Generalised Repair for Overlay Networks

Barry Porter, François Taïani and Geoff Coulson
Computing Department
Lancaster University, Lancaster, 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.*

1. Introduction

Overlay networks [12] 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 [25, 6, 30, 9, 11].

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 [27] 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 [19] 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, a Gnutella-like resource-sharing overlay [28] 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 similar approach to that of Chord (e.g. [25, 30, 24]), but modified to fit the precise operation of the overlay (such as CAN's n-dimensional coordinate space [24]). 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 suboptimal 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 which offers reusable and flexible building blocks for overlay repair. Our approach eliminates the necessity to reinvent the

 This work was published in the proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS 2006), Leeds, UK, pp. 132–142.

wheel, simplifies the design of overlays, and provides a flexible basis for the construction of *tailorable* repair services. Our design has been guided by the following principles:

Separation of generic and specific aspects of 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 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. 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.

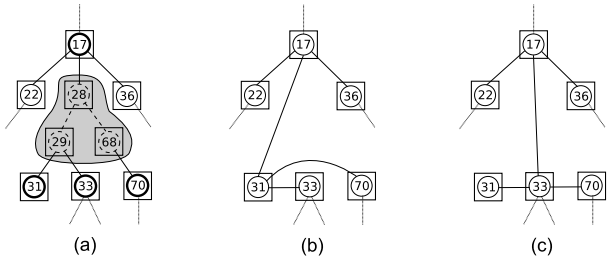


Figure 1. (a) An overlay with a failed section (nodes 28, 29 and 68); and (b) and (c), possible repairs of this failure. Hosts are shown as squares, and overlay nodes as circles. Physical network connections are not shown.

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 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. We can then use this 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* (what we call 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'[1].

In the next section we describe the agreement protocol and other central algorithms in detail, and also discuss example repair strategies. We then 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 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 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 for this). We do not however assume

[1] Formally, this adds a second parameter *voterSet* to the classical consensus primitives of *propose(value,voterSet)* and *decide(value,voterSet)* [8]. 'Self-defining constituency' refers to the fact that in addition to the traditional properties of *validity*, *agreement*, and *termination* we require that agreement be reached only if *all* non-failed members of *voterSet* execute *propose(..,voterSet)* with the *same voterSet* value.

any particular timeliness properties for messaging. We also assume that overlay nodes are related to each other only through a local 'neighbouring' relationship that constitutes the overlay structure (i.e. no global knowledge is held anywhere in the system).

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 (this is 'fail-stop', as defined in [2]). While we allow that nodes may continue to fail as repairs are ongoing, we assume that such failures do not occur at such a rate that our algorithm can't 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, repair-specific information (discussed in the following sections) and, optionally, application-specific data. For redundancy, such a 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 [16]. We recognize that the state returned by a scalable backup service may not be fully up-to-date, and for simplicity assume that overlays can cope with this (we note that many current overlays are explicitly designed to cope with minor inconsistencies due to the conflicting need to scale to 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. 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 [8]). We discuss the implications of employing less-than-perfect failure detectors in section 3.

We do not have space in this paper to consider the implementation of the distributed backup and failure detection services, and leave these as the subject of future work.

2.2. An overview of the three-phase repair algorithm

The overall repair algorithm is presented in figure 2 and illustrated in figure 3. The algorithm is executed by each node p in the overlay, and operates in 3 main phases. Initially, the algorithm blocks until p's failure detector informs it that one of its neighbours has failed. When a failure is detected, p enters phase 1 of the algorithm by executing CONSTRUCTFAILEDSECTIONVIEW, the 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 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 view[2]. As discussed below, repair-strategy related information that is to be used in phase 3 may be disseminated during phase 2. This is simply achieved by piggybacking 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 atomically[3] (i.e. within the REPAIRBEGIN and REPAIREND brackets), and a local, per-node, *repair log* is employed 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 (which is possible outside the REPAIRBEGIN / REPAIREND bracket). If the latter happens, the algorithm loops back 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 [22].

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.

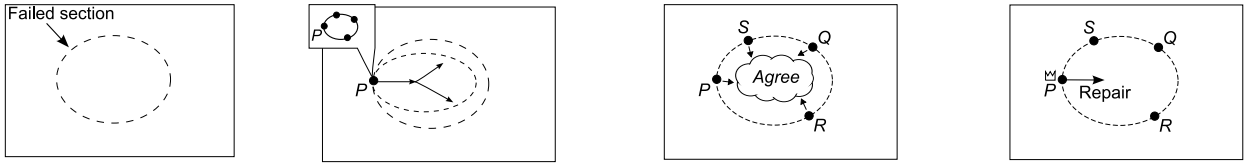
As the above view-construction process occurs asynchronously, *repair activity* from phase 3 of a separate ex-

[2] As discussed later, the protocol caters for complications that can arise when multiple instantiations of the protocol concurrently execute in the same area of the overlay.

[3] Standard replication techniques can be used here [23] to temporarily protect the coordinator from failure until the repair is complete—the important factor is to ensure repair progress is not 'lost', as discussed in [22].

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

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



(a) An overlay section crashes.

(Phase 1) Each border node that detects a failed neighbour constructs its own view of the extent of the failed section.

(Phase 2) Border nodes enter an agreement protocol to converge on a common view of the failed section and select a repair coordinator.

(Phase 3) The repair coordinator performs the repair while being monitored by the remaining border nodes.

Figure 3. The three main phases of our repair algorithm

ecution 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 a locally maintained log, and in their own backups. 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.

If a node p has *multiple* failed neighbours, it treats each one as part of a *separate* failed section, constructing a view of each failed section using the approach described above. It then determines a *ranking* of these different views and returns only the highest one. This mechanism is important in avoiding deadlock; this subtle issue is discussed in more detail later. The ranking relationship[4] is also a key factor in solving the 'self-defining constituency problem' discussed in section 1, as it gives potential constituents a common understanding of which failed section to address first—thus ensuring 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 algorithm[5].

[4]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 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.

[5]An obvious optimisation here is, if p's border set contains only itself, to move straight to phase 3 since no agreement is needed. Note that for readability, our pseudo-code does not show such optimisations.

2.4. Phase 2: Agreeing a repair coordinator

In phase 2, a node p attempts to obtain an agreement with its fellow border nodes on the extent of the failed section to be repaired. Contrary to traditional consensus protocols [8], this 'agreement' must deal with the self-defining constituency problem. In particular, as there is no prior knowledge of which nodes 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. It is inspired by the consensus algorithm for strong failure detectors presented by Chandra et al [8], but with suitable modifications to deal with the problems mentioned above.

Received messages are filtered before being passed to AGREEONVIEW, 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 $(BNodes_p, FSection_p)$ to its fellow border nodes and attempts to obtain their agreement about this view. When it terminates, AGREEONVIEW returns a vector of values $V_p[p_k]$ ($p_k \in BNodes_p$) that contains the "opinions" of p's fellow border nodes. $V_p[p_k] =$ accept if p_k has invoked AGREEONVIEW with exactly the same view as p; $V_p[p_k] =$ reject if p_k disagrees with p's view. $V_p[p_k] = \perp$ ("bottom") if p_k failed before giving its opinion.

As in [8], 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 [8].

Each round starts with p sending its view to all $BNodes_p$, and collecting responses from those nodes (lines 5-6 of figure 4). The **wait** statement at line 6 ensures that a node only deals with messages that either (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 of ongoing node failure while view construction is under way. This is illustrated in figure 5. A reject message is an opinion vector with reject at the sender's position.

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.[6] AGREEONVIEW itself terminates at p when all nodes in $BNodes_p$ 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.

```
1:  procedure AGREEONVIEW_p(BNodes_p, FSection_p)
2:      V_p[p_k] ← ⊥ for all p_k ≠ p
3:      V_p[p] ← accept
4:      for r ← 1 to the size of BNodes_p - 1 do
5:          SEND ⟨r, BNodes_p, FSection_p, V_p⟩ TO all
              BNodes_p not rejecting 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 lower-ranked view I receive (if
              all waited-on nodes have failed, stop waiting)
7:          for each received message do
8:              put received opinions (≠ ⊥) in V_p
9:          end for
10:     end for
11: end procedure
```

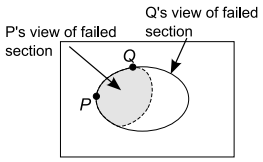
Figure 4. The pseudo-code of AGREEONVIEW when executed by node p

If a node p obtains an opinion vector that only contains accept tags, this means that all nodes in $BNodes_p$ 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 repair 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.

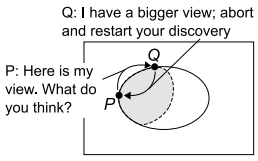
SELECTCOORDINATOR deterministically returns a repair coordinator for a view passed as a parameter. Since all border nodes involved in the same instantiation of the algorithm get the same opinion vector from the agreement protocol, all nodes agreeing on a common view are guaranteed to select the same 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 dynamic criteria (e.g. communication latencies or available resources).

We now discuss how we deal with the complications in-

[6] A 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 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) An overlay section fails and is discovered by P. Due to ongoing failures, that failed section grows further. 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. View conflict resolution

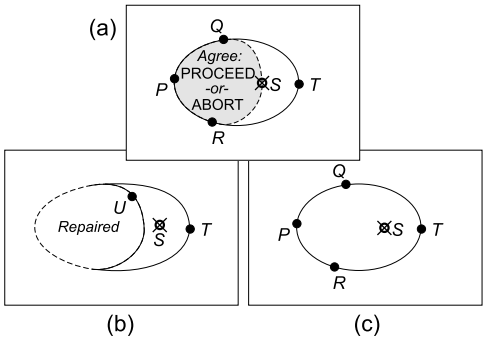


Figure 6. (a) A cascading failure, in which border node S fails. Repair either proceeds anyway (b), or aborts and views converge to a larger one (c)

troduced 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 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 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 discovery of a (now smaller) failed section, which includes the node U in its border set. 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, their backup data is updated to reflect the changes performed by the repair operation (as in figure 6 (b)), thus helping to ensure conflicting view convergence.

When the repair has completed, it is added to the local repair log (line 13 of figure 2), and, similarly to pre-AgreeOnView message filtering, the queue of buffered messages is checked (line 14), and a reject message sent to the sender of any view message containing nodes that have just been repaired (removing those 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 logical 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 (e.g. 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 [17].

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 dif-

ferently (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 logical structure of the overlay network to 'cut out' the failed nodes, and 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 were 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, 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 interconnected in a suitable way. 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 [1]: because this is tailored to overlay simulation it provides us with a higher level of abstraction than a network simulator such as ns-2 [4].

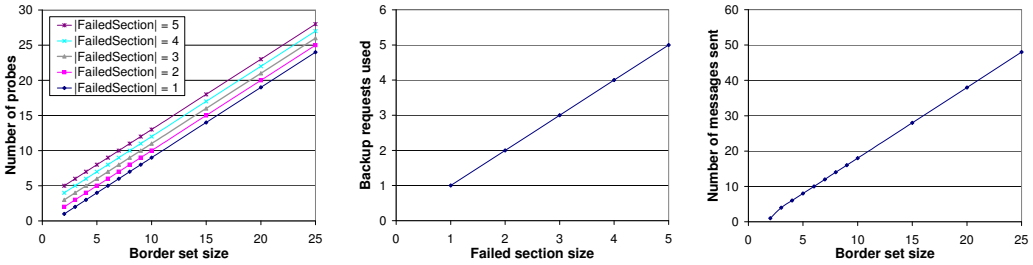
3.1. Evaluation under normal conditions

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 a repair coordinator (phase 2); and (iv) the number of messages sent to actually enact a repair (phase 3).

To evaluate these four elements, we simulated the repair algorithm on a randomly-connected overlay 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.

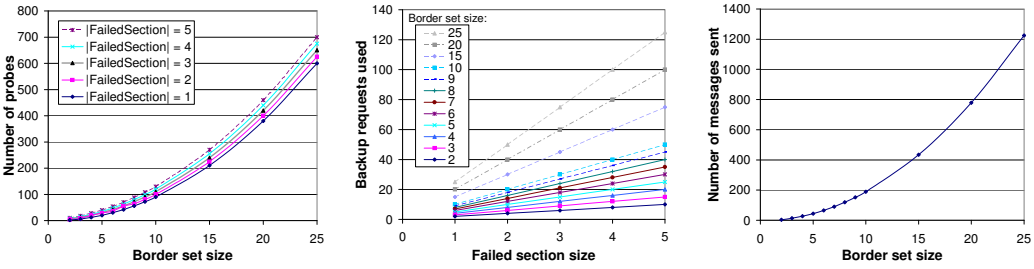
The results in figure 7 show measurements of the first three of the above-mentioned factors as seen by each individual border node (in parts (a), (b) and (c) of the figure, respectively). We also show the aggregated effect on the whole border set in parts (a), (b) and (c) of figure 8. In all these graphs, 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; and 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 this point.

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



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

Figure 7. Per-border-node overhead during repair



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

Figure 8. Combined overhead at border nodes during repair

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 expressed[8] 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 (excluding 'finger' nodes, which are refreshed continuously). This would yield a message count of 9 for a failed section of size 1, and 33 for a failed section of size 5, including failure detection probe usage, backup accesses[9], 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 66). 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 [29] would have a repair cost proportional to the size of the tree.

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

[8]This equation is 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. 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.

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

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 implications of this.

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, and 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.

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

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 *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.

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 specific conditions under which it is incurred, this overhead is acceptable. As an 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

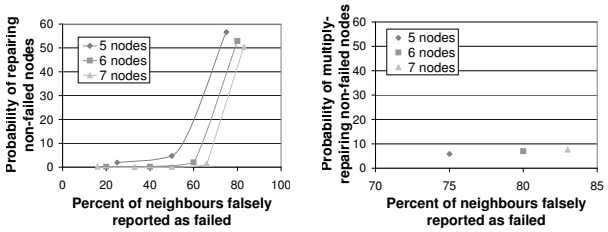
As with any repair approach, our repair algorithm is adversely affected by *false positives*, whereby a failure detector wrongly declares a node as having failed. There are two cases to consider.

Case 1 involves a node p being told incorrectly that one or more of its neighbours has failed, going on to construct a view of the supposed failed section, and proposing this view to its supposed border set. There are two possible subcases 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. 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 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 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



(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 9. Effects of false positives

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.

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

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 [26], 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 by enacting repairs.

Consensus [3, 8] and leader election [18, 26] are both well-studied fields, but current work does not address the 'self-defining constituency' problem; i.e. the interdependency 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). [7] and [2] address consensus with *unknown* and *uncertain* participants, respectively. Although these works are similar to the self-defining consistency problem we have introduced, our work is different in that we allow the existence of *competing* protocol instances (each promoting its own constituency), and arbitrate using view ranking.

Generalised failure detection and backup provision (i.e. redundancy) have also been addressed in the literature: [16] 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 [24]), with positive results in terms of the number of messages required to maintain a given degree of redundancy; and [31] explores considerations in designing a failure detection protocol in overlay networks, examining desirable properties for general failure detection. Both are complementary to our own work; and indeed the approach described in [16] could be almost directly plugged-in as our distributed backup service where the overlay being supported is a DHT.

Our approach might also be viewed as relating to the group communication paradigm [10]. However, our algorithm 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.

We note that a particular class of "gossip"-style overlays (e.g. [15]), which use loose, probabilistic and stochastic rules to self-organize, are *not* generally suitable to be supported by our approach. This approach to overlay construction and maintenance could in fact be seen as the conceptual opposite to our own, and indeed recent work has suggested a generalized gossip-like protocol to self-organize into various different topologies that exist today [20]. Such approaches provide an interesting alternative to our own.

Our wider approach to overlay network repair has some similarities to self-* work such as the autonomic computing initiative [14]. While there is a lot of work in this area such as self-stabilization [5] and self-optimisation [13], 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 9 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 [17] 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 [21].

In future work, we intend 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 overlays in the context of the above-mentioned Gridkit platform.

References

[1] AgentSpace distributed alg. prototype & evaluation tool. http://www.comp.lancs.ac.uk/computing/users/porterbf/

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

[3] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[4] 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, 2000.

[5] 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*, pages 144–153, California, 1995.

[6] 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.

[7] D. Cavin, Y. Sasson, and A. Schiper. Consensus with unknown participants or fundamental self-organization. In *Third International Conference on Ad hoc Networks and Wireless (ADHOC-NOW 2004)*, pages 135–148, Vancouver

[8] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *JACM*, 43(2):225–267, 1996.

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

[10] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[11] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46, 2001.

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

[13] J. Dowling, E. Curran, R. Cunningham, and V. Cahill. Collaborative reinforcement learning of autonomic behaviour. In *Proc. of the 2nd International Workshop on Self-Adaptive and Autonomic Computing Systems*, pages 700–704, 2004.

[14] A. Ganek and T. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42:1:5–19, 2003.

[15] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. SCAMP: Peer-to-peer lightweight membership service for large-scale group communication. In *Proc. of the 3rd International workshop on Networked Group Communication*, 2001.

[16] 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.

[17] P. Grace, G. Coulson, G. Blair, and B. Porter. Deep middleware for the divergent grid. In *Proc. of Middleware 2005, LNCS*, volume 3790, pages 334–353

[18] A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1):60–87, 1990.

[19] 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 *Proc. of the Fourth Symposium on Operating System Design and Implementation (OSDI)*, pages 197–212, October 2000.

[20] A. Montresor, M. Jelasity, and O. Babaoglu. Chord on demand. In *Proc. of IEEE P2P 2005*, pages 87–94

[21] B. Porter, G. Coulson, and D. Hughes. Intelligent dependability services for overlay networks. In *Proc. of Distributed Applications and Interoperable Systems 2006 (DAIS'06)*, volume 4025 of *LNCS*, pages 199–212, Bologna, Italy

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

[23] D. Powell, editor. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, 1991.

[24] 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.

[25] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *LNCS*, 2218:329, 2001.

[26] S. Singh and J. F. Kurose. Electing "good" leaders. *Journal of Parallel and Distributed Computing*, 21:184–201, 1994.

[27] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press

[28] B. Yang and H. Garcia-Molina. Designing a super-peer network. In *Proc. of the 19th International Conference on Data Engineering*, Bangalore, India, March 2003.

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

[30] 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.

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