

Generalised Repair for Overlay Networks

Barry Porter, François Taïani and Geoff Coulson

Computing Department, Lancaster University, UK

(barry.porter, francois.taiani, geoff)@comp.lancs.ac.uk

Abstract

We present and evaluate a *generic* approach to the repair of overlay networks which identifies general principles of overlay repair and embodies these as a reusable service. At the heart of our approach is an algorithm that discovers the extent of a failed section of any type of overlay, and assigns responsibility to carry out the repair. The repair strategy itself is ‘pluggable’ and can be tailored to the requirements of a specific overlay type or instance. Our approach is *efficient* in terms of the number of repair-related message exchanges it incurs; *scalable* in that it involves only nodes in the locality of the failed section of the overlay; and *resilient* in that it correctly handles cases in which multiple adjacent nodes fail simultaneously, and it tolerates new failures that occur while a repair is underway. The benefits of our approach are that: (i) it extracts and encapsulates best practice in repair for overlays; (ii) it simplifies the design and implementation of new overlays (because repair issues can be treated orthogonally to basic functionality); and (iii) it supports *tailorable* levels of dependability for overlays, including pluggable repair strategies.

Keywords Overlay networks, dependability, fault-tolerance, self-healing, distributed systems, recovery algorithms

1. Introduction

Overlay networks [1] are application-level distributed systems that are architecturally situated between the infrastructure network (e.g. the IP layer) and the end-user application. They typically offer specialised virtual network topologies (e.g. trees or rings), or application-specific services ‘which are outside the scope of the underlying network (e.g. application-level multicast or ad-hoc routing). Their use is increasingly common and the set of overlay types in use is becoming increasingly diverse [2, 3, 4, 5, 6, 7].

Most overlay networks provide some mechanism for *self-repair* so that the loss of nodes does not unduly affect the overlay’s functioning. Such repair mechanisms are essential, as overlays typically operate in hostile environments in which nodes run on unstable machines that are subject to crash or to be switched off. One well-known example of a repair mechanism is that adopted by the Chord distributed hash-table (DHT) overlay [8] which redundantly stores

data on multiple nodes, and ensures that requests for data on lost nodes are redirected to nodes holding replicants. As another example, the Overcast content-dissemination overlay [9] maintains its tree structure in the face of node loss using a strategy in which child nodes maintain ‘ancestor lists’ and attempt to locate and attach to a surviving ancestor if their current parent fails. As a third example, an implementation of a Gnutella-like resource-sharing overlay [10] might recover from super-node crashes by promoting suitable leaf nodes to super-node status.

Although these various repair strategies work, there are three main problems with this ad-hoc, per-overlay approach to repair. First, there is a lot of re-invention of the wheel, as overlay designers repeatedly solve the same fundamental problems. This is particularly true in the DHT field: many DHTs adopt a basically similar approach to that of Chord (e.g. [2, 4, 11]), but modified to fit the precise operation of the overlay (such as CAN’s n-dimensional coordinate space [11]). Second, overlays are harder to design than they otherwise would be given that the repair protocol needs to be embedded in the overlay’s functional behaviour. And third, because repair is just one of a much wider set of concerns, the repair approach adopted by overlay designers is often sub-optimal or not as flexible as might be desired. For example, one approach to repairing the failed nodes in the tree-based overlay of figure 1(a) would be to restore nodes 28, 29 and 68 on other hosts. But where it is difficult to find suitable hosts, an alternative approach, with examples shown in figure 1 (b) and (c), would be to repair the tree without actually restoring the failed nodes. As further examples, it may be useful in some installations to increase or decrease the degree of redundancy in a DHT by deciding whether or not to restore failed nodes; or, in Gnutella, to proactively migrate the leaf nodes of a failed super-node to another super-node.

We have therefore designed a *generic* approach to overlay repair which offers reusable and flexible building blocks for overlay repair. Our approach eliminates the necessity to reinvent the wheel, simplifies the design of overlays, and provides a flexible basis for the construction of *tailorable* overlay repair services. Our design has been guided by the following principles:

Separation of generic and specific aspects of overlay repair. We separate overlay repair 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 separation is motivated by the desire to support and harness diverse environments through configurability and *intelligent self-configuration*: in this case supporting alternative repair strategies that make different tradeoffs depending on their deployment environment.

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

Aggregated failure handling. Rather than restrict ourselves to treating *individual* overlay nodes as the unit of failure detection and repair, we want our approach to generalise naturally to deal with failed *sections* of overlay, as illustrated in figure 1. 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.

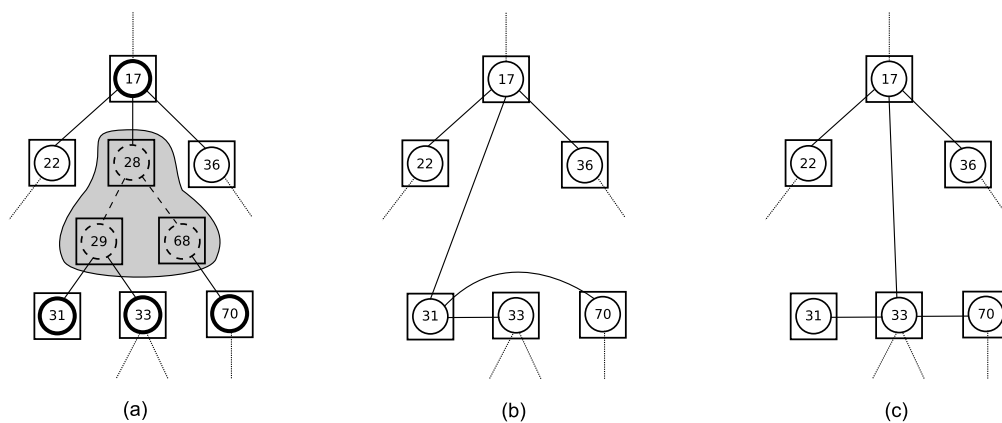


Figure 1. (a) An overlay network with a failed section (nodes 28, 29 and 68); and (b) and (c), possible repairs of this failure. Hosts are shown as squares, and overlay nodes as circles. Physical network connections are not shown.

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 sits well with our goals. 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.

We have therefore taken a fundamentally different tack: The core of our approach to generic overlay repair is a localized ‘agreement protocol’ that enables the set of nodes bordering a failed section of an overlay (i) to 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. We can then use this generic agreement protocol 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* (i.e. 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). We refer to this phenomenon, which is the major characteristic of the problem space that we address, as the ‘self-defining constituency problem’.

The remainder of the paper is structured as follows. First, section 2 describes the agreement protocol and other central algorithms in detail, and also discusses example repair strategies. Then we empirically evaluate our approach in section 3 and discuss related work in section 4. Finally, we offer concluding remarks in section 5.

2. The central algorithms

2.1. System model and assumptions

We consider an overlay 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. We assume that, provided the recipient’s underlying network address is known, 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). We do not however assume any particular timeliness properties for messaging. Also, we do not assume that overlay nodes have any global knowledge about other nodes in the overlay; instead, we assume that they are related to each other only through a ‘neighbouring’ relationship that constitutes the overlay structure.

We assume that nodes may fail at any time, and that when they do so they do not subsequently interact with the rest of the overlay (‘fail-stop’). While we allow that nodes may continue to fail as repairs are ongoing, we do assume that such failures do not occur at such a rate that our algorithm is unable to keep up with them.

In terms of infrastructure, we assume the existence of a *distributed backup service* 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 (see page 7) 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 [12]. We recognize that the state returned by a scalable backup service might not be completely up-to-date, and for simplicity we assume that overlays can cope with this situation (we note that many current overlays are explicitly designed to cope with minor inconsistencies due to the conflicting need to scale to very large numbers of nodes).

Finally, we assume the availability of per-node *failure detectors* which are used to probe the liveness/ failure status of nodes. Ideally, these would be ‘perfect’ failure detectors (i.e. any failed node is eventually detected, and non-failed nodes are never suspected of failure—no false-positives [13]) but this is not an absolute requirement. We discuss the implications of various ‘strengths’ of failure detectors in section 3.

2.2. An overview of the three-phase repair algorithm

```

1: procedure REPAIRALGORITHMp
2:   loop
3:     wait for some of p’s neighbours to fail
4:     repeat
5:       (BNodesp, FSectionp) ← CONSTRUCTFAILEDSECTIONVIEW()           ▷ PHASE1
6:       Vp[] ← AGREEONVIEW(BNodesp, FSectionp)                       ▷ PHASE2
7:       until Vp[] only contains accept                                ▷ Agreement on the failed section
8:       coordinator ← SELECTCOORDINATOR(BNodesp, FSectionp)
9:       if p = coordinator then                                       ▷ I am the coordinator
10:        REPAIRBEGIN
11:          ENACTREPAIR(BNodesp, FSectionp)                             ▷ PHASE3
12:          SEND ⟨coordinator, (BNodesp, FSectionp), repairOK⟩ TO all nodes in BNodesp
13:          ADDREPAIRTOREPAIRLOG()
14:          REJECTVIEWSCONTAININGREPAIREDNODES()
15:        REPAIREND
16:       else                                                             ▷ I am not the coordinator
17:         wait until ⟨coordinator, (BNodesp, FSectionp), repairOK⟩ received or coordinator ∈ FailedNodesp
18:       end if
19:     end loop
20: end procedure

```

Figure 2. Pseudo-code of the repair algorithm when executed by node *p*

The overall repair algorithm is presented in outline in figure 2 and illustrated in figure 3. The algorithm is executed by each node *p* in the overlay, and operates in *three* main phases. Initially, the algorithm blocks until *p*’s failure detector informs it that one of its neighbours has failed. Once a failure has been detected, *p* enters phase 1 of the algorithm by executing CONSTRUCTFAILEDSECTIONVIEW, the basic job of which is to discover *p*’s ‘view’ of the extent of the failed section (i.e. is it just the one neighbour that has failed, or are there more failures lurking behind?), and of the set

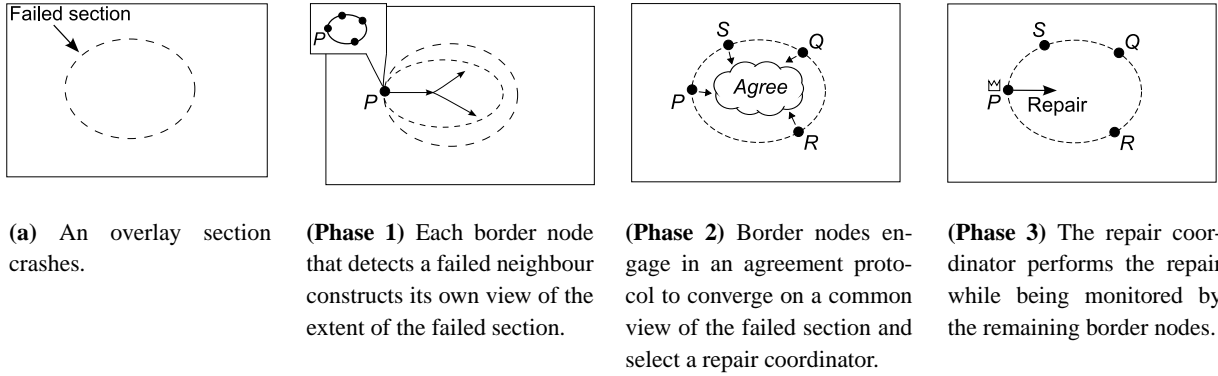


Figure 3. The three main phases of our repair algorithm

of nodes *bordering* this failed section (referred to as *border nodes* which together comprise the *border set*). These two aspects of p 's view are returned in $FSection_p$ and $BNodes_p$ respectively.

Next, node p executes phase 2 (AGREEONVIEW). This involves negotiation among border nodes to elect one distinguished border node that will take responsibility for enacting the repair (the so-called *repair coordinator*). This negotiation involves each border node exchanging its view with all the other border nodes and, eventually, all the border nodes coming to a single agreed view. The algorithm loops around phases 1 and 2 until all nodes have agreed this common view¹. Repair-strategy related information that is to be used in phase 3 may be disseminated during phase 2. This is simply achieved by piggy-backing the information on phase 2 messages.

Finally, in phase 3 a pluggable repair strategy is executed by the chosen repair coordinator. The repair is carried out transactionally² (i.e. within the REPAIRBEGIN and REPAIREND brackets), and a *repair log* is employed to help prevent nodes from 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 (which is possible outside the REPAIRBEGIN / REPAIREND bracket). If the latter happens, the repair loops back around to the very beginning.

In the remainder of this section we expand on the above outline. Due to space limitations, we do not present a proof of the algorithm's correctness (i.e. that every failed node is repaired exactly once); for this, the interested reader is referred to a complementary report [14].

¹As discussed later, the protocol caters for complications that can occur due to the fact that multiple instantiations of the protocol might be concurrently executing in the same area of the overlay.

²Standard replication techniques can be used here to temporarily protect the repair coordinator from failure until the repair is complete.

2.3. Phase 1: Discovery of the extent of the failed section

A node p enters phase 1 on detecting the failure of a neighbour, and calls `CONSTRUCTFAILEDSECTIONVIEW` to discover the extent of the failed section, which may consist of multiple nodes (see line 5 of figure 2).

`CONSTRUCTFAILEDSECTIONVIEW` requests from the distributed backup service the backed-up state of its failed neighbour, and from this extracts details of its neighbour's neighbours. It checks each of these with the failure detector to determine their status. If any are alive, they are added to p 's border node 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 detector, and this procedure continues recursively until every connection path from p 's failed neighbour terminates (transitively) in a node believed to be alive.

As the above view-construction process occurs asynchronously, *repair activity* from phase 3 of a separate execution of the repair algorithm might occur concurrently, and this might cause the resulting view to refer to overlay sections that have since been repaired. To avoid this, repair coordinators log their repairs in their own backups and the view construction algorithm checks the failed section to ensure that no nodes are contained in the repair log of any node in the failed section or border set. If this occurs, the view is abandoned and view construction re-starts. Because we assume that nodes stop failing for long enough for the algorithm to complete, a consistent view will eventually emerge through this process.

Note that even with this consistency safeguard, the view a node constructs of a failed section might not reflect the current state of the overlay, as nodes might fail at any time and the failure detector might show arbitrary latencies. In particular, p 's border set might include nodes that have only just failed. Our algorithm is designed to cater for such cases, as explained in the following sub-sections.

If a node p has *multiple* failed neighbours, it treats each of these as a part of a *separate* failed section, and constructs a view of each such failed section using the approach described above. It then determines a *ranking* of these different views and returns only the highest of these. This mechanism is important in avoiding deadlock; this subtle issue is discussed in more detail later. The ranking relationship³ is also a key factor in solving the 'self-defining constituency problem' discussed in section 1, as it gives potential constituents a common goal (of using the widest,

³The 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 failed nodes in the view (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.

live constituency), which ensures positive progress in constituency formation.

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 algorithm⁴.

2.4. Phase 2: Agreeing on a repair coordinator

In phase 2, a node p attempts to obtain an agreement from its fellow border nodes on the extent of the failed section to be repaired. Contrary to traditional consensus protocols [13], this ‘agreement’ must deal with the self-defining constituency problem. In particular, as there is no prior knowledge of who should take part in a particular agreement, multiple attempts to agree on a view of a failed section may be active concurrently, and might therefore conflict or lead to deadlock.

The agreement protocol is captured in the `AGREEONVIEW` procedure, shown in figure 4. In order to deal with concurrent agreement attempts, received messages are filtered before being passed to `AGREEONVIEW`. This is so that any views containing nodes that have already been repaired according to the local repair log are rejected (as explained below). All other messages are buffered in a message queue ready to be extracted by `AGREEONVIEW`.

In invoking `AGREEONVIEW`, a node p shows its view of a failed section ($BorderNodes_p, FailedSection_p$) to its fellow border nodes and attempts to obtain their agreement about this view. When it terminates, `AGREEONVIEW` returns a vector of values $V_p[p_k]$ ($p_k \in BorderNodes_p$) that contains the “opinions” of p ’s fellow border nodes. $V_p[p_k] = \text{accept}$ if p_k has invoked `AGREEONVIEW` with exactly the same view as the local node p ; $V_p[p_k] = \text{reject}$ if p_k disagrees with p ’s view. $V_p[p_k] = \perp$ (“bottom”) if p_k failed before giving its opinion.

The internals of `AGREEONVIEW` are inspired by the consensus algorithm for strong failure detectors presented by Chandra et al [13]; but with the difference that, contrary to Chandra’s original protocol, `AGREEONVIEW` must avoid deadlock when dealing with conflicting views, and deal with the self-defining constituency problem. The protocol is structured as a series of asynchronous ‘rounds’ in which each border node waits to receive a message from every other node in its border set before proceeding to the next round. The use of rounds 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 the rationale behind ‘rounds’ in Chandra et al’s original protocol please see [13].

Each round starts with p sending its view to all $BorderNodes_p$, and collecting responses from those nodes (lines 5-

⁴An obvious optimisation is, if p ’s border set contains only itself, to move straight to phase 3 since no agreement is needed.

```

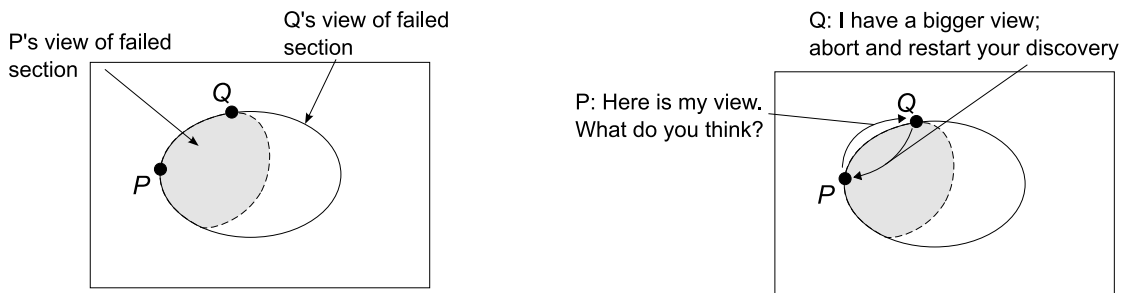
1: procedure AGREEONVIEWp(BorderNodesp, FailedSectionp)
2:   Vp[pk] ← ⊥ for all pk ≠ p
3:   Vp[p] ← accept
4:   for r ← 1 to the size of BorderNodesp - 1 do
5:     SEND ⟨r, BorderNodesp, FailedSectionp, Vp⟩ TO BorderNodesp that have not rejected my view or failed
6:     wait for a message in round r from each node I have sent to, unless they are reported failed, rejecting any
7:       lower-ranked view I receive (if all nodes I was waiting for have failed, stop waiting)

8:     for each received message do
9:       put received opinions (≠ ⊥) in Vp
10:    end for
11:  end for
12: end procedure

```

Figure 4. The pseudo-code of AGREEONVIEW when executed by node p

6 of figure 4). The conditions of the **wait** statement at line 6 ensure that a node only deals with messages that (i) agree with its view and are in the same round, or (ii) contain a lower-ranking view, for which a reject message is sent to the sender. This second condition is needed to arbitrate between conflicting failed section views that might appear in cases on ongoing node failures while view construction is under way. This is illustrated in figure 5. A reject message is simply an opinion vector with a reject tag at the sender's position.



(a) An overlay section crashes and is discovered by P. Due to ongoing failures, that failed section continues to grow. Q discovers the full extent of the failed section. P needs Q's agreement to proceed on its view, and vice versa. There is a potential deadlock as P and Q do not run the same protocol instance.

(b) P contacts Q (line 5 of figure 4) with its view. P's view is ranked lower than Q's view, so Q examines P's message at line 6 and subsequently rejects it.

Figure 5. Conflict resolution between concurrent agreement attempts

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.⁵ AGREEONVIEW itself terminates at

⁵A practical optimization between rounds is to ensure that opinions that nodes are known to already possess are not sent to them again. If a node

p when all nodes in $BorderNodes_p$ have either (i) executed AGREEONVIEW with exactly the same view, (ii) rejected this view as described earlier, or (iii) failed before being able to do either of the above. All nodes that took part in a given view will see the same values in their opinion vectors, so they can make a deterministic decision based on the same information.

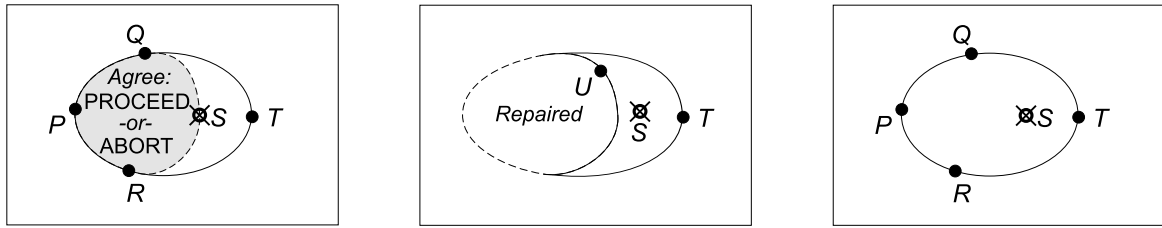
If a node p obtains an opinion vector that only contains accept tags, this means that all nodes in $BorderNodes_p$ have invoked AGREEONVIEW on the same view and thus agree with p . In this case, p uses SELECTCOORDINATOR (line 8 in figure 2) to select a coordinator. If the opinion vector contains values other than accept, the node returns to the start of the repeat-until loop on line 4 in figure 2.

SELECTCOORDINATOR deterministically returns a repair coordinator for a view passed as a parameter. Since all border nodes involved in the same instantiation of the algorithm get the same opinion vector from the agreement protocol, all nodes agreeing on a common view are guaranteed to select the same coordinator. Furthermore, if additional phase 3 related information is piggy-backed on accept tags in the view-agreement protocol, SELECTCOORDINATOR can make an optimal choice based on various dynamic criteria (e.g. communication latencies or available resources).

We now discuss how we deal with the complications introduced by the occurrence of multiple conflicting failed section view agreements. Such cases occur when the failure of one or more border nodes (i.e. cascading failures) interferes with an agreement attempt already in progress. Such a case is shown in figure 6 (a): Nodes P, Q, R and S all have the same view of the failed section they delineate and have just started a corresponding agreement protocol. S then fails, triggering the detection of a new and larger failed section by T, which includes the original failed section. T's view then *competes* with that of the surviving nodes P, Q, and R. The algorithm must ensure that only *one* of these views is agreed to ensure repair consistency (i.e., to avoid multiply-repaired nodes).

There are two possible outcomes from this scenario, shown in parts (b) and (c) of figure 6. The first outcome is possible if any of P, Q and R received the opinion "I agree" from S before S is detected as failed; if this occurs our algorithm ensures that the opinion of S is propagated to all of P, Q and R, and so their opinion vectors are not missing any information, and contain only accept tags. In this case phase 2 completes successfully, and a repair coordinator is selected as explained earlier. As long as that coordinator is not S, repair proceeds. Any view received by the coordinator from T is rejected on repair completion (line 14 of figure 2), allowing T to re-start view construction and

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. Note that for readability, our pseudo-code does not show such optimisations.



(a) P, Q, and R either agree that “S agreed” before failing, or that “S failed” before participating.

(b) If P, Q, and R agreed that “S agreed” before failing, S belongs to a *new* failed section and will be repaired separately, with border nodes U and T.

(c) If P, Q, and R agreed that “S failed” before participating, P, Q, and R restart in phase 1, including S in their failed section views and T as a border node.

Figure 6. (a) A cascading failure, in which border node S fails. Repair either proceeds anyway (b), or aborts and views converge to a larger one (c)

discovery of a (now smaller) failed section. This is illustrated in figure 6 (b).

The second outcome is possible either if no opinion is received from S before it is detected as failed by all of P, Q and R, or if S is selected as repair coordinator. In both of these cases, P, Q and R re-start the overall repair algorithm, and their views will converge with that of T. This possibility is illustrated in figure 6 (c).

2.5. Phase 3: Enacting the repair

The node that was chosen as repair coordinator performs the repair at line 11 of figure 2 by calling the procedure ENACTREPAIR. If some of the border nodes have failed after agreeing on the view (as in figure 6 (b)), their backup data is updated to reflect the changes performed by the repair operation, helping to ensure conflicting view convergence.

Once the repair has been completed, ADDREPAIRTOREPAIRLOG is called (line 13 of figure 2). Repair logs are used during view construction as explained on page 7. Similarly to pre-AGREEONVIEW message filtering, REJECTVIEWSCONTAININGREPAIREDNODES is then called (line 14) which checks the queue of buffered messages, and sends a reject message to the sender of any view message that contains nodes that have just been repaired, removing the view messages from the local node’s buffer.

We now discuss two possible implementations of the ENACTREPAIR procedure—i.e. pluggable repair strategies.

2.5.1. Strategy 1: Node restoration. ‘Node restoration’ simply replaces each failed node with a new overlay node on an alternative host, injecting the relevant backup data into each new node. Thus, the structure of the overlay does not change. Before entering phase 2, a border node using the node restoration repair strategy first locates a suitable

alternative host (i.e. one with sufficient resources) for each node in its failed section view. For host location we currently rely on a resource discovery service provided by our Gridkit middleware platform [15].

Selection is also biased by proximity to the border node itself; i.e. closer hosts will promote faster repairs and faster interaction of the border node with those restored nodes when repairs are complete. Because of this proximity bias, and the different timings of attempts to locate resources by each border node, each border node is likely to choose differently (depending on the diversity of the environment) regarding which hosts to use to restore failed nodes. To arbitrate between these different choices, each border node deterministically ‘scores’ its repair strategy, in terms of its suitability to the overlay. Scores are also adjusted depending on the resources of the border node itself such as bandwidth and processor speed—attributes that have an immediate effect on repair speed. A border node’s score is sent to the other border nodes during phase 2 by piggy-backing it on accept tags, so that by the end of the view agreement protocol each border node knows the repair score of every other border node. The border node with the best score is then selected as the repair coordinator, and enacts its suggested repair strategy in phase 3 (this simply consists of re-instantiating the failed nodes on the chosen hosts from backup copies; the bulk of the work has already been done). In case there is a tie, the border node with the highest ID is chosen from among the tied nodes.

2.5.2. Strategy 2: Structural adaptation. Unlike the node restoration strategy, ‘structural adaptation’ seeks to change the structure of the overlay network to ‘cut out’ the failed nodes, and to migrate the service(s) they were providing to surviving nodes, in an appropriate way for the specific overlay. Before entering phase 2, each border node develops a strategy of how it would perform the adaptation if it was the coordinator. We use a model of adaptation that can be generically applied to different overlays, and is equally applicable to structured overlays like Chord, loosely structured overlays like multicast trees, and unstructured overlays like Gnutella. This model is based on the observation that these overlays can all be structurally repaired by having the repair coordinator act as a ‘hub’ to restore the connectivity and service formerly provided by the failed section without restoring any of the lost nodes.

An example of this is shown in figure 1 on page 3, which depicts a multicast tree overlay. Each border node uses the same information: There are four live nodes, 17, 31, 33 and 70, that need to be connected to each other in some suitable configuration. Each border node bases its repair score on how ‘good’ a hub it would make to the others. There are therefore four distinct possibilities for post-repair structure depending on which border node acts as the hub (though only two are shown in figure 1, with nodes 31 (b) or 33 (c) acting as hubs).

Each border node scores itself according to its free resources, and optionally according to input obtained from the overlay itself (the overlay is given the opportunity to base the border node’s score on its own metrics of importance). We have designed a generic API to achieve this, though we do not have space to discuss it in this paper. Scores are again included with view agreement protocol accept messages in phase 2, and the border node with the highest-scoring repair is selected as the coordinator, enacting its suggested repair in phase 3. A similar approach can be applied to DHTs like Chord, where the node with the highest ID (therefore closest to the appropriate new position of recovered DHT data) has the highest repair score, and in Gnutella 0.6, where the least heavily loaded super-peer has the highest repair score, and will take on the leaf-peers of failed super-peers.

3. Evaluation

We now present an evaluation of our repair algorithm. We first examine (in section 3.1) the operation of the algorithm under normal operating conditions; then (in section 3.2) we additionally consider the complicating factors of cascading failure and false positives reported by the failure detector. We employ a combination of simulation and analysis. For the simulations, we used a home-grown distributed algorithm simulator: because this is tailored to overlay simulation it provides us with a higher level of abstraction than a network simulator such as ns-2 [16].

3.1. Evaluation under normal operating conditions

The four major factors in the per-repair overhead of the repair algorithm are: (i) the number of failure detection probes incurred; (ii) the number of backup accesses incurred when constructing failed section views (phase 1); (iii) the number of messages incurred to agree on a failed section and repair coordinator (phase 2); and (iv) the number of messages sent to actually enact a repair (phase 3).

To evaluate these 4 elements, we simulated the repair algorithm on a randomly-connected overlay⁶ 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 algorithm began) and all of them assume no false positives in failure detection. As mentioned, we discuss the effects of these complicating factors in section 3.2.

The results in figure 7 show measurements of the first three of the above-mentioned factors as seen by each

⁶The actual topology of the target overlay is completely immaterial to the results obtained in this section.

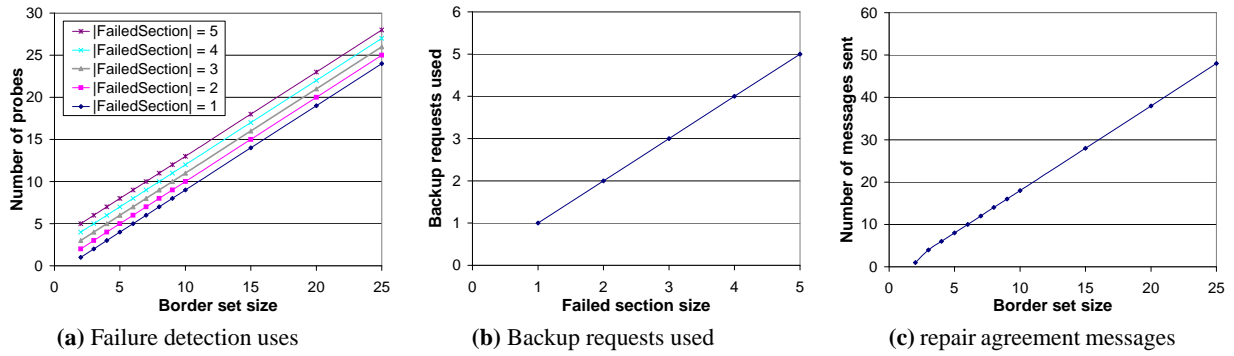


Figure 7. Per-border-node overhead during repair

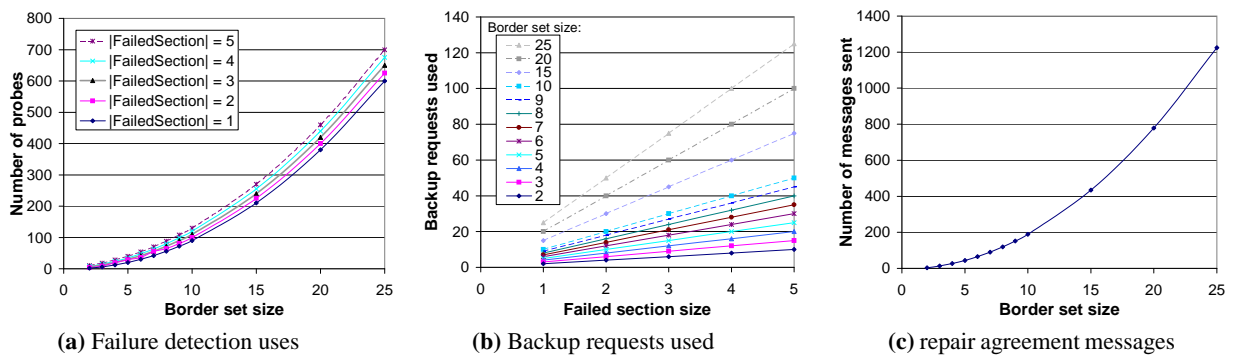


Figure 8. Combined overhead at border nodes during repair

individual border node (in parts (a), (b) and (c) of the figure, respectively). We also show the aggregated effect on the whole border set in parts (a), (b) and (c) of figure 8. In all 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. The y-axes represent the number of messages incurred. We do not present results for border sets larger than 25 because the trends are already clear by that point.

The graphs show that the number of failure detection probes and agreement messages per border node grows 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 8 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.

As can be seen, the algorithm scales nicely in terms of the load on each border node. Analytically, the number of

failure detection probes incurred by a border node can be expressed as $(b + (f - 2))$ (where b is the size of the border set and f is the number of nodes in failed section)⁷; the number of backup requests that occur is simply equal to f ; and the number of view agreement messages sent in a run can be expressed⁸ as $(b - 1) \times 2$.

Turning now to the fourth factor, i.e. repair enactment, the number of messages involved here is of course dependent on the repair strategy used—for example structural adaptation (see section 2.5.2) incurs *no* extra messages as non-coordinator nodes simply replace failed links with a link to the coordinator, and vice versa. Node restoration (see section 2.5.1) typically incurs 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 in this paper). 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.*

To put this in context, any failure in the Chord ring-based overlay would involve just *two* border nodes (yielding a message count of 7 for a failed section of size 1, and 23 for a failed section of size 5, including failure detection probe usage, backup accesses, and agreement messages). Similarly, the failed section in figure 1 would result in a border set of size 4 (yielding a repair message count of 54). In both cases the repair overhead is independent of the size of the overlay. While this is also true of Chord, it is by no means always the case. For example, a tree using a rejoin-at-root repair strategy [17] would have a repair cost proportional to the size of the tree.

3.2. Evaluation of key complicating factors

In this section we provide an analysis of the important ‘complicating factors’ of (i) cascading failures and (ii) false positives reported by the failure detector.

3.2.1. Cascading failures. As mentioned, the algorithm accommodates the failure of border nodes at any time, but assumes that ongoing failures will stop for long enough for the algorithm to be able to complete. We now examine the effects of ongoing border node failures on the algorithm to gain insight into how long a period of stability is required.

⁷The 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 algorithm’s execution, and a node does not need to check itself for failure during failed section view construction.

⁸This equation is for non-coordinator nodes; the repair coordinator must additionally send out repairOK messages to the border set on repair completion, and so incurs an additional $b - 1$ messages. Note that 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.

As there are many different points at which border nodes can fail, this is a complex and difficult issue to analyse. Our approach is to examine the *worst* possible case, which occurs when a border node fails at the very start of the algorithm without providing an opinion, as shown in figure 6 on page 11, and thus a full series of rounds must be executed to no effect.

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.

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

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 we assume 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 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 is therefore *less* with more failed border nodes within a single border set, but *more*—and *cumulative*—when successive agreement attempts are aborted due to having failed nodes in their associated border sets.

Overall, we can summarise by stating that the worst case ongoing failure scenario will result in a message count which is an order of magnitude higher than the normal case. We feel that, given the strong robustness property provided, and the rare conditions under which it is incurred, this overhead is acceptable. Furthermore, theoretical results [18] seem to indicate that such costs are unavoidable if the desired level of robustness is to be ensured. In any case, the overheads for typical failed sections are entirely acceptable. For example, in the particular worst-case scenario shown in figure 6 (c), each surviving border node incurs an additional cost of only 7 messages: $(4-1) + ((4-2) \times (4-1-1))$.

3.2.2. Failure detection inaccuracy. As with any repair approach, our repair algorithm is adversely affected by *false positives*, whereby a failure detector wrongly declares a node as having failed. There are two cases to consider.

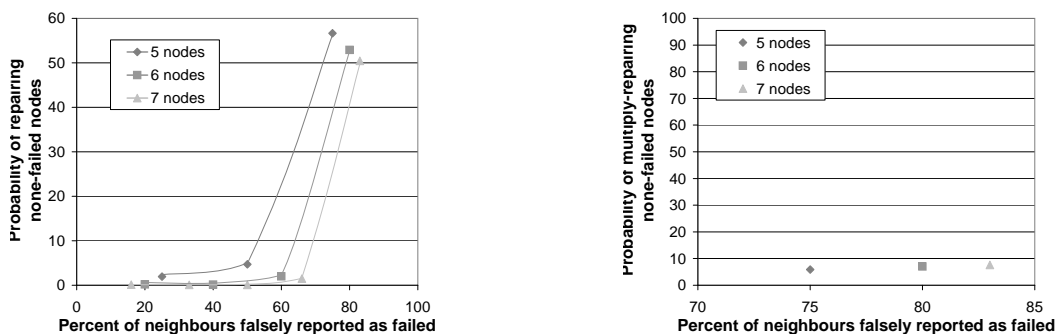
Case 1 involves a node *p* being told incorrectly that one or more of its neighbours has failed, and going on to

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 be aborted, and the only adverse effect will be p 's wasted effort. 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 algorithm will erroneously proceed to repair non-failed nodes.

Fortunately, we can straightforwardly recover from such errors by sending a signal to nodes in a failed section prior to phase 3 (not shown in our pseudo-code). Nodes that are wrongly assumed to have failed and 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 algorithm 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.

To investigate the probabilities of the above two cases occurring, we simulated simple fully-connected mesh overlays of increasing node population, reporting to each node a percentage of its neighbours as failed when they had not in fact failed. The results are shown in figure 9. Figure 9 (a) pertains to the first case (sub-case 2), with the x-axis showing increasing percentages of false positives per node, and the y-axis the probability of repairing non-failed nodes. Curves are given for three different mesh sizes. Figure 9 (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. For case 1, the results show that a very large percentage of per-node false positives is needed for there to be a significant chance of a non-failed node being repaired, and, further, that the more highly connected an overlay is, the less chance there is of this occurring. For case 2, the results show that the probability of multiple concurrent repairs of a node is very small—much smaller than that of repairing a non-failed node in case 1.



(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 result in concurrently repaired nodes, with corresponding probabilities of such repairs occurring

Figure 9. Effects of false positives on the algorithm's performance

This initial analysis is encouraging and points to interesting future research regarding the effects of false positives.

We hope to contribute more in this area in a future paper.

4. Related work

Our repair algorithm can be viewed as a combination of a kind of ad-hoc group formation and 'preference-based' leader election [19], with the important difference that the algorithm attempts to find a *stable* region of overlay network (failed section) to operate on, and also itself makes *changes* to the overlay network when making repairs.

Consensus [20, 21] and leader election [22, 19] are both well-studied fields, but current work does not address the 'self-defining constituency problem'; i.e. the inter-dependency that arises between those who are *agreeing* (the border set) and that which they are *agreeing to* (the failed section, and thus constituency of the border set itself).

Generalised failure detection and backup provision (i.e. redundancy) have also been addressed in the literature: [12] describes interesting work on providing redundancy generically in DHTs, irrespective of the structure of the DHT (such as a ring or n-dimensional coordinate space [11]), with positive results in terms of the number of messages required to maintain a given degree of redundancy; and [23] explores considerations in designing a failure detection protocol in overlay networks, examining desirable properties for general failure detection. Both of these efforts are complementary to our own work; and indeed the approach described in [12] could be almost directly plugged-in as our distributed backup service if the overlay being supported is a DHT.

Our approach might also be viewed as relating to the group communication paradigm [24]. However, our al-

gorithm fundamentally differs from group communication systems in the way that groups are formed. Rather than relying on a system wide nameserver or some other means of discovering a group to join, ‘groups’ in our case are formed locally and spontaneously as potential border nodes discover a failed section.

Our wider approach to overlay network repair, using lightweight decentralized protocols, has some similarities to self-* work such as the autonomic computing initiative [25]. While there is a lot of work in this area such as self-stabilization [26] and self-optimisation [27], we are not aware of any that concretely addresses decentralized repair of a distributed system, allowing configurability at the point of repair based on dynamic criteria.

5. Conclusion

We have proposed a novel approach to the repair of overlay networks, which is generically applicable to any overlay, and entirely decentralized (i.e. localized to the area of overlay that has failed). The approach extracts and encapsulates best practice in repair for overlays, simplifies the design and implementation of new overlays (because repair issues can be treated orthogonally to basic functionality), and supports *tailorable* levels of dependability for overlays, including pluggable repair strategies.

Despite its genericity, the repair algorithm performs well in common failure scenarios and is indeed comparable with the ‘native’ repair algorithms used by many specific overlays. For example, as shown in section 3.1, it incurs only 7 messages to repair a failed Chord node. Furthermore, our approach performs almost equally well with a single node failure or a failed section of overlay, and is robust to further failures that occur during its ongoing execution.

The described protocols have all been implemented within a component-based framework that is provided as part of our Gridkit platform [15] which has explicit support for overlay deployment. An overview of the architecture of the implementation, which includes ‘pluggable’ backup and failure detection services, is given in [28].

In future work, we are particularly planning to focus on developing and defining the precise consistency semantics required from a decentralized backup service, in order to make further guarantees in this important area. In addition, we plan to further investigate the impact of false positives in failure detection, the effects of infrastructure network partitions, and the empirical behaviour of our algorithm in a range of overlay types in the context of the above-mentioned Gridkit platform.

References

- [1] D. Doval and D. O'Mahony, "Overlay networks: A scalable alternative for p2p," *IEEE Internet Computing*, vol. 7, no. 4, pp. 79–82, 2003.
- [2] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," *LNCS*, vol. 2218, p. 329, 2001.
- [3] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "SCRIBE: A large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in communications (JSAC)*, 2002.
- [4] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," UC Berkeley, Tech. Rep. UCB/CSD-01-1141, April 2001.
- [5] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel, "ALMI: An application level multicast infrastructure," in *3rd UNIX Symposium on Internet Technologies and Systems (USITS '01)*, San Francisco, CA, USA, March 2001, pp. 49–60.
- [6] Y. Chawathe, S. McCanne, and E. A. Brewer, "RMX: Reliable multicast for heterogeneous networks," in *INFOCOM*. Tel Aviv, Israel: IEEE, mar 2000, pp. 795–804.
- [7] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," *Lecture Notes in Computer Science*, vol. 2009, p. 46, 2001.
- [8] 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*. ACM Press, 2001, pp. 149–160.
- [9] 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)*, October 2000, pp. 197–212.
- [10] URL, "<http://rfc-gnutella.sourceforge.net/developer/stable/index.html>."
- [11] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content addressable network," UC Berkeley, Berkeley, CA, Tech. Rep. TR-00-010, 2000.
- [12] 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*, Trondheim, Norway, July 2005.
- [13] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.
- [14] B. Porter, F. Taïani, and G. Coulson, "Generalizing repair for overlay networks," Lancaster University, Tech. Rep. PTC-06-01, 2006. [Online]. Available: <http://www.comp.lancs.ac.uk/computing/users/porterb/>
- [15] P. Grace, G. Coulson, G. Blair, and B. Porter, "Deep middleware for the divergent grid," in *Proceedings of Middleware 2005*, *LNCS*, vol. 3790, 2005, pp. 334–353.
- [16] 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*, vol. 33, no. 5, pp. 59–67, 2000.
- [17] M. Yang and Z. Fei, "A proactive approach to reconstructing overlay multicast trees," in *IEEE INFOCOM*, Hong Kong, March 2004.
- [18] P. Dutta and R. Guerraoui, "Fast indulgent consensus with zero degradation," in *Proc. 4th European Dependable Computing Conference (EDCC'02)*, vol. LNCS 2485. Toulouse (France): Springer-Verlag, 2002, pp. 191–208.
- [19] S. Singh and J. F. Kurose, "Electing "good" leaders," *Journal of Parallel and Distributed Computing*, vol. 21, pp. 184–201, 1994.
- [20] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.
- [21] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *J. ACM*, vol. 32, no. 4, pp. 824–840, 1985.
- [22] A. Itai and M. Rodeh, "Symmetry breaking in distributed networks," *Information and Computation*, vol. 88, no. 1, pp. 60–87, 1990.
- [23] S. Zhuang, D. Geels, I. Stoica, and R. H. Katz, "On failure detection algorithms in overlay networks," in *Proceedings of INFOCOM'05*, Miami, FL, USA, March 2005.
- [24] G. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: a comprehensive study," *ACM Computing Surveys*, vol. 33, no. 4, pp. 427–469, 2001.
- [25] A. Ganek and T. Corbi, "The dawning of the autonomic computing era," *IBM Systems Journal*, vol. 42:1, pp. 5–19, 2003.
- [26] R. W. Buskens and R. P. Bianchini, Jr., "Self-stabilizing mutual exclusion in the presence of faulty nodes," in *FTCS-25: 25th International Symposium on Fault Tolerant Computing Digest of Papers*, Pasadena, California, 1995, pp. 144–153.
- [27] J. Dowling, E. Curran, R. Cunningham, and V. Cahill, "Collaborative reinforcement learning of autonomic behaviour," in *Proceedings of the 2nd International Workshop on Self-Adaptive and Autonomic Computing Systems*, 2004, pp. 700–704.
- [28] B. Porter, G. Coulson, and D. Hughes, "Intelligent dependability services for overlay networks," in *Proceedings of Distributed Applications and Interoperable Systems 2006 (DAIS'06)*, ser. LNCS, vol. 4025. Bologna, Italy: Springer-Verlag, June 2006, pp. 199–212.