

# Intelligent Dependability Services for Overlay Networks

Barry Porter, Geoff Coulson, and Daniel Hughes

Computing Department, Lancaster University, Lancaster, UK  
{barry.porter, geoff, hughesdr}@comp.lancs.ac.uk

**Abstract.** Application-level overlays have emerged as a useful means of offering network services that are not supported by the underlying physical network. Most overlays employ proprietary dependability mechanisms to render them more resilient to node failure; but the use of proprietary approaches leads to duplication of effort during development and adds design complexity. In this paper we propose *generic* dependability services which simplify the design of overlays. Our services are fully decentralized and are configurable to take advantage of current network conditions, which can enable us to make better repairs following failures.

## 1 Introduction

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

As overlays become more widely deployed, their *dependability* becomes an ever more critical issue. In current practice, every overlay implements its own dependability mechanisms. For example, Chord [7] employs backup links and data replication, and Overcast [8] uses a specialised tree-repair strategy, both of which are intended to provide resilience in a single environment consisting of end-user hosts. In our present work, we seek to add dependability to overlay networks as a *generic service* which can operate in multiple environments including that of ‘overlay deployment environments’, which are becoming more common [9, 10, 11]. Our primary goal is to address fault tolerance issues in a systematic and re-usable manner, and thus to both simplify and enhance the design and deployment of dependable overlays. An associated, more general goal is to explore autonomic dependability in large scale overlay-based distributed systems, where self-configuring services can make intelligent decisions at runtime.

This paper presents an approach to providing generic dependability services to overlay networks, which can benefit from multiple configurations to intelligently provide redundancy and enact repairs in an overlay, taking into account the current environmental conditions in which the overlay is operating. In doing so,

we argue that our services can improve the quality of repairs made to an overlay following failures, and present preliminary results in support of this.

The rest of this paper is organized as follows: In section 2 we examine several overlays and their proprietary fault-tolerance mechanisms. In section 3 we introduce our general approach to building dependability services, and in section 4 we present our initial results from evaluating our approach in an overlay deployment environment. Finally, section 5 discusses related work, and section 6 presents concluding remarks and aspirations for future work.

## 2 Analysis of Dependability in Overlay Networks

Our general observation is that while most overlays provide resilience in the face of node failure, the mechanisms used are targeted at a single environment, where each overlay node resides on an end-user host in an Internet-like environment. As a result, the approach taken by many overlays to repair themselves is to remove the failed nodes from the overlay and to attempt to continue to provide the same service without them. This causes a cumulative degradation of the overlay's functioning over repeated failures, as more nodes are lost. We also observe that the physical capabilities of an overlay's hosting nodes are typically not taken into account, which can further tend to degrade the functioning of the overlay. A final observation is that as overlays become more widely deployed and used in more demanding application areas, dependability becomes an increasingly pressing concern. For example, overlay dependability is crucial in Grid environments due to the large volumes of data that are typically handled [12].

To demonstrate that these limitations pervade a wide range of overlay types we now look in more detail at DHT overlays (in section 2.1), content dissemination overlays (in section 2.2) and flooding overlays (in section 2.3).

### 2.1 DHT Overlays

The general purpose of this class of overlay is to provide an efficient and scalable "key-based routing" facility in which a message can be routed in  $O(\log N)$  hops (where  $N$  is the total number of nodes in the overlay) to a target node that is designated by a given key. In Chord [7], for example, all the nodes are organised as a logical ring. Chord nodes each maintain a so-called *finger list*—a list of increasingly distant nodes around the ring. This is used for  $O(\log N)$  routing towards a target node. Chord nodes also store the IDs of their immediate "successor" and "predecessor" nodes in the ring so they can still make  $O(N)$  progress at times when the finger list is incomplete. One use of Chord is as a distributed data repository. In such an application, a data item which is submitted for storage in the repository is stored at the node whose ID is closest to a hash of the data. Pastry [1] works in a similar way to Chord, as does Tapestry [3], although the latter is organised as a mesh rather than a ring. Another popular DHT overlay, CAN [13], is organised such that nodes have zones of responsibility in a distributed coordinate space.

Despite their differences, all the above-mentioned overlays have a similar approach to dependability. In particular, when used as a data repository, they increase availability of the data by replicating data items on the  $n$  nodes whose ID is “closest” to the hash of a stored piece of data. The response to a node failure is to update the links in the routing tables of the affected nodes to reflect the change, and also to restore the number of replicants of data items stored at the failed node by copying them to further nodes.

The general disadvantage of this approach is that the self-repair algorithm permanently increases the load on the surviving nodes and reduces the total amount of redundancy in the overlay, as the same volume of data is redundantly stored at less hosts. We also observe that the physical resources of a node are generally not taken into account in DHT overlays; a node is given an ID and is expected to be able to store all data hashed to that ID. This expectation may be not be workable in a highly heterogeneous system that includes a significant number of poorly-resourced hosts.

## 2.2 Content Dissemination Overlays

Content dissemination overlays [14, 15, 4] deliver streaming content to multiple users in a scalable manner. They are typically organised as a tree with the sender at the root. Each non-root node receives data from its parent, and forwards it to each of its children using a point-to-point link.

TBCP [14] is a good representative of this class of overlay. TBCP builds a single rooted tree, and new nodes join the tree by first contacting the root node on a published or well-known address. The root node decides if it wants to accommodate the joining node as one of its direct children; if not, it forwards the join request to its most suitable child. This process recurses until the joining node finds a place in the tree. For performance reasons, TBCP attempts to build a tree that reflects the structure of the underlying IP network—i.e. the nodes contained in each sub-tree should tend to share IP-level locality. Decisions about whether to accept a node as a direct child or to pass it on are made on this basis. Another approach to maintaining a close correspondence between an overlay multicast tree and the underlying IP topology is to employ a network metric such as round trip time between nodes [4].

In terms of dependability, if a node fails, the default response in many such overlays is for all the nodes below the failure point to re-join via the root of the tree. The drawback of this is that the resultant bottleneck can cause traffic to be significantly disrupted during the (possibly extensive) re-building phase. One possible optimisation is to have each node record a “backup parent” [15]; but this also has complications: if the child of a failed node re-locates to a backup parent, it brings the entire sub-tree below it, which can result in a poorly balanced tree. Another approach is to simply assign the grandparent as the backup parent [8]. This keeps the tree balanced, but can increase the out-degree of the new parent, which may in the future place additional strain on that node potentially beyond its capacity [15]. The key points are that in each approach there is additional

stress on some parts of the rest of the tree as a result of the failure, and that this is *cumulative* over multiple repairs.

In addition, when attaching a new node to an overlay, current schemes tend not to take into account the characteristics (e.g. in terms of processing power and link speed) of the underlying host machine: if a tree is built on top of hosts with greatly differing capabilities, it may not perform with adequate quality of service (QoS). In an extreme case, for example, if a low power PDA, or a PC with a dialup connection, is given many children it would have difficulty sending data out to all of these at a sufficient rate.

### 2.3 Flooding Overlays

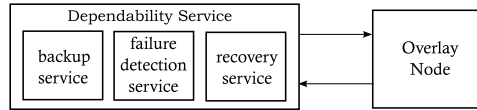
This simple type of overlay is typically used to locate and acquire resources in a distributed environment. Messages are flooded to a (subset of a) node's neighbours, and the neighbours pass these messages onto *their* neighbours etc. This continues until either the target resource is located, the edge of the overlay is reached, or a maximum hop-count that messages are permitted to travel is reached. Unlike DHTs, queries in flooding overlays typically only reach a subset of the overlay's nodes (termed the "search horizon"), which means that there are no guarantees about locating resources that exist in the overlay.

Early flooding overlays such as Gnutella v0.4 [16] made no provision at all for dependability because it was assumed that resources would naturally be replicated over multiple nodes, and that flooding would likely locate a suitable copy. Version 0.6 of Gnutella, however, adds the notion of "super-peers" to the architecture in an attempt to enhance scalability. In v0.6, end-user computers are viewed as "leaf-peers" that do not directly engage in flooding. Instead, each leaf-peer attaches to a super-peer which manages a number of leaf-peers and maintains a list of resources held by these. This architecture reduces the number of nodes that engage in flooding and therefore increases the search horizon. As a side effect, however, it impacts dependability [17] in that the failure of a super-peer requires the leaf-peers it was supporting to locate another active super-peer. Furthermore, as the number of super-peers drops, the load on the remaining ones clearly increases, which tends toward the emergence of bottlenecks in the overlay.

## 3 Approach

### 3.1 Overview

Architecturally, we use a "Dependability Service" component, which can load and configure sub-services which address an area of dependability. An instance of the dependability service component and its sub-services resides alongside each overlay node, as shown in figure 1. Each service internally uses only 'soft state' (i.e. state that can be re-built from instantiation simply by existing in the environment), so that services are inherently self-repairing. Currently, our design uses three major sub-services: i) a *backup* service, ii) a *failure detection* service and iii) a *recovery* service.



**Fig. 1.** The dependability services, horizontally composed with an overlay node

Before discussing each of these services, we first introduce the concept of `accessinfo` and `nodestate` records. In order to externally manage an overlay, we first elected to create a specification of what constitutes an overlay. Rather than providing fully transparent services, we are interested in taking the application-specific needs of each overlay into account, and in order to do this we needed to create a certain ‘model’ to which an overlay must conform.

Our overlay model has two basic ‘types’: `Accessinfo` records and `nodestate` records. An `accessinfo` record represents an overlay node’s ID, or a neighbour link to another node in the overlay. Each overlay node will therefore have one `accessinfo` record giving its logical node ID, and a collection of records providing its list of neighbours. `Accessinfos` are expected to have ‘context’ (such as ‘NodeID’, ‘child’ or ‘successor’) included in them, though the internal structure of `accessinfo` records is *unknown to the services*, and entirely the decision of the overlay. They are named as such because they represent not only a way to store the structure of the overlay on a per-node basis, but they also allow service instances on neighbouring nodes to communicate with each other by passing to their overlay node an `accessinfo` record and a message to deliver to the service at the target node, thus exposing the structure of the overlay to the services.

`Nodestate` records represent any other state that is required by the overlay to be persistent across failures, such as resources in Chord or resource indices at Gnutella super-peers<sup>1</sup>. Again, their constitution is the choice of the overlay; our services simply know that they represent ‘extra state’ of a node that is not directly related to the overlay’s core structure. We use both of these typed, ‘black-box’ objects to generalize overlays sufficiently for what we need to achieve, while still allowing significant room for specialization by overlays of their needs by filling in the black boxes in ways uniquely appropriate to them.

Our general approach to dependability centres on *decentralization*, *configurability* and *intelligent self-configuration*. We now discuss the three services mentioned above, outlining available configuration options in each.

### 3.2 Backup Service

The backup service is used to redundantly store `accessinfo` and `nodestate` records belonging to an overlay node in case that node fails. This data is stored on one or more appropriate backup hosts in the overlay (other than the host of

<sup>1</sup> Note overlays are not required to have `nodestate`; some overlays like simple multicast trees have only structural data, which is captured in full by `accessinfo` records.

the origin overlay node), and constitutes everything about a node that is needed to make repairs should it fail.

Our current implementation uses a simple ‘push’ variant where overlay nodes notify the backup service when neighbours or other overlay state is added or removed at that node. The local backup service at a node then transmits the collection of `accessinfo` and `nodestate` records to other nodes. In terms of configurability, the backup service can store more or less complete backup copies of each node on different hosts, providing a simple way to increase or decrease the amount of redundancy in the overlay.

In future work, we intend to make the backup service self-configuring, so that it can increase or reduce the amount of redundancy in the overlay according to observed regional stability of networks and hosts. Additionally, backups should ideally be stored at the most stable hosts in the overlay with the most free resources. We also seek to take advantage of the way our overlay model allows individual handling of `accessinfo` and `nodestate` records, such that we only alter existing backups to add new ‘fragments’ of data about a node instead of re-saving the full collection of these each time a change is made.

### 3.3 Failure Detection Service

The failure detection service is used to detect node failures in a decentralised manner. The service is currently implemented in the form of an overlay that is used to monitor the nodes of one or more “target” overlays that require dependability. Having detected a failed node, an instance of the failure detection service informs the recovery service instance(s) on the neighbour(s) of that failed node. A number of overlay types could be used for failure detection, with various protocols, but our current implementation makes use of gossip protocols [18].

Because distributed failure detection is already a well-researched area in its own right, we do not pursue this aspect of the dependability service in this paper.

### 3.4 Recovery Service

The general behaviour of a recovery service component is, on learning of the failure of a neighbouring node from the failure detection service, to create a strategy to repair the overlay. We currently use two different methods to achieve this; i) recovering failed nodes on alternative hosts, and ii) adapting the overlay to perform the same duties without the failed nodes.

When recovering failed nodes on alternative hosts, the service first discovers suitable hosts (i.e. with sufficient free resources), then instantiates new overlay nodes on those hosts, and injects backed up data into each, essentially re-creating each failed node. At nodes that neighboured these failed nodes, `accessinfo` records are manipulated by the service so that they point to the newly restored node(s). This is the most transparent method of repair, as the overlay structure does not change, but clearly requires the availability of suitable alternative hosts and the ability to locate them. We return to this issue in section 4.1.

When adapting the structure of the overlay to operate without the failed nodes, the service needs to know how to perform the adaptation; that is, what the structural and behavioural rules of the overlay are. We can use various levels of interaction with the overlay in order to acquire such information when a repair needs to be made, but we present the most generic approach here which requires least interaction with the overlay. It uses the observation that several types of overlay can be structurally repaired using the same procedure, by adding all outward-pointing ‘perimeter’ connections from the failed section of overlay (i.e. neighbour links from failed nodes to neighbouring live ones) to one selected live neighbour of the failed section, and adjusting connections at all other live neighbours of the failed section to point to that same selected neighbour. Any recovered `nodestate` is also inserted into the selected node.

Using either of the above repair ‘styles’, or a combination of both, it is necessary to select a *coordinator* to actually carry out the repair. To do this, the nodes neighbouring (or ‘bordering’) a failed node or failed section of overlay discover each other by locating the backup of a failed neighbour, extracting its neighbours, and testing each for failure, then recursing the procedure, until each link terminates (transitively) in a node reported to be alive. These nodes then communicate with each other, using an agreement algorithm to select one of them as the repair coordinator. We do not have space to present the algorithm in detail here, but interested readers are referred to [19]. Briefly, its properties include that only the nodes bordering a failed section (or single failed node) of overlay are involved in its repair, limiting the effort of repair to the affected area of overlay. This is because recovery service instances are initially only concerned with the failure of their direct neighbours, expanding their area of concern as they discover additional connected failed nodes. The algorithm is resilient to further node failures while repairs are taking place, and is also able to select the repair coordinator based on dynamically-acquired data at the time of the failure, such as free resources on each border node and their network latencies.

## 4 Evaluation

### 4.1 Gridkit: Overlays as Part of Middleware Services

The environment on which we focus for our evaluation is an overlay deployment environment called Gridkit [10], which is a middleware service supporting communication-based Grid systems in diverse networks. Gridkit and its overlay networks are constructed from software *components*, and overlay networks are used as a substrate for ‘interaction types’ requested by an application (e.g. multicast, publish-subscribe), operating in a heterogeneous Grid-like environment. The architecture of overlay nodes is specified into a *control*, *state* and *forwarder* component, allowing overlays to be composed in a ‘stack’ to provide advanced services (a typical example is using Scribe [2] atop Pastry for publish-subscribe style communication), where messages travel up and down the stack (e.g. between forwarder components). More details on Gridkit are available in the literature [10].

For our dependability services, we are particularly interested in two aspects of the environment that Gridkit operates in; its *heterogeneity*, where hosts of massively variable capabilities are connected together in a suitable overlay, and its *altruistic* nature. This latter aspect can be harnessed by Gridkit’s *resource discovery* framework, a service capable of discovering Gridkit-enabled hosts in the network with specified types and levels of resources.

When our recovery service is used in such an environment, it can *switch* between the structural adaptation and node restoration repair styles as available Gridkit hosts and resources dictate. This is a powerful ability, meaning that the overlay may not need to ‘degrade’ at all following a failure, as failed nodes can be restored on alternative Gridkit hosts, maintaining (if possible) a constant node (and host) population through failures. If suitable hosts are *not* available, the recovery service can simply employ structural adaptation to repair the overlay as normal.

We now present results from a Gridkit-like environment which show how this kind of intelligent, configurable repair can be beneficial in practice. We employ the following criteria in evaluating the dependability service:

- *ongoing memory use* refers to the average ongoing memory load on the hosts used by the overlay;
- *average request handling load* refers to the average number of user requests handled by a host per second;
- *average recovery time* refers to the time taken to recover failed node from the time of detection of the failure;
- *messaging overhead* refers to the total amount of maintenance-related overlay traffic per unit time.

In the following, we first, in section 4.2 present a detailed quantitative analysis of the dependability service in comparison to the proprietary dependability mechanisms supported by the Chord DHT. Then, in section 4.3, we offer a more general qualitative analysis.

## 4.2 Comparison with Chord’s Dependability Mechanisms

To perform this evaluation, we developed Java software that emulates a set of hosts as operating system processes and inter-host links as IPC calls. We then ported the backup and recovery services (using only the node restoration repair style) to this environment. Failures are simulated in terms of a script and notified to overlay nodes running on the simulated hosts as if by the failure detection service. On top of this, we developed two Chord implementations: one is standard Chord [7], and the other is a modified Chord that replaces Chord’s proprietary dependability mechanisms with our dependability APIs.

Our experimental Chord configurations employed a successor list size of 2, and an identifier space size of 8. We used ring sizes of 12 nodes, but included 17 ‘Gridkit’ hosts, each of which was capable of hosting an arbitrary number of nodes; initially 12 hosts supported a single node each, and the others were

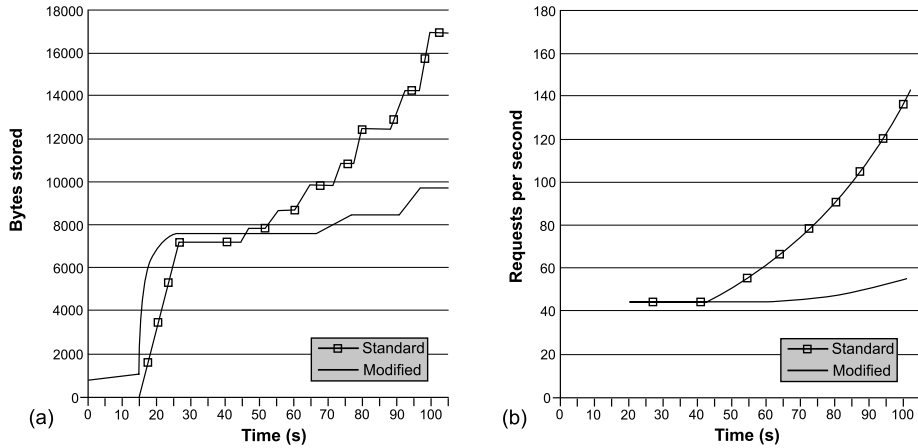
idle<sup>2</sup>. The rings were used to store a set of 60 different data files. Each of these, which were of identical size, was hashed to a key and stored on the node closest to that key. Although the assignment of files to nodes was ‘random’, the same assignment was used for each experimental run. Each node duplicated its state on one additional node—in the standard Chord case, through replication; in the modified Chord case, via the backup service. In each run of the experiment, we observed the effects of failing 7 hosts, one every 10 seconds. This resulted in the standard Chord version being left with 5 nodes at the end of the run, and the modified version being left with 12 (as the 7 failed nodes were recovered by the dependability service). The file data was injected into the ring at time  $T+20$ , and the first failure occurred at  $T+42$ . All the results presented below are averaged over all hosts used by the overlay in question (n.b. obviously by the end of the standard Chord runs there were only 5 hosts in use; whereas there were 10 by the end of the modified Chord runs).

In terms of *ongoing memory use*, figure 2 (a) shows the average memory load on the overlay’s hosts. It can be seen that, in the case of standard Chord, the load increases steadily from the time of the first host failure (at time  $T+42$ ). However, in the case of modified Chord, because the total load is spread over more available hosts, the average load is much smaller. Note that the slight increases at times 68 and 91 are due to the fact that a host is from that point supporting 2 nodes (as mentioned, there are 12 nodes but only 10 hosts in use at the end of the run). Note also that between times 26 and 42 (which is a failure free period) modified Chord consumes slightly *more* memory than standard Chord. This is due to the overhead of storing backups in a generic fashion. In conclusion, this experiment confirms that modified Chord in the face of node failure can spread the load over a wider range of hosts than standard Chord and thus reduces host memory overload and consequent service degradation.

We evaluated *average request handling load* by counting the number of requests arriving in each second at each host. A single designated node generated requests for a random selection of 15 of the 60 files stored in the ring at a rate of 15 requests per second. Figure 2 (b) shows that the average request handling load is similar for the two cases until  $T+42$ , when the first host failure occurs. From this point onward, in the case of standard Chord, a constant number of requests is being handled by a shrinking number of hosts—therefore the request handling load steadily climbs. In the case of modified Chord, however, the number of hosts stays around the same so that the request handling load is roughly constant (again the slight increase is due to two hosts supporting two nodes). In conclusion, this experiment confirms that our approach can maintain the request handling patterns of the original ring topology across node failures, and therefore reduce bottlenecks. Note, incidentally, that request handling load does *not* translate directly to average request latency, because network latency must also be taken into account in this. In fact, standard Chord will tend to a *lower* average network latency as failures occur, simply because there are fewer nodes

---

<sup>2</sup> The nature of our criteria is independent of the size of the Chord ring involved, so there is no loss of generality in using such a “small” ring.



**Fig. 2.** Average memory load (a) and requests per second (b) at hosts used by the overlay

left in the ring. However, our approach will increasingly “win” as the ring becomes more loaded with data—as this happens, the per-host request handling load will progressively overshadow the effects of network latency.

We evaluated *average recovery time* in modified Chord simply by measuring the average latency between failure detection and recovery completion. This was measured as 179ms. In standard Chord, of course, nodes are not recovered: instead the predecessor of the failed node simply re-designates the failed node’s successor as its successor; therefore recovery time is negligible. Thus the standard Chord time is close to 0ms. Essentially, we are paying “up front” for later payoffs in terms of improved memory use and reduced request handling load. This tradeoff is increasingly in our favour as more failures occur—recovery time in our approach is constant (and quite small) for each failure, while the degradation caused by compensating for the loss of a node in standard Chord is cumulative as failures increase.

Finally, we measured overlay *messaging overhead* in terms of the numbers of overlay maintenance-related messages. More specifically, we totalled the byte count of these per second and divided by the number of hosts involved. The results are shown in figure 3. It can be seen that there is a small start-up cost incurred by modified Chord between time 0 and the time of the data injection (at  $T+20$ ). This is due to backups being taken as the ring is built; it tails off as the ring stabilises. Following data injection, both cases suffer a spike; this is larger in the case of modified Chord, again due to the overhead of creating generic backups. Subsequently, however, modified Chord fares slightly better than standard Chord—except transiently when failures occur. The reason for the higher ambient overhead of standard Chord is the need to continuously maintain the successor list; modified Chord does not have this requirement, although it does need to

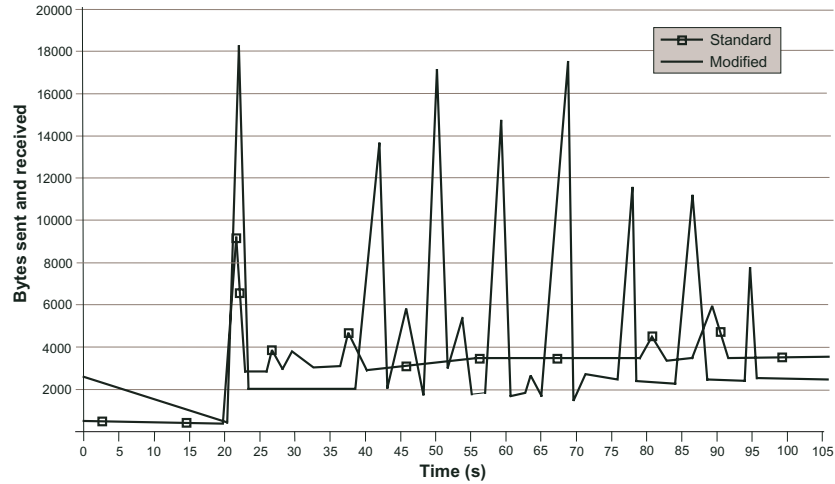


Fig. 3. Average number of overlay maintenance-related bytes per second per host

maintain the Chord finger table as this is an inherent part of the operation of the overlay. In conclusion, the recovery “spikes” are traded off for a generally improved level of service in terms of messaging overhead. As above, the overhead is transient while the adverse effects of *not* restoring nodes is permanent and cumulative.

To summarise, repeated failures in Chord using its proprietary dependability approach lead to reduced overall redundancy in the overlay, and more stress being put on other nodes in the long term to compensate for the failures. By restoring failed nodes in Gridkit where possible, we maintain the structure and integrity of the overlay across failures, and the benefit of this grows progressively with the amount of data stored in the ring and number of node failures.

### 4.3 Qualitative Evaluation

We now extrapolate from the DHT-specific arguments above to the other overlay classes mentioned in section 2.

First, consider content dissemination overlays such as TBCP. In such overlays, the ongoing memory use and request handling load criteria are not applicable as the purpose of the overlay is simply to forward “live” data. The main difference relates to the average recovery time criterion. There are basically two cases depending on the tree repair strategy used. First, consider the “rejoin at root” strategy (see section 2.2). Here, the node restoration approach avoids two pathologies: i) overloading of the (bottleneck) root node; and ii) long outages in cases where the failed node has many descendents. Second, consider the “backup parent” strategy. Here, our approach can avoid the pathology of structural degradation. In particular, it avoids stressing the backup parent which might have to deal

with a larger number of children than it is equipped for; and it avoids situations in which the tree may become unbalanced if a large subtree is moved from one branch to another. These pathologies can be avoided by recovering the failed node on an alternative host and re-integrating it into the overlay in the same logical position, removing the need for any re-configuration of the tree. There is a trade-off involved in all of these situations. For simple cases (e.g. where the failed node has few descendents), the proprietary methods may be faster; the benefit of restoring nodes becomes particularly apparent in large trees.

Second, consider flooding overlays such as Gnutella v0.6. In terms of ongoing memory use, request handling load, and maintenance-related messaging overhead, the benefits are similar to those seen in the Chord case—as super-peers fail, rather than their memory use and request handling load burdening other surviving super-peers, the failed nodes are simply restored on alternate hosts when possible. The corresponding drawbacks are also similar to the Chord case. The main difference between the Chord and Gnutella cases is in terms of recovery time (for super-peers). In standard Gnutella v0.6, leaf nodes that were attached to a failed super-peer must locate an alternative super-peer, which may in some cases require the intervention of the user. In a dependability service enhanced Gnutella, however, leaf nodes would be automatically informed of the location of the recovered super-peer.

## 5 Related Work

Classic work on fault-tolerance services for distributed applications has focused on management ‘frameworks’ [20, 21]; these often have hierarchical arrangements with various dedicated ‘managers’ (usually replicated for fault-tolerance) to recover from failures. In contrast, we seek to develop entirely *decentralized* services which are horizontally composed with the application, affording us scalability and enhanced service resilience, and removing reliance on administrated infrastructure to host our services.

The Resilient Overlay Network project [22] highlights the usefulness of overlays to improve levels of service beyond those of the physical network, but RON is aimed at providing dependable communications over the Internet using an overlay and does not address the failure of overlay nodes themselves. While our approach can also be used to provide dependable communications by introducing dependability to a target overlay network, it is more general and focuses not only on the overlay surviving, but also on any data in the overlay being persistent.

There are some overlays such as Narada [23], and some simple flooding overlays (e.g. Gnutella v0.4), that employ dependability mechanisms which do not degrade over multiple recovery operations, and which do take account of host resources. However, the number of such overlays is sufficiently small that our approach is still very widely applicable; moreover the overlay-specific dependability techniques of these overlays are generally not suitable for overlays of different types. We are not aware of any work except ours that is aimed specifically at making overlays themselves dependable in a generic way.

Finally, our work is related to general trends in autonomic computing research [24] in that it is decentralized, using relatively lightweight components distributed throughout an overlay to monitor and manage it, and our services are self-configuring.

## 6 Conclusions

The heart of our proposal is to offer dependability to overlays in the form of generic services, which intelligently configure as appropriate to an overlay's environment. We have presented an example using an overlay deployment environment, where intelligent selection of a repair strategy can improve the performance of an overlay following multiple failures. The generalization and extension of existing overlay dependability mechanisms as external services allows commonly applicable standards of fault-tolerance across a wide range of overlays, and we have shown that the price of such genericity is not prohibitively high.

We currently have implemented both node restoration and structural adaptation repair styles in our recovery service, and have basic implementations of our failure detection and backup services.

In our future work, we intend to bring similar intelligence to our backup service, taking advantage of our overlay model to store only changes to an overlay node's `accessinfo` or `nodestate` records, and storing backups at the most suitable (i.e. highly resourced and stable) nodes, as well as varying the amount of redundancy used depending on the relative stability of the overlay.

We are also interested in helping to deal with network heterogeneity; as we discussed in section 2, many of today's overlays are not good at distributing load according to the resources of their members' hosts, and we believe that an additional service can address this issue by re-distributing load appropriately.

## References

1. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science* **2218** (2001) 329
2. Castro, M., Druschel, P., Kermarrec, A.M., Rowstron, A.: SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)* (2002)
3. Zhao, B.Y., Kubiawicz, J.D., Joseph, A.D.: Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley (2001)
4. Pendarakis, D., Shi, S., Verma, D., Waldvogel, M.: ALMI: An application level multicast infrastructure. In: 3rd USNIX Symposium on Internet Technologies and Systems (USITS '01), San Francisco, CA, USA (2001) 49–60
5. Chawathe, Y., McCanne, S., Brewer, E.A.: RMX: Reliable multicast for heterogeneous networks. In: INFOCOM, Tel Aviv, Israel, IEEE (2000) 795–804
6. Clarke, I., Sandberg, O., Wiley, B., Hong, T.W.: Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science* **2009** (2001) 46

7. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: 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) 149–160
8. Jannotti, J., Gifford, D.K., Johnson, K.L., Kaashoek, M.F., O’Toole, Jr., J.W.: Overcast: Reliable multicasting with an overlay network. In: Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI). (2000) 197–212
9. Touch, J.: Dynamic internet overlay deployment and management using the x-bone. In: ICNP ’00: Proceedings of the 2000 International Conference on Network Protocols, Washington, DC, USA, IEEE Computer Society (2000) 59
10. Grace, P., Coulson, G., Blair, G., Mathy, L., Yeung, W.K., Cai, W., Duce, D., Cooper, C.: GRIDKIT: Pluggable overlay networks for grid computing. In: DOA ’04: Proceedings of Distributed Objects and Applications, Cyprus (2004)
11. Li, B., Guo, J., Wang, M.: iOverlay: A lightweight middleware infrastructure for overlay application implementations. In: Proceedings of IFIP/ACM/USENIX Middleware, Toronto, Canada (2004)
12. Pallickara, S., Fox, G.: NaradaBrokering: A distributed middleware framework and architecture for enabling durable peer-to-peer grids. In: Middleware. (2003) 41–61
13. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content addressable network. Technical Report TR-00-010, UC Berkeley, Berkeley, CA (2000)
14. Mathy, L., Canonico, R., Hutchison, D.: An overlay tree building control protocol. Lecture Notes in Computer Science **2233** (2001) 76
15. Yang, M., Fei, Z.: A proactive approach to reconstructing overlay multicast trees. In: IEEE INFOCOM, Hong Kong (2004)
16. URL: <http://rfc-gnutella.sourceforge.net/developer/stable/index.html> (2000)
17. Yang, B., Garcia-Molina, H.: Designing a super-peer network. In: Proceedings of the 19th International Conference on Data Engineering, Bangalore, India (2003)
18. Renesse, R.V., Minsky, Y., Hayden, M.: A gossip-style failure detection service. Technical Report TR98-1687, Cornell University (1998)
19. Porter, B., Taïani, F., Coulson, G.: Generalizing repair for overlay networks. Technical Report PTC-06-01, Lancaster University (2006)
20. Marzullo, K., Cooper, R., Wood, M.D., Birman, K.P.: Tools for distributed application management. IEEE Computer **24:8** (1991) 42–51
21. Bagchi, S., Whisnant, K., Kalbarczyk, Z., Iyer, R.K.: The chameleon infrastructure for adaptive, software implemented fault tolerance. In: Symposium on Reliable Distributed Systems. (1998) 261–267
22. Andersen, D.G., Balakrishnan, H., Kaashoek, M.F., Morris, R.: Resilient overlay networks. In: Symposium on Operating Systems Principles. (2001) 131–145
23. Chu, Y.H., Rao, S.G., Zhang, H.: A case for end system multicast. In: Measurement and Modeling of Computer Systems. (2000) 1–12
24. Ganek, A., Corbi, T.: The dawning of the autonomic computing era. IBM Systems Journal **42:1** (2003) 5–19