in' only.

overlay node calls methods on its local service instance, and a service instance calls methods on the overlay node it is supporting.

4.1.1 A simple, extensible API

Figure 12 shows the details of SONAR's simple main API—the *ISONAR* interface, with five methods. The figure also shows the overlay's simple main API, encapsulated in the *ISONARNode* interface, to be implemented by an overlay node.

These two interfaces are very lightweight, yet provide for most of SONAR's functionality. The figure also shows some of the extended interfaces of SONAR. For an overlay to be repairable by the recovery service, it must implement the *IRepairableNode* interface, and for an overlay to have its nodes be *restorable* (i.e. re-created on other hosts) by the recovery service, its nodes must additionally implement the *IRestorableNode* interface.

Similarly, if the overlay wishes to use a 'push' backup service implementation, notifying SONAR of changes in nodestate as and when they occur, it can acquire through *ISONAR* a reference to a backup service which exports the *IPushBackupService* interface. This extensible

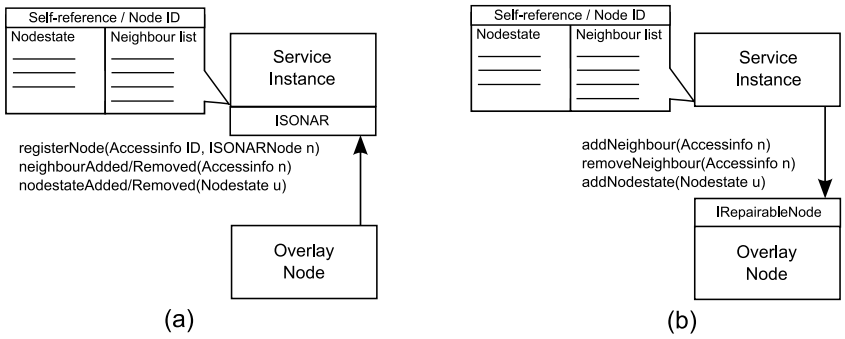


Figure 13: API interactions: (a) An overlay node exposing its accessinfos and nodestates; and (b) SONAR inspecting and adapting these (e.g. at repair time).

API architecture can potentially be taken as far as is desired in terms of the detail level which it provides access to, in both 'directions'.

4.1.2 An overlay model

As mentioned, in addition to implementing the relevant interfaces, overlays must conform to a particular 'model' which SONAR understands. This model is still highly generic, and crucially, allows overlays to express their unique properties without SONAR needing to understand them.

To conform to this model, overlay implementations are required to structure their nodes in terms of two key abstractions: *accessinfos* and *nodestates*. The former describes the topological position of a node, providing its 'neighbour' links, and the latter describes any additional elements of state that is important to the overlay. Based on these abstractions, which are described in detail shortly, SONAR can observe each node in terms of both *topology* and *state*, and can also *adapt* the topology and state as required to carry out repairs.

Figure 13 depicts the full 'model' of an overlay node and its interactions with SONAR. Note that the overlay's implementation of *IRepairableNode* implies implementation of *ISONARNode*. The below subsections define and discuss accessinfos, nodestates and the interactions between the service and the overlay.

Accessinfos Accessinfos are used to expose to SONAR the connectivity of an overlay node with its 'neighbours', and also to enable SONAR instances to communicate with each other. This communication, which is useful for example to allow the service to send a backup of a node to another node, is achieved using the overlay's own topology, and enables the peer-to-peer

operation of SONAR. In terms of its representation, an accessinfo is a record that refers to an overlay node and encapsulates sufficient information to allow a message to be sent by the overlay to that node. The internals of an accessinfo are entirely opaque to SONAR, and they are assumed to be 'serializable' so that they can be marshalled by SONAR for transport and storage purposes. Their opacity to SONAR permits overlays to construct these data items entirely as they wish, thereby promoting the genericity of SONAR to a wide range of overlays, while providing *enough* information to SONAR by clearly defining a 'role' for this data.

When an overlay node first comes into existence, it is expected to provide its local SONAR instance with an accessinfo that refers to itself, which is used to identify and associate various kinds of data with the node. This is achieved by calling *ISONAR.registerNode(Accessinfo nodeRef, ISONARNode n)*. This call also provides the SONAR instance with a local object reference n on which it can call overlay-side API methods.

Following this self-advertisement, each node is expected to keep its local SONAR instance informed about changes to its 'local' topology—i.e. its connectivity to neighbouring nodes. This is achieved using *ISONAR.neighbourAdded/Removed(Accessinfo n)*. With the accessinfos passed in these calls, the SONAR instance is able to communicate with peer instances associated with the given neighbours by using *ISONARNode.sendToService(Accessinfo n, byte[] message)*, illustrated in figure 14. This call requests that the overlay node send the given message to the SONAR instance associated with the specified neighbour (*nodeRef*). The receiving overlay node subsequently calls *ISONAR.deliverMessage(byte[] message)* to deliver the message to its local SONAR instance.

The above deals with basic topology management and the enabling of peer-to-peer communication for SONAR. However, this basic topological description is often not enough to properly capture the overlay's subtleties because it does not take into account the various topological 'roles' that might be played by certain nodes in certain overlays. For example, ring-based overlays may comprehend the roles of 'successor' and 'predecessor', while tree overlays may comprehend the roles of 'parent' and 'child'. To enable such semantic information to be expressed by the overlay, accessinfos can be 'tagged' by the overlay with arbitrary contextual information. As with accessinfos themselves, the nature of this information is opaque to SONAR. Accessinfos can be

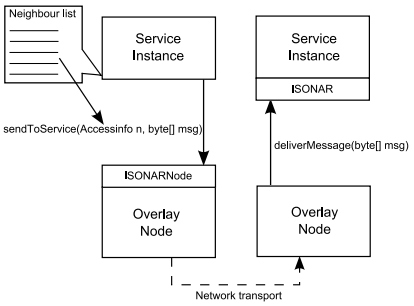


Figure 14: Neighbours exposed by the overlay are used by the service to send data to other service instances

compared by SONAR purely on the node to which they refer, or purely on the context which they represent (a 'standard' comparison of two accessinfos is context-sensitive as well as node-reference-sensitive). The overlay-side API provides this capability through the *ISONARNode* interface, and this simple added information can be useful to repair strategies created by the recovery service (discussed in section 5.3).

Nodestates Nodestates are used to encapsulate state that an overlay node is interested in having restored when the node is recovered. They are optional and do not need to be used by overlays that don't need to maintain persistent state. Like accessinfos, nodestates are assumed to be 'serializable', and the internals of nodestates are opaque to SONAR.

Overlay nodes pass nodestates to SONAR using *IPushBackupService.nodestateAdded /Removed(Nodestate u)*. If there is a failure, *IRepairableNode.addNodestate (Nodestate u)* can be called by SONAR on an appropriate target node to restore the data encapsulated in these nodestates to the overlay (the issue of which node to select for this restoration is discussed in detail in chapter 5).

Like the contents of nodestates, the implementation of *addNodestate()* is entirely up to the overlay. For example, a DHT overlay node may expose each file stored locally as an individual nodestate unit, and implement *addNodestate()* to map to the DHT *store()* operation, thereby routing the data to the correct place in the overlay. Alternatively, a super-peer in a Gnutella-like overlay may store its resource index as a nodestate, and implement *addNodestate()* as a 'merge' operation to merge any existing local resource index with the provided one.

4.1.3 Repair actions

The above demonstrates how SONAR can modify the topology and re-distribute the state of an overlay using a common 'model' with the general IRepairableNode interface. Beyond this base API, additional repair strategies and overlay input are supported through "progressive disclosure"; a set of optional interfaces available to the overlay developer wishing to have a greater understanding of, or level of control over, the service's operations. This is exemplified here through the *IRestorableNode* interface, as shown in figure 12.

This interface is designed to allow failed overlay nodes to be fully *restored* on alternate hosts, as opposed to compensatory topology modification. This strategy is useful in Grid-like deployments, where resources may be more plentiful. Restored nodes must be provided with the 'node ID' accessinfo of the failed node they are replacing (an accessinfo originally used with *registerNode()*), achieved with the *IRestorableNode.setNodeID()* method. Both of these repair types are generically applicable to a range of overlay networks; a discussion of this, and of the mechanics of a safe, decentralised repair approach, are provided in chapter 5.

Following this example, different 'repair strategies' can be developed and used when appropriate, according to the current state of the overlay's deployment environment, provided any additional interfaces relating to specific repair strategies are implemented. Implementing such interfaces allows the overlay to configure which strategies it would like to support (a runtime examination by the service of implemented interfaces permits this), and indeed further expansion of the overlay-side API can allow arbitrary overlay guidance on how a repair strategy is performed, achieved by the strategy querying the overlay before repair enactment. A discussion and evaluation of the benefits of dynamically selecting different repair strategies at the time of repair is detailed in section 7.2.

4.2 Case studies

4.2.1 Chord

A detailed view is now given of the overlay developer's task in making an overlay SONAR-compatible. Chord [76] is used as an example in this section, as it is well-known and easy to understand without being trivial—a further example based upon the TBCP overlay [58] is given

```
class ChordAccessinfo implements java.io.Serializable {
  public static final int NODE_ID = 0, SUCCESSOR = 1, PREDECESSOR = 2;
  public RemoteNode nodeReference;
  public int context;

  public ChordAccessinfo(RemoteNode nodeReference, int context) { .... }

  public boolean equals(Object o) {
    if (o instanceof ChordAccessinfo) {
      ChordAccessinfo chk = (ChordAccessinfo) o;
      if ((chk.nodeReference.equals(nodeReference))
          && (chk.context == context))
        return true;
        else
        return false;
      }
      else return false;
    }
  }
```

Figure 15: An accessinfo implementation for Chord

in the following section. The presentation here is based on an actual modification of an existing Chord prototype [1] as used in Lancaster's Gridkit middleware.

Instantiating the abstractions Chord has two main topological characteristics: (i) a ring structure, in which nodes are linked clockwise and anti-clockwise by 'successor' and 'predecessor' links respectively; and (ii) the use of per-node 'finger tables' that provide O(log N) routing for any key (as the ring alone would only provide O(N) routing).

For the purposes of this case study, the ring structure is exposed to the service in terms of accessinfos, and the finger tables of nodes are modelled in terms of nodestates. This latter choice makes sense because it is known that the finger table will be refreshed shortly after a repair anyway, and so backed-up finger table data is used only to speed up this process (providing a fairly good jump start). However, it would also have been possible to expose finger tables in terms of accessinfos by calling *neighbourAdded/Removed()* as each finger link changes, with an appropriate 'finger' context (see below).

Having made this decision, the next step is to define an accessinfo class as shown in figure 15. The Chord implementation uses Java RMI as its communication protocol, and so the *RemoteNode* reference is a wrapper around an RMI remote reference (and Chord node ID). The new accessinfo class also has a comparator method and a 'context' variable, which is used to tag the record with contextual data of 'successor', 'predecessor' or 'NodeID'. It is serializable for network transport and flat storage.

A similarly simple 'Chord nodestate' class is then created (not shown) to contain a node's

```
public void sendToService(ChordAccessinfo toNodeID, byte[] message) {
  toNodeID.nodeReference.receiveServiceMessage(message);
  }
public ChordAccessinfo createAccessinfo(ChordAccessinfo toNode, ChordAccessinfo useContext) {
  return new ChordAccessinfo(toNode.nodeReference, useContext.context);
  }
public boolean nodeRefsEqual(ChordAccessinfo r1, ChordAccessinfo r2) {
  return r1.nodeReference.equals(r2.nodeReference);
  }
public boolean contextsEqual(ChordAccessinfo r1, ChordAccessinfo r2) {
  return r1.context == r2.context;
  }
public void addNodestate(StoredFingerTable unit) {
  refreshFingerTable(unit.table);
  }
public void addNeighbour(ChordAccessinfo neighbour) {
  //check what kind of neighbour it is
  if (neighbour.context == ChordAccessinfo.SUCCESSOR)
    setSuccessor(neighbour.nodeReference);
    else if (neighbour.context == ChordAccessinfo.PREDECESSOR)
    setPredecessor(neighbour.nodeReference);
  }
public void removeNeighbour(ChordAccessinfo neighbour) {
  //check what kind of neighbour it is
  if (neighbour.context == ChordAccessinfo.SUCCESSOR)
    setSuccessor(null);
   else if (neighbour.context == ChordAccessinfo.PREDECESSOR)
    setPredecessor(null);
  }
```

Figure 16: The implementation of the *ISONARNode* and *IRepairableNode* interfaces for Chord

finger table; this is simply an array of remote references. This completes the definition of Chord under SONAR's model. This is clearly both Chord-specific (with customized contexts), and implementation-specific (with an RMI remote reference used as the accessor detail).

Using the API To be repairable by SONAR, the Chord node implementation must implement seven methods, shown in figure 16. SONAR is implemented in Java, and uses top-level Java Objects to represent accessinfos and nodestates, but to simplify the presentation here 'implicit casting' to the ChordAccessinfo class (and corresponding nodestate class 'StoredFingerTable') is assumed in the provided code extracts.

Note the *receiveServiceMessage()* call, shown near the top of figure 16, is a remote method call, from which the recipient overlay node calls *deliverMessage()* on its SONAR instance to deliver the sent message (as in figure 14 in section 4.1). Finally, note in the *addNeighbour* and *removeNeighbour* methods how Chord uses the *context* data that it attaches to accessinfo records—this data is crucial to Chord, and without it no understanding would be possible of whether a neighbour being added / removed by the service was a successor or predecessor.

In order to expose the overlay's topology and state to the service, the overlay node imple-

```
private void stabilize() {
  ...

  // Updating successor
  RemoteNode successor =  getSuccessor();
  RemoteNode rn = successor.getPredecessor();
  // Check to see if the successor's predecessor is greater than Node ID
  if ((rn != null) && (ChordNodeID.inRange(rn.nodeID, myNodeID, successor.nodeID))) {
    // If it is, set node successor to the successor's predecessor
    sonar.neighbourRemoved(new ChordAccessinfo(successor, ChordAccessinfo.SUCCESSOR));
    setSuccessor(rn);
    sonar.neighbourAdded(new ChordAccessinfo(rn, ChordAccessinfo.SUCCESSOR));
    }
  ...
  }
```

Figure 17: The modified implementation of the stabilize method in Chord

```
public ChordAccessinfo setNodeID(ChordAccessinfo nodeID) {
   setChordNodeID(nodeID.nodeReference.chordID);
   return new ChordAccessinfo(myRemoteReference, ChordAccessinfo.NODE_ID);
   }
```

Figure 18: The implementation of *setNodeID()* for Chord

mentation also needs to call six methods on SONAR at various points during its execution. *registerNode()* is called on startup, then a reference to the default backup service acquired with *getServiceInstance()*, and topology information is then provided as it becomes available (or changes) using *neighbourAdded/Removed()*. Successor changes occur within Chord's *stabilize()* method, of which an extract is shown in figure 17 (just 2 lines have been added, highlighted in *italics*).

Chord's *notify()* method was also modified in the same way to expose changes in the node's predecessor. In both cases, it was decided not to internally store neighbour links as instances of the new ChordAccessinfo class, but instead to create them only when Chord interacts with the service. This decision was made because an existing overlay was being modified, and it was therefore desirable to make as few changes to it as possible.

Finally, when a Chord node updates its finger table, a similar procedure to that shown in figure 17 is again performed, but this time using the methods *IPushBackupService.nodestateAdded/Removed()* of the selected backup service, where the old finger table is removed from persistent storage with the service, and the new one added, as nodestate.

Optional additional implementation To allow the recovery service to choose at runtime between topology-modifying and node-restoring repair strategies, the optional interface IRestor-

ableNode was implemented, with its *setNodeID()* method as in figure 18.

This completes a version of Chord which can be maintained by SONAR, so that the service provides all aspects of redundancy, failure detection, and recovery. It is believed that all overlay-side instrumentation is trivial for the overlay developer, as demonstrated by the code extracts above—in total, the modified version of Chord is 80 lines (10%) longer than the 'standard' version. Chord's functional behaviour has not been altered in any way, so it is fully compatible with existing applications.

Nevertheless, it is additionally possible for such applications, again with minor modifications, to themselves take advantage of SONAR to ensure that their data is safe across node failures. To achieve this, applications simply need to wrap their data as nodestates and call *addNodestate()* and *removeNodestate()* on the backup service as data is added to/removed from the local node.

4.2.2 TBCP

To provide a fuller picture of how the API can be used by different overlays, an example of its use by the TBCP application-level multicast overlay [58] is now given. TBCP is in many respects simpler than Chord, with no data storage responsibilities, but as mentioned in section 2.1.1, provides a richer set of repair candidates should a node fail (due to the many valid configurations of a tree).

TBCP is also interesting in contrast to Chord because it has a single 'distinguished' node—the root—which has special status in the overlay as the source of multicast data. For this section, it is assumed that the tree is being used as a general group multicast structure, such that any node in the tree can send data to the overlay via the root node (rather than the root node being truly irreplaceable as the single source of e.g. a multicast video stream).

Instantiating the abstractions Most TBCP nodes have only one kind of information; the identities of their child nodes. When data arrives at a TBCP node, it is simply forwarded to these child nodes. The TBCP implementation used also provides each tree node with the identity of the root node, should they wish to send data to the overlay. The root node, of course, also knows that it is the root of the tree (a fact established by the TBCP protocol).

Child links are clearly a good mapping for accessinfo records, exposing the topology of the

```
class TBCPAccessinfo implements java.io.Serializable {
  public static final int NODE_ID = 0, CHILD = 1;
  public ITreeNode nodeReference;
  public String groupID;
  public int context;

  public TBCPAccessinfo(IDTreeNode nodeRef, String groupID, int context) { ... }

  public boolean equals(Object o) {
    if (o instanceof TBCPAccessinfo) {
      TBCPAccessinfo chk = (TBCPAccessinfo) o;
      if ((chk.nodeReference.equals(nodeReference))
          && (chk.context == context))
        return true;
        else
        return false;
      }
      else
      return false;
    }
  }
```

Figure 19: An accessinfo implementation for TBCP

overlay to the service. A TBCPAccessinfo class is therefore created to contain these links (plus node IDs, as with Chord). This is shown in figure 19. Again, the TBCP implementation uses RMI as its communication protocol, so the *ITreeNode* field represents a remote reference. As with Chord, the new accessinfo class has a context attribute (set to either 'child' or 'node ID'), a comparator method, and is serializable.

The only other salient information a node may have is if it is the root of the tree. This is implemented as a 'nodestate' record, for which a very simple 'isRoot' flag is used. Should the root node fail, this flag will be presented to a suitable alternative node during repair, which may then assume the duties of the root node (more detail on this procedure is given below).

Using the API As with Chord, seven methods must be implemented by TBCP to be repairable by the service. These are shown in figure 20. Again, the implementation is abstracted slightly for readability, replacing 'Object' with 'TBCPAccessinfo' or 'RootStatus' as appropriate.

To expose TBCP's topology to the service, the overlay node implementation calls *registerNode()* at startup, as is required, and acquires a reference to the default backup service with *getServiceInstance()* (though technically only the root node needs this). The *neighbourAdded/Removed()* methods are again used to inform SONAR about topology changes—TBCP's *joinTree* method deals with all such topology changes, and an extract of the modified version of this is shown in figure 21. Again, just 2 lines have been added.

Finally, a node exposes its root status (if it is the root) by calling *addNodestate()* on the backup service when it is made aware by the overlay protocol that it is the root node.

```
public void sendToService(TBCPAccessinfo toNodeID, byte[] message) {
  toNodeID.nodeReference.receiveServiceMessage(message);
  }

public TBCPAccessinfo createAccessinfo(TBCPAccessinfo toNode, TBCPAccessinfo useContext) {
  return new TBCPAccessinfo(toNode.nodeReference, useContext.groupID, useContext.context);
  }

public boolean nodeRefsEqual(TBCPAccessinfo r1, TBCPAccessinfo r2) {
  return r1.nodeReference.equals(r2.nodeReference);
  }

public boolean contextsEqual(TBCPAccessinfo r1, TBCPAccessinfo r2) {
  return r1.context == r2.context;
  }

public void addNodestate(RootStatus unit) {
  //the fact that I'm being given RootStatus implies I'm the root now
  setIsRoot(true);
  //re-advertise myself as the root
  ...
  }

public void addNeighbour(ChordAccessinfo neighbour) {
  addChild(neighbour.nodeRef);
  }

public void removeNeighbour(TBCPAccessinfo neighbour) {
  removeChild(neighbour.nodeRef);
  }
```

Figure 20: The implementation of the *ISONARNode* and *IRepairableNode* interfaces for TBCP

```
public ITreeNode joinTree(String groupID, Vector distances, ITreeNode newNode) {
  ...

  //
  // Find best parent node for joining node (myself, or one of my children) ...
  //
  ...

  if (nodeParent == this) {
    addChild(groupID, newNode);
    sonar.neighbourAdded(new TBCPAccessinfo((IDTreeNode) newNode, groupID, TBCPAccessinfo.CHILD));
    ...
    //now deal with possible displaced child
    if (treeConfiguration.getDisplacedChild() != null) {
      ITreeNode removedNode=treeConfiguration.getDisplacedChild();
      removeChild(groupID, removedNode);
      sonar.neighbourRemoved(new TBCPAccessinfo((IDTreeNode) removedNode, groupID, TBCPAccessinfo.CHILD));
      ...
      }
  ...
  }
```

Figure 21: The modified implementation of the joinTree method in TBCP

Optional additional implementation To take advantage of resource-rich deployment environments, it is desirable again to implement the *IRestorableNode* interface, permitting SONAR to fully restore failed nodes on alternate hosts if appropriate. In the case of TBCP, this node simply returns its own ID, discarding the incoming node ID, as it is not critically important to maintain node IDs as in Chord.

Because root node repairs are permitted (such that root status is re-located appropriately if the root node fails), this creates a subtle issue for non-root nodes, which have a reference to the root node should they wish to send data to the overlay—if the root 'moves', they will have a reference to the failed root node instead of the new, post-repair root. The implementation used here circumvents this issue by using a middleware resource discovery infrastructure—if a root repair occurs, the new root node advertises itself with the resource discovery service, and when nodes wish to send data, they check with the discovery service that their reference to the root is valid. Outside of this implementation detail, a simple solution would be for a new root node to use the overlay to multicast its root status to all nodes as part of the repair.

There are of course some issues which SONAR does not address, in particular here the problem of lost messages due to failures in a multicast overlay (i.e. messages sent while repairs are taking place). This is generally out of scope of the work described in this thesis, but there are other efforts to address such concerns (e.g. [49]), which could be integrated into SONAR if desired.

4.3 Summary

This chapter has presented the API which allows SONAR to be used by overlay networks (originally published in [62]), demonstrating the effort needed by overlay developers to take advantage of the generic self-repair capabilities offered by this work. A detailed case study of two different overlays has been provided, showing how SONAR can be used to support different overlays—fulfilling requirement 2 (*genericity & re-usability*) from chapter 2.

5 A SONAR Repair Protocol

The previous chapters have discussed the context of this work, the overall design and architecture of SONAR, and the means by which overlay developers can use it to support node failures. This chapter presents the detail of how the recovery service in particular works—how the failure of an overlay node can be repaired in a generic and decentralised way, while allowing configurability & adaptivity in the way repairs are made.

The approach taken in this work is first motivated, and the contributions it makes are outlined in section 5.1. The protocol is then presented in section 5.2, and example repair strategies that can be used—the major adaptable feature of the protocol—are discussed in section 5.3. Following this, the protocol's correctness is discussed in greater detail in section 5.4, and the chapter is summarised in section 5.5.

5.1 Motivation and contributions

The design of the repair protocol was motivated by three major goals:

Separation of generic and specific aspects of repair. An overriding goal was to create a *foundation* of repair; a mechanism which could be used as a common building block atop which various repair strategies could be deployed and used. As such, repair is conceptually separated into two parts: a *generic* part in which the extent of a failure is detected and delineated; and a *repair specific* part in which a repair strategy is selected and enacted. This is designed to support diverse environments and high degrees of configurability: in this case supporting alternative repair strategies that make different tradeoffs depending on their deployment environment.

Localised repair. It is important that only nodes in the *locality* of a failed node or failed section are involved in coordinating its repair. This is essential to guarantee the *scalability* of the approach. In very large overlays, such as Internet-scale P2P networks, it would be infeasible to involve centralised services or nodes beyond the failure locality.

Aggregated failure handling. Rather than be restricted to treating *individual* overlay nodes as the unit of failure detection and repair, the approach was designed to generalise naturally

to deal with *failed sections* of overlay, thus being potentially capable of handling any severity of node failures. This is especially beneficial where the virtual structure of an overlay corresponds somewhat to the underlying physical topology, 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 key barrier to 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 went well with the goals of this work. In particular, global coordination does not scale, probabilistic approaches don't lend themselves to consistency, and pre-defined repair strategies clearly do not meet the need for genericity.

A fundamentally different approach is therefore proposed: The core of this approach to generic overlay repair is a localised 'agreement protocol' that enables the set of nodes bordering a failed section of an overlay (i) to discover and agree on the extent of the failed section; (ii) to agree on a repair action to be taken; and (iii) to select a coordinator from among themselves to manage the repair. This protocol can then be used as a common basis for the support of different repair strategies. In this approach, however, a pernicious inter-dependency arises between those who are *agreeing* (a group termed the 'border set') and that which they are *agreeing to* (i.e. the extent of the failed section, which implies the constituency of the 'border set' itself). This phenomenon, which is the major characteristic of the problem space that is addressed, is referred to as the 'self-defining constituency problem'[2].

The culmination of these features is a protocol supporting decentralised, *collaborative* repair of an arbitrary distributed system—nodes with a mutual interest in a failure are put in contact

[2]Formally, this adds a second parameter *voterSet* to the classical consensus primitives of *propose(value,voterSet)* and *decide(value,voterSet)* [15]. 'Self-defining constituency' refers to the fact that in addition to the traditional properties of *validity*, *agreement*, and *termination* it is required that agreement be reached only if *all* non-failed members of *voterSet* execute *propose(..,voterSet)* with the *same voterSet* value.

with one another, provided with a consistent view of the nature of the failure, and permitted to create an agreed repair strategy, potentially based on information they provide to each other using the protocol (such information is naturally and consistently propagated if desired). The decided action can then be enacted safely—it is done so in isolation such that the action or its sub-actions will not be repeated.

More generally, provided that assumptions hold, the repair protocol guarantees that every failed node in the system will be repaired (using a repair strategy of the system's choice) exactly once.

5.2 The Repair Protocol

5.2.1 Assumptions

An overlay is considered as consisting of a potentially very large number of nodes that are deployed in an underlying infrastructure network (e.g. the Internet). Nodes are identified using overlay-specific unique identifiers. Provided the recipient's underlying network address is known, it is assumed that any node can send a message to any other node, and that message communication is reliable (i.e., barring network partitions, any message sent is eventually delivered; TCP/IP semantics are sufficient). However, no particular timeliness properties are assumed for messaging.

In terms of overlay topology, no global knowledge is assumed—overlay nodes are assumed to be related to each other only through a 'neighbouring' relationship that constitutes the overlay structure to be repaired in case of failure.

Nodes may fail at any time, but when they do so they do not subsequently interact with the rest of the overlay ('fail-stop', as defined in [7]). Nodes may continue to fail as repairs are ongoing, but it is assumed that such failures do not occur at such a rate that the protocol is unable to keep up with them. It is assumed that there are no physical network partitions, though logical (overlay) network partitions are naturally supported.

In terms of infrastructure, the existence of a *distributed backup service* is assumed, as discussed previously, from which the state of a failed node can be obtained by any non-failed node. This state includes the neighbour links of nodes, other repair-specific information (i.e. 'repair logs', as

discussed later) and, optionally, application-specific data. To create redundancy, such a backup service would likely re-use the same hosts that support the overlay itself, with one possible implementation being a replication approach such as that described in [29]. It is possible that the state returned by a scalable backup service might not be completely up-to-date, and for simplicity it is assumed that overlays can cope with this situation (many current overlays are explicitly designed to cope with minor inconsistencies due to the conflicting need to scale to very large numbers of nodes). However, up-to-date repair logs must be consistently available from backups.

Finally, the availability of per-node *failure detectors* is assumed, which are used to probe the liveness/ failure status of other nodes. For theoretical correctness, a 'perfect' failure detector is required (i.e. any failed node is eventually detected, and non-failed nodes are never suspected of failure—no false-positives [15]). A weaker failure detector does not prevent the protocol from working, but reduces its deterministic guarantee of repairing every failed node exactly once to a probabilistic one, which is explored in section 7.1.2.

5.2.2 An overview of the protocol

```
1:  procedure RepairProtocol_p
2:      loop
3:          wait for some of p's neighbours to fail
4:          repeat
5:              (BNodes_p, FSection_p) ← ConstructFailedSectionView()          ▷ Phase1
6:              SelectRepairStrategy(BNodes_p, FSection_p)
7:              V_p[] ←AgreeOnView(BNodes_p, FSection_p)                       ▷ Phase2
8:          until V_p[] only contains accept          ▷ Agreement on the failed section
9:          coordinator ← SelectCoordinator(BNodes_p, FSection_p)
10:         if p = coordinator then                                  ▷ I am the coordinator
11:             PrepareRepair(BNodes_p, FSection_p)
12:             AddRepairToRepairLog()
13:             EnactRepair(BNodes_p, FSection_p)                              ▷ Phase3
14:             Send ⟨coordinator, repairDetails, repairOK⟩ to all nodes in BNodes_p
15:             RejectViewsContainingRepairedNodes()
16:         else                                                 ▷ I am not the coordinator
17:             wait for ⟨coordinator, repairDetails, repairOK⟩ or coordinator ∈ FailedNodes_p
18:         end if
19:     end loop
20: end procedure
```

Figure 22: Pseudo-code of the repair protocol when executed by node p

The overall repair protocol is presented in outline in figure 22 and illustrated in figure 23. The protocol is executed by each node p in the overlay, and operates in *three* main phases. When p's failure detector reports that one of its neighbours (i.e. nodes whose failure p has a direct interest

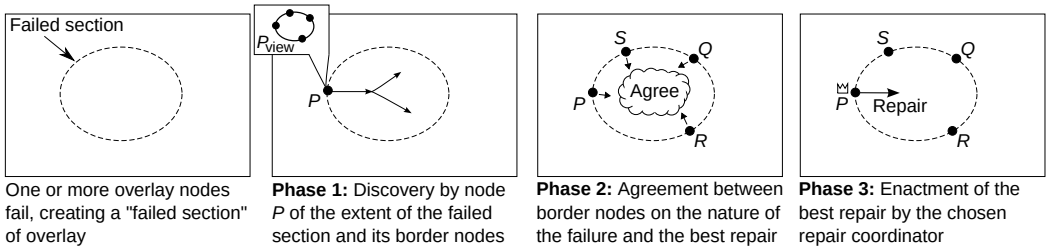


Figure 23: The three main phases of the repair protocol

in) has failed, it enters phase 1 of the protocol. p constructs a 'view' of the failure (which may be one or more nodes), and discovers the other live nodes surrounding the failure (the 'border nodes'). This is performed by the ConstructFailedSectionView procedure, which returns a failed section and border set, in $FSection_p$ and $BNodes_p$. Node p then chooses a repair strategy for this presumed failed section with SelectRepairStrategy.

Next, node p executes phase 2 of the protocol with the AgreeOnView procedure. In this procedure, p exchanges its view with the border set it has discovered, and those border nodes negotiate until they come to a single agreed view. Border nodes whose views are rejected at this stage will return to phase 1 to re-consider their view, and then re-enter phase 2—this negotiation continues until all nodes surrounding a given failure come to an agreement about that failure. Repair strategy information from SelectRepairStrategy is also disseminated during phase 2. Following agreement, one of the border nodes is selected as the repair coordinator.

In phase 3, the chosen coordinator executes the selected repair strategy. The coordinator logs the intent of its repair in a local, per-node *repair log* to help prevent nodes being repaired more than once. While the repair is being carried out, the other border nodes wait until either they receive a repairOK message from the coordinator, or they detect that the coordinator has failed. If the former happens, any local repair duties of non-coordinator nodes for the chosen repair strategy are carried out, completing the repair, and if the latter happens, the protocol loops back to the very beginning.

In the remainder of this section the above outline is expanded upon.

5.2.3 Phase 1: Discovery of the extent of the failed section

In Phase 1, a node p is told about the failure of one of its neighbours, and calls ConstructFailedSectionView (referred to above, line 5 of figure 22) to discover of the extent of the failed section, which may consist of multiple nodes.

To achieve this, ConstructFailedSectionView requests from the backup service the backed up state of a failed neighbour q, and from this extracts details of q's neighbours. It checks each of these neighbours with the failure detection service to determine their status. If any are alive, they are added to p's border set; if any are reported failed, they are added to p's failed set, and their backups are acquired from the backup service. These nodes' neighbours are then checked with the failure detection service, and this procedure continues until every connection path from p's failed neighbour terminates (transitively) in a node believed to be alive.

As this proceeds, p checks any repair logs it finds in node backups, and ensures the resulting view is consistent with past repair actions (updating backups it has found if necessary to reflect these repair actions). If p neighbours multiple failed sections, it chooses the 'highest ranked' of them. Ranking of failures is integral to the protocol, and allows arbitration between 'competing' views—i.e. failed section views proposed by different nodes that contain some, but not all, of the same nodes in their failed sections.

The ranking relationship is defined as follows: (i) identical views have the same rank; (ii) the ranking between non-identical views is assessed based on the number of nodes in their failed section (so the view with the most nodes is ranked highest); (iii) non-identical views with the same number of nodes are discriminated using node IDs.

When ConstructFailedSectionView returns, p has a border set (which always includes itself) and a failed set, and is ready to enter phase 2 of the protocol. Note that because of the progressive discovery of phase 1, the discovered failed section may contain nodes currently being repaired, and may contain border nodes that have actually failed—the protocol is designed to handle such problems.

5.2.4 Phase 2: Failed section agreement and repair coordinator selection

In phase 2, a node p attempts to obtain agreement with its fellow border nodes on the extent of the failed section to be repaired. This is a consensus problem, and its solution is inspired by the consensus algorithm for strong failure detectors presented by Chandra et al [15]. This algorithm has some key properties that make it suitable for use here, but it is significantly modified to be able to deal with the particular problems here—the 'consensus group' is formed on-demand by a progressive discovery mechanism (phase 1) and thus different initiators of the protocol may develop different, conflicting views of the membership of this consensus group.

This phase of the protocol therefore has a group of nodes trying to agree with each other on who belongs in the group (and by implication, what the failed section is). The agreement protocol is captured in the AgreeOnView procedure, shown in figure 24.

Before being passed to AgreeOnView, all received messages are first filtered so that any views containing nodes that have already been repaired according to the local repair log are rejected (enclosing the offending repair log to assist the sender in its following phase 1). All other messages are buffered in a message queue ready to be extracted by AgreeOnView.

```
 1: procedure AgreeOnView_p(BNodes_p, FSection_p)
 2:     V_p[k] ← ⊥ for all k ≠ p
 3:     V_p[p] ← accept
 4:     OpposingNodes_p ← ∅
 5:     for r ← 1 to the size of BNodes_p − 1 do
 6:         Send ⟨r, BNodes_p, FSection_p, V_p⟩ To all non-failed nodes in BNodes_p \ OpposingNodes_p
 7:         PendingNodes_p ← BNodes_p \ OpposingNodes_p
 8:         msgReceived_p ← ∅
 9:         repeat
                wait | for next msg in buffer msg_k = ⟨r_k, BNodes_k, FSection_k, V_k⟩
                     | such that ((BNodes_k, FSection_k) = (BNodes_p, FSection_p) ∧ r = r_k) or
                     |                 (BNodes_k, FSection_k) is lower ranked than (BNodes_p, FSection_p)
                     | (stop waiting and goto 19 if all PendingNodes_p are reported failed)
11:             removeMessageFromBuffer(msg_k)
12:             if (BNodes_k, FSection_k) is lower ranked than (BNodes_p, FSection_p) then
13:                 Send ⟨r_k, BNodes_k, FSection_k, V_k[p] = reject⟩ To k      ▷ Reject lower-ranked view
14:             else
15:                 msgReceived_p ← msgReceived_p ∪ {msg_k}                    ▷ Take opinion on my view
16:                 PendingNodes_p ← PendingNodes_p \ {k}
17:             end if
18:         until PendingNodes_p = ∅ or all PendingNodes_p are reported failed
19:         Update V_p with received opinions in msgReceived_p ≠ ⊥
20:         Add to OpposingNodes_p any nodes rejecting my view
21:     end for
22: end procedure
```

Figure 24: The pseudo-code of AgreeOnView when executed by node p

As in [15], AgreeOnView is structured as a series of asynchronous 'rounds' in which each consensus participant (border node) sends its view to, then waits to receive a view message from, every other node in its border set before proceeding to the next round. This ensures that all nodes in a border set acquire uniform knowledge about the opinions (or failure status) of all other nodes in that border set.

For this reason, the consensus algorithm which inspired this protocol was uniquely suitable—it is actively inclusive and fair, such that every border node is *asked* to provide its opinion on a proposed view, which allows the rejection mechanic and thus view arbitration when different border nodes have different views. This also ensures that no information is 'missed' about other views being considered concurrently or about other repairs in progress, as might happen if border nodes only waited for opinions from a certain percentage of other nodes, for example.

When receiving the views of other border nodes, p will handle views which are either equal to its own and in the same round, or are lower ranked (as in the 'wait' condition on line 10). Those which are lower-ranked are rejected by p (line 13), forcing their proposers to re-consider their views, and enabling arbitration between conflicting views. This mechanic is based upon the assumption that the larger view is more recent (after more failures have occurred), and thus more relevant. Since only one view with which to enact a repair may be agreed upon, one of the views must be retracted.

Such cases can occur as in figure 25, where a particular node's failure detector shows a higher 'latency' than others, or when border nodes fail soon after phase 1 without providing an opinion in phase 2—in this case the rejection mechanism avoids potential deadlock whereby two nodes await acceptance of their respective, conflicting views from each other.

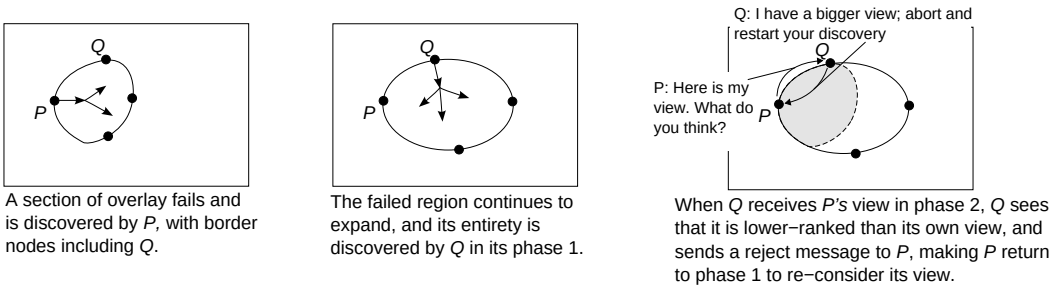


Figure 25: An example scenario using view rejection—in this case avoiding deadlock

A round completes when messages related to p's view have been received from all fellow border nodes that have not yet been detected as having failed, and have not rejected the proposed view. After each round, p updates its local knowledge vector V_p using the messages collected in the last round. If a message says that a node m is rejecting p's local view, p adds m to $OpposingNodes_p$, as it is not interested in this view. Rounds continue at p, however, to ensure that opinions are propagated to all nodes that *are* interested in this view[3].

When AgreeOnView finishes, if a node p obtains an opinion vector that only contains accept tags, this means that all nodes in $BNodes_p$ have invoked AgreeOnView with the same view and thus agree with p. In this case, p uses SelectCoordinator (line 9 in figure 22) to select a repair coordinator, and proceeds to phase 3. If the opinion vector contains values other than accept, the node returns to the start of the repeat-until loop on line 4 of RepairProtocol in figure 22, returning to phase 1 to re-consider its view.

SelectCoordinator deterministically returns a repair coordinator for a view passed as a parameter. Since all border nodes involved in the same instantiation of the protocol get the same opinion vector from the agreement protocol, all nodes agreeing on a common view are guaranteed to select the same coordinator. If, for example, border nodes score their repair designs according to a common policy, this can be piggy-backed on accept tags in the view-agreement protocol, along with an indication of each border nodes' own capabilities, so that SelectCoordinator can make an optimal choice according to various dynamic criteria (communication latencies, energy reserves, available resources, etc.).

5.2.5 Phase 3: Enacting the repair

The node that was chosen as repair coordinator first performs any repair-strategy-specific preparatory actions, then logs its repair intent, and finally performs the repair at line 13 of figure 22 by calling the procedure EnactRepair. If some of the border nodes failed after agreeing on the view, their backup data will be updated if necessary in future when nodes find this repair log and conflicting ex-border node backups in phase 1, helping to ensure conflicting view convergence.

[3]Note that a practical optimisation between rounds is to ensure that opinions that nodes are known to already possess are not sent to them again. If a node sees that all nodes in its border set know everything (after two rounds, in the best case), and there is no missing information (i.e. $\perp$), it can finish AgreeOnView. A complete version of the protocol's pseudo-code, including this optimisation, is given in appendix-A.

This also permits a repair coordinator to fail while repair is in progress, and still have the intent of that repair carried out from its repair log. This does require some care in the design of repair strategies, and also relies upon the assumption that up to date repair logs *must* be found in backups of nodes.

Once the repair has been completed, a repairOK message is sent to the border set, containing information about local actions those nodes should take to complete the repair. RejectViews-ContainingRepairedNodes is then called (on line 15) which checks the queue of buffered messages, and as in the pre-AgreeOnView filtering mechanism, a reject message is sent to the sender of any view message that contains nodes that have just been repaired (removing those messages from p's buffer).

The following section discusses several repair strategies that conform to this required behaviour—that a log of their intent implies they will definitely be carried out.

5.3 Repair strategies

5.3.1 Additive repairs

This kind of repair encompasses any strategy that *creates* or *restores* nodes as part of the repair, thereby adding to the overlay (even if only re-adding elements that had previously existed).

This is the most challenging form of repair, as it requires that a failed node not be restored / replaced and integrated into the overlay more than once, else the overlay's structure and functionality could easily become corrupted.

Example approaches The kinds of additive repair considered in this work are those in which selected nodes from a failed section are restored on alternative hosts—a technique useful in Grid-like environments, where additional resources are often available beyond those currently used by the overlay.

In these instances, restoring selected nodes can fully maintain the capacity and availability that the overlay had before the failure, placing no extra burden on surviving overlay nodes to take on the responsibilities of failed nodes. The simplest form of such a repair is to restore every failed node on alternative hosts. This is shown for TBCP in figure 26—the same repair style is clearly also applicable to other overlays.

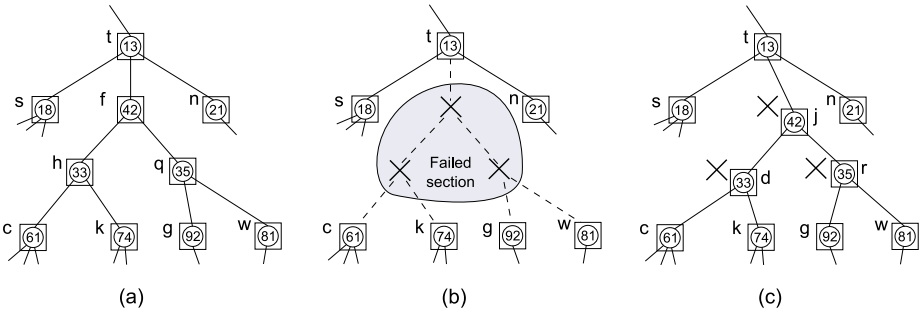


Figure 26: (a) Part of a tree overlay (b) A failure develops (c) A repair suggests restoring the failed nodes on alternative hosts. Overlay nodes are identified with numbers, and physical hosts with letters.

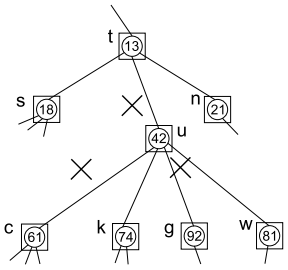


Figure 27: A partial restoration of a failed tree section

However, it may be just as effective in some cases to restore only *some* of the failed nodes, if resource availability dictates that this is the best course of action. Consider again the failures shown in figure 26. If the repair design for the failure shown in the tree-overlay observes that the deployment environment has a resource discovery system, and this discovery system reports a single available host of suitable capability, then a single overlay node can be restored, with topology adaptation as shown in figure 27. The same tactic is again easily possible in other kinds of overlays.

Overlay infrastructure that has been restored in this way can later be automatically removed or consolidated according to a desired policy—for example, when all of the surrounding overlay nodes have departed the system.

Scoring and strategy selection The ability for a node to consider many repair strategies is useful in selecting the best possible strategy for the current dynamics of the deployment environment, but the amount of time that is spent considering different strategies (i.e. in SELECTREPAIRSTRATEGY) is clearly traded against the speed with which repair is made. The fact

that *multiple* nodes (i.e. all those in a border set) are often involved in considering repair strategies can be leveraged as a way to more quickly develop a rich set of repair possibilities—in this case, use of resource-discovery can bias results by preference of their proximity to the requesting border node.

Scoring of additive repairs can be based on many factors—generic ones being measures of the combined physical capacities of the hosts proposed for restored nodes, and the speed of their proposers for actual repair enactment. However, with additional overlay input through extended APIs, scores can be weighted on factors more relevant to the overlay's needs—in a multicast tree, for example, bandwidth and latency of a candidate host's network connections may be more important that its memory and processing power.

Mechanics of repair In the SELECTREPAIRSTRATEGY method, a border node considering making an additive repair will use a resource discovery service to locate suitable hosts on which to restore selected overlay nodes. It will generate the accessinfo node IDs of the to-be-restored overlay nodes, and save them for later use if chosen as the repair coordinator.

If chosen as the coordinator, the border node restores each selected node on the host determined by SELECTREPAIRSTRATEGY, and then sends the repairOK message to the rest of the border set. On receiving this message, the border nodes examine the repair design sent (including the pre-created accessinfos of new nodes), and update their failed neighbour links with replacements to the designed alternative nodes of the repair.

The ability for SONAR to *create* accessinfo records, specifying attached context, is essential here to ensure proper modifications can be made of overlay topology to include new nodes. As an example, consider the SONAR instance at node 74 in figure 27. It must update the node's 'parent' link to point to the restored node 42, instead of the failed node 33. To do this, SONAR uses the API to request its overlay node to create an accessinfo that *points to* node 42 (using e.g. node 42's 'ID' accessinfo), and that has *the context of* its accessinfo that pointed at node 33. In this way, the overlay will be informed with the *addNeighbour* API method that it should add a new link to node 42 as a parent—without SONAR needing to know the meaning of 'parent'.

Discussion The major difficulty in this kind of repair is that the repair action is neither atomic or repeatable. A further difficulty is that these kinds of repair are reliant upon their chosen support infrastructure (from SELECTREPAIRSTRATEGY) being accessible (i.e. not failed) until the repair is complete.

If, for example, 3 nodes are to be restored to repair a failed section, and so 3 hosts were selected on which to restore them, if the 3rd host z is failed by the time the repair is in progress, the coordinator *cannot* diverge from its planned repair action—i.e. it must perform the rest of the planned repair action *around* z, and the surrounding overlay of the node intended to be restored on host z must be connected to the (non-existent) node. They will then enter phase 1 for this node, and repair it separately.

If a coordinator *did* diverge from its logged repair strategy, its failure would mean other border nodes may not be able to locate all restored nodes, if their locations differ from those that are logged, therefore inviting corruption of overlay state.

This further mandates that, even though the node destined for z never actually existed, its *backup* must exist, else this additional phase 1 cannot be successful. Backups of nodes to be restored must therefore in fact be created and stored before the nodes themselves are created—i.e. in the PREPAREREPAIR procedure, just before the repair is logged. This fulfils the requirement of repair preparation, in that it has no effect on the running overlay network (such steps can only be taken after repair logging has been done).

5.3.2 Subtractive repairs

This kind of repair involves the removal of failed overlay elements and re-distribution of responsibility among existing overlay elements to maintain functionality after failures.

Example approach These kinds of repair are suitable when resources beyond the scope of the overlay are limited—particularly in end-user-host deployments. In such scenarios, the surviving nodes in the overlay must take on the responsibilities of the failed nodes, and the overlay's topology must be re-factored to remove those failed nodes. In the repair protocol, the obvious nodes to be subject to this re-factoring and re-distribution of responsibility are the border nodes—the only questions being which border node(s) should take on the added responsibility, and how

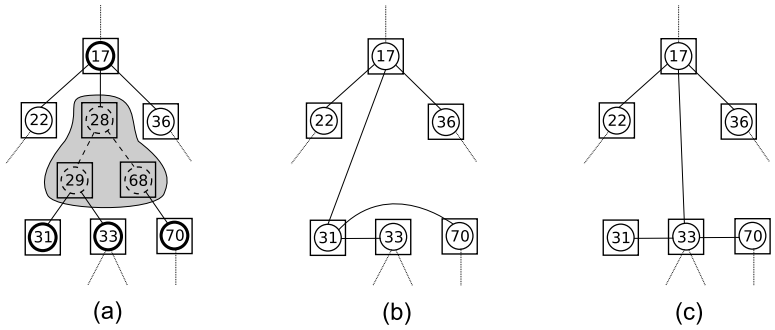


Figure 28: (a) Part of a tree overlay, with a failed region; (b) A single-hub subtractive repair with node 31, and (c) node 33 as the repair coordinator

to re-factor the overlay structure between the border nodes.

As with additive repairs, there are many choices available in this kind of repair, for consideration by SelectRepairStrategy. A simple 'single hub' approach is described here, in which one of the border nodes in an agreed repair is selected to take on the responsibilities of all the nodes in the failed section. This kind of repair is generic to many overlays, and is illustrated for a tree-based overlay in figure 28, which shows the effects of two different border nodes being selected as the 'hub' (and therefore the coordinator) of the repair.

Scoring and strategy selection Scoring of this kind of repair can again be based on many factors; either generic, or specific to a certain overlay. For example, the coordinator may score its ability to serve as a single hub based on how many neighbours its overlay node is currently supporting, so the border node with the least neighbours (and often therefore workload) gets the highest score. Specific properties such as bandwidth and link latency can be taken into account simply with an overlay-specific strategy module.

Mechanics of repair An overlay node considering this kind of repair will evaluate its own suitability as a single hub during SelectRepairStrategy, and enclose this evaluation in messages exchanged with other border nodes during phase 2. If chosen as repair coordinator, a node simply assigns itself the workload from the failed section by adding each of the other border nodes as neighbours (with appropriate context), and adding all retrieved nodestate to its overlay node.

The coordinator then sends a repairOK message to the rest of the border set, as normal. On

receiving this message, the border nodes examine the repair design sent, and replace their failed neighbour links with links to the repair coordinator (again with appropriate attached context).

The ability of SONAR to create accessinfos based on the nodes they refer to and the context embedded in them is again critical here, for a similar reason to additive repairs—when replacing a node's failed neighbour, that replacement link must be of the correct context so that the overlay knows how to use the provided accessinfo. Again, using the API, this is a simple procedure.

Finally, because this kind of repair creates no new entities, the PrepareRepair procedure has nothing to do, and so the intended repair is immediately logged, and then carried out.

5.4 Correctness, determinism and probability

Theoretical correctness of the described protocol means that 'every failed overlay node is involved in exactly one repair action'. Put another way, there is a time eventually when no failed node is left un-repaired (i.e. *progress*), and at no point is any failed node repaired more than once (*safety*). One approach to proving that these conditions are never broken is to examine all of the scenarios that the protocol can be presented with, and demonstrate that none of them can prevent positive progress or violate the protocol's safety guarantees.

As this also serves to better explain the protocol's design, this approach will be taken here. Of course, this also requires proving that all possible scenarios have been accounted for.

The presentation here will therefore proceed with the phases of the protocol, beginning with 'phase 0', in which the protocol is waiting for a failure to occur. Using this approach, it is hoped that the coverage of possible scenarios can be easily verified by the reader.

At all times, the repair protocol's view of the 'world' is provided and controlled exclusively by the failure detection and backup services, and so this discussion is guided primarily by the view these two services provide, and of course the possibility of nodes failing at any time.

Finally, throughout the following discussion, both the 'allowed' scenarios are explored, as permitted by the assumptions of the protocol, and 'hypothetical' scenarios—those which the assumptions disallow, but are interesting to explore to discover why the assumptions are as presented in section 5.2.1. Hypothetical scenarios are indicated with *italicised* paragraph headings.

5.4.1 Phase 0

The scenarios this 'phase' can encounter are relatively trivial, but are included for completeness. The protocol is at this point waiting for one of its neighbour nodes to be reported as failed by the failure detection service. It can still receive repair protocol messages from other nodes, and this section aims to prove that progress cannot be impaired, or correctness compromised, at this point.

Accurate failure report When the local failure detector reports the failure of a neighbour node, phase 1 is entered. The proof for this is self-evident from the algorithm in figure 22; after leaving line 3, there is simply no other path but directly to line 5, which initiates phase 1.

It is also clear that the protocol will not advance beyond line 3 unless a neighbour of p is reported failed. The failure detector is at this stage interested only in p's neighbours, and thus will not provide notifications about the failure of any other nodes in the system.

Failure detection latency The failure detector class assumed for the protocol states that *any failed node is eventually detected*, meaning that the failure detector can show arbitrary (but finite) latency in detecting the fact that a node is failed. When a section of overlay FS_i fails, a border node p in phase 0 that is slower than other border nodes of FS_i may receive phase 2 messages from nodes with 'faster' failure detectors.

When such messages arrive, they will either be simply buffered by p, or *rejected* if they refer to nodes that p has been involved in the repair of. Buffered messages can only be removed during phase 2 (line 11 of figure 24) or phase 3 (line 15 of figure 22), which can only be reached after phase 1, thus supporting failure detection latency at this stage.

False positive (hypothetical) If the local failure detector of p falsely declares a neighbour as failed, p will enter phase 1, as above. This is unavoidable.

5.4.2 Phase 1

While an algorithm has not been provided in this chapter for phase 1, it is sufficiently simple that the description in section 5.2.3 is unambiguous[4]. Nevertheless, the progressive discovery

[4]Those interested in seeing pseudo-code for this phase are referred to appendix-A.

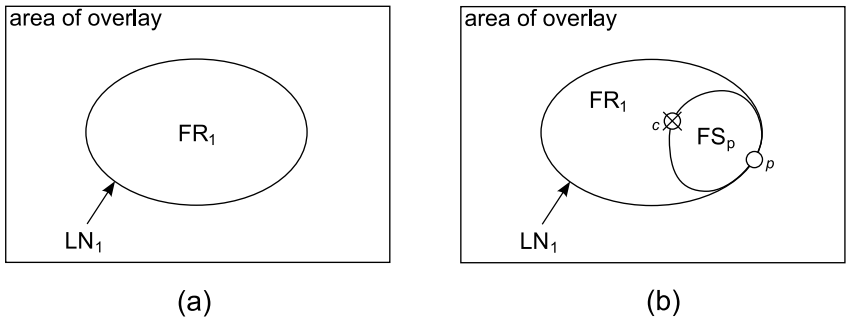


Figure 29: (a) A failed region of overlay FR_1, and the live nodes surrounding it LN_1, (b) A node p with a view containing some of FR_1, due to the belief that node c is alive (and thus a border node) due to failure detector latency

involved in this phase can cause a number of complex scenarios to arise, and again this section aims to prove that all such scenarios are dealt with such that positive progress is always made, and correctness is not violated as a result.

Failure detection latency During phase 1, p can be informed that a node c it inquired about is alive, when in fact c is failed. p will then enter phase 2, and attempt to get agreement from its assumed border nodes, including c. This scenario can be generalised as follows: There exists a failed region of overlay network FR_1, in which all nodes were transitively connected to one another through the neighbouring relationship, and all nodes are actually failed. This is shown in figure 29.

The live nodes surrounding FR_1, shown as the set LN_1, can in phase 1 either create views that are smaller than or equal to FR_1—smaller views are due to failure detection latency, as in figure 29 (b). Whichever view size was established by phase 1, this view is carried into phase 2, in which the node proposing it attempts to get agreement from its assumed border set.

Again, this is self-evident from the mechanism of phase 1, and the fact that phase 2 necessarily follows phase 1.

From this position, three basic scenarios are possible, two of which are illustrated in figure 30. Multiple instances of such scenarios are possible, with many different view sizes, but ultimately equate to those shown in figure 30.

Condition (a) is a threat to the progress of the protocol, because node p proposes its view FS_p, expecting a response from node g in its border set, while node g simultaneously proposes

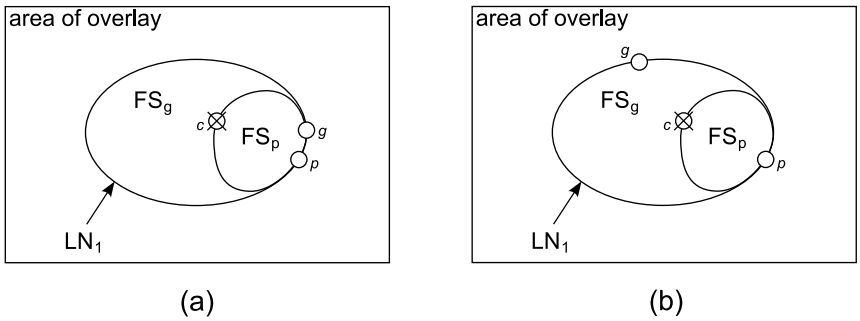


Figure 30: (a) Node p has a view FS_p which is smaller than the full failed section, but includes node g as a border node; g itself discovered the entire failed section FS_g during phase 1, so their views conflict in phase 2, (b) A node p holds the view FS_p, which contains only itself as a border node, and node g has the larger view FS_g, and awaits node p's opinion on that view

its larger view FS_g, expecting a response from node p. If neither node cared about each others' views, they would wait for each other forever; this is avoided by the ranking relationship discussed in section 5.2.1. During phase 2, a node is allowed to process messages in its buffer pertaining to *lower-ranked views*, as well as messages pertaining to agreeing views. Because of this, g will take p's view message from its buffer, satisfying the 'wait' conditions of line 10 of figure 24, and will *reject* the view because it is lower ranked, on line 13 of the same procedure.

Node p will receive this reject message, thus having a response from node g, and will be able to proceed. It must eventually be notified that node c is failed, and will therefore complete all of its rounds. When it does so, p's opinion vector V_p will contain a reject tag from node g, and no data at all from node c, so the condition 'until $V_p[]$ contains only accept' on line 8 of figure 22 will be false, requiring p to loop to line 5, re-starting phase 1.

During this process, node g will have been waiting for p to either accept or reject its view, and p must *eventually* produce the same view as g—when it does, g will be able to proceed to the next 'round' of the AgreeOnView procedure, and complete phase 2.

Condition (b) presents a similar issue, but this time node p is the only genuinely live border node in its proposed border set. Node g again awaits p's opinion on its view, but this time p is not circularly waiting for a response from g. In this case, p must again *eventually* detect that all nodes in its proposed border set have failed, and will thus stop waiting for opinions from them (again as in the wait condition on line 10 of figure 24). When this occurs, p will complete all of its rounds, and again will discover that its opinion vector V_p does *not* contain only accept tags,

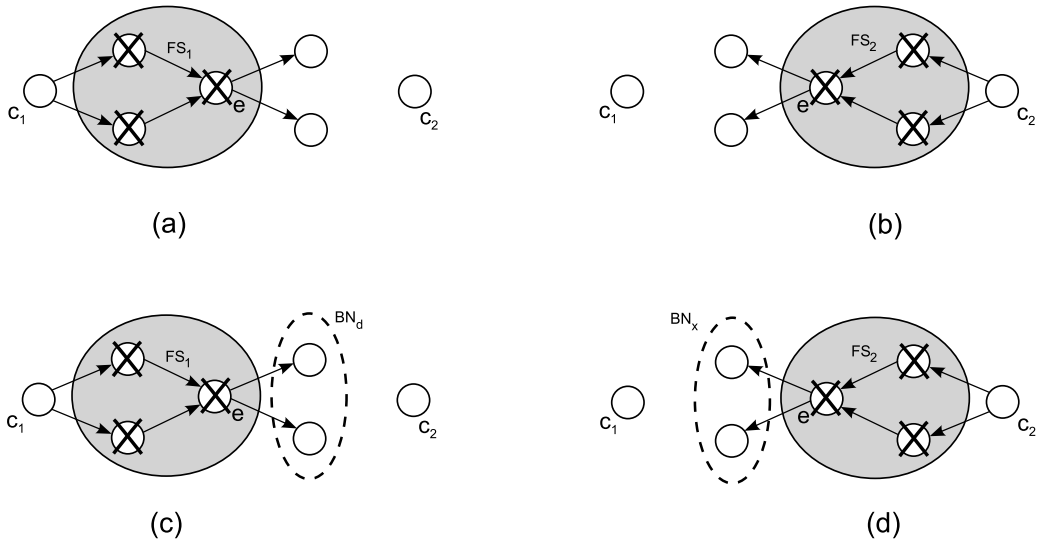


Figure 31: Views constructed by nodes c_1 (a) and c_2 (b); either the nodes in BN_d (c) or BN_x (d) must have failed before being able to provide opinions to c_1 or c_2, respectively

requiring that it return to phase 1 to re-build its view. Again, it must eventually converge with node g's view, at which point the protocol will proceed as normal and repair the failed nodes.

One other scenario can be created by phase 1, which is *overlapping* views—two views which share some, but not all, failed nodes. This scenario is shown in figure 31. This is potentially dangerous, because it appears as if nodes c_1 and c_2 in figure 31 could both repair node e. However, even though the invocation of phase 2 with such views at two different nodes is possible, it is *not* possible for both of c_1 and c_2 to enter phase 3 from this position.

Lemma A: More formally, it is impossible for an invocation of phase 2 by any two pairs of nodes with two views (BS_1, FS_1) and (BS_2, FS_2) where $FS_2 \cap BS_1 \neq \varnothing$ *and* $FS_1 \cap BS_2 \neq \varnothing$ to both move to phase 3; one or both must return to phase 1.

This guarantee is created in part by the assumptions on the failure detector of *no false positives*, and additionally by the fact that phase 3 cannot be entered without an opinion from *every* node in the border set—failed nodes cannot give opinions, and either the border nodes circled in BN_d or BN_x (or both) *must* have failed before having an opportunity to send an opinion relating to the proposed views of c_1 or c_2, such that either c_1's completion of phase 2 will result in a return to phase 1, c_2's will do so, or both will return to phase 1, depending on the order of events. The presence of the nodes in BN_d and / or BN_x in a border set must therefore be due to failure detection latency.

The reason for this is as follows; consider figure 31, and imagine that c_1 creates its view as in part (a), at which point nodes in BN_d are alive. The nodes in BN_d transmit phase 2 agreement to c_1, permitting it to transition to phase 3 and make repairs. If this happened, node c_2 *could not* have completed its phase 1 view construction before the nodes in BN_x had actually failed, since c_1 has already been told that they have failed (necessarily to get its agreement from BN_d). Because it will receive no relevant opinions from the failed nodes in BN_x, node c_2 cannot therefore proceed to phase 3, and must return to phase 1. Other event orderings create similar impossibilities, guaranteeing lemma **A**.

The protocol therefore handles all possible conditions created by failure detection latency during phase 1, in terms of safety, so that a failed node cannot be involved in multiple repairs (i.e. instances of phase 3). Of course, failure detection latency can impede the *progress* of the protocol, potentially indefinitely. This is because, if a node in phase 1 is continuously informed that a collection of nodes (e.g. those in BN_x in figure 31) is alive, it will never get agreement from them (simply waiting in phase 2). A perfect failure detector is required to *eventually* suspect every failed process, but the duration of *eventually* is clearly an important factor in the capability to make repairs—but is generally beyond the control of the repair protocol.

False positives (hypothetical) Continuing the discussion of the previous sub-section, false-positives *allow* a scenario in which two different nodes can transition from phase 2 to phase 3 with two views (BS_1, FS_1) and (BS_2, FS_2), such that $FS_2 \cap BS_1 \neq \varnothing$ *and* $FS_1 \cap BS_2 \neq \varnothing$—and a node e appears in both FS_1 and FS_2.

For this reason, the assumption of no false positives is used. However, it is useful to investigate the boundary between what is, and is not, possible—i.e., what are the *weakest* possible assumptions in which solving a problem is possible.

Failure detectors have long been classified into well-known types, originally by [15], where 'perfect', or $\mathcal{P}$, is the strongest possible failure detector (impossible to implement in asynchronous systems). Example other classes are:

- Strong ($\mathcal{S}$); The failure detector eventually permanently suspects all crashed processes, and *a* correct (non-crashed) process is never suspected.

- Eventually perfect ($\Diamond\mathcal{P}$); The failure detector eventually permanently suspects all crashed processes, and eventually no correct (non-crashed) process is suspected to be failed.

- Eventually strong ($\Diamond\mathcal{S}$); The failure detector eventually permanently suspects all crashed processes, and *a* correct process is eventually not suspected to be failed.

This classification system has proven highly useful to the development and examination of consensus algorithms for over a decade. The system was originally created, however, with a fixed set of processes in mind, that were to agree on a value. In a scenario like phase 2, the classification makes sense because it can be said, for example, that with a failure detector of class $\mathcal{S}$, each live border node does not wrongly suspect one other non-failed border node (i.e. the same one continuously) for the duration of phase 2.

Unfortunately, applying this to the progressive discovery mechanism of phase 1 is difficult; perhaps the most accurate interpretation is to say that, from a *complete* view discovered in phase 1, an $\mathcal{S}$-class failure detector will not have suspected one non-failed node of having failed—i.e. one single border node is known to be accurately reported as alive. The problem is that this is actually an extremely *weak* assumption, even though the failure detector used is relatively strong—essentially allowing only one node *from the entire overlay* to be known as not failed, since the entire overlay is potentially within the purview of phase 1's construction and discovery.

In the context of phase 1, a more useful way to classify a failure detector is by the distributed *correlation* of false positives among border nodes entering phase 2 (i.e. do they have common false positives in order to agree). This kind of correlation model is examined in section 7.1.2, when the protocol's resilience to false positives is evaluated.

One further comment on false positives in phase 1 is warranted, in that introducing the possibility of false positives during view construction indirectly creates another possible deadlock (i.e. prevention of progress) scenario. It was mentioned in section 5.2.1 that the existence of false positives do not prevent the protocol from functioning, so it is important to discuss this here. The problem arises when a node h proposes a view (BS_t, FS_t) in phase 2 in which it has wrongly been informed that some of the nodes in FS_t have failed. If some of the nodes in BS_t do not agree that their direct neighbours have failed, they will have no interest in the proposed view, and so h will wait forever for their opinions.

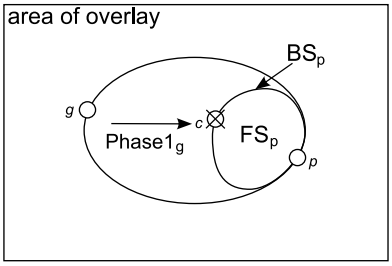


Figure 32: Phase 1 is executed by node g, and walks 'through' a failed border node of a repair in progress by node p

The situation can rapidly worsen if another node has proposed a view in which all the nodes it suspects have failed are accurately reported as such, and includes h in its border set. h will never respond to this message if the proposed view is not lower ranked than the view h itself is trying to propose, and so an entire system can rapidly become deadlocked, unable to make any repairs due to awkwardly placed false positives.

The presented protocol can be augmented to counter this possibility, which is discussed in appendix-A.

Repair in progress A node in phase 1 can walk failed nodes that are currently being repaired. Such a scenario can occur as in figure 32. The failure encompassed by the border set BS_p first occurs, and those border nodes agree upon the nature of the failure, and select node p to enact the repair. Before repair occurs, another failure develops 'outside' the border BS_p, for which node g enters phase 1. The border node c fails during this process, and node g adds both c and all of FS_p to its failed section view.

g must, however, reach the coordinator that is handling the repair of FS_p, and add it to its view as a border node. When the coordinator (p, in the above figure) completes its repair, it will observe a phase 2 message from g, and will see that it contains nodes that p has just repaired. It then uses the procedure RejectViewsContainingRepairedNodes, executed from line 15 of the RepairProtocol in figure 22 to reject this view. g will return to phase 1, and will use the reject information from node p to update the backup of failed node c, taking account of whatever repair action occurred. Should node p fail at some point during or after repair, before being able to reject g's view, g will eventually re-start phase 1 (because it will detect p as failed and stop waiting for its opinion in phase 2), and either find node backups that reflect the recent repair,

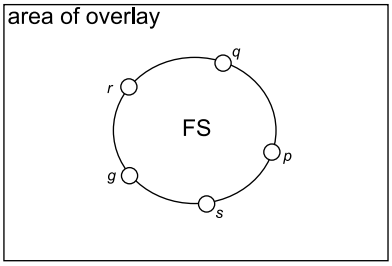


Figure 33: Nodes about to enter phase 1 in attempt to discover a failed section

or locate p's repair log and again perform a latent repair.

Agreement in progress Similarly to the case of executing phase 1 during a repair in progress, a failed border node during phase 2 can cause a node outside that border to walk the failed section *inside* the border as part of its phase 1, creating a larger failed section view FSV_u. If the in-progress phase 2 results in agreement and a transition to phase 3, the phase 3 coordinator will enact the repair and again reject the received view FSV_u as it contains already repaired nodes. If the in-progress phase 2 fails to agree, it will revert to phase 1, and will *not* discard the received view FSV_u, the recipients instead handling it in their next phase 2 as normal.

Backup location failures (hypothetical) Phase 1 uses the simplifying assumption that *the state of a failed node can be obtained by any non-failed node.* This section examines what happens if the backup service fails to perform to this assumption, and the minimum performance required by the backup service for the protocol to be able to make repairs (i.e. *make progress*).

Consider the border set shown in figure 33.

If the above assumption is not used, clearly any one border node can simply fail to locate a needed backup while discovering the failed section in phase 1—this failure can prevent it from discovering the full extent of the failed section, and the full border set (in the worst case, a node could fail to find the backup of a failed neighbour, thus being unable to make *any* phase 1 progress).

A border node that fails to complete phase 1 successfully will not normally provide any meaningful phase 2 input, which prevents any other border nodes of the same failed section from making any progress, as they will await the opinion of a node which will never provide one.

This is a relatively trivial problem to solve by having border nodes *share* backup data that

they have acquired, so that all border nodes are able to participate in phase 2 as long as *one* border node was able to locate all the necessary backups[5].

A more complex 'backup failure' scenario is when each border node could only locate *some* of the backups needed to navigate through a failed section in phase 1. If two such border nodes have enough backups to find one another, but perhaps not enough to find *other* border nodes, they can share the backups they were able to find, and hopefully both be able to then make further progress to perhaps discover a third border node. Further progress may be possible following communication with this additional node, and so on—while this is an interesting theoretical graph problem, it is beyond the scope of this work.

Backup drift (hypothetical) Finally for phase 1, the out-of-date-ness (or 'drift') of acquired backups is examined—what happens if a border node is able to acquire all necessary backups when constructing a failed section view, but some of them are out of date?

This issue has two sides; drift which loses *overlay* updates, and drift which loses *repair* updates (either border node updates for a repair action, or repair log data). These are now examined in turn.

Loss of overlay updates

This is a problem which can, under different conditions, both impede the progress and compromise the safety of the protocol. Both conditions pertain to the *structural* data contained in backups, which defines the failed section views constructed in phase 1. Progress can be impeded if the backup of a node f contains links to 'border nodes' that do not consider f to be a neighbour, and thus have no interest in the failure of f (such that they will provide no phase 2 input, preventing the proposer of the view from leaving phase 2). This can happen if, for example, f altered its position in the overlay's topology, failing at some point afterwards, and this positional change was not retrieved in its backup.

This is a complex problem, due to the distributed nature of the backup service and repair protocol—different border nodes might have acquired backups of failed nodes that vary in their recency, assuming multiple copies of a node's backup are stored at different locations (as is usually

[5]This is actually one of two possible solutions; the other is for border nodes that were unable to locate enough backups to generate a complete failed section view to 'concede' in phase 2—so, a node receiving a view message in which the view contains nodes it locally believes have failed, but for which it could not itself develop a complete view, can send an 'ACCEPT' message in response (along with a note of 'no repair strategy')

the case for high resilience). If border nodes interested in the same failed section of overlay are able to find paths to one another in phase 2, even if they find backups of nodes in their failed sections with different timestamps, then these border nodes may be able to assist each other in combining and sharing their most recent backups to form a more up-to-date view.

If *no* nodes have backups which point only to border nodes that are interested in this failure (so all proposed border sets contain at least one 'uninterested' node due to backup drift), then clearly this will impede the progress of the protocol indefinitely, or at least until more up to date backups are located / provided.

In terms of *safety*, backup drift can conceptually permit a failed node to be involved in multiple repairs, again due to topological data in backups being out of date. This is conceivable if an overlay node f re-located to a different area of the overlay's topology, and then failed. The border nodes building a failed section view of the original region in which f was located may find backups of the old neighbours of f which still note f as a neighbour. Concurrently, the border nodes building a failed section in the most recent region of overlay in which f was located may find backups of nodes from *that* region which list f as a neighbour, and therefore f is subject to be part of two different repairs.

Loss of repair updates

The state written to repair logs simply *must* be available to phase 1 instances, in order to prevent nodes from being repaired multiple times. As long as repair logs are available, the backups of failed ex-border nodes do not necessarily need to reflect repairs that have been made, as 'latent repairs' can be made by phase 1 nodes discovering backup state conflicting with information found in repair logs.

Even so, the requirements above are clearly quite strong, and actually impossible to guarantee without 'stable storage', which is nonexistent in many overlay deployments. Calculating the *probability* of the above conditions occurring is a very complex problem on its own, involving many factors, and is beyond the scope of this work.

This work has endeavoured to design a repair protocol which can theoretically guarantee that every failed node in an overlay is repaired exactly once; this section has begun to show how difficult this is in the context of an overlay network. As noted, this theoretical guarantee was

targeted based on a desire to design a *fully generic* protocol, capable of performing any repair strategy safely, most notably 'additive' repairs.

Having the capability to restore nodes is, as seen so far, very difficult. This is because it is an additive, *non-repeatable* action; the integrity of the overlay would be at serious, undetectable risk if a node was restored twice. Interestingly, *subtractive* repair strategies *are* repeatable—it is guaranteed by the nature of the systems considered that a node cannot in actuality be *removed* twice. Even if the recovery service attempts to do this, it cannot cause any damage, because something removed cannot physically be removed again, and the attempt to do it can cause no damage by trying.

Adapting the structure of an overlay to repair it is therefore much easier to support, and requires less strong assumptions. A discussion of the relative merits of different repair strategies in this context is deferred to the discussion of phase 3.

5.4.3 Phase 2

So far, scenarios that can arise due to phase 1 have been examined in detail. This section examines phase 2 in the same way. It is assumed that phase 1 has generated valid views—i.e. with up to date backups, and without false positives. The ramifications of not having such views from phase 1 have already been discussed.

Normal case When things go well, with all nodes agreeing on the same failed section view, phase 2 proceeds as already described, and must result in a transition to phase 3 for all border nodes involved in the same phase 2 instance; all such nodes will select the same coordinator, which will enact a repair in phase 3.

Border node failure Border node failures are expected to occur, and are handled by the protocol. The way they are handled depends on the timing of the failure. There are essentially two cases, classified with the notion of *phase 2 survivors*—border nodes that do not fail until phase 2 is complete.

Using this concept, there are then border nodes which fail *before* any phase 2 survivors receive any phase 2 messages from them, and those which fail *after* a phase 2 survivor receives a phase

2 message from them. Due to the properties of the protocol, if any phase 2 survivor receives a failed border node's opinion, then *all* phase 2 survivors will receive that opinion.

A transition to phase 3 is possible only in the latter of the above two cases—if all border nodes that fail do so *after* having sent a phase 2 agreement message to at least one phase 2 survivor. This ensures that all phase 2 survivors will receive opinions from all nodes in the given border set, and assuming they are all 'accept', can proceed to select a repair coordinator.

A phase 2 opinion must be received from *every* border node participating in that instance of phase 2 for the reasons given in the discussion of failure detection latency in phase 1—to allow 'failed' (i.e. $\perp$) to constitute an 'opinion' would allow nodes that failed long before a phase 1 view construction to be treated as submissive to a view, and therefore allow a node to ignorantly repair only a small part of a larger failed section that is concurrently being considered for repair by others.

Therefore, if a node's opinion vector contains any $\perp$ entries, it *must* return to phase 1 to re-try view construction.

Selecting a failed border node as coordinator If phase 2 is successful, so no border nodes failed before a phase 2 survivor received their agreement, then it is still possible that a repair will *not* be made from this agreement, because a failed border node may be selected as coordinator. This may seem counter-intuitive, and it should be simple for border nodes using the SelectCoordinator procedure to simply avoid border nodes marked as failed.

However, it is not possible to tell *when* a candidate coordinator failed, and therefore whether or not it completed its repair action before it failed. If a border node therefore observed that the ideal coordinator (based on repair strategy selection) had 'failed' according to the local failure detector, and that border node unilaterally decided to pick the 'next best' coordinator from the remaining live border nodes, which happened to be itself, then it can easily repair nodes that have actually already been repaired by the ideal coordinator *before* it failed.

All border nodes must therefore choose the same coordinator, regardless of whether that node is locally believed to have failed. If, after coordinator selection, a border node sees that this coordinator has failed, it will revert to the beginning of the repair protocol, and re-enter phase 1, as it will still have failed neighbours.

It will then eventually add the failed coordinator to its view, and locate its repair log (if present). Based on the information in the repair log, it can then carry out the local effects that the repair should have had if a repairOK message had been received from the coordinator. If no repair log was present in the coordinator's backup, then it failed before it was able to make its suggested repair, and so the larger border set will attempt to agree and repair the larger failure, now including the failed coordinator.

False positive (hypothetical) False positives have already been discussed in the context of phase 1, and it is assumed for this section that phase 1 has developed a view which contains no false positives. However, with a less-than-perfect failure detector, border nodes can be incorrectly reported as failed *during* phase 2.

As with the consensus algorithm on which this phase is based, it is tolerant of false-positive reports of all but one member of the border set (a member that must never be falsely reported to *any* border set member). This is not explored any further here, and the interested reader is referred to [15] for a discussion of the original theory relating to this.

5.4.4 Phase 3

Finally, the conditions that can occur during phase 3 are examined, most of which relate to the failure of the repair coordinator at various points during this phase. It is shown that these conditions cannot result in a violation of the protocol's correctness criteria, and cannot indefinitely impede its progress.

Coordinator failure before PrepareRepair This causes no impact other than to slow down the speed of repair. Other border nodes in this border set will eventually detect the failure of the coordinator, re-enter phase 1, locate the coordinator's backup and find that no repair took place, then enter into phase 2 to attempt to agree on and repair a now larger failed section view.

Coordinator failure during PrepareRepair The behaviour of the PrepareRepair procedure for a repair strategy must be such that it performs no operations affecting the live overlay network; this is true of the repair strategies described in section 5.3. Failure during this procedure is therefore equivalent to the above.

Coordinator failure before AddRepairToRepairLog This is equivalent to the above.

Coordinator failure before EnactRepair At this point, the intended repair has been logged in the coordinator's backup, which means the intended effects of the repair *will* take place eventually. The surviving border nodes in this border set will eventually detect the coordinator's failure, enter phase 1 as they must still have a failed neighbour, and construct a failed section view up to the coordinator.

They will find a logged repair in the coordinator's backup, which contains information that the other nodes in the failed section view have already been subject to a repair (whether the repair actually completed or not). The border nodes will then observe that they are not consistent with the repair log, and carry out the locally appropriate actions of the repair, as detailed by the repair strategy for non-coordinators (again, see section 5.3 for examples).

This will remove the failed neighbour originally causing this failed section view, possibly replacing it with another failed node, depending on the actions of the repair strategy—if so, phase 1 is re-initiated for this new failed neighbour.

Coordinator failure during EnactRepair Partial completion of a repair strategy ultimately equates to the same situation as the above case, though partial execution of the repair strategy (e.g. with some nodes restored) may leave less to repair in the follow-up iteration of the repair protocol. This works as shown in figure 34; a repair coordinator, p, begins enacting an additive repair strategy, and is able to restore nodes k and w before p itself fails. The other border nodes q, r, g and s revert to phase 1, re-constructing the original failed section in (a), but now find node p's repair log, and enact the effects of that repair locally; node s for example swaps its link to failed node w with one to the restored node w, no longer having any failed neighbours. The nodes that p did not restore before it failed become the subject of a future, smaller failed section agreement attempt, and the failed node p itself becomes part of a separate failed section (c).

Coordinator failure before send repairOK With the repair strategy executed in full, the repairOK message is used to inform border nodes that repair has completed, and that they may enact any elements of the repair local to their node. As discussed in section 5.4.3, if no repairOK

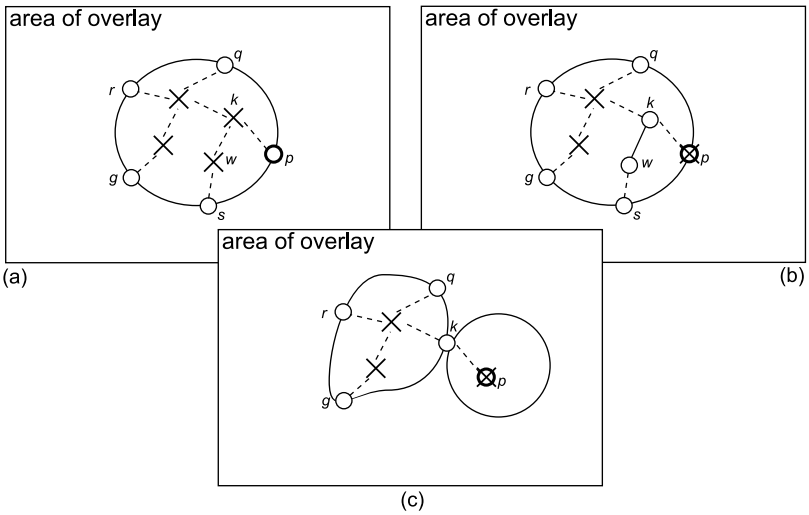


Figure 34: The coordinator of a repair (a), fails mid-way through its repair, having restored failed nodes k and w (b). The nodes it was not able to repair become part of a future agreement attempt and repair (c).

messages are sent, the border nodes will eventually detect the failure of the coordinator, include it in a future failed section view, and discover its repair log in phase 1. They will then enact the repair effects on their local node in the same way as if they had received repairOK from the coordinator.

Coordinator failure during SEND repairOK A repairOK message is sent to every border node, an action which is not atomic, such that some border nodes may be sent a repairOK message, and some may not. Those that do receive it will have enacted any local elements of the repair, as normal, and those that do not receive it will detect the failure of the coordinator and proceed as in the previous case.

Coordinator failure before REJECTVIEWSCONTAININGREPAIREDNODES The purpose of this procedure is to examine the local node's message buffer for any view messages containing nodes that the coordinator has just repaired. From the point of view of the sender of this view, this coordinator is a member of its border set, and it is awaiting an opinion on its view in round 1 of phase 2.

This procedure therefore supplies a reject message to any such border nodes, as it has just repaired some or all of the nodes contained in the proposed failed section. This allows those waiting nodes to proceed past round 1 of their phase 2 instances, and because a reject was sent

from this coordinator, they will communicate with that node no further. If, instead of sending reject messages to these nodes, the coordinator fails before being able to do so, the nodes waiting for an opinion will eventually detect the failure, and will stop waiting for an opinion from this node.

Either way, therefore, these waiting border nodes will not be able to proceed to phase 3, and when they revert to phase 1 they will eventually locate backups which reflect the repair action (or they will detect the repair log of this node, which equates to the same thing).

Coordinator failure during RejectViewsContainingRepairedNodes This scenario is ultimately equivalent to that above, except that *some* waiting nodes will receive a reject message from the coordinator, and some will see it fail without providing an opinion.

Border node failure A border node that fails just before or during the repair process in phase 3 will eventually have its stored backup updated to reflect the changes made by the repair (i.e. the topology and state modifications that would have been made to that border node had it lived to see the repair).

This is guaranteed to happen as long as the repair coordinator logs its repair, and this repair log is located in the coordinator's backup should it fail.

Stable storage and subtractive repairs It has already been demonstrated that stable storage is necessary to ensure that every failed node is involved in exactly one repair. This, in turn, is necessary in particular when dealing with additive repairs, where failed nodes are being restored—performing such an action twice is potentially corruptive.

However, if it is for the moment assumed that only *subtractive* repairs are ever made, the assumption of stable storage can be weakened. The assumption is essentially used so that repair logs are guaranteed to be found, and thus already-restored nodes are not forgotten about. Since a failed node cannot be removed from the overlay twice, the issue becomes a simpler one of *consistency*.

Assume that repair logs are always lost. If a subtractive repair is partially completed due to coordinator failure as in figure 35 (b), such that some of the border nodes g and s have their topology / state updated, but some do not, this creates a problem in the follow-up phase 1 and

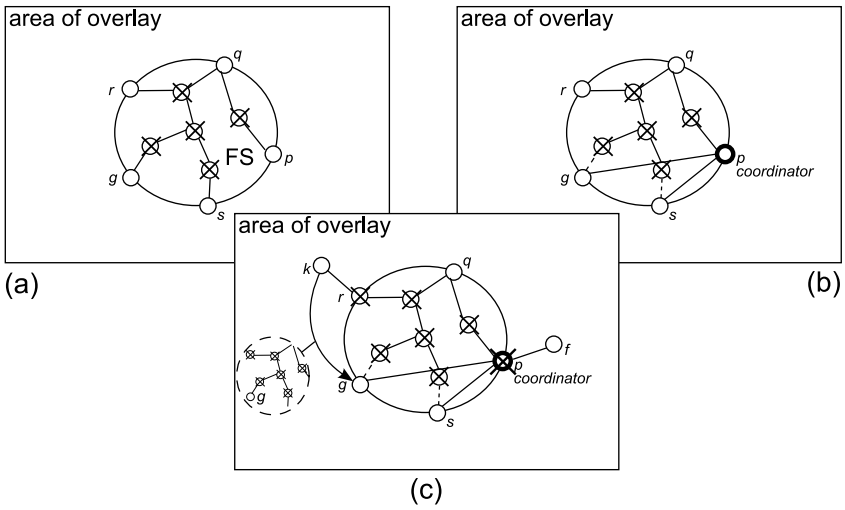


Figure 35: Different nodes observe different, contradictory structures of the overlay due to lost repair logs following the failure of coordinator node p mid-repair

2 instantiations, as nodes g and s may have views proposed to them which refer to failed nodes that according to their neighbour lists do not have any relationship to them—i.e. they are not interested in the failure of these nodes. In figure 35, nodes g and s have their links to nodes in the failed section FS replaced with links to node p (part (b)). Node k, which has not locally seen any effects of this repair, then proposes the original view FS to nodes g and s (part (c)), but these nodes no longer consider themselves as neighbours of the proposed view.

This scenario has occurred because the nodes g and s have an inconsistent view of the overlay network—their view is actually more up-to-date, but they do not know why or how. Now, consider altering the assumption; instead of repair logs always being lost, repair logs are only lost if all *effects* that they imply are *also* lost.

This is difficult to uphold for additive repairs, since the effects of an additive repair are actually newly instantiated nodes in the system, as well as local changes at border nodes, and these new nodes will not simply disappear if all repair logs referencing them are lost. However, for subtractive repairs it is simpler; when a border node receives a repairOK message, it logs the repair immediately. It then performs its 'side' of the repair, and logs the local *effects* of the repair (i.e. topology / state changes). Coupled with the new assumption that the backup service cannot return a node backup that contains the effects of the repair at this node, and does not contain the repair log (a reasonable assumption, given the logged-after status), this guarantees

that either a border node (or its backup) contains both the effects of the repair *and* the repair log (allowing latent phase 1 repairs on any other border nodes not containing the effects of the repair), *or* a border node backup contains only the repair log, or neither.

Looking at the full picture, this means that if no repair log is found from any border node, then no effects of the repair actually exist anymore, and so it is safe to perform any repair in this region again. This is, however, only valid if only subtractive repairs are permitted for the lifetime of the system.

Summary As has been shown in the previous sections, repair logs are essential to the correctness of the protocol in ensuring nodes are not repaired twice, and furthermore, repair logs *must* be found along with a node's backup, and these logs *must* be up-to-date with all repairs carried out by that node.

In systems as dynamic as many overlay network deployments, this simply cannot be guaranteed—it necessitates 'stable storage' for repair logs. However, as mentioned, the guarantee of repairing every node exactly once is necessary *only* for additive repair strategies in which nodes are restored. In subtractive repair strategies, the guarantee can be lessened to repairs of a failed node occurring *at least once*, which permits some important changes to the assumptions and details of the protocol, allowing it to operate in much more hostile—and realistic—environments.

It is important to know the limits of possibility, and this section has endeavoured to find those limits, demonstrating a theoretically correct, fully general repair protocol, which can safely perform any kind of repair, and showing the necessary conditions for such a protocol to operate correctly. The possible reductions in these necessary conditions have been shown when reducing the generality of the repair protocol to making *subtractive* repairs only—still general to many overlays, but not fully general in terms of possible repair strategies.

Even so, making coordinated *subtractive* repairs safely has shown limits of what is possible in terms of the necessary failure detectors and backup guarantees. Further reductions in the assumptions on these two important support services necessarily reduces the correctness of the protocol from *deterministic* to *probabilistic*. This is revisited in section 7.1.

5.5 Concluding remarks

Distributed systems today are moving ever more towards the entities in those systems having complex inter-relationships, exemplified by the peer-to-peer model, perhaps the ultimate demonstration of role multiplicity, where services are used by a peer, and services of a similar kind are *provided* by that same peer to other peers.

These complex inter-relationships of suppliers and providers woven throughout a distributed system work through *collaboration* between many entities to provide a needed service. This chapter has presented a repair protocol which also operates in an explicitly collaborative way.

This collaboration ultimately allows the repair protocol to be fully generic in the kinds of repairs that it can support, from subtractive adaptation of the overlay to additive augmentation. The cost of providing this support has been demonstrated, and the boundaries of possibility explored. This in turn fulfils requirements 1–4 (scalability, genericity to overlays, ability to harness the strengths of a deployment environment, and genericity to those deployment environments) for the recovery service in particular. An earlier version of this work was published in [63], along with a supporting technical report [64].

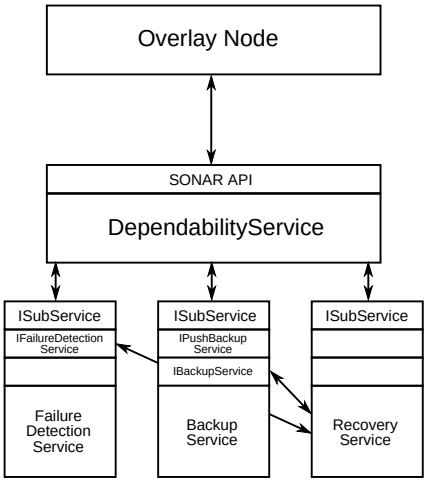


Figure 36: The implementation architecture of SONAR, showing relationships between services

6 Implementation

This chapter discusses the implementation details of SONAR, beginning with the overall architecture, and then detailing the individual sub-services. SONAR was implemented as a module within the Gridkit middleware, and was designed as a collection of easily replaceable components, following the philosophy of OpenCOM (Gridkit's underlying component model). The implementation is entirely in Java.

6.1 Architecture

The architecture of SONAR as seen by overlays has already been discussed in chapter 4. However, the internal architecture of SONAR's implementation is slightly more complex than this external view, with a high degree of modularity and configuration options. An overview is given in figure 36.

The central component is the *DependabilityService*. This is the overlay's primary point of access to the functionality of SONAR, serving as the entry-point for topology change notifications and SONAR message delivery. It is also responsible for loading and configuring sub-services. All standard features of the API are handled here, such as *registerNode()* and *neighbourAdded()*, and these are propagated to sub-services as appropriate.

All sub-services have a standard interface, and may also implement extended interfaces to describe specific capabilities. The standard interface is *ISubService*, which is very similar to

the main SONAR API, with the methods *registerNode()*, *unRegisterNode()*, *deliverMessage()*, *neighbourAdded/Removed()* and *getServiceName()*.

When the DependabilityService component is then informed through the main API that its local overlay node has added a neighbour, for example, this event is then forwarded to the currently loaded sub-services using their standard *ISubService* interface. This allows each sub-service to handle these events as they wish.

SONAR message transport is also achieved through the central DependabilityService component; when a sub-service wants to send a message, it requests that the DependabilityService do this, providing it with the message to send and the accessinfo representing the destination. The DependabilityService component passes this request on to the local overlay node, using the overlay-side API call *sendToService(...)*. When this message arrives at the destination overlay node, that node delivers it to the local DependabilityService component, which in turn delivers the message to the destination sub-service.

Each sub-service therefore has a reference to this central component, and uses it to send messages and acquire references to other sub-services as required. The recovery service needs access to a failure detector, for example, and so can request a reference to a component matching a given interface type from the DependabilityService.

6.2 Failure detection service

The failure detection service has an extended interface with the methods *setListenerOf()*, *unSetListenerOf()*, *hasNodeFailed()* and *isNodeFailed()*. The former two control 'registrations' of nodes of interest, such that the registering entity will be informed of changes in the failure status of registered nodes. *hasNodeFailed()* performs an immediate 'active' check to see if the given node is believed to be failed (e.g. by sending probes and waiting for a response or timeout), while *isNodeFailed()* checks if a registered node of interest is currently believed to be failed (so this call invokes no additional network activity).

Users of the failure detection service implement the interface *IFailureDetectionListener*, with the two methods *nodeSuspected(nodeID)* and *nodeNotSuspected(nodeID)*; these are called when the status of registered nodes of interest changes.

The implementation of the failure detector is a very simple one, and sends probes to all registered nodes of interest. If they do not respond within a certain amount of time, they are marked as failed. If at any point in the future they *do* respond, they are marked as alive once more, and the registered entities informed of the change.

6.3 Backup service

The backup service has two extended interfaces, *IPushBackupService* and *IBackupService*. The former is used by the overlay to advertise changes in nodestate, and the latter is used to acquire backups (e.g. by the recovery service during repairs). The backup service is notified about changes in topology through the general *ISubService* interface, so that it can distribute backups of the local node appropriately.

The implementation is again simple; a service instance periodically 'floods' a backup of the nodes on its host to all of its immediate overlay neighbours. Those neighbours re-forward the backup to *their* neighbours (other than the sender), and so on, for a given number of hops. Each successive backup is given a higher 'timestamp', so older backups are overwritten with newer ones. The implementation of both this, and the failure detection service, could be improved, but were not the focus of this work, and their described implementations are sufficient for general use.

6.4 Recovery service

The recovery service has no specialised interface, as it cannot be asked to do anything (other than its implicit tasking by being loaded in the first place). However, it does implement the *IFailureDetectionListener* interface, so that it can be notified of the suspected failure of nodes it is interested in (and obviously implements the *ISubService* interface, as all sub-services must).

When loaded, the recovery service acquires references to a backup and failure detection service instance, and registers with the failure detection service all of the overlay node's advertised neighbours, so that the recovery service will be informed if they are believed to have failed.

The service then becomes passive until a failure is reported. Meanwhile, the failure detection service probes registered nodes, and the backup service creates backups of the local nodes and

distributes these for safe keeping as described above.

On the report of a failure, the recovery service enters phase 1 of the protocol described in chapter 5; it uses the *getBackupOf()* method of the *IBackupService* interface to get the stored backups of each node detected as failed, examines the neighbours of these failed nodes as detailed in their backups, and uses the *hasNodeFailed()* method of the *IFailureDetectionService* interface to actively probe the status of these neighbours.

Once this discovery phase is complete, the recovery service registers all the border nodes with the failure detection service, using *setListenerOf()*, so that it is told if a border node is failed (and thus will stop waiting to receive an opinion from that border node).

Phase 2 then executes, and if successful phase 3 is entered; the coordinator logs and enacts the chosen repair, and the other border nodes again use *setListenerOf()* to receive notification of the failure of the coordinator. The coordinator uses an additional, internal interface on the backup service to attach repair logs to its node's backup.

6.5 Summary

This chapter has described the major implementation details of SONAR, which can be downloaded from [1] along with the rest of Gridkit. It has shown the modularity internal to SONAR, which promotes modifications to or switching of the implementations of each sub-service without needing to change other sub-services.

7 Evaluation

This chapter begins by first evaluating in isolation the repair protocol approach described in chapter 5, to provide an understanding of its behaviour. Following this, SONAR as a whole, including the repair protocol, is empirically evaluated in comparison with the proprietary repair approaches found in a representative group of overlay networks. Finally, the genericity of SONAR is examined, considering its applicability to current overlays, and looking ahead to possible future overlay designs.

Throughout this chapter, the optimised version of the repair protocol is used (such that a border node will not send to another border node an opinion it knows has already been received). All simulation described here is performed with a simulator developed in parallel with this work, which is particularly adept at expressing and evaluating peer-to-peer systems in detail, using a specially designed language and virtual machine [3].

7.1 Repair protocol evaluation

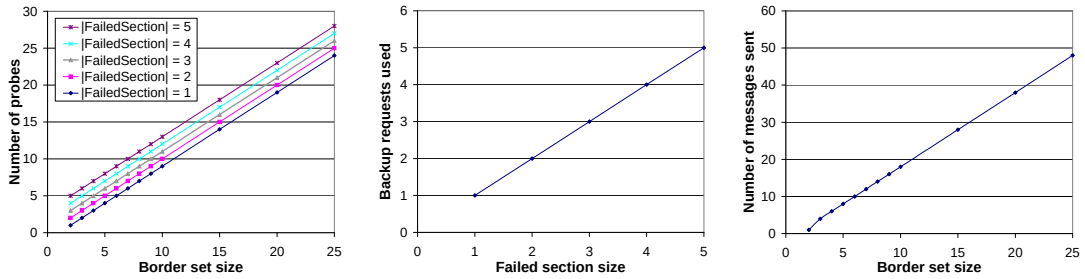
This section examines in more detail the behaviour of the repair protocol described in chapter 5. Its behaviour is examined abstractly, without reference to any particular overlay, to demonstrate its general properties. The behaviour of the protocol in 'normal' conditions is first explored—i.e. scenarios when its assumptions hold. This shows the normal costs involved in using the protocol.

Following this, the protocol is examined under the stresses of a more practical deployment, with experimentation and analysis of two key complicating factors—cascading failures, in which border nodes fail during an agreement attempt, and false positives reported by the failure detector.

This last scenario necessarily makes the protocol's correctness *probabilistic* rather than *deterministic*, and so this section investigates the probabilities of the protocol performing correctly under these conditions.

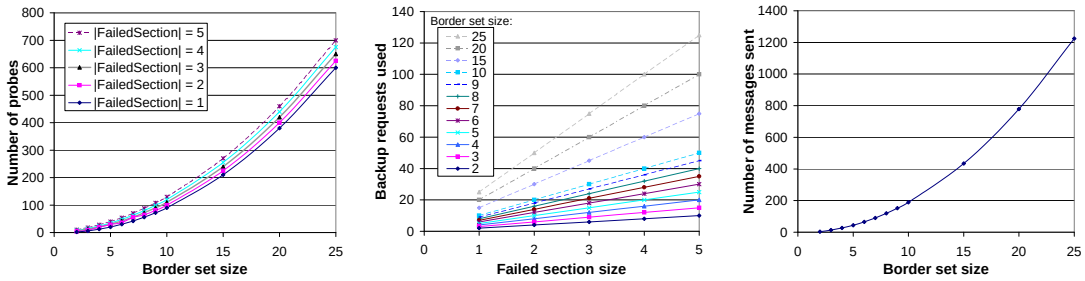
7.1.1 Evaluation under normal conditions

The four major factors in the per-repair overhead of the repair protocol are: (i) the number of failure detection probes incurred; (ii) the number of backup accesses incurred when constructing



(a) Failure detection uses **(b)** Backup requests used **(c)** Repair agreement messages

Figure 37: Per-border-node overhead during repair



(a) Failure detection uses **(b)** Backup requests used **(c)** Repair agreement messages

Figure 38: Combined overhead at border nodes during repair

failed section views (phase 1); (iii) the number of messages incurred to agree on a failed section and a repair coordinator (phase 2); and (iv) the number of messages sent to actually enact a repair (phase 3).

To evaluate these four elements, the repair protocol was simulated on an overlay network using scenarios with increasing border set sizes and failed section sizes. None of these scenarios involved cascading failures (i.e. there were no additional failures once the protocol began) and all of them assume no false positives in failure detection. As mentioned, the effects of these complicating factors are discussed in section 7.1.2.

The results in figure 37 show measurements of the first three of the above-mentioned factors as seen by each individual border node (in parts (a), (b) and (c) of the figure, respectively). The aggregated effect on the whole border set is shown in parts (a), (b) and (c) of figure 38. In all of these graphs, the x-axes represent either the size of the border set or the size of the failed section, depending on which of these factors is most relevant in the particular case; and

the y-axes represent the number of messages incurred.

The graphs show that the number of failure detection probes and agreement messages per border node grow linearly with border set size, and that the growth in backup accesses is linear with failed section size. In terms of the combined overhead incurred by an entire border set, the graphs in figure 38 show that the number of failure detection probes and agreement messages grows polynomially with border set size, and that the growth in backup accesses is linear with both failed section and border set size.

The protocol therefore scales quite nicely in terms of the load on each border node. Analytically, these first three factors can be expressed per-border-node as follows:

i. The number of failure detector 'uses' incurred by a border node is $(b + (f - 2))$, where b is the size of the border set and f is the number of nodes in the failed section. The reason for the '-2' is that the initial notification of a neighbour failure is not counted as a use of the failure detector during the protocol's execution, and a node does not need to check itself for failure during failed section view construction.

ii. The number of backup requests that occur is simply equal to f, i.e. the number of nodes in the failed section.

iii. The number of view agreement messages sent is $(b - 1) \times 2$ (for non-coordinator nodes; the repair coordinator must additionally send repairOK messages to the border set on repair completion, and so incurs an additional $b - 1$ messages). This is because, in the best case, there are 2 rounds, and each border node needs to send a message to every other border node (i.e. excluding itself) in each round. This equation does not apply to the special case of 2 border nodes, for which only 1 message is needed for non-coordinator nodes, and 2 for the coordinator.

Looking at the fourth factor of cost, i.e. repair enactment, the number of messages involved here is dependent on the repair strategy used. For example, subtractive repairs incur *no* extra messages as non-coordinator nodes simply replace failed links with a link to the coordinator, and vice versa. Additive repairs typically incur one extra message per restored node to instantiate and add state to that node, in addition to the cost of resource discovery to locate suitable hosts

(not considered here). Generally, these costs are minor in terms of message overhead compared to those incurred in phases 1 and 2.

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

7.1.2 Evaluation of key complicating factors

This section provides an analysis of the important 'complicating factors' of (i) cascading failures and (ii) false positives reported by the failure detector.

Cascading failures The repair protocol accommodates the failure of border nodes at any time, but assumes that ongoing failures will stop for long enough for the protocol to be able to complete. The implications of this are now examined.

As there are many different points at which border nodes can fail, this is a complex issue to analyse. The most useful analysis is of the *worst* possible case, and this is examined here. The worst case occurs when a border node fails at the very start of the protocol without providing an opinion, and a full series of rounds must be executed to no effect (the round-reducing optimisation being unusable here as there is missing data from the failed border node).

In such cases, the number of additional messages sent by each border node can be found using equation 1, in which $failedBNodes$ is the number of failed border nodes in the border set, and b is the size of the border set (including the failed members).

$$\underbrace{(b-1)}_{\text{round 1}} + \left(\underbrace{(b-2)}_{\text{remaining rounds}} \times \underbrace{((b-1) - failedBNodes)}_{\text{surviving nodes}} \right) \tag{1}$$

In equation 1 the first term $(b-1)$ represents the round 1 messages sent by each surviving border node to all other border nodes (including failed ones, which it is assumed have not yet been reported failed). The second term, $(b-2) \times ((b-1) - failedBNodes)$, represents the messages sent by each surviving border node to complete all remaining rounds, with each surviving node sending messages to all other surviving nodes in each remaining round. All failed border nodes that do not provide an opinion are necessarily detected as failed in round 1 by all surviving

border nodes, because border nodes must wait in a round to receive a message from every other border node, unless that node is reported failed. Using equation 1, the overhead of handling border node failures is *less* with more failed border nodes within a single border set, but *more*—and *cumulative*—when successive agreement attempts abort due to having failed nodes in their border sets.

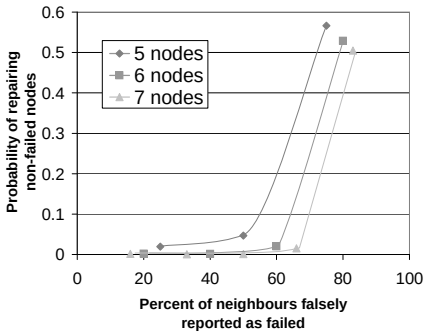
As an example, assuming a border set of size 4 with one failed border node, each surviving border node would incur a cost of 7 messages to abort the agreement attempt and be able to return to phase 1; $(4-1)+((4-2)\times(4-1-1))$.

Further agreement messages would then be required as normal in the follow-up phase 2 to actually make the repair. From the previous sub-section, it can be seen that a normal-case agreement between 4 border nodes would cost 6 messages per border-node (and an additional 3 for the coordinator).

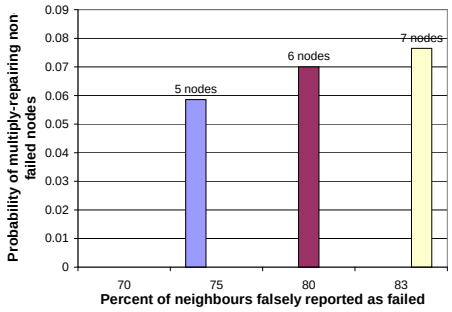
The worst case ongoing failure scenario may therefore result in a message count almost double the normal case (as an abort is needed plus the follow-up agreement). Given the strong properties of the protocol, however, and the specific conditions under which these higher costs are incurred, this overhead is reasonable.

Failure detection inaccuracy As with any repair approach, the repair protocol is adversely affected by *false positives*, where a failure detector wrongly declares a node as having failed. There are two major cases to consider here.

Case 1 involves a node p being told incorrectly that one or more of its neighbours has failed, going on to construct a view of the supposed failed section, and proposing this view to its supposed border set. There are two possible sub-cases here. The first, and most likely, is that the other supposed border nodes simply disagree with p's view (because at least some of their failure detectors did not report a false positive). Here, the agreement attempt will fail, and the only adverse effect will be p's wasted effort, involving phase 1 and on the order of $(b-1)\times 2$ phase 2 agreement messages ('propose' from p and 'reject' from the other $b-1$ border nodes). The second sub-case occurs when the failure detectors of every node in the supposed border set are *all* uniformly incorrect such that they all agree with p. Here, the protocol will erroneously proceed to repair non-failed nodes.



(a) Case 1: probabilities of repairing non-failed nodes, with increasing percentages of false positives and increasing mesh size



(b) Case 2: minimum required percentage of false positives to potentially cause concurrently repaired nodes, with corresponding probability of such repairs occurring

Figure 39: Effects of false positives

However, such errors can be straightforwardly countered by sending a signal to nodes in a failed section prior to phase 3. Nodes wrongly assumed to have failed which receive this signal are expected to respond either by terminating or by rejoining the overlay using some overlay-specific mechanism. Thus, although some wasted effort has been incurred, there is no threat to the integrity of the overlay from faulty failure detection.

Case 2 is more problematic. This happens when false positives occur in the following context: (i) multiple concurrent executions of the repair protocol are in operation and therefore multiple border sets are concurrently being formed; *and* (ii) within each of these concurrent executions all members of the respective border sets agree on their failed section views; *and* (iii) more than one of these border sets includes one or more of the *same* nodes in its failed section view. This unfortunate combination of circumstances can lead to multiple repair coordinators operating on the links of supposed failed nodes without knowledge of each others activities. This, in turn, can lead to race conditions which can potentially damage the integrity of the overlay.

Developing a completely general analysis of the probabilities of the above two cases occurring is a challenging problem, as it depends both on the topology of the overlay (of which there are many possibilities, as discussed in chapter 2) and on the specific placement of false-positives within that topology. This thesis does not attempt to fully model this problem for every possible topology, instead simplifying it slightly to provide a general outline of the effects.

To achieve this, simulations were performed in simple fully-connected mesh overlays of increasing node population. Each node in these topologies was then informed that a percentage of its neighbours had failed, when in fact they had not. For each mesh size, for each percentage of false positives used, the nodes reported as failed to each node were changed, such that every possible 'pattern' was tried for that percentage level, to then obtain an overall probability of erroneous repair due to false positives.

The results of this are shown in figure 39. Figure 39 (a) pertains to the first case (sub-case 2, in which repairs go ahead), with the x-axis showing increasing percentages of false positives per node, and the y-axis the probability of repairing non-failed nodes at those percentages. Curves are given for three different mesh sizes. Figure 39 (b) pertains to the second case, giving the minimum false positive percentage at which any possibility of concurrently repairing nodes was observed, and the corresponding probability of this happening.

In both cases, the highest percentages of false-positives used essentially mean that each node wrongly believes every other node except one (and itself) to be failed. The 100% false-positives cases are not shown, as these obviously mean all nodes will repair each other with probability 1. The graphs show that a high percentage of false-positives is needed under this model (over 50% at every node) for any significant probability of erroneously repairing non-failed nodes in case 1. Further, an even higher percentage of false-positives at every node (75% and above) is needed in order to have any possibility of multiple coordinators concurrently enacting repairs involving the same node(s)—and even then this probability is very low.

This demonstrates the power of the consensus approach used in the repair protocol, which forces the independent failure detectors of border nodes to agree with one another on failure status, and while a perfect failure detector is required for its theoretical correctness, it takes significant errors from the failure detector to cause problems.

7.2 Comparative evaluation

SONAR is now evaluated as a whole, including the repair protocol—as stated in chapter 4, the genericity of a service must not be cost-prohibitive in terms of its overhead and performance compared with specific solutions. However, genericity *does* come at a price, and this section eval-

uates the viability of using SONAR instead of a custom-built dependability solution, examining the Chord [76], TBCP [58] and Gnutella super-peer [83] overlays.

The majority of this evaluation is performed using simulation, but it is believed that the factors being considered here—in particular the performance differentials—are applicable equally to real deployments. TBCP is additionally evaluated in a deployment on PlanetLab [2] to give an idea of the real performance on SONAR.

Throughout this section the following evaluation criteria are used:

- The speed with which repairs are completed
- The performance overhead of the fault-tolerance approach (both in failure-free conditions and during repairs)
- The disruption due to failures of the overlay's service provision
- The performance of the overlay after failures

For the above factors, SONAR's performance is evaluated when it is permitted to use subtractive repairs only, to mirror the kind of repairs made by the custom-built dependability solutions of the respective overlays. In each case, however, the use and effects of additive repairs are additionally considered for SONAR.

In all simulation graphs, time is expressed as 'virtual time', an abstraction used by the simulator to accurately model a distributed (i.e. inherently parallel) system with very high resolution. A single unit of virtual time elapses for a simulated node when it has executed a single (high-level) instruction.

7.2.1 Chord

Two versions of Chord were created for simulation; one uses Chord's proprietary repair mechanism as in [76], and the other uses SONAR for its repair. Both versions used identical failure detection techniques, and identical amounts of redundancy. Network sizes of 300 nodes were used, and over the course of an experiment 50% of the nodes were gradually failed by selecting several random nodes to fail at regular intervals.

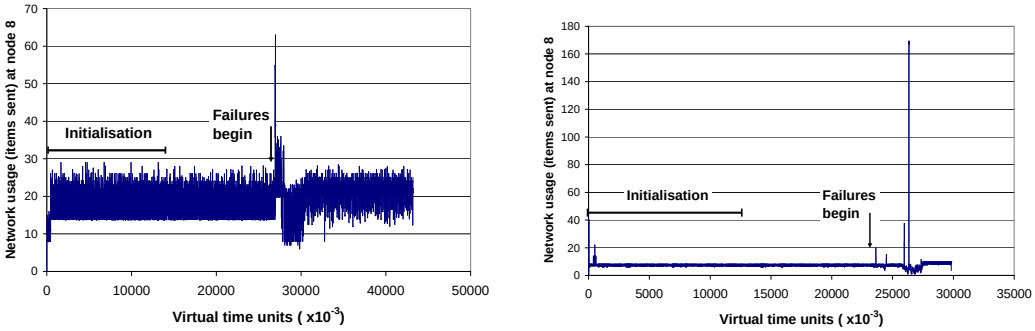


Figure 40: Standard Chord (left) and SONAR-Chord (right) network usage at node 8

Repair speed On average, a repair in the SONAR-enabled version took 1.1 times that of the version using the proprietary repair technique.

This is reasonable, given the amount of repair options offered by the protocol, and the guarantees provided—to relate this to real-time, Chord's proprietary repair mechanism takes on the order of milliseconds to complete, and consists simply of a node switching its successor from a failed node to the next live node on its successor list.

Performance overhead Per-node performance can be usefully measured here in terms of CPU load, memory usage and network usage. The impact on CPU load of running SONAR alongside overlay nodes is negligible, and is not considered further.

Figure 40 shows the overall fault-tolerance related network usage (in terms of the volume of data sent) of the two versions of Chord (both at node 8, a 'survivor') throughout the experiment. Standard Chord uses slightly more network resources during failure-free periods, due to its continuous successor list maintenance (whereas the SONAR backup service 'backs off' as the network stabilises); the SONAR version does however show a number of 'spikes' in network usage; these are due to larger failure groups occurring, requiring more messages to repair.

Figure 41 shows the overall average memory usage of the two Chord versions throughout the experiment. They show the period before any failures occur, up to time 4800, showing the higher memory load of SONAR, and the period during failures when the memory usage of both versions increases steadily; this is due to there being less nodes in the Chord rings at which to store the same amount of DHT data.

The additional memory costs seen in SONAR are due to genericity—ultimately, more data

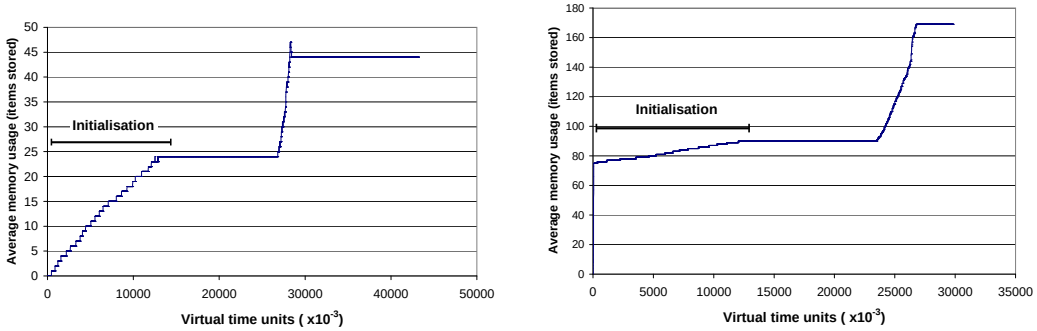


Figure 41: Standard Chord (left) and SONAR-Chord (right) average memory usage

must be stored to achieve the same resilience level. In the two failure free periods, this overhead is roughly 4 times that of standard Chord. These costs include necessary meta-data enclosed with backups, in addition to repair logs from the recovery service.

There are two main conclusions that can be drawn from these results. Firstly, in the failure-free period at the start of the experiments (up to the marked points) the 'ambient' messaging overhead of using SONAR can be observed—slightly lower than standard Chord, due to the removal of successor list maintenance.

Secondly, during some failures, SONAR uses more network resources, above that during failure-free periods, while standard Chord uses roughly the same amount. This is because the SONAR repair protocol described in section 5 is pro-active in discovering the failed section (i.e. sending failure detection probes), and requires agreement messages to be sent to reach consensus on the failure extent and repair strategy. By contrast, standard Chord nodes simply make a local update to their successor link as they detect that it is failed (replacing it with the next node from their successor list).

Disruption due to failures Repair speed has been examined in isolation, but not the disruption that is caused to overlay service while failures (and their repairs) are ongoing. This is tested by attempting to acquire keys from the Chord ring while failures are present in the overlay. Obviously, with a longer repair time, it can be expected that SONAR-Chord will experience higher temporary performance degradation due to failures.

In fact, standard Chord failed to acquire 27% of keys in these circumstances, and SONAR-Chord failed to acquire 47%. Following the conclusion of all repairs, both versions had 100%

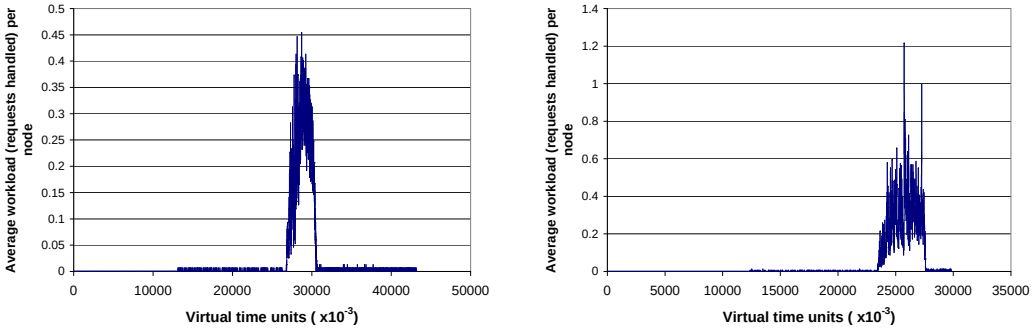


Figure 42: Standard Chord (left) and SONAR-Chord (right) average workload

success at acquiring keys (indicating that no data was lost, and no failures were unable to be repaired).

Overlay performance Figure 42 shows the average per-node 'workload'—i.e. number of requests being handled by each node—for the two versions of Chord. As with the memory load graphs, a node's workload increases steadily over the course of the experiment, as there are less nodes servicing the same number of requests. The sharp drop at the end of the experiment is caused by the end of stress testing, and a much less intense final 'full-scan' to ensure all starting DHT resources are present in the network following the completion of all repairs. The slight difference in the profiles of the two graphs is caused mainly by the occasional spike in SONAR as a failure, and slightly slower repair, causes a build-up of requests at the point in the ring 'before' that failure.

Using additive repairs In all of the previous, SONAR used subtractive repairs to recover the overlay after a failure—i.e., mirroring the strategy used by the overlay's proprietary fault-tolerance mechanism. However, with a simple configuration option, SONAR can be permitted to use additive repairs where possible. The rest of SONAR remains identical to the above, but when a failure occurs, nodes will consider repairs that restore those failed nodes on alternative sufficiently resourced hosts from outside the overlay.

Of course, this assumes the existence of such hosts, but this is not an unusual scenario in a Grid-like deployment. Using additive repairs, such that 50 'spare' suitable hosts are assumed to exist, figure 43 shows the resulting average workload over time.

This clearly achieves a generally lower workload over time than the version using only sub-

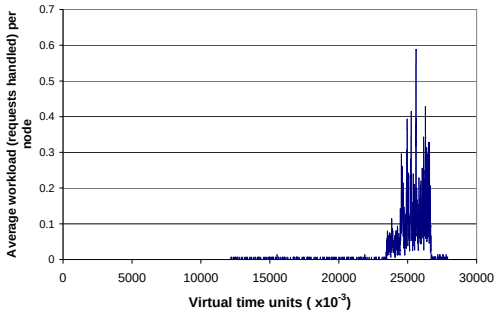


Figure 43: Chord-SONAR average workload with additive repairs

tractive repairs, as can be expected—there are more hosts remaining in the overlay by the end of the experiment. This demonstrates the power of adaptation to the deployment environment that can be leveraged from a strong recovery protocol. Moreover, this adaptivity is generic to a wide range of overlays, which can straight-forwardly harness the same benefits.

Summary While the cost of using SONAR is generally higher than the proprietary fault-tolerance of Chord, this is most evident with the memory overhead of a relatively basic backup service implementation. General messaging overhead is slightly less that standard Chord, and repair overhead is more, taking slightly more time and using briefly more resources. However, all of these costs are reasonable for a generic service, especially given the adaptivity and modularity it possesses; even the most significant impact—disruption due to failures—is just 20 percent higher with SONAR.

7.2.2 TBCP

Again, for TBCP, two versions were created, one with custom-built fault-tolerance mechanisms, and the other supported by SONAR. In this case network sizes of 1000 nodes were used, and again 50% of the nodes were failed over time by selecting several random nodes to fail at regular intervals. The approach adopted for TBCP's proprietary fault-tolerance mechanism is a system in which each node remembers its grand-parent, and the tree root node, so that a node with a failed parent simply re-initiates the standard 'join' procedure either at its grand-parent, or if that node is also failed, at the tree root. Any children of such displaced nodes remain with them in this process. This approach is fairly simple to implement, and maintains a good tree 'shape'

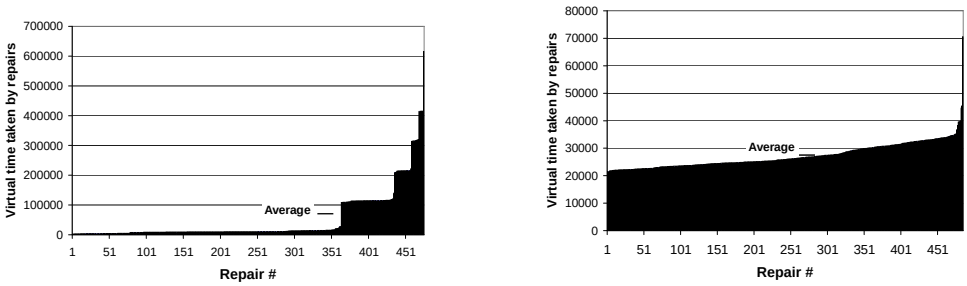


Figure 44: Standard TBCP (left) and SONAR-supported TBCP (right) repair times distribution.

(i.e. in terms of balance) [84].

SONAR-supported TBCP was also evaluated on PlanetLab [2], to examine the behaviour of SONAR in a more realistic setting—this is a time-consuming process, and so was only performed for SONAR-supported TBCP. The PlanetLab deployment was of 150 hosts, with similar configuration parameters to the simulated version, apart from the failure amounts, which in PlanetLab were at 80% of the overlay (giving more data to analyse from these smaller experiments). This PlanetLab deployment is also used here as a basis to assess how frequently the 'complicating factors' of the repair protocol occur in a real environment.

Repair speed In simulation, the average repair time of standard TBCP was 1.8 times slower than that of the version supported by SONAR.

This is explained by the re-join-at-grand-parent/root strategy employed for the proprietary TBCP repair mechanism. As mentioned above, this maintains a very good tree shape, but can take a significant time to 'stabilise' as displaced nodes find new positions in the tree, particularly for failures close to the root.

To put this in context, figure 44 shows the spread of repair times in the two simulated versions—the fastest repairs were 1,868 simulation time units for standard TBCP, and 21,444 for SONAR-supported TBCP; and their slowest repairs were 615,144 and 70,533 respectively. Standard TBCP therefore performs faster in 'easy' cases where nodes fail close to the leaves of the tree, but becomes much worse near the root, as a node re-joining at a high-up grand-parent or at the root will likely need to traverse a significant proportion of the tree to find the ideal place. SONAR has repair time behaviour which takes longer than 'easy' tree repairs, but deviates far less, giving it a lower average.

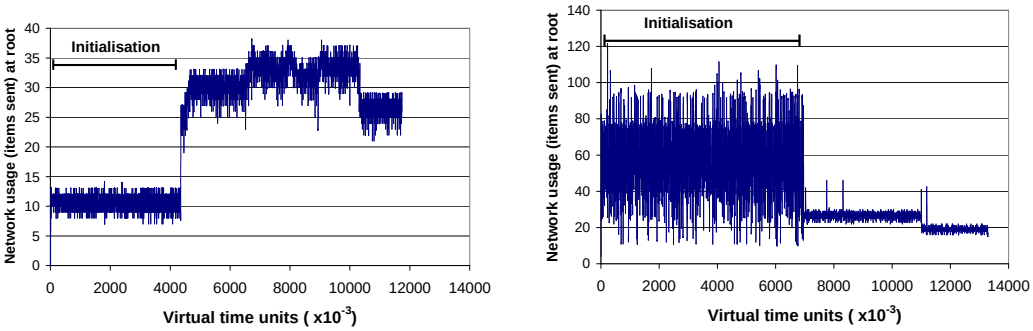


Figure 45: Standard TBCP (left) and SONAR-TBCP (right) network usage at the root node in simulations.

The PlanetLab tests under similar conditions show that SONAR took an average time of 30s to repair a failure. While this appears to be a long time, it should be noted that PlanetLab is a very heavily loaded environment [21] with many distributed systems sharing its resources. Furthermore, the deployment is truly global, with nodes in America, Europe and Asia.

Performance overhead Again, the major two measures of overhead are network usage and memory usage. Figure 45 shows the network usage of the two TBCP versions, as sampled at the root node, which is representative for these purposes of the network in general, and also shows us fine details.

The large jump in the standard TBCP version occurs when test data begins to be sent through the tree, at around time 900. The network usage of the SONAR-supported version is clearly higher during the initial construction of the overlay—this is due to the 'active' backup service, which has to send full node backups around when changes occur, as opposed to the standard version's duties being limited to simply remembering which nodes are its grand-parent and root. The tree protocol takes some time to stabilise, and so the backup service is unable to 'back off' in its dissemination aggressiveness until around time 1500. Failures began soon afterwards, and network overhead from this point onwards is comparable to that of standard TBCP, except for occasional spikes as a larger repair is needed. The SONAR version's overhead drops slightly at around time 2250 because one of the root's children fails at this point, and the resulting subtractive repair observes a less-loaded border node than the root to act as the repair hub and take on the children of this failed node, leaving the root node with one less child node from this point onwards (just before this drop, a brief spike can in fact be seen highlighting the overhead

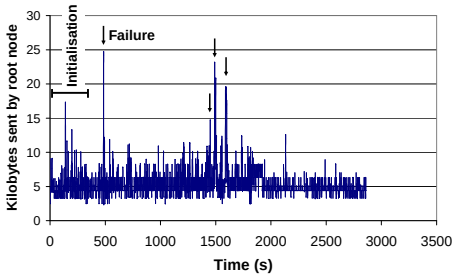


Figure 46: SONAR-supported TBCP network usage at the root node on PlanetLab.

of repair agreement here).

Figure 46 shows the SONAR-supported version's network usage in its PlanetLab deployment, giving a flavour of the real costs involved. The figure shows the root node's network usage over time in terms of kilobytes sent; simulation results are more abstract, showing the message sizes sent in terms of the number of elements their packets contain.

The PlanetLab graph in figure 46 is a slightly different shape during the initialisation of the overlay; this is because a somewhat (manually) optimised overlay construction protocol was used on PlanetLab to more quickly build the tree with reduced stress on PlanetLab itself. It has no effect on the post-initialisation behaviour of SONAR—the general trend of which is the same as simulation, with reasonably low normal network usage, at around 5KB per second on average. The node shown was involved in 4 repairs throughout the experiment (marked with arrows), and these cause briefly higher volumes of network traffic, as also occasionally occurred in simulation.

Turning now to memory overhead, figure 47 shows SONAR-supported TBCP's memory usage in simulation. This again is due to node backup storage. Standard TBCP has no notable memory usage, and so its graph is not shown. While memory usage here is measured in simulation units of storage, by relating this to the PlanetLab network usage graph, we can see that memory usage peaks at around 170KB.

Disruption due to failures During the entirety of each experiment, data was being sent through the tree. This data was numbered sequentially, and each node recorded how many pieces of data it missed throughout the simulation. In the standard TBCP tests, nodes missed an average of 15.9% of all data items sent over the course of the experiment, with an average of 0.3% lost in one continuous 'gap'. With SONAR, nodes missed an average of 13.2% of all

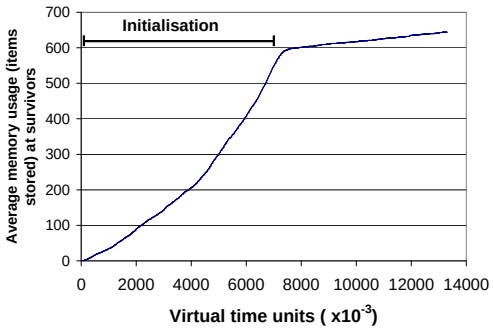


Figure 47: TBCP-SONAR memory usage

data items sent, with an average of 0.6% lost in one continuous gap. The higher overall loss in standard TBCP is due to the disruption caused by the re-join repair approach. While this is good for maintaining an optimal, balanced tree 'shape', it takes time for all children of a failed node to find new places in the tree, and can cause some 'displacement' of other nodes in the process.

Overlay performance The performance of TBCP is measured here by how much 'workload' its nodes have—i.e. how much each node has to contribute to the overlay. In TBCP this equates to the number of *children* a node has in the tree—how many nodes it has to re-forward data to. By the end of the simulations, standard TBCP had an average child count of 0.85; SONAR-supported TBCP was marginally higher at 0.98. Bearing in mind that standard TBCP's re-join approach derives an optimal child count for the tree-building protocol, SONAR performs reasonably well in distributing workload in its repairs[6]. The PlanetLab deployment of TBCP performed slightly worse, with an average child count of 1.3. This is higher simply because of different locations of failures in the PlanetLab tests, and because more of the overlay was failed in the PlanetLab tests than in simulation, at 80% rather than 50%.

Using additive repairs In TBCP, the use of additive repairs means less of an added burden on surviving nodes from taking up the extra child-count workload described above. This time, 100 'spare' external hosts were assumed to be available, and the enabling of additive repairs in

[6]These figures may seem surprisingly low, but they are drawn downwards by the number of leaf-nodes in the tree, i.e. nodes with no children.

SONAR resulted in the average child count being at 0.92, instead of the average of 0.98 achieved with subtractive repairs only (therefore becoming closer to the optimal).

Again, but for this simple configuration switch, the rest of SONAR remained unchanged.

Complicating factors Finally, to give some insight into how likely it is that some of the 'complicating factors' of the repair protocol occur in practice, in a selection of TBCP PlanetLab experiments run, in which there were around 100 repairs each, an average of just 20% of these repairs in each experiment had a border node failure. The need to expend much higher effort to handle border node failures was therefore relatively rare in these experiments with randomly selected failures, though obviously this is dependent upon the precise failure patterns that occur.

In terms of false-positives, looking again at results from PlanetLab, an average of just 4.2% of the nodes checked for failure by each node were inaccurately reported as failed. Clearly this is dependent upon the failure detection protocol used, and its configuration suitability to its deployment, but this gives an indication of the frequency of false-positives in a practical deployment setting, which is generally far too low to cause dangerous conflicting repairs.

Summary The cost of using SONAR is again higher than the proprietary fault-tolerance of TBCP, and as with Chord this is most evident with the memory overhead of the backup service implementation. General messaging overhead of SONAR is again on a par when the overlay is stable, with backup maintenance costs creating higher overhead during periods of instability (such as overlay construction), and brief periods of additional network overhead during repairs. The impact of failures when using SONAR is on average *less*—i.e. it takes less time than the re-join at grand-parent / root strategy, and there is therefore less data missed in total throughout the tree. SONAR does finish with a slightly higher workload per-node, but it is not significant enough to make use of the generic service infeasible.

7.2.3 Gnutella Super-Peer overlay

Finally, a Gnutella-like super-peer overlay network is examined under simulation. Again, two versions were created, one with custom-built fault-tolerance mechanisms, and the other supported by SONAR operating within the super-peer network. Network sizes of 320 nodes were used,

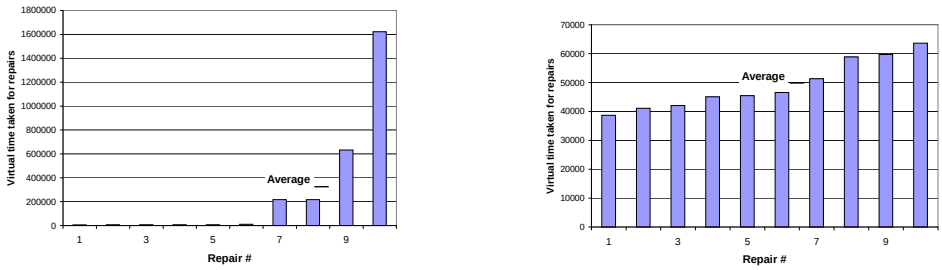


Figure 48: Standard Gnutella (left) and SONAR-supported Gnutella (right) repair times distribution.

and this was divided into 20 super-peers and 300 leaf-peers evenly distributed among the super-peers[7]. 50% of the super-peers were failed over time. The Gnutella-like overlay's proprietary fault-tolerance approach was for super-peers to advertise to their connected leaf-peers a list of alternative super-peers should a super-peer fail. If a leaf-peer detected the failure of its primary super-peer, it would switch to one of these alternatives. Additionally, super-peers themselves need to maintain their own connectivity, ensuring they have sufficient links to other super-peers after failures both for effective searching and to avoid becoming isolated from the rest of the network. This custom-built approach for Gnutella is a common strategy in super-peer overlays.

Repair speed SONAR again out-performed standard Gnutella in terms of average repair speed, with standard Gnutella's repairs taking on average 5 times longer than those under SONAR. Similarly to TBCP, this is because *some* repairs in standard Gnutella are much slower, caused by some leaf-peers taking a long time to locate an alternate super-peer. Figure 48 shows the distribution of repair times for both versions, and as with TBCP shows that while many repairs in standard Gnutella were faster to complete than under SONAR, there is much more deviation in repair times with standard Gnutella.

By contrast, rather than being leaf-peer centric, SONAR repairs involve only the super-peers, who select a repair coordinator, and this coordinator notifies displaced leaf-peers of their new super-peer as part of its nodestate (i.e. resource table) injection.

[7]The smaller network size here is due to the higher per-node intensity of Gnutella nodes (and similarly Chord nodes), which perform more tasks under the Gnutella protocol than does TBCP, making them much more time-consuming under the high-resolution simulation tool used. The network sizes used for these two overlays are still realistically large, and the decentralised nature of SONAR renders it unaffected by network size as demonstrated with the larger TBCP tests (i.e., SONAR will scale as long as the overlay itself is designed to do so).

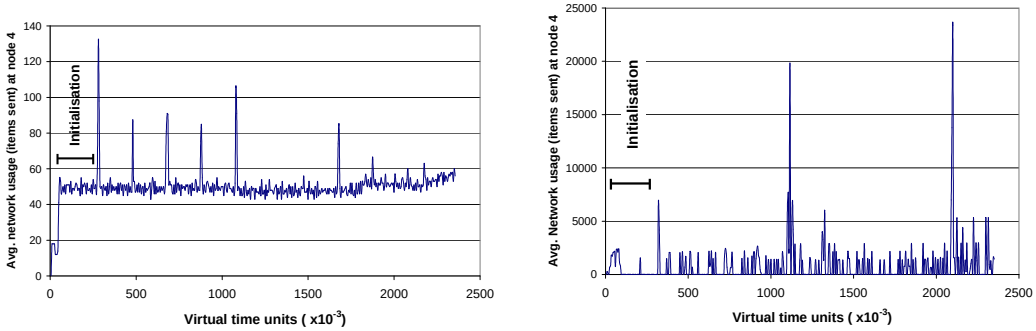


Figure 49: Standard Gnutella (left) and SONAR-Gnutella (right) network usage at node 4

Performance overhead Again, the major two measures of overhead are network usage and memory usage. Figure 49 shows the results for network usage for the two versions of Gnutella.

In the case of Gnutella, the initialisation process is very fast, and failures begin almost immediately at time 60. From this point forward, both versions experience a series of spikes in their respective graphs, with those in SONAR being more frequent and to a higher value. The spikes seen in standard Gnutella are due to the additional network traffic expended by super-peers to locate other super-peers to restore their desired connectivity level within the super-peer network.

In SONAR, the smaller and more frequent 'spikes' that can be seen in the network usage graph occur when a super-peer's state changes, requiring the updating and re-distribution of the backup of that super-peer. These changes to node state occur much more in Gnutella throughout the experiment than in the other two overlays, and this is because when a leaf-peer downloads a resource, it advertises the fact that it is now sharing this resource with its super-peer. As a result, the super-peer updates its resource index, which requires a new backup to be created and distributed by SONAR, as resource indices are stored as nodestate.

The even higher spikes in SONAR occur when a larger repair is undertaken, involving a larger number of border nodes. Because the Gnutella super-peer network is a relatively densely connected mesh topology, more border nodes are generally involved in repairs, making them more expensive. This connection density additionally affects the backup service, which has more immediate neighbours to distribute its node backups to.

Figure 50 shows the average memory usage results for both Gnutella versions. The general profiles of the two graphs are the same, increasing over time as less remaining super-peers take

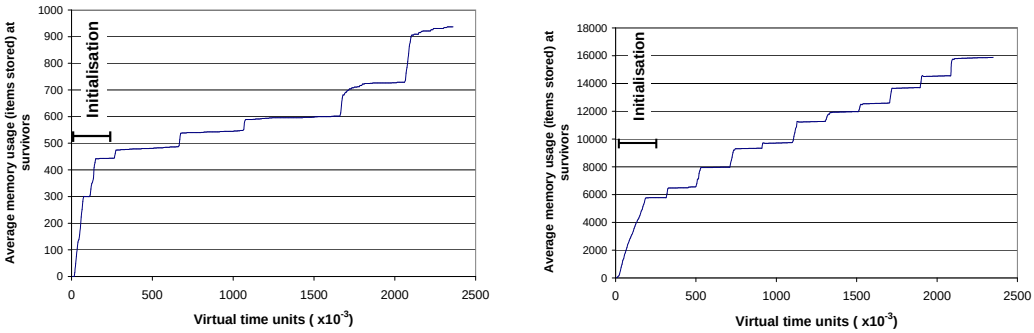


Figure 50: Standard Gnutella (left) and SONAR-Gnutella (right) average memory usage

on more responsibilities, but the SONAR version uses more memory on average per-node, again with the costs of generic backup and repair log storage (the difference more evident here because super-peer backups are relatively large).

Disruption due to failures During each experiment, requests for data were being made from leaf-peers; this involves sending their super-peer a lookup request, along with a search string, the super-peer flooding this lookup through the super-peer network for a certain TTL, and each receiving super-peer returning to the originating leaf-peer a list of other leaf-peers that host matching resources. The originating leaf-peer then selects one of these results, and contacts that leaf peer directly to download the resource. Finally, that leaf-peer adds the downloaded resource to its local list of shared resources, and advertises this fact with its super-peer. Throughout the simulation, leaf-peers kept a record of how many pieces of data were unobtainable, to gauge the disruption caused by failures.

Despite the significant difference in repair times, the results here are very close—in the standard tests, nodes missed a total of 0.49% of requested items, and in the SONAR tests, nodes missed a total of 0.43%. This closeness is explained by the fact that some repairs in standard Gnutella were completed more quickly than in SONAR, with just a few taking much longer (and of these long repairs, some proportion of leaf-peers would have found an alternative super-peer early on, completing their 'step' of the repair quickly).

Overlay performance Figure 51 shows the average resource index size of super-peers in the two Gnutella versions, which equates to the workload of super-peers in terms of how many requests they will be handling.

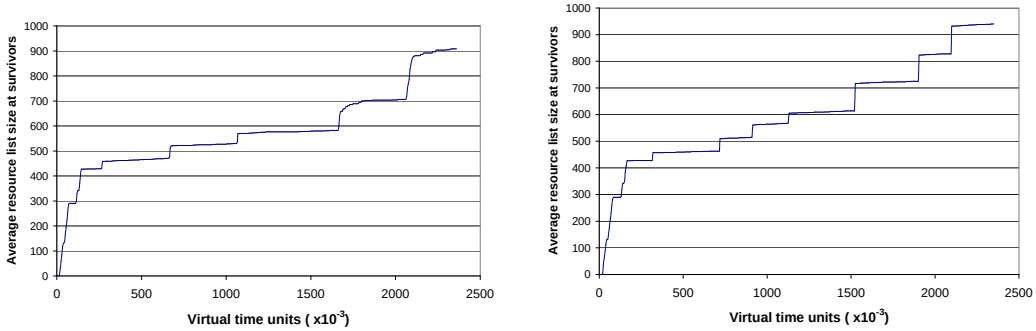


Figure 51: Standard Gnutella (left) and SONAR-Gnutella (right) average resource list sizes

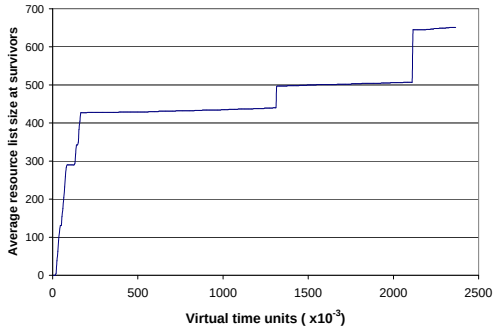


Figure 52: Gnutella-SONAR (additive) resource list sizes

SONAR workload finishes very slightly higher than standard Gnutella, seemingly creating very slightly more workload per super-peer. This is actually because slightly *more* requests were *successfully made* in the SONAR experiment due to repairs being faster, and thus resources were replicated at slightly more leaf-peers following successful resource download.

Using additive repairs Again, a simple configuration option enables additive repairs, allowing failed super-peers to be *restored* on alternative hosts where possible. With 5 such hosts made available, figure 52 shows the resulting average workload of super-peers at the end of the experiment, represented by their resource list sizes.

As can be expected, this is significantly lower than when only using subtractive repairs, as the workload can be distributed across more hosts.

Summary The cost of using SONAR for Gnutella is again higher in terms of messaging overhead at super-peers, this time more than with the previous two overlays considered. There are two reasons for this; firstly that the backup service implementation needs to disseminate

more frequent changes to the state of super-peers, and secondly that the more densely-connected mesh of the super-peer network in Gnutella means that repairs incur more overhead (as more border nodes need to agree). However, repair speed with SONAR is on average actually faster in fully completing repairs than the per-leaf-peer approach of standard Gnutella, and interestingly the SONAR repair model creates less work for leaf-peers, in that their 'side' of the repair is automated by the super-peer coordinating the repair.

7.3 Genericity evaluation

The genericity of SONAR is difficult to exhaustively evaluate; and impossible to definitively evaluate in terms of its genericity to future overlay networks, which may have vastly different designs to current overlays. However, the key factors that make overlays compatible with SONAR can be established, and from this the compatibility of arbitrary overlays—present and future—can be found by quickly examining these criteria.

Genericity has two major dimensions—the *overlay model* defined by the SONAR API in chapter 4, and the *service implementations*, such as the proposed recovery service implementation from chapter 5. For example, while this recovery service implementation may not suit all overlay network designs, an alternative implementation may still be possible while keeping the general overlay model and possibly the same failure detection and backup service implementations.

7.3.1 Overlay Model & API

The overlay model has the key simple concepts of accessinfos and nodestates. Accessinfos may have context tags, and collectively describe the topology that the overlay wishes to recover after failures. Nodestates are optional, and contain any other state that an overlay would like to make persistent. It is believed that this model is completely generic to every overlay design—every overlay by definition has topology, and some overlays have other state. The make-up of these two elements is left deliberately 'open' from SONAR's point of view, allowing arbitrary definition by the overlay. The two-way interaction model of the API is similarly minimal in its requirements, and open to overlay definition.

Given that any overlay can potentially *use* SONAR, then, with this minimal and open API, the questions are simply these: Are there overlays for which using a separated resilience service

does not make sense; and are there overlays for which the specific recovery protocol of chapter 5 is inappropriate?

In addressing the first of these two questions, the findings of chapter 2 are briefly re-visted. Four major overlay classes were covered; content dissemination, structured DHTs, semi-structured resource discovery, and unstructured / randomised overlays.

It was found that overlays of the former three classes have definitive 'functionality' and 'resilience' parts of their protocols. Content dissemination overlays have tree repair strategies, DHTs have replication, and semi-structured overlays have super-peer recovery strategies. All must clearly also have failure detection protocols.

Unstructured / randomised overlays can be distinguished from this group, as their resilience mechanism is often one-and-the same with their functional control protocols—a powerful example of this is that, in a gossip overlay, the fact of a node not having been heard from as part of normal gossiping can be used as failure detection. The normal topology maintenance protocol then automatically compensates for this by pruning the offending node from its 'view' and soliciting for alternative nodes to gossip with. As mentioned, however, such protocols are not currently capable of automatically persisting non-topological state such as DHT data, instead relying on external factors for this (such as the 'automatic' societal replication that occurs in Gnutella as more users download a file).

A general rule-of-thumb is then this: If the resilience mechanism of an overlay is clearly different to its control protocol, then it is suitable to externalise this responsibility to SONAR. Any overlay developer will know if they find themselves writing code beyond the overlay's functional protocols which is there only to deal with failures—if, for example, a failure detection protocol is *added* beyond the functional needs of the overlay, this is a good point to consider re-using a common service like SONAR.

From this point, a developer's mind is on the question *will a common service work with my overlay?* This is answered by the functionality provided by the sub-service implementations. Each is now examined in turn.

7.3.2 Service implementations

Failure detection is entirely generic, regardless of the implementation used—failure detection protocols simply send their own messages to a given node, and receive similar messages from nodes. The vast majority of failure detection protocols therefore always 'work', the only question being how *well* each one works. This is affected mostly by the density of a topology, and the expected speed of its hosting infrastructure (e.g. hosts and networks), and is thus a design choice. Again, however, virtually *any* protocol will work in general settings.

Backup is similarly generic; backup data is shipped from its source host to other hosts in the overlay, for lookup in case of failure. The data which is handled is that specified by the SONAR APIs—topology data and optional additional state. Again, the only question is the performance of this service in its deployment environment; its basic functionality will always work.

The only unknown is then the recovery service—to the overlay developer, will the manner in which it makes modifications to my overlay be accurate, and allow my overlay to continue to function as normal after a repair?

This in turn depends on the chosen implementation of the recovery service. The protocol discussed in chapter 5 is now examined in terms of its assumptions on the way an overlay works.

It operates by discovering a border of live nodes around a failed node / failed region of overlay, and selecting one of these border nodes as the repair coordinator. This coordinator is provided with the following information: The agreed membership of the border set; the agreed membership of the failed section; and the backups of each node in the failed section.

Using only this information, the coordinator must be able to repair the overlay. Strictly speaking, repair strategies are designed to be pluggable, and so in the worst case the overlay developer could create their own small module which takes the above information and derives a repair for SONAR to enact. This capability aside, the two *generic* repair strategies suggested in this thesis are i) additive repairs in which various numbers of nodes from the failed section are restored, and ii) single-hub subtractive repairs in which border nodes replace their failed links pointing into the failed section with a link carrying the same context to the coordinator, and the coordinator creates a link to each other border node carrying the context of its own link(s) that pointed into the failed section.

Strategy (i) is valid as long as a failed overlay node can 'cope' with being physically re-located while still representing its old topological and semantic place in the overlay. For example, a re-located Chord node will retain the same hashed ID that it had before it failed, not generating a new one based on its new physical location, as this would change the semantic meaning of the node in relation to the rest of the overlay. One example scenario in which this may *not* work is in an overlay where a node relied on a particular piece of hardware being present—not usually the case in commonly-used overlays, but possible in more specialised ones.

Strategy (ii), with topology adaptation, is valid as long as the topology between border nodes can be constructed legally in the way that the single-hub coordinator approach works—this is acceptable in Chord's core ring structure, TBCP's tree structure and the mesh-network of the Gnutella super-peer overlay (and by implication similar overlays to these three like SkipNet and Overcast). Essentially, this compatiblity is a quick decision for the overlay developer based on their knowledge of their overlay.

Taking a broader view, ultimately each sub-service of SONAR is a pluggable component, and others can potentially be developed if the existing ones are entirely unsuitable for a given overlay. The idea, however, is that each single pluggable service variant is generic to a wide range of overlays, and a wide range of deployment environments, as has been shown with the three developed in this work, and so far fewer variants are needed overall than is the case with per-overlay implementations.

7.4 Summary

This chapter has examined the properties of the repair protocol proposed in chapter 5, showing that its 'normal case' performance is good, scaling well per border node. It has additionally been shown that the repair protocol's performance in the face of cascading failures is reasonable, and that it is robust to the presence of false positives, requiring a large per-node percentage to create any problems. This chapter has also demonstrated the viability of the SONAR generic self-repair approach as a whole with three representative overlay types, showing that its additional overhead is acceptable, and that while it performs slightly worse in scenarios of very frequent overlay change and high interconnectivity (and could thus perhaps be improved for such cases) its

overall performance and flexibility is good. Finally, the genericity of SONAR has been examined, showing that the prototype implementation is applicable to a wide range of overlays, and that the modularity of the approach—both at the level of sub-service pluggability and repair strategy pluggability—allows it to be used in alternate configurations when necessary, often while still being able to keep some standard elements like the failure detection service in those alternate configurations.

8 Future directions

In this chapter, the state of this work is examined in terms of the next steps that could be taken to further this research. The discussion here follows the three-service composition, and then widens to view SONAR as a whole.

The backup service is the most obvious area of future work; while some current work could fill its role, or simple implementations could suffice for many cases, a fully configurable, adaptive implementation of this service would be beneficial—one which takes into account the observed regional stability of the overlay, the resources available on hosts for storing backups, and their perceived reliability, to provide a service which reacts to changes in the deployed overlay to provide a high assurance service while minimising overhead.

Such a service could also take advantage of different deployment environments such as Grid systems to leverage resources outside the overlay—such as regional, dedicated backup 'servers'—to further improve performance.

The failure detection service is in less need of attention, with several good alternatives available today for its implementation.

In terms of recovery, the proposed repair protocol is, of course, one of many possible implementations that could be encapsulated within the recovery service. It is however unique in its ability to safely enact any form of repair, enabled with the notion of *collaboration* between the parties interested in a given failure (defined as the 'border set' in chapter 5). This approach does have limits, as exposed in the evaluation—the more nodes that collaborate in a repair, the more costly it is to make that repair.

An interesting future direction would therefore be to examine methods of more finely controlling which nodes will collaborate in which repairs, beyond simply re-using the overlay's neighbouring relationship; such a capability would be particularly useful in densely connected topologies where border sets may become quite large. One approach to this has been alluded to already in the case of Chord. Here, the topology of Chord exposed to the service can be only the main ring structure, with finger table links are stored as nodestate. This essentially identifies a 'core' topology in Chord which must be maintained, and a 'secondary' topology which is automatically maintained by the overlay's functional protocol so long as the core topology is intact.

It may be interesting to take this model further, perhaps to the point where core topologies can be automatically identified, reducing the overhead involved in the repair protocol. Other ways of scoping the nodes 'interested' in a given node's failure may also be useful to explore, such as only those nodes which have higher physical capacities.

Transparently building a kind of 'repair interest' overlay within the overlay being supported may therefore be useful to get the most out of repairs by giving responsibility to limited numbers of nodes who are ideal repairers. An interesting trade-off to consider here is that the more densely connected these 'interest' relationships are, the slower repairs become as more nodes must agree, but conversely the damage due to false-positives is *avoided* with a higher probability, because the failure detector opinions of more nodes are being shared, and it becomes less likely that all of them are incorrect.

Taking a wider view of the overlay network research community, new overlays are still too often seen as being tied to a particular application and deployment environment—the concept of an overlay as a single, re-usable system component is only just emerging, with work like Gridkit and JXTA providing tools to support this model. Thus, the interplay between different overlays is only just starting to be considered, where overlays are composed as a stack, and when multiple overlays overlap and occupy the same overall system.

In these cases, fault-tolerance approaches can go further by ensuring that failure detection and redundancy protocols streamline their activities where possible (e.g. where a host runs multiple nodes either in a stack or as separate systems) so that overhead is minimised—and repair itself can be envisaged as leveraging cooperative hosts, and working with overlapping overlays, to provide a globally balanced experience.

There is also an interesting scope for generalised *optimisation* modules—while such a service was considered early in this work, it was necessary to develop the repair aspects of the design first, and this kind of intelligent, adaptive optimisation service is in itself a major undertaking, much like the backup service.

In general, the further modularisation and composition of overlays presents an interesting challenge, and one which can take overlays fully from single systems to interoperating and co-supporting virtual networks.

9 Conclusions

This thesis has presented the current state-of-the-art of overlay networks and their built-in resilience mechanisms, and surveyed existing work on generalised fault-tolerance support services for distributed systems. The shortcomings of current per-overlay resilience approaches have been addressed, showing that they create added work for overlay developers, particularly when the fault-tolerance mechanisms are clearly distinct from the overlay's functional protocols. Further, there is routine duplication of this work, especially in functionally similar overlays like Chord and SkipNet, where the functional similarity results in near-identical fault-tolerance approaches created by their designers. With these ad-hoc approaches to fault-tolerance, there is a lack of support for diverse deployment environments, and a lack of unified, easy to understand 'configuration parameters' across all overlays in terms of their fault-tolerance, making the job of application developers more difficult.

Furthermore, with the exception of failure detection protocols, which are well developed and well-suited to peer-to-peer environments, classic fault-tolerance services for distributed systems have been shown to be unsuited to deployment with overlay networks; almost all rely on additional managed infrastructure, require strict consistency rules unsuited to large-scale systems, or require systems to be built in very specific ways.

In response to these shortcomings, this thesis has proposed SONAR, a suite of fully decentralised services to support the self-repair of a wide range of overlay networks, providing a highly configurable and re-usable service which achieves a separation of concerns between an overlay's functionality and its fault-tolerance. The various facets of SONAR meet each of the requirements set down following a survey of existing work in both overlays and fault-tolerance.

Each sub-service operating with SONAR is entirely decentralised, and naturally scales with the overlay being supported, meeting requirement (1) of scalability. The separation of concerns is achieved using a carefully designed API, promoting bi-directional interaction between SONAR and the overlay it is supporting. By abstracting an overlay into its key components of topology and state, SONAR is able to maintain a high degree of genericity. By allowing the overlay to define its own semantic meaning within these key components, and further in the enactment of repair itself, the specific requirements and properties of individual overlays are

naturally supported. Detailed case studies of the Chord and TBCP overlays have demonstrated the applicability of this approach in diverse settings. This meets requirement (2) of re-usability and helps towards requirement (5) of genericity to past, present and future overlays.

Three major areas of fault-tolerance have been identified as being necessary for self-repair; redundancy (backup), failure detection, and recovery. Current work for both the backup and failure detection services has been reviewed, and a new approach to recovering from failures in an overlay has been proposed for the recovery service. This approach leverages distributed consensus, scoped to the area of a failure, to provide a strong repair primitive upon which many different kinds of repair can be used. Building on this, the concept of adaptive repair has been proposed, in which each failure is repaired in a way appropriate to the dynamics of the deployment environment. This is enabled by the collaborative nature of the repair protocol, ensuring safety for the most technically difficult repairs—adding new nodes to the overlay to replace failed ones. This repair protocol meets requirements (3) and (4), being generically applicable across multiple deployment environments, and being able to adapt to its deployment environment as the properties of that environment change.

Finally, the strong modularity of the SONAR implementation architecture has been presented, which promotes pluggability of its sub-components, and is well suited to changing small parts of SONAR while retaining others when necessary. This further supports meeting requirement (5) of genericity to past, present and future overlays—both at the level of repair strategies under the proposed repair protocol, and the level of sub-services themselves, when a particular implementation does not fit well with a specific overlay's design.

All of this work has been evaluated in multiple respects. The properties of the proposed repair protocol have been examined, demonstrating its scaling properties with increasing failed section sizes, its ability to react to cascading failures with reasonable overhead, and its resilience to false-positives. The performance of SONAR as a full self-repair service has been evaluated against custom-built fault-tolerance solutions for three representative overlays, demonstrating its acceptable overhead as a generic service, its generally good performance matching the custom-built approaches in many respects, and its flexibility beyond those custom-built solutions. And finally, the genericity of the overall approach to overlays past, present and future has been ex-

amined, showing that an open, minimal API provides a great deal of flexibility, and a modular architecture provides further support for diversity, both with pluggable sub-services and pluggable repair strategies under the generic repair protocol. Combined, these provide support for many foreseeable overlay designs, in addition to those already tested in the evaluation.

In summary, this thesis thus contributes the following to the research fields of overlay networks and fault-tolerance:

- A modular fault-tolerance service architecture suited to working in a peer-to-peer environment, providing generalised self-repair for crash-failures in overlay networks
- A way to effectively separate the concerns of fault-tolerance and functionality in overlay networks, using a semantically-open overlay model and overlay-service API
- A novel, decentralised repair approach suitable for use with a range of overlays, which is able to dynamically select repair strategies as appropriate for each failure

This thesis has shown that the separation of an overlay's fault-tolerance from its functionality is *desirable* in many cases due to the shortcomings of current approaches, is *possible* with an appropriate API and generic repair protocol, and is *feasible* in terms of the performance of the generic service and the additional overhead it brings.

With the emergence of overlay networks as the common approach to deploying new global services, the provision of a common, configurable and adaptive dependability service is a positive step towards the formalisation of overlays, and thus their viability as a highly re-usable system component. Looking to the future, further research into an adaptive, generalised backup service would be highly beneficial, as may a similarly generalised 'optimisation' service, and finding alternative and possibly automated ways to scope the involved parties in each collaborative repair may yield interesting performance results.

References

[1] Gridkit middleware public release. https://sourceforge.net/projects/gridkit/.

[2] PlanetLab. https://www.planet-lab.org/.

[3] ProtoSpace distributed algorithm prototype & evaluation tool. http://www.comp.lancs.ac.uk/computing/users/porterbf/.

[4] E. Anceaume, M. Gradinariu, and A. Ravoaja. Incentives for p2p fair resource sharing. In *The fifth IEEE International Conference on Peer-to-Peer Computing*, pages 253–260, Germany, August 2005.

[5] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *Symposium on Operating Systems Principles*, pages 131–145, Banff, Canada, October 2001.

[6] S. Bagchi, K. Whisnant, Z. Kalbarczyk, and R. K. Iyer. The chameleon infrastructure for adaptive, software implemented fault tolerance. In *Symposium on Reliable Distributed Systems*, pages 261–267, Indiana, USA, October 1998.

[7] Z. Bar-Joseph, I. Keidar, and N. Lynch. Early-delivery dynamic atomic broadcast. In *Proceedings of the 16th International Symposium on DIStributed Computing (DISC)*, pages 1–16, Toulouse, France, October 2002.

[8] S. Behnel and A. Buchmann. Overlay networks - implementation by specification. In *Proceedings of the International Middleware Conference (Middleware2005)*, pages 401–410, Grenoble, France, November 2005.

[9] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, pages 256–267, California, USA, February 2003.

[10] K. Birman. Replication and fault-tolerance in the ISIS system. In *Proceedings of the 10th ACM Symposium on Operating Systems*, pages 79–86, Washington, USA, December 1985.

[11] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in network simulation. *Computer*, 33(5):59–67, May 2000.

[12] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive microreboots: A soft-state system case study. *Performance Evaluation Journal*, 56(1-3):213–248, March 2004.

[13] M. Castro, M. Costa, and A. Rowstron. Should we build gnutella on a structured overlay? *SIGCOMM Computer Communication Review*, 34(1):131–136, January 2004.

[14] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in a cooperative environment. In *Proceedings of SOSP'03*, pages 298–313, New York, USA, October 2003.

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

[16] Y. Chawathe, S. McCanne, and E. A. Brewer. RMX: Reliable multicast for heterogeneous networks. In *INFOCOM*, pages 795–804, Tel Aviv, Israel, March 2000.

[17] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: a distributed anonymous information storage and retrieval system. In *International workshop on Designing privacy enhancing technologies*, pages 46–66, Berkeley, California, United States, July 2001.

[18] M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, pages 160–178, Heidelberg, Germany, November 2001.

[19] B. F. Cooper. Trading off resources between overlapping overlays. In *Proceedings of Middleware 2006*, pages 101–120, Melbourne, Australia, November 2006.

[20] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. A component model for building systems software. In *Proceedings of IASTED Software Engineering and Applications (SEA04)*, pages 684–689, Cambridge, MA, USA, November 2004.

[21] F. M. Cuenca-Acuna and T. D. Nguyen. Self-managing federated services. In *Proceedings of the 23nd IEEE International Symposium on Reliable Distributed Systems*, pages 240—250, Florianopolis, Brazil, October 2004.

[22] R. de Lemos and J. L. Fiadeiro. An architectural support for self-adaptive software for treating faults. In *Proceedings of the Workshop on Self-Healing Systems (WOSS'02)*, pages 39–42, Charleston, South Carolina, USA, November 2002.

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

[24] E. Durfee and V. Lesser. Negotiating Task Decomposition and Allocation Using Partial Global Planning. *Distributed Artificial Intelligence*, 2:229–244, January 1989.

[25] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[26] P. Felber and A. Schiper. Optimistic active replication. In *Proceedings of 21st International Conference on Distributed Computing Systems (ICDCS'2001)*, pages 333–341, Phoenix, Arizona, USA, April 2001.

[27] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. SCAMP: Peer-to-peer lightweight membership service for large-scale group communication. In *Proceedings of the 3rd International workshop on Networked Group Communication*, pages 44–55, London, UK, November 2001.

[28] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, March 1999.

[29] A. Ghodsi, L. O. Alima, and S. Haridi. Symmetric replication for structured peer-to-peer systems. In *The 3rd International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, page 12, Trondheim, Norway, July 2005.

[30] P. Grace, G. Coulson, G. Blair, L. Mathy, W. K. Yeung, W. Cai, D. Duce, and C. Cooper. GRID-KIT: Pluggable overlay networks for grid computing. In *DOA '04: Proceedings of Distributed Objects and Applications*, pages 1463–1481, Cyprus, October 2004.

[31] P. Grace, G. Coulson, G. Blair, and B. Porter. Deep middleware for the divergent grid. In *Proceedings of Middleware 2005*, pages 334–353, Grenoble, France, November 2005.

[32] P. Grace, G. Coulson, G. Blair, and B. Porter. Addressing network heterogeneity in pervasive application environments. In *Proceedings of the 1st International Conference on Integrated Internet Ad-hoc and Sensor Networks (Intersense 2006)*, page 20, Nice, France, May 2006.

[33] S. Guha, N. Daswani, and R. Jain. An Experimental Study of the Skype Peer-to-Peer VoIP System. In *Proceedings of The 5th International Workshop on Peer-to-Peer Systems (IPTPS '06)*, pages 1 – 6, Santa Barbara, CA, February 2006.

[34] I. Gupta, T. Chandra, and G. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 170–179, Newport, USA, August 2001.

[35] N. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, Washington, USA, March 2003.

[36] V. Hingne, A. Joshi, T. Finin, H. Kargupta, and E. Houstis. Towards a pervasive grid. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 207, Washington, DC, USA, April 2003.

[37] Y. hua Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 1–12, Santa Clara, California, United States, June 2000.

[38] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: analysis, module and applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, pages 381–390, Pasadena, CA, June 1995.

[39] D. Hughes, G. Coulson, and J. Walkerdine. Free riding on gnutella revisited: the bell tolls? *IEEE Distributed Systems Online*, 6(6), June 2005.

[40] T. Iimura, H. Hazeyama, and Y. Kadobayashi. Zoned federation of game servers: a peer-to-peer approach to scalable multiplayer online games. In *Proceedings of the Workshop on NetGames, SIGCOMM04 Workshops*, pages 116–120, Portland, Oregon, August 2004.

[41] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI)*, pages 197–212, San Diego, California, USA, October 2000.

[42] M. Jelasity and O. Babaoglu. T-man: Gossip-based overlay topology management. In *Proceedings of the 3rd International Workshop on Engineering Self-Organising Applications*, pages 1–15, Utrecht, The Netherlands, July 2005.

[43] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Proceedings of the 5th International conference on Middleware*, pages 79–98, Toronto, Canada, October 2004.

[44] M. Jelasity, A. Montresor, and O. Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. In *Proceedings of the 1st International Workshop on Engineering Self-Organizing Applications (ESOA'03)*, pages 265–282, Melbourne, Australia, July 2003.

[45] D. R. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *SPAA '04: Proceedings of the 16th annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 36–43, Barcelona, Spain, June 2004.

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

[47] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 179–188, San Diego, California, USA, June 2007.

[48] G. Kola, T. Kosar, and M. Livny. Phoenix: Making data-intensive grid applications fault-tolerant. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 251–258, Pittsburgh, USA, November 2004.

[49] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of SOSP'03*, pages 282–297, New York, USA, October 2003.

[50] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*, pages 190–201, Cambridge, MA, USA, November 2000.

[51] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: Exploiting the semantics of distributed services. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 43–57, Quebec, Canada, August 1990.

[52] N. Leibowitz, M. Ripeanu, and A. Wierzbicki. Deconstructing the kazaa network. In *WIAPP '03: Proceedings of the The Third IEEE Workshop on Internet Applications*, page 112, San Jose, CA, USA, June 2003.

[53] B. Li, J. Guo, and M. Wang. iOverlay: A lightweight middleware infrastructure for overlay application implementations. In *Proceedings of IFIP/ACM/USENIX Middleware*, pages 135–154, Toronto, Canada, October 2004.

[54] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SOSP '05: Proceedings of the twentieth ACM Symposium on Operating Systems Principles*, pages 75–90, Brighton, United Kingdom, October 2005.

[55] O. Marin, M. Bertier, and P. Sens. DARX - a framework for the fault-tolerant support of agent software. In *Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE'03)*, pages 406–418, Denver, Colorado, USA, November 2003.

[56] O. Marin, P. Sens, J.-P. Briot, and Z. Guessoum. Towards adaptive fault tolerance for distributed multi-agent systems. In *Proceedings of the European Research Seminar on Advances in Distributed Systems (ERSADS'01)*, pages 195–201, Bertinoro, Italy, May 2001.

[57] K. Marzullo, R. Cooper, M. D. Wood, and K. P. Birman. Tools for distributed application management. *IEEE Computer*, 24(8):42–51, August 1991.

[58] L. Mathy, R. Canonico, and D. Hutchison. An overlay tree building control protocol. In *NGC '01: Proceedings of the Third International COST264 Workshop on Networked Group Communication*, volume 2233, pages 76–87, London, UK, November 2001.

[59] T.-W. Ngan, D. S. Wallach, and P. Druschel. Enforcing fair sharing of peer-to-peer resources. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 149–159, Berkeley, California, February 2003.

[60] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An application level multicast infrastructure. In *3rd USNIX Symposium on Internet Technologies and Systems (USITS '01)*, pages 49–60, San Francisco, CA, USA, March 2001.

[61] B. Porter, G. Coulson, and D. Hughes. Intelligent dependability services for overlay networks. In *Proceedings of Distributed Applications and Interoperable Systems 2006 (DAIS'06)*, pages 199–212, Bologna, Italy, June 2006.

[62] B. Porter, G. Coulson, and F. Taïani. A generic self-repair approach for overlays. In *Proceedings of the Workshop on Reliability in Decentralized Distributed Systems (RDDS)*, pages 1490–1499, Montpellier, France, October 2006.

[63] B. Porter, F. Taïani, and G. Coulson. Generalised repair for overlay networks. In *Proceedings of the Symposium on Reliable Distributed Systems (SRDS)*, pages 132–142, Leeds, UK, October 2006.

[64] B. Porter, F. Taïani, and G. Coulson. Generalizing repair for overlay networks. Technical Report PTC–06–01, Lancaster University, 2006.

[65] J. A. Pouwelse, P. Garbacki, D. H. J. Epema, and H. J. Sips. The Bittorrent P2P file-sharing system: Measurements and analysis. In *4th International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 205–216, Ithaca, NY, USA, February 2005.

[66] H. V. Ramasamy, A. Agbaria, and W. H. Sanders. Semi-passive replication in the presence of byzantine faults. Technical Report UILU-ENG-04-2202, University of Illinois, February 2004.

[67] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, pages 68–79, Berkeley, CA, USA, February 2003.

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

[69] R. V. Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. Technical Report TR98-1687, Cornell University, 28, 1998.

[70] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, pages 127–140, Boston, MA, USA, June 2004.

[71] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proceedings of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, pages 267–280, San Francisco, California, USA, March 2004.

[72] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of Middleware 2001*, pages 329–350, Heidelberg, Germany, November 2001.

[73] J. Sacha, J. Dowling, R. Cunningham, and R. Meier. Discovery of stable peers in a self-organising peer-to-peer gradient topology. In *Proceedings of the 6th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'06)*, pages 70–83, Bologna, Italy, June 2006.

[74] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking*, San Jose, California, USA, January 2002.

[75] C. Shelton, P. Koopman, and W. Nace. A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems. In *Proceedings of the Eighth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS'03)*, pages 156–163, Guadalajara, Mexico, January 2003.

[76] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. 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*, pages 149–160, San Diego, California, United States, August 2001.

[77] L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz. OverQoS: An Overlay Based Architecture for Enhancing Internet QoS. In *1st Symposium on Networked Systems Design and Implementation (NSDI)*, pages 71–84, San Francisco, CA, March 2004.

[78] D. Thain, T. Tannenbaum, and M. Livny. *Grid Computing: Making The Global Infrastructure a Reality*, chapter 11 - Condor and the Grid, pages 299–335. John Wiley, 2003.

[79] J. Touch, Y.-S. Wang, V. Pingali, L. Eggert, R. Zhou, and G. G. Finn. A global x-bone for network experiments. In *Proceedings of IEEE Tridentcom 2005*, pages 194–203, Trento, Italy, March 2005.

[80] URL. http://rfc-gnutella.sourceforge.net/developer/stable/index.html.

[81] URL. http://www.darkridge.com/~jpr5/doc/gnutella.html.

[82] J. Verbeke, N. Nadgir, G. Ruetsch, and I. Sharapov. Framework for peer-to-peer distributed computing in a heterogeneous, decentralized environment. In *Proceedings of the 3rd International Workshop on Grid Computing*, pages 1–12, Baltimore, MD, USA, November 2002.

[83] B. Yang and H. Garcia-Molina. Designing a super-peer network. In *Proceedings of the 19th International Conference on Data Engineering*, pages 49–60, Bangalore, India, March 2003.

[84] M. Yang and Z. Fei. A proactive approach to reconstructing overlay multicast trees. In *IEEE INFOCOM*, Hong Kong, March 2004.

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

[86] Y. Zhu and B. Li. Overlay multicast with inferred link capacity correlations. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*, pages 54–63, Lisboa, Portugal, July 2006.

[87] S. Zhuang, D. Geels, I. Stoica, and R. H. Katz. On failure detection algorithms in overlay networks. In *Proceedings of INFOCOM'05*, pages 2112–2123, Miami, FL, USA, March 2005.

```
1:  procedure RepairProtocol_p
2:      loop
3:          wait for some of p's neighbours to fail
4:          repeat
5:              (BNodes_p, FSection_p) ← ConstructFailedSectionView()          ▷ Phase1
6:              SelectRepairStrategy(BNodes_p, FSection_p)
7:              V_p[] ←AgreeOnView(BNodes_p, FSection_p)                       ▷ Phase2
8:          until V_p[] only contains accept          ▷ Agreement on the failed section
9:          coordinator ← SelectCoordinator(BNodes_p, FSection_p)
10:         if p = coordinator then                                  ▷ I am the coordinator
11:             PrepareRepair(BNodes_p, FSection_p)
12:             AddRepairToRepairLog()
13:             EnactRepair(BNodes_p, FSection_p)                              ▷ Phase3
14:             Send ⟨coordinator, repairDetails, repairOK⟩ to all nodes in BNodes_p
15:             RejectViewsContainingRepairedNodes()
16:         else                                              ▷ I am not the coordinator
17:             wait for ⟨coordinator, repairDetails, repairOK⟩ or coordinator ∈ FailedNodes_p
18:         end if
19:     end loop
20: end procedure
```

Figure 53: Pseudo-code of the repair protocol when executed by node p

A Full round-optimised repair protocol pseudo-code

This appendix presents complete pseudo-code for the repair protocol of chapter 5. It is given with minimal commentary, as its functionality and behaviour have already been described in full. To begin, for completeness, figure 53 shows the overall protocol, and is identical to the previous appearance of this figure.

The procedure ConstructFailedSectionView(), marking phase 1, is shown in figure 54, along with a supporting procedure ConstructFailedSectionView() in figure 55.

The SelectRepairStrategy() procedure (called from line 6 of figure 53) is not shown as it is pluggable, and the implementations suggested are trivial. AgreeOnView() is shown in figure 57, and has been slightly modified to include a call to the ProcessRound() procedure, which expands upon how the end-of-round processing is performed, shown in figure 58. AgreeOnView() also here contains the round-reducing optimisation discussed in section 5.2.4. Finally, the pre-AgreeOnView() message filtering procedure is shown in figure 56.

```
1: procedure ConstructFailedSectionView_p
2:     HighestRankedBN ← ∅
3:     HighestRankedFS ← ∅
4:     for all q ∈ failed neighbours do
5:         repeat
6:             noconflict ← true
7:             (BNQ, FSQ, repairLogs) ← ConstructFailedSection(q)
8:             if any node in FSQ has a backup conflicting with a repair log then
9:                 update affected backups in FSQ to resolve conflicts
10:                noconflict ← false
11:            end if
12:        until noconflict
13:        if (BNQ, FSQ)isrankedhigherthan(HighestRankedBN, HighestRankedFS) then
14:            HighestRankedBN ← BNQ
15:            HighestRankedFS ← FSQ
16:        end if
17:    end for
18:    return (HighestRankedBN, HighestRankedFS)
19: end procedure
```

Figure 54: Pseudo-code of phase 1 when executed by node p

```
1: procedure ConstructFailedSection_p(q)
2:     BNodes ← ∅
3:     FSection ← q
4:     for all u ∈ q's neighbours do
5:         if u is reported failed then
6:             (NxtBNodes, NxtFSection) ← ConstructFailedSection(u)
7:             BNodes ← BNodes ∪ NxtBNodes
8:             FSection ← FSection ∪ NxtFSection
9:         else
10:            BNodes ← BNodes ∪ {u}
11:        end if
12:    end for
13:    return (BNodes, FSection)
14: end procedure
```

Figure 55: Pseudo-code of ConstructFailedSection when executed by node p

```
1: procedure ReceiveViewMessage(⟨r, BNodes_k, FSection_k, V_k⟩ from k)
2:     if FSection_k contains incarnation IDs of nodes I have repaired before then
3:         Send ⟨r_k, BNodes_k, FSection_k, V_k[p] = reject, repair_log⟩ To k
4:     else
5:         addMessageToBuffer(⟨r, BNodes_k, FSection_k, V_k⟩, k)
6:     end if
7: end procedure
```

Figure 56: The pseudo-code of ReceiveViewMessage

```
1: procedure AgreeOnView_p(BNodes_p, FSection_p)
2:     KB_p[|BNodes|][|BNodes|][|BNodes|]                    ▷ Initialise knowledge base
3:     KB_p[rn][c][k] ← ⊥ for all rn and c, and all k ≠ p
4:     KB_p[rn][c][p] ← accept for all rn and c
5:     OpposingNodes_p ← ∅
6:     for r ← 1 to the size of BNodes_p − 1 do
7:         if KB[r][k][j] for every k and j is ≠ ⊥ then
8:             return KB_p[r][p]
9:         end if
10:        for each non-failed node q in BNodes_p \ OpposingNodes_p do
11:            Send ⟨r, BNodes_p, FSection_p, KB_p[r][p] \ KB_p[r][q]⟩ To q
12:        end for
13:        PendingNodes_p ← BNodes_p \ OpposingNodes_p
14:        msgReceived_p ← ∅
15:        repeat
               wait: for next msg in buffer msg_k = ⟨r_k, BNodes_k, FSection_k, V_k⟩
                     such that ((BNodes_k, FSection_k) = (BNodes_p, FSection_p) ∧ r = r_k) or
                               (BNodes_k, FSection_k) is lower ranked than (BNodes_p, FSection_p)
16:                  (stop waiting and goto 19 if all PendingNodes_p are reported failed)
17:            removeMessageFromBuffer(msg_k)
18:            if (BNodes_k, FSection_k) is lower ranked than (BNodes_p, FSection_p) then
19:                Send ⟨r_k, BNodes_k, FSection_k, V_k[p] = reject⟩ To k    ▷ Reject lower-ranked view
20:            else
21:                msgReceived_p ← msgReceived_p ∪ {msg_k}                   ▷ Take opinion on my view
22:                PendingNodes_p ← PendingNodes_p \ {k}
23:            end if
24:        until PendingNodes_p = ∅ or all PendingNodes_p are reported failed
25:        ProcessRound(msgReceived_p, OpposingNodes_p)
26:    end for
27:    return KB_p[r][p]
28: end procedure
```

Figure 57: The pseudo-code of AgreeOnView when executed by node p

```
1: procedure ProcessRound_p(msgReceived_p, OpposingNodes_p)
2:     for all ⟨sender, r, BNodes_j, FSection_j, V_j⟩ ∈ msgReceived_p do
3:         for all n ∈ BNodes_j do
4:             if V_j[n] = reject then
5:                 KB_p[r + 1][p][n] ← reject
6:                 KB_p[r + 1][sender][n] ← reject
7:                 OpposingNodes_p ← OpposingNodes_p ∪ {n}
8:             else if V_j[n] = accept then
9:                 KB_p[r + 1][p][n] ← accept
10:                KB_p[r + 1][sender][n] ← accept
11:            end if
12:        end for
13:    end for
14:    KB_p[r + 2][n][m] ← KB_p[r + 1][n][m] for every n and m
15: end procedure
```

Figure 58: The pseudo-code of ProcessRound when executed by p

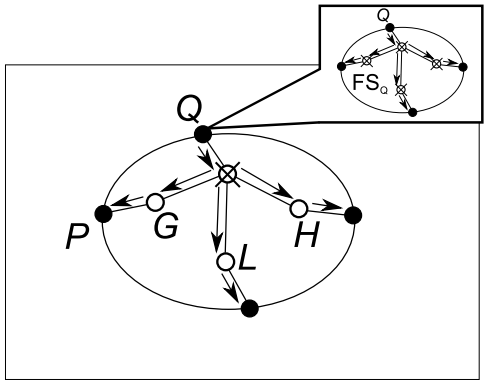


Figure 59: A node Q proposes a view including false positives (of nodes G, H and L), in which other border nodes have no interest

B Avoiding deadlock caused by false-positives

This appendix follows on from chapter 5 and presents the specific case of deadlock that can be caused by the occurrence of false positives, and provides a solution to avoid such cases. This deadlock situation can occur as in figure 59, where some border nodes do not care about a view being proposed to them because they (rightly) do not share the belief that their neighbours have failed, as suggested by that view.

However, the proposing node must get an opinion from all nodes in its border set, else it will not be able to proceed to take part in any future agreement attempts (and thus repairs)—for example, if node L later proposes a view to Q which suggests a border set of Q, L, H and G surrounding a single failed node, node Q will reject this view as it is lower-ranked than its own view.

The resolution to this problem is conceptually simple; border nodes note the time at which a view was proposed to them (necessarily including some of their neighbours in its failed section), and if after a certain amount of time D those border nodes have not detected that their associated neighbours have failed, they will send an FD_REJECT message as their opinion for this view. In the above example, this allows node P to reject Q's view if P does not believe its neighbour G is failed (after waiting a time D following Q's proposal).

However, this creates one additional subtle problem; a node must be able to *change its opinion*. This is a unique scenario which does not occur anywhere else in the protocol. Thus, node P which previously communicated the opinion FD_REJECT for a given view from node Q

may legitimately later detect the failure of its associated neighbours (G in the above example), effectively changing its opinion to $ACCEPT$. This can cause a problem, because this new $ACCEPT$ opinion could arrive first at Q, and be 'used up' by Q; the $ACCEPT$ will not then be sent again by P, causing a further potential deadlock scenario.

To address this, opinions have 'dependencies' attached to them; nodes remember if they are changing their minds about a view, and associate a dependency with their new opinion on their previous opinion. Nodes receiving opinions then note which opinions they have 'used'. If a node Q has an opinion x in its message buffer, and Q observes that it has not fulfilled a dependency that this opinion has, then Q does *not* use x, instead leaving it in its message buffer.

Q will at some point receive the previous opinion y which x depends upon being processed first, and Q will use this opinion in its current view agreement instance, not x[8]. If the communication protocols in the environment guaranteed ordering in all cases, of course, this would not be a problem—however, it is likely that in such a case node P will use two different connections to node Q over which to send messages, as it is not actually a neighbour of Q; the first connection will be used to send an FD_REJECT, then P will ignore Q until it proposes its own view, for which it may use a new connection to contact Q.

[8]This obviously requires in addition that Q won't change the opinion of a view instance from y to x simply because it has used y and still has x waiting. Instead, Q will save x for the next agreement instance—this is a trivial local addition to the protocol.