Using Current Uptime to Improve Failure Detection in Peer-to-Peer Networks

Richard Michael Price

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
The University of Birmingham, UK
March, 2010

Abstract:

Peer-to-Peer (P2P) networks share computer resources or services through the exchange of information between participating nodes. These nodes form a virtual network overlay by creating a number of connections with one another. Due to the transient nature of nodes within these systems any connection formed should be monitored and maintained to ensure the routing table is kept up-to-date.

Typically P2P networks predefine a fixed keep-alive period, a maximum interval in which connected nodes must exchange messages. If no other message has been sent within this interval then keep-alive messages are exchanged to ensure the corresponding node has not left the system. A fixed periodic interval can be viewed as a centralised, static and deterministic mechanism; maintaining overlays in an predictable, reliable and non-adaptive fashion.

Several studies have shown that older peers are more likely to remain in the network longer than their short-lived counterparts. Therefore using the distribution of peer session times and the current age of peers as key attributes, we propose three algorithms which allow connections to extend the interval between successive keep-alive messages based upon the likelihood that a corresponding node will remain in the system.

By prioritising keep-alive messages to nodes that are more likely to fail, our algorithms reduce the expected delay between failures occurring and their subsequent detection. Using extensively empirical analysis, we analyse the properties of these algorithms and compare them to the standard periodic approach in unstructured and structured network topologies, using trace-driven simulations based upon measured network data. Furthermore we also investigate the effect of nodes that misreport their age upon our adaptive algorithms and detail an efficient keep-alive algorithm that can adapt to the limitations network address translation devices.

This thesis is dedicated to my Grandfather Derrick Price,
who passed away in December 2009.

Acknowledgements

Firstly I am indebted to my supervisor Peter Tino whose endless patience and softly-spoken wisdom has encouraged and inspired me since my days as a undergraduate.

I would also like to thank my co-supervisor Georgios Theodoropoulos for pushing me to present my work when I thought it of little note.

To my friends, for being the most wonderful of distractions: David Brooks; Zeke Dixon; Ruth German; Luke Jackson; Iain Marshallsay; Robert Minson; Richard O'Carroll and Michaela Russell.

To my family for cherishing me despite not being exactly sure what it is that I do. Dad, Mom and Adam: I love each and every one of you.

Lastly - for the blessed assurance that Jesus is mine - to whom I owe everything.

Contents

List of Figures

List of Algorithms

1 Introduction

1.1 Background

Increasingly, Peer-to-Peer (P2P) networks are being used to share computer resources or services through the exchange of information between participating nodes. Examples include file sharing systems such as Gnutella [23], BitTorrent [13] and eMule [20]; distributed hash table based (DHT) structured networks such as Chord [62], Pastry [54], OneHop [39], CAN [50] and Kademlia [42].

Typically without any centralised control or hierarchical organisation,

P2P networks have become increasingly poplar as they provide a good substrate for sharing data, distributing content and multicasting information. By removing the bottleneck of the more traditional client-server approach, P2P networks can scale effectively, forming large networks through self-organisation. Nodes within these networks form a virtual network overlay by creating a number of connections with one another. Due to the transient nature of nodes within these systems any connection formed should be monitored and maintained to ensure the routing table is kept up-to-date.

Connection maintenance in P2P networks is primarily concerned with detecting failures that have left the network without informing their neighbours. Once a failed routing table entry has been detected it can then be replaced with an another node that is currently present in the network. While the approaches to failure detection are numerous the simplest solution is the most employed. This thesis contributes to this work by presenting an adaptive approach to failure detection based upon the observation that the time a peer has already spent in the network influences the time it's likely to remain.

1.2 Problem Statement

The primary function of P2P networks is to facilitate the sharing of information between peers which relies upon up-to-date and well maintained connections. As nodes leave the network connections fail, without maintenance routing tables gradually deteriorate and the efficiency of the resulting network's structure declines as new nodes join the network and existing nodes leave.

Clearly, the maintenance of routing tables in P2P networks is an important undertaking. Maintenance strategies need to be designed and imple-

mented based upon their efficiency in terms of the bandwidth they consume and their performance in terms of detecting failures quickly.

The current state of the art approach to maintaining connections within P2P networks is to define a fixed keep-alive period, a maximum interval in which connected nodes must exchange messages. The size of this fixed interval is often determined without consideration of the rate of churn or individual nodes in a centralised, static and deterministic fashion.

However, several studies have shown that older peers are more likely to remain in the network longer than their short-lived counterparts. This thesis investigates how the time a node has already spent in the network can be used to make more effective failure detection algorithms.

1.3 Contributions of this Thesis

This thesis explores the issues involved in designing and implementing adaptive failure detection algorithms that utilise current uptime in Peer-to-Peer networks. The contributions of this thesis are:

1. Three heuristic failure detection algorithms termed - ProbKA, PredKA and BudgetProb - which, given the distribution of session times is known, utilise the time a node has already spent in the network to determine when the next keep-alive message should be sent.

2. The extensive empirical analysis of our algorithms compared against the standard periodic approach using real network data in both unstructured and structured networks.

3. The investigation of alternative methods of ascertaining a node's current age and the affect of nodes misreporting their age.

4. We investigate and empirically analyse keep-alive algorithms that adapt to the limitations of network address translation devices.

1.4 Structure of the Thesis

This thesis is structured as follows:

- Chapter 2 introduces Peer-to-Peer networks as the background to the failure detection problem.
- Chapter 3 presents an up-to-date review of failure detection research, in the context of peer-to-peer networks.
- Chapter 4 overviews several empirical measurement studies of real Peer-to-Peer networks. Highlighting, how the time nodes spend within networks can be accurately measured and modeled. Finally the chapter summarises how this information has been proposed as a heuristic in existing research.
- Chapter 5 begins by explaining how knowledge regarding the time nodes spend within networks can be used to predict when they will depart. This Chapter then introduces our adaptive algorithms which predict the likelihood of a node being online based upon it's current uptime.
- Chapter 6 and 7 present the thesis' contribution for adaptive failure detection algorithms in unstructured and structured networks respectively, including the experimental frameworks and evaluations of each.
- Chapter 8 investigates how keep-alive intervals can be effectively and efficiently adapted to the limitations of NAT devices.

- Chapter 9 summarises the results and contributions of this thesis.

1.5 Publications

The inspiration for this research began by investigating how to model Peer-to-Peer networks realistically.

- Work with the dPeerSim project produced the paper "Analysis of a self-organizing maintenance algorithm under constant churn" presented within the Proceedings of the 2008 International Symposium on Applications and the Internet-Volume [47], highlighted insights into the self-organising nature of Peer-to-peer networks and how understanding churn could be used to improve maintenance within these networks.

- The application of current uptime based failure detection algorithms to unstructured networks, described in Chapter 6, was first presented at the Fifth International Conference on Collaborative Computing in 2009 under the title; "Still alive: Extending keep-alive intervals in P2P overlay networks" [48].

- The paper "Adapting to NAT timeout values in P2P Overlay Networks", presented here in Chapter 8, was presented at The Ninth International Workshop on Performance Modeling, Evaluation, and Optimization of Ubiquitous Computing and Networked Systems (PMEO-UCNS) in 2010 [49].

2

Introduction to Peer-to-Peer networks

This Chapter will introduce and discuss several different Peer-to-Peer networks which are the background to, and the general problem domain of, this thesis.

Peer-to-Peer (P2P) networks are an increasingly popular, if often infamous, substrate of distributed systems. Since the introduction and eventual demise of the legally strangled Napster service [44], P2P networks have received a great deal of public and research attention. One of the main draws of these networks is that anyone can participate sharing resources or services through the exchange of information between participating nodes.

Participants in P2P networks are often referred to as *peers*, *nodes* or *clients*. In this thesis we use the terms peers and nodes interchangeably to refer to individuals within a network. Whereas the term client is used to refer to the particular software application used to access the network. Examples of contemporary P2P clients include Limewire [38], Bearshare [4], Vuze [69], eMule [20] and μTorrent [43].

Nodes form a virtual network overlay by creating connections with other nodes. Two connected nodes are often referred to as neighbours, with each node's list of *neighbours* being called its *routing table*. P2P networks often differ in respect to the number of neighbours they keep. In a network of size N, the range of protocols varies from networks that maintain a fixed number of connections irrespective of network size [13, 23], to others that maintain $O(logN)$ neighbours [62, 54, 51] to fully connected networks that maintain $O(N)$ neighbours per node [39]. We begin our introduction of P2P networks with unstructured networks.

2.1 Unstructured Networks

2.1.1 Gnutella

The first generation of peer-to-peer file sharing networks genuinely began with the creation of Gnutella in the year 2000 [23]. Gnutella utilises a very simple and trustworthy protocol which has proved to be highly effective, this has allowed Gnutella to not only become but also remain one of the most popular file sharing protocols despite the development of many alternative applications [45].

When a Gnutella client, which can also be referred to as a *servent*, joins a network it creates a number of direct connections to other clients running

the protocol. Originally, the Gnutella program was distributed with a predefined list of reliable peers to allow new clients to bootstrap into the network. Currently, clients either connect by contacting nodes they've previously seen or by requesting a list of addresses from a web cache server.

Once a client has connected, additional peers can be easily found by existing connections reporting their own neighbours. By sending a *Ping* message, Gnutella clients can discover information regarding other peers in the network. Clients that receive a Ping message forward it to their own neighbours and respond with a *Pong* message which includes its own address, listening port and information regarding the amount of data it has made available to the network. The client that sent the original Ping message can then choose to form connections with these newly discovered peers. The cost of joining a Gnutella network is minimal as no further data needs to be transferred. Furthermore when a client leaves the network only its neighbours need to be contacted as there is no centralised authority to inform.

Originally the Gnutella platform considered all peers as equal, with each peer connected to a number of others selected entirely at random. Although this created a robust network, the significant heterogeneity amongst peers limited the scalability of the entire overlay.

Gnutella $v0.6$ [59] introduced *ultrapeers* allowing self-nominated and more capable peers to become the backbone of the network. Since Gnutella $v0.6$, most peers join a Gnutella network as *leaf peers* maintaining a small number of connections which must include at least one ultrapeer. The default settings of the most popular versions of Gnutella, currently LimeWire and BearShare [38, 4], determine the average number of connections peers maintain. As a result, most leaf peers tend to maintain just four connections with other peers, whereas ultrapeers maintain thirty connections by default.

Clients within a Gnutella network locate files by broadcasting queries to their neighbours, typically these queries follow the form <*predicate, ip*>. Clients that receive a query will forward it to their neighbours and so on until a predefined limit of network hops, the *time to live*, have been reached. Therefore, searching within a Gnutella network very much follows the pattern of a epidemic algorithm; gradually spreading from a single location and encompassing an increasingly large number of nodes each cycle, as shown in Figure 2.1. Clients attempt to match any files they possess to the predicate sent with any successful matches being communicated to the original client directly via the ip sent in the query. Clients do not forward duplicate queries or queries that exceed their time to live.

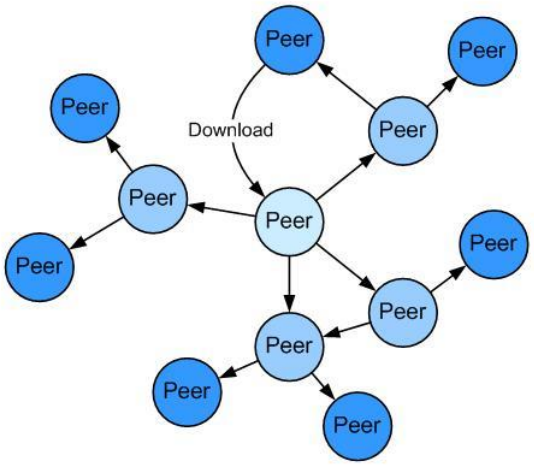


Figure 2.1: Gnutella resolves queries using an epidemic algorithm.

The strength of Gnutella is its simplicity, the low join and leave costs result in easy and efficient creation and maintenance of networks. Due to the inherent randomness of creating networks using this method, they are also robust as high degree networks are unlikely to partition. The routing of successful queries directly back to the original client also helps minimise the number of messages sent. This was not an original feature of the Gnutella

protocol but was later added to improve the scalability of networks. The likelihood of a successful search within a Gnutella network can also be easily increased by raising neighbourhood size and the number of hops a query is allowed to take. Furthermore, the format of predicates broadcasted throughout the network is not restrictive, and can be used to find numerous files with similar attributes.

However, the decentralisation of Gnutella does have its disadvantages. Finding rare data in the network may be impossible even if it exists, the total cost of bandwidth exponentially increases as you increase the number of allowed hops a query is allowed to take. Its upon this basis, Ritter in [52], predicted that the Gnutella network could not scale effectively and a single exhaustive search in a large Gnutella network would require a massive amount of data to be sent across the internet. At the time Gnutella was being heralded as the successor to the popular but legally strangled Napster service [44], which Ritter had helped to develop.

However, these two services were based upon quite different principles and are, as such, not directly comparable. Although Ritter's predictions did have reasonable grounding they could not facilitate for the distribution of both data and queries in a P2P network such as Gnutella. There is a massive amount of replication of data within a Gnutella network and it is this replicated data that the majority of queries are searching for. This allows a Gnutella network to set fairly low neighbourhood sizes and maximum hop distances without sacrificing a great deal of search accuracy.

2.1.2 BitTorrent

The BitTorrent [13] P2P protocol focuses on replicating typically large files to many clients. BitTorrent quickly became a popular alternative to the

more traditional client-server model of file mirroring and is currently used by many software publishers to distribute popular software updates to many users simultaneously.

A file distributed by BitTorrent is divided into numerous smaller fragments called *chunks*, typically around 256kB each, by the initial host known as the *seed*. Users are often called *peers*, with peers that have a complete copy of the file called *seeds* and peers with an incomplete copy called *leechers*. Peers can select from a number of clients that manage the transfer of files using the BitTorrent protocol.

Peers wishing to download a file, download a *torrent* file which connects them to the centralised component known as the *tracker*, as shown in Figure 2.2. The tracker acts as a rendezvous point for all peers, typically supplying a random subset of 50 peers already active within the torrent. A peer, however, will only seek to maintain between 20-40 connections to other peers. Multiple fragments can then be downloaded in parallel to peers from the initial seed. Once the distribution of these fragments has begun, the peers themselves can begin sharing the fragments they hold to other peers. When a peer collects all the fragments it becomes an additional seed and purely uploads fragments to other peers. The group of seeds and leechers that are distributing a file are collectively known as a *swarm*.

By choking and unchoking individual connections the BitTorrent protocol regulates the exchange of chunks between peers using two simple principles. Firstly, a "tit-for-tat" policy encourages cooperation between peers by, when possible, only sending a chunk to peers whom a peer has received data from. Secondly, while peers maintain connections to at least 20 neighbours at a time, they limit the number of peers they serve simultaneously to just four. A seeding peer will send data to the four peers with the highest download

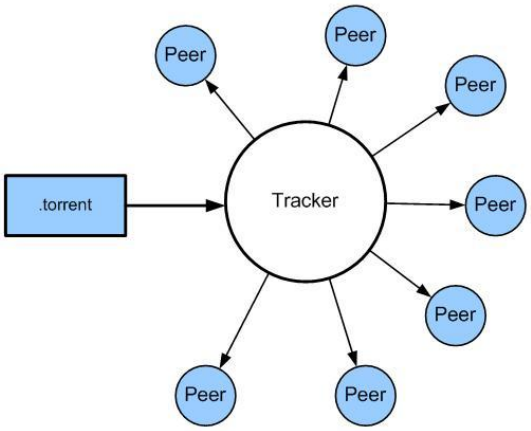


Figure 2.2: Peers in BitTorrent locate one another through a centralised tracker by downloading a torrent file.

rate, whilst a leeching peer sends data toward the four highest uploaders. Every ten seconds peers re-evaluate the upload/download rates for all peers and every thirty seconds peers optimistically unchoke a random peer hoping to potentially discover a better service. BitTorrent clients will therefore keep just one fifth of their connections active at any time, with the remaining connections kept in reserve.

BitTorrent also implements a chunk selection algorithm, typically the *rarest first policy* which seeks to upload the least duplicated chunk and therefore the most needed to the set of connected peers. However, when a peer first joins a torrent it seeks to download any available chunk in order to begin the download process as quickly as possible. Another interesting feature of BitTorrent networks is their tracker, that acts as a centralised rendezvous point for the network. Nodes contact a tracker upon joining, when sending periodic updates and, if they leave gracefully, upon their departure. As the tracker logs all this data it provides us with the arrival and departure times of actual peers to the nearest second, giving those with access to these tracker logs an insight into the inner workings of BitTorrent networks.

By dividing the load across all peers BitTorrent avoids common bottlenecking problems associated with more traditional single source downloads. In fact, a file can still be successfully downloaded by any peer even if all seeds have been removed from the system, as long as the collective swarm maintains a distributed copy of the file. As fragments are shared, the number of distributed copies within the swarm slowly increases giving the overall system increased flexibility and robustness.

BitTorrent is limited in the respect that it only allows the downloading of files that are listed in the available trackers. Searching for files actually entails locating trackers via a third-party search engine. It is these search engines that authorities target in an attempt to limit the amount of illegal file sharing across the internet. However, users of such services are likely to migrate from one site to another when a popular illegal file-sharing search engine is shut down.

2.2 Structured Networks

Distributed hash table (DHT) based structured P2P networks construct overlays by deterministically positioning both nodes and data within a network. The position of nodes and data within the network is determined by first assigning a unique key. Each key is then hashed to produce an *identifier* which then occupies a fixed position within the overlay.

As all clients have access to the DHT, information can be located using the operation *lookup*(*key*) which will return the IP address of the node responsible for that key. In file-sharing applications, files are often stored at their originating node(s) and a pointer to each of these nodes is inserted for every file in the overlay by hashing each file's key. Files can then be found

using the $lookup(key)$ operation, which will locate the pointer and resolve the query by returning the IP address of the node(s) holding the file. Data can be inserted in the network using the $insert(key, dataitem)$ operation.

2.2.1 Chord

Chord [62] is a DHT based structured P2P network that scales logarithmically with the number of nodes. Chord orders identifiers onto an *identifier circle*, giving Chord a ring structure called the *Chord ring*.

Using a consistent hash function such as $SHA-1$ [19] each node and key in the Chord ring is given a m-bit identifier. Keys are most often items available through the network such as data, resources and files. The size of m is selected to ensure the probability of two keys colliding and hashing to the same identifier is negligible. The size of the Chord ring is given by 2^m.

A node's identifier is often assigned by hashing its IP address, while a key's identifier is produced by hashing the key. Chord assigns the key κ to the *successor node of* κ, the first node whose identifier is equal to or greater than κ denoted by $successor(\kappa)$. Items such as data with the key κ are therefore assigned to the node $successor(\kappa)$. Consistent hashing ensures with a high probability that keys are equally distributed amongst nodes. As a result, in a network with N nodes and K keys each node will be responsible for K/N keys on average.

In its simplest form each node in a Chord network only maintains a connection to its immediate successor. Lookups can then be forwarded around the ring towards the desired identifier until it reaches the node responsible for that key. Of course a single connection is the smallest possible routing table a node can maintain and remain connected. Accordingly routing in this manner is inefficient with lookups being resolved on average after $O(N/2)$

hops, as illustrated in Figure 2.3.

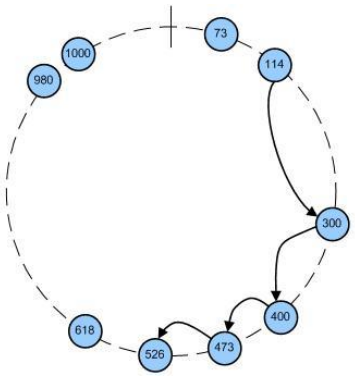
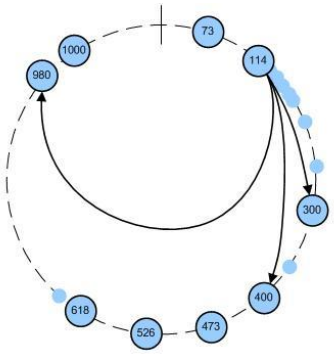


Figure 2.3: Simple routing in Chord.

To improve routing efficiency each node n maintains a routing table containing up to m entries called the *finger table*. Each finger table entry targets a specific point in the overlay and is filled by the first node that succeeds that point. The i^{th} entry in the finger table of node n contains the first node that succeeds n by at least 2^{i-1} for $1 \leq i \leq m$. The first entry points to the node's immediate successor and each consecutive entry points to a node at increasingly large distances away in the identifier-space. This results in a node being connected to more nodes who are closer in identifier-space than those that are further away. Overall, Chord maintains a connection to $O(logN)$ distinct finger table entries, as the example in Figure 2.4 shows.

By forwarding lookups greedily through finger tables, to the node closest to the desired identifier, Chord is capable of efficiently locating any key in a network of size N nodes in just $O(log_2 N)$ hops on average. To ensure such efficient routing nodes, within Chord have to maintain their routing tables. The *stabilization* procedure, which is run periodically by each node, not only ensures its successor and predecessor pointers are correct but also each routing table entry is pointing towards the correct node.

Furthermore, it is also generally accepted by the DHT community that



IP	Target	Position
82.256.198.131	114 + 1 = 115	300
82.256.198.131	114 + 2 = 116	300
82.256.198.131	114 + 4 = 118	300
82.256.198.131	114 + 16 = 130	300
82.256.198.131	114 + 32 = 146	300
82.256.198.131	114 + 64 = 178	300
82.256.198.131	114 + 128 = 242	300
143.275.149.30	114 + 256 = 370	400
63.23.18.28	114 + 512 = 626	980
-	114 + 1024 = 114	self

Figure 2.4: An example Chord finger table.

Chord's ring topology offers some flexibility in neighbour selection. A common way to allow for more flexible neighbour selection within Chord is too choose the owner of a random key that succeeds a node n within the range $[2^{i-1}, ..., 2^i]$ [24, 41, 28].

Chord can resolve lookups in either a *iterative* or *recursive* style. Using iterative routing, the node initiating a lookup conducts all the communication: asking each node in turn for the closest known node to the desired destination. While in recursive routing, nodes forward lookups to the next node until it reaches the appropriate successor as shown in Figure 2.5.

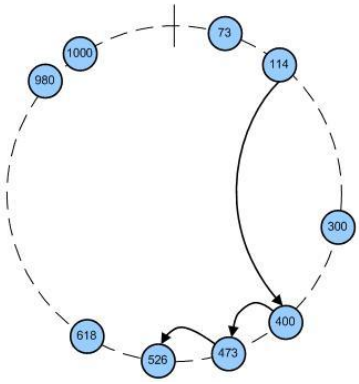


Figure 2.5: Efficient routing in Chord.

Nodes joining and leaving the network cause the accuracy of each node's

finger table to gradually degrade, causing routing to become less efficient. To maintain efficient routing each node in a Chord network must ensure each finger table entry is correct and pointing toward the node responsible for the desired identifier. At the very least Chord must ensure each node's successor pointer is up to date to ensure consistent routing. As Chord relies upon the successor pointer being correct its position is checked as often as every second.

Maintaining a Chord overlay comes at a cost. Each node not only pings all entries in its finger table to check their liveness but also initiates the *stabilization* protocol which updates and replaces successor and finger table entries. Each node n at regular intervals picks a finger table entry i and initiates a lookup for the target of entry i namely $n + 2^{i-1}$. The finger table entry i is then replaced by the result of this lookup if necessary. Stabilization is scheduled on a periodic basis in Chord [62], but in DKS - which to some extent is a generalisation of the Chord system - stabilization occurs reactively when a change in network topology is detected [1].

Chord was amongst the first DHT based structured networks to be developed and as such is often used as a benchmark system to compare P2P networks against. Furthermore, a number of potential applications of Chord have been proposed including a Cooperative File system [15] and a Chord-based Domain Name Service (DNS) system [14].

2.2.2 Pastry

Pastry [54] like Chord is intended to be a general platform for a variety of P2P Internet applications. Accordingly they share a number of characteristics; each Pastry node has a unique numeric identifer determining its position in a circular identifier space, lookups are expected to be resolved in just $O(logN)$

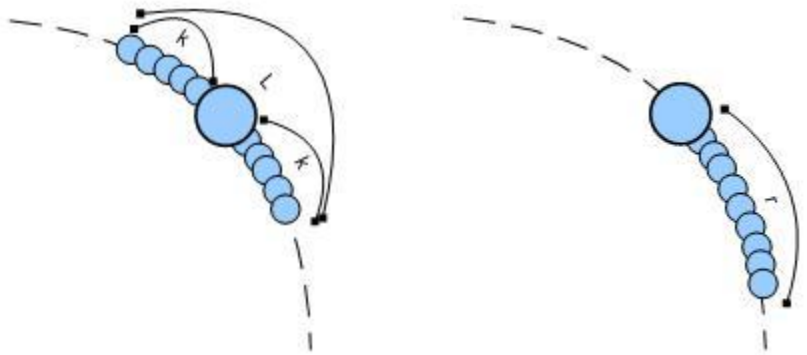
steps, whilst each node maintains a routing table containing $log_{2^c}(N)$ rows. Where c is a configuration parameter, typically set to 4. Unlike Chord, Pastry offers more flexibility in choosing routing table entries and each node is responsible for the keys it's numerically closest to; not just the keys it immediately succeeds.

Nodes in Pastry are assigned a 128-bit node identifier called the *node id*. Node id's are assumed to be uniformly distributed throughout the identifier space. Each Pastry node maintains a routing table containing $[log_{2^c}N]$ rows with 2^c-1 entries each. Where c is a configuration parameter that determines the size of the routing table and is usually set to 4. The nth row of the routing table of node A contains entries that share the first n digits of the A's node id but do not share the $n+1$th digit of A's node id. Nodes with an appropriate prefix fill each space in the routing table, with each entry being associated with an IP address. As many nodes are likely to satisfy the appropriate prefix constraint, nodes in a Pastry will often select the node with lowest latency to be included within the routing table. Furthermore Pastry node's often forward lookups to the node with the lowest latency that matches the required identifier prefix. Due to the uniform distribution of node identifiers on average $[log_{2^c}N]$ rows are populated.

Pastry was one of the first DHTs to support sequential neighbours, the equivalent of Chord's successor list. Each node in a Pastry network maintains a *leaf set* of the $2l$ nodes immediately preceding and succeeding its position as show in Figure 2.6a. The members of node $A's$ leaf set $L(A)$ are referred to by $L(A)_i$, with $-l \leq i \leq l$ where $L(A)_0$ is the owner of the leaf set node A.

The original version of Chord did not support sequential neighbours but choose to augment its routing information once it proved to be an effective

strategy to improve network robustness. Each node in a Chord network maintains a *successor list* containing r nodes that succeeded its position in the overlay, as illustrated in Figure 2.6b. As a result all r successors would have to fail simultaneously to completely disrupt routing around a Chord ring. The typical ring geometry as used in both the Chord and Pastry DHTs encourages a natural ordering of nodes. Each node can maintain a set of sequential neighbours who either immediately precede or succeed its position in the overlay.



(a) (b)

Figure 2.6: Pastry's leafset and Chord's successor list.

Sequential neighbours play an critical role in network recovery, when nodes fail important routing information may be lost; until nodes can repopulate their routing tables they must use alternative nodes to route messages through. Sequential neighbours provide these alternative routes thus providing more flexible and robust routing around a network overlay, by maintaining just a single sequential neighbour a network can still route messages. However, as sequential neighbours are just as likely to leave the network as any other node the size of this set is often selected to minimise the probability that all node's in the set fail within a small period of time. Typically, the size of a sequential neighbour set L is logarithmic in proportion to the size

of the system. For example the default leaf set size in FreePastry [18], an open-source Java implementation of Pastry maintained by Rice University is 24. Sequential neighbours provide a DHT with increased *static resilience* enabling messages to still be routed, albeit with decreased efficiency, in the presence of multiple failures allowing the recovery algorithms time to restore the network.

Nodes detect failures in their leaf set using the same keep-alive mechanism used for most distributed indexes. Each node within a leaf set is expected to send keep-alive messages every k seconds. If a node does not send a keep-alive message as expected, a probe message is sent and if no response is given the node is assumed to have left the network. As leaf sets provide alternative routes through which messages can be sent, it is important that they are consistent with the actual state of the network. If over a period of time all members of a node's routing table and leaf set have failed a node would become forcefully disconnected from the network.

2.2.3 OneHop

OneHop [39] maintains a complete list of nodes in every node's routing table, allowing lookups to be resolved in a single step. While a OneHop node does not maintain open connections with all other nodes in the network, it does store and maintain the information required to contact all other nodes when required. The accuracy of this information determines the success rate of any query issued. As in Chord and Pastry each node is assigned a random 128-bit identifier uniformly distributed on a ordered identifier ring. Each node has a predecessor and a successor in the identifier ring and periodically sends keep-alive messages to these nodes to ensure they are still present and no new nodes have come between them.

Keep-alive messages detect changes in the network and nodes are informed of these changes through dissemination trees which propagate event information. First the identifier ring is divided into several contiguous intervals called *slices*, with each slice having a *slice leader*. When a new node joins it learns of its slice leader and routing table information from one of its neighbours. Each slice is then further divided into equal-sized units, with each unit having a *unit leader*. Upon detecting a new node joining, or an existing node failing, ordinary nodes notify their slice leader. Slice leaders aggregate all these event notifications for an interval of t_{big} seconds before informing all other slice leaders. Every t_{wait} seconds slice leaders then inform all unit leaders in their respective slice, who in turn inform their successor and predecessors. By piggy-backing information on keep-alive messages, these other nodes then propagate this information in one direction: if they received information from their predecessor they inform their successor and vice versa until the unit boundary is reached.

As with Chord and Pastry, OneHop relies on successor and predecessor pointers for correctness. This ensures even if all other membership information is incorrect, a lookup will eventually succeed after re-routing. Accordingly OneHop sends keep-alive messages to successors and predecessors every second to ensure they are up-to-date.

2.3 Summary

Peer-to-Peer networks provide a good substrate for the creation of large scale distributed systems. One of the key reasons for this is their resilience to node failures. This Chapter has surveyed several network overlays each of which maintain different structures and different levels of routing state. Routing

state provides each network with a certain amount of *static resilience*, the extent to which overlay networks can route effectively before routing state is restored.

However, static resilience is only one aspect of a network's overall resilience, which also includes *data replication* and *routing recovery* [28]. While data replication ensures node failures do not result in loss of data through methods such as caching, routing recovery replaces failed nodes in routing tables. Critically, both static resilience and routing recovery rely upon failed nodes being detected accurately and quickly. Without accurate failure detection unnecessary maintenance overhead may be incurred. For example, unless cached nodes are available to repair failed routing table entries, routing recovery may resort to expensive recovery mechanisms such as the stabilization mechanism in Chord. Furthermore, to exploit static resilience within an overlay, a node must know which neighbours have failed. Once again relying on accurate and timely detection of failed nodes.

To this end, every node in almost all contemporary P2P networks must spend bandwidth on the maintenance of its routing state to ensure its connections are up-to-date. Our focus is not upon the comparison of P2P overlay networks, but upon improving failure detection within network overlays to further augment the existing resilience of these systems. To further explore this issue the next chapter examines existing failure detection mechanisms and highlights areas for improvement.

3

Existing Failure Detection Techniques

Peer-to-peer (P2P) networks share computer resources or services through the exchange of information between participating nodes. These nodes form a virtual network overlay by creating connections with one another. While joining a P2P network necessitates contacting other nodes, leaving a network does not. Nodes may leave a P2P overlay *ungracefully*, i.e without informing their neighbours. Due to the transient nature of nodes within P2P systems any connection formed should be monitored and maintained to ensure the routing table is kept up-to-date. Without maintenance routing tables gradually deteriorate and the efficiency of the resulting network's structure declines

as new nodes join the network and existing nodes leave.

Routing table maintenance is comprised of two parts, failure detection and node replacement. Failure detection is the discovery of neighbours that have left the network ungracefully. Each entry in a routing table specifies a connection to another node, should a neighbour leave the network ungracefully the remaining node's routing table is left ignorant to the change in network topology. Once a failed routing table entry has been detected it can then be replaced with an another node that is currently present in the network. The method of replacing routing table entries is often network protocol specific with routing tables using cached nodes where available.

This Chapter presents a review of failure detection research in the context of P2P networks beginning with passive maintenance and then the widely used standard keep-alive algorithm, before describing gossip-based algorithms and predictive mechanisms.

3.1 Passive Maintenance

The simple alternative to maintenance would be not to monitor connections at all, instead relying on data packets to be sent between neighbours. When data packets are sent through a failed connection they eventually timeout and the failure of the packet is reported back to the sending node, essentially detecting the failure. It is this passive approach to failure detection and routing table maintenance that the DKS family of P2P applications promotes [1]. Active failure detection consumes additional bandwidth, this may be unnecessary in P2P systems where the number of lookups and data insertions is significantly higher than the number of joins, leaves and failures. However, the passive approach of DKS is often inadequate as data traffic may not

be sufficient to ensure each connection is checked regularly. Furthermore failures would only be detected when a node is needed [51], with data packets timing out and incurring potentially expensive delays. Similarly, a study by Krishnamurthy [32] highlighted that when churn is high a periodic active approach performs better than the passive correction on change approach.

3.2 The Standard Keep-Alive Algorithm

The Standard Keep-Alive (SKA) algorithm is employed by virtually all P2P overlay networks to detect the departure of ungraceful nodes. Currently, all connections are maintained in unstructured networks, such as Gnutella [23] and BitTorrent [13] and structured networks such as Chord [62], Pastry [54] and Bamboo [51] by the standard keep-alive algorithm.

Algorithm 1 SKA()

```
for all i in routing_table do
  if T^i_since ≥ k then
    probe i
  end if
end for
```

Typically two connected nodes each assume the other to be "alive" in the network for a duration of time defined by the *keep-alive period*, k. This keep-alive period defines the maximum interval that a connection between two nodes should remain inactive. If the time since a node was last contacted, T_{since}, is greater than a single keep-alive period, keep-alive messages are exchanged to ensure the connection is still alive. The time since a node was last contacted is stored separately for each individual connection a routing table maintains as shown in the pseudo-code for the SKA algorithm is given in Algorithm 1.

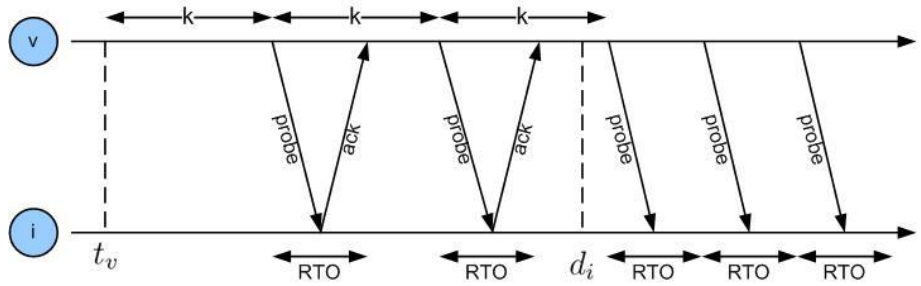


Figure 3.1: The Standard Keep-alive algorithm. Nodes exchange keep-alive messages every k seconds.

As illustrated in Figure 3.1, at time t_v node v connects to node i and sends one keep-alive message, often called a *probe*, every k seconds. Nodes that are part of the network respond to a received keep-alive message by returning an *acknowledgment* message. As nodes that have left the network do not respond, this allows any failed connection to be detected and subsequently replaced.

The response to a probe is expected before the round-trip timeout (RTO) expires. The structured DHT Bamboo calculates the RTO based for each neighbour on previously observed round-trip times [51]. If a keep-alive acknowledgement is not received once the RTO has expired, the neighbour is considered to have failed.

In practice unacknowledged keep-alive messages are re-sent multiple times at short intervals. This process is shown in Figure 3.1 when node i departs the network at time d_i. Re-sending unacknowledged probes minimises the risk of false negatives, where a node is wrongly considered to have failed due to a keep-alive message being somehow lost in the underlying network.

Strictly speaking, acknowledgement messages are only required when routing tables are asymmetrical. Within symmetrical routing tables - when node i is present in node v's routing table and v is present in node i's table - a keep-alive from i to v is enough to inform v that i is still alive. However, in

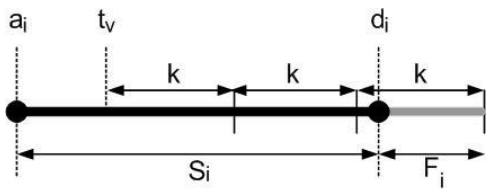


Figure 3.2: Node i's session time $S_i = d_i - a_i$. Its neighbour node v connects to i at time t_v and begins periodically sending keep-alive messages with an interval of k. Node i's departure at time d_i is not detected by v until the next keep-alive message goes unacknowledged, resulting in a failure detection delay period of F_i.

asymmetrical routing tables a node can only determine its neighbour is online by receiving an acknowledgement message. As keep-alive messages with acknowledgements can be applied to all types of routing tables, we solely focus upon them in this study.

Many existing P2P systems send keep-alive messages according to a standard and fixed periodic interval, this interval is typically determined by the application developer and therefore uniform across all nodes in a network. The designers of BitTorrent [13] for example have set the default keep-alive period to $k = 120$ seconds. Not only is the SKA algorithm simple to implement, it is also simple to calculate the number of keep-alive messages sent during a given period. In a network of N nodes with D being the average node degree, there are $2ND$ keep-alive messages sent and subsequently acknowledged every k seconds.

Figure 3.2 illustrates node i arriving in the network at time a_i and departing at time d_i, its *session time* $S_i = d_i - a_i$. At time t_v node v connects to node i and only then begins to send periodic keep-alive messages with an interval of k seconds to i. At time d_i node i departs the network ungracefully, node v does not learn of this departure until the subsequent keep-alive that is sent but goes unacknowledged. As a result there is a *failure detection de-*

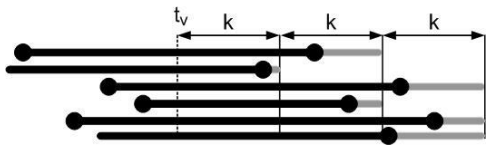


Figure 3.3: Nodes connect to neighbours and therefore depart the network at any point within an interval k with equal probability. This departure is not detected until the next keep-alive message, incurring an average delay of $k/2$.

lay period F_i during which node v falsely believes that node i is still present within the network.

The size of the incurred mean delay period is directly proportional to the keep-alive period k. As neighbour selection is typically not dependent upon current session times and nodes must select neighbours already present within the network, the join time t_v can occur at any point during node v's session time with equal probability. Therefore the leave time d_i of a neighbour falls uniformly at random within any single keep-alive interval. This departure is not detected until the next keep-alive message, incurring a mean delay of $k/2$ [70], as illustrated in Figure 3.3.

Keep-alive messages also serve a very important second purpose - they prevent connections from becoming inactive and, as a result, being removed by Network Address Translation (NAT) devices. NAT devices allow several nodes within a private network to share a single public IP address. This allows private home networks and business intranets to interface with public networks such as the Internet.

However due to limited resources and vulnerability to denial of service attacks, NAT devices cannot indefinitely hold the state of their translation tables. As a result, idle connections are eventually expired and connection states removed after a NAT *timeout period*. Every time a packet is sent through a connection the NAT device at the other end restarts the timeout

period. Keep-alive messages therefore serve as artificial packets, forcing NAT devices into resetting the timeout period and keeping the connection alive.

To avoid connections becoming idle, connected nodes must periodically exchange *keep-alive messages* at an interval shorter than the timeout period. As a result of their dual purpose and simplicity, keep-alive messages are widely used throughout all types of networks to maintain connections between nodes.

A fixed periodic interval can be viewed as a centralised, static and deterministic mechanism, maintaining overlays in an predictable, reliable and non-adaptive fashion. While the periodic interval can be manually adjusted in response to network conditions, in practice this may be difficult to do. Not only would all nodes in the network have to be individually contacted, defining a suitable interval is dependent on factors such as current network conditions which may be difficult to obtain and furthermore may change rapidly.

3.2.1 Bidirectional Forwarding Detection

The standard keep-alive algorithm is typically configured to detect failures within a few seconds, an acceptable level for most applications. However with the rise in popularity of Voice over IP (VOIP) applications such as Skype [3]; a more aggressive failure detection mechanism is required to avoid even a sub-second loss of signal which would be detectable to the human ear.

Bidirectional Forwarding Detection (BFD) [31] is an interoperable standard protocol that aims to provide short failure detection times at a low-overhead. The BFD protocol is very similar to the standard keep-alive mechanism described above. Both mechanisms detect failures when nodes fail to respond to artificial messages. Like the SKA mechanism, BFD can be con-

figured to detect failures by one node sending "hello" or "still alive" packets to another node at a set rate. Should a packet not be received after a set duration the sending node is considered to have failed. Similarly to SKA, BFD can also be configured to send "echo" packets which must be acknowledged by the receiving node.

At just 24 bytes on top of the UDP and IP header, BFD packets are small and detect the failure of both nodes and links. Unlike keep-alive messages, BFD packets do not reset the NAT expiration timers on routers. Furthermore in routers with a distributed architecture, BFD packets can be processed on interface modules, whereas other routing protocol packets (such as keep-alive messages) are processed by the control plane. This allows BFD packets to be processed quicker by avoiding a router's main processor.

The main advantage BFD possesses over standard keep-alive is that it can be configured to send control packets at very short intervals at a lower overhead. However we consider the standard keep-alive mechanism as a more general and widely used approach to failure detection as the mechanisms behind BFD are identical.

Overall, algorithms seeking to improve the standard keep-alive algorithm typically do not alter the underlying mechanism of inserting artificial packets into the network and expecting acknowledgements in response. Instead, as the next two sections detail, improvements to SKA can can be broadly classified into two approaches: gossip-based and predictive algorithms.

3.3 Gossip-based Algorithms

Gossip-based algorithms share information of node failures amongst neighbouring nodes once a failure has been detected. Gossip-based failure detec-

tion algorithms are generally complimentary to other failure detection mechanisms helping to improve their performance in terms of cost, scalability and by reducing failure detection time.

The use of gossip-based algorithms in networks was pioneered by the Clearinghouse project [17] to resolve inconsistencies in distributed databases. Gossip based or sharing algorithms mimic the behaviour of epidemic algorithms: once a node learns of new information it is said to be *infected*, and this infection is spread amongst neighbouring nodes, propagating the information throughout the network.

In [68], Renesse et al. introduce a gossip-based failure detection service in which a group of nodes monitor members of a network. Each node in the network maintains a list of other member's addresses and their corresponding *heartbeat counter* which is used for detecting failures. Every T_{gossip} seconds each node increments its own heartbeat counter and selects one other member at random to send it's membership list to. When a node receives such a *gossip* message, it merges the list in the message with its own using the maximum heartbeat counter for each member. If a member's heartbeat counter has not increased for more than T_{fail} seconds, then the member is considered failed and is removed from membership lists after $T_{cleanup}$ seconds. Using random and periodic communication this gossip-based approach improves scalability and the average failure detection time. Users can then query the monitoring nodes to learn which services are currently available.

Zhuang et al. in [70], empirically analyse the performance of four gossip-based algorithms against the standard keep-alive algorithm described above. The first approach evaluated by Zhuang et al. is the Sharing Negative Information with Backpointer State (SN+BPTR) algorithm. In SN+BPTR nodes share negative information, "node is down", with the neighbours of a

failed node.

More formally, each time that node X sends a keep-alive message to node Y it also learns of all Y's neighbours $n(Y)$ and stores them in its backpointer state. If node X subsequently discovers Y has failed it then informs the neighbours in $n(Y)$ of this event. Zhuang et al. refer to this sharing of negative information as *boosts*. Nodes in $n(Y)$ will then remove node X from their routing table if they receive b consecutive boosts within a time window T_{boost}. They then show how this time window T_{boost} can be configured such that node's are unlikely to be falsely removed due to network issues such as packet loss but are removed with high probability when a neighbour does fail.

The remaining three algorithms investigated by Zhuang et al. are essentially variations on the SN+BPTR approach. The second algorithm - the sharing negative information (SN) algorithm - simply shares negative information without maintaining backpointer state. Backpointer state is knowledge of another node's neighbours, who receive boost messages when a node fails. Without backpointer state, a node simply forwards boosts to all of it's own neighbours. The effectiveness of this approach relies on the probability of two neighbours sharing a third mutual neighbour.

The third and fourth algorithms share negative and positive information with and without maintaining backpointer state SNP+BPTR and SNP respectively. Sharing positive information, a node is online, reduces the number of false positives at the expense of increase control overhead due to the additional positive information messages.

Zhuang et al define four performance metrics which they use to analyse and compare their algorithms: failure detection time, probability of a false positive, control overhead and packet loss. The *failure detection time* is

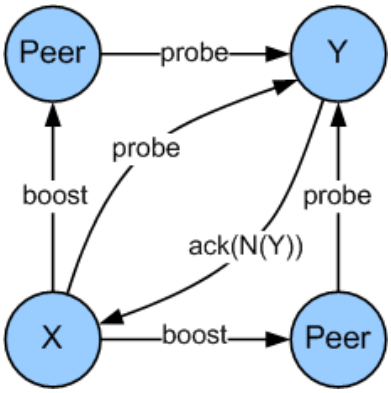


Figure 3.4: The SNP+BPTR algorithm.

the time elapsed from a failure occurring and subsequently being detected. Larger failure detection times cause failed nodes to remain in routing tables longer, which may result in queries being forwarded to them. The *probability of a false positive* (falsely detecting a failure) is especially serious when nodes share information. Algorithms that are prone to falsely detecting nodes as down not only incorrectly replace their own routing table entries but, when combined with gossip mechanisms, may cause needless gossip messages to be sent and potentially other neighbouring nodes to incorrectly alter their routing tables. The *control overhead* considers the cost in terms of bandwidth spent on maintenance. As the primary purpose of a P2P network is rarely limited to maintaining that network's state, minimising the cost of maintenance contributes to the efficiency of the network. Finally the *packet loss* rate measures how reliable routing is when packets are lost due to forwarding messages to a failed neighbour. Packet loss leads to expensive message timeouts, causing responses to a user's query to be delayed.

Zhuang et al. find in the absence of network failures, algorithms with backpointer state reduce the failure detection time and the cost of maintenance when compared with the traditional SKA approach. These algorithms

allow nodes to learn of failures from their neighbours without having to detect it for themselves: essentially working as a group to detect failures rather than as individuals.

Although the SN+BPTR and SNP+BPTR consumes more bandwidth, due to boosts and inclusion of the backpointer state in acknowledgements, for the SKA approach to achieve the same failure detection time it would have to send keep-alive messages at a faster rate consuming more bandwidth than the SN+BPTR and SNP+BPTR algorithms. However, when network failures are considered, nodes that share information continue to reduce the average failure detection time but at a increased control overhead when compared to the SKA approach.

The authors of Bamboo [51] investigate and compare alternative to ways to maintain the structure of the Pastry DHT [54]. Bamboo's maintenance algorithm periodically exchanges node's leaf set with a randomly selected member of that leaf set. In Bamboo when a node sends its entire leaf set to a random neighbour, that neighbour must respond with its own leaf set. The first send message is called a leaf set *push* with the reply called a leaf set *pull*. The authors detail how pushing and pulling augments the networks resilience to bad leaf set states especially when nodes are temporarily unable to receive leaf set updates. Whereas in FreePastry [18], nodes only send their leaf set to other members of the set once they have learnt of a change in the network. Unlike in FreePastry a Bamboo node will be able receive updates regarding changes in the network the next time it shares it's leaf set with another node, whereas FreePastry nodes must wait to be informed.

By running the Bamboo algorithm on a periodic basis rather than according to changes as they are detected in the network, the maintenance process is decoupled from the rate of churn. This is important on two re-

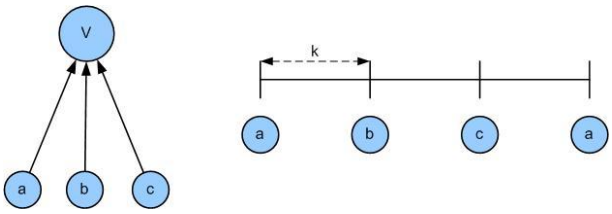


Figure 3.5: Nodes a, b, c are all connected to node V.

lated aspects. Firstly all changes in each period are aggregated into a single message: in FreePastry a message is typically sent for each change as soon as it is detected. While FreePastry nodes are informed of changes sooner, it's at the expense of $O(L^2)$ messages being sent, where L is the size of the leaf set. Whereas the leaf sets in a Bamboo node converge in $O(logL)$ phases. Secondly, adding messages to the network nodes generate additional congestion. An overly congested link may cause a node to receive keep-alive messages in a untimely fashion and as a result wrongly consider a node to have failed. As FreePastry recovers from failures reactively its response will be to send more messages informing all leaf set members of this perceived network change. As detailed in [51] this creates a positive feedback loop adding even more congestion and further increasing the chance that other neighbours will be wrongly perceived to have failed.

The Cooperative Keep-alive (CKA) algorithm presented by Dedinski et al in [16], effectively reduces the overhead of keep-alive messages through cooperation between nodes with mutual neighbors. Although the frequency of keep-alive messages remains the same, instead of nodes responding to all it's neighbors in parallel, they propose that each node coordinates it's keep-alive messages so they are sent in sequence by it's common neighbors.

As illustrated in Figure 3.5, node V directs it's incoming connections from node's a, b and c to send keep-alive messages one after another. Newly established incoming connections are simply added to end of the existing

sequence of messages and failed connections are removed.

In a system with no churn, the CKA algorithm stabilises with node's receiving a keep-alive request and sending a keep-alive reply every k seconds, resulting in $2N$ messages every k seconds. This compares favorably to the SKA algorithm which in the same period of time produces DN messages, where D is the average node degree, which is $\frac{D}{2}$ more keep-alive traffic than the CKA approach.

Nodes using the CKA algorithm only send a keep-alive message to a individual neighboring node every Dk seconds. Should a failure occur, incoming connections would detect the failure after $\frac{Dk}{2}$ seconds on average. Whereas the SKA approach detects failures after just $\frac{k}{2}$ seconds on average.

To address the issue of incurring long failure detection delays, Dedinski utilises neighbourhood set flooding. Neighbourhood set flooding is an epidemic style algorithm that gradually informs mutual neighbours of a node's failure, in a similar fashion to the SN+BPTR algorithm presented in [70].

Consider an example network where nodes X and Y are connected and share some neighbouring nodes. Upon detection the failure of node Y, node X iteratively contacts the mutual neighbours of $n(Y)$ as it becomes aware of them. As node X may have an outdated view of all the neighbours of Y these neighbours may remain uninformed should Y fail. Using neighbourhood set flooding the nodes $n(Y)$ not only probe Y themselves to ensure news of the failure is correct, they also send a copy of their known neighbours of Y to node X. If node X subsequently learns of any new neighbours of Y it then also informs them of Y's failure and the process repeats until X has informed all the known neighbours of Y. Despite X using neighbourhood set flooding, there still may be cases where Y's neighbourhood is partitioned. In this case the failure of X will be eventually discovered after at most Dk seconds by

all nodes.

The CKA algorithm - by using a simple scheduling algorithm and neighbourhood set flooding - can dramatically reduce the number of keep-alive messages exchanged whilst still maintaining the network efficiently. By reducing the cost of maintenance the number of connections per node can be increased, improving the connectivity and routing efficiency of the overall network.

Although highly effective, the cooperative-keep-alive algorithm presented in [16] assumes that all connected nodes have an incentive to maintain their mutual connections. However, many P2P overlays often maintain unidirectional connections resulting in the receiving node having little if any incentive to share information. Furthermore, malicious nodes could exploit the information sharing aspect in such a network and deliberately missinform their neighbours by scheduling keep-alive messages at inappropriate periods or by including nodes that do not exist in their mutual neighbour reports.

Overall, as gossip-based algorithms only send additional messages once a failure has been detected they incur minimal additional overhead. When compared against the SKA algorithm, by allowing nodes to share information and detect failures in parallel, gossip-based approaches reduce the failure detection delay significantly.

3.4 Predictive Keep-Alive Mechanisms

Predictive keep-alive mechanisms attempt to use features of the underlying network - primarily churn - to predict when nodes will leave the system. Based on such predictions these mechanisms adjust the frequency at which keep-alive messages are sent to detect failures more efficiently. The efficiency

of keep-alive messages can be improved in two ways: firstly by sending keep-alive messages only when necessary thereby reducing the bandwidth incurred by maintenance and secondly by minimising the delay between failures occurring and being detected.

Chen et al. [12] specify the quality of service (QoS) of failure detectors in distributed systems which suffer probabilistic message losses and network delays. The failure detection model studied by Chen suspects a process to have failed if a failure detector has not received a heartbeat message after some period of time. As a result these failure detectors can mistakenly assume a process has failed if the heartbeat message has been somehow lost or delayed. Accordingly, Chen et al propose three primary metrics for the QoS specification of failure detectors; which includes the time taken to detect failures, how often mistakes reoccur and how long mistakes last for. Given a set of QoS requirements the authors show how the parameters of failure detectors can be computed even when the probabilistic behaviour of these systems is not known.

However, as described earlier, this thesis only examines failure detectors that actively probe nodes and expect to receive an acknowledgement message in response. If an acknowledgement message is not received then a probe can be resent a number of times to reducing the probability of falsely detecting a failed node to a minimum. Therefore we do not evaluate our proposed failure detection algorithms based upon the occurrence of false detections as they can be effectively eliminated by simply tuning the number of times a probe is resent. We do however consider the detection time of failures as a key performance metric.

Liben-Nowell et. al in [36, 37] consider the problem of continuous churn in the Chord network and how to determine the rate at which the stabilization

procedure should be run. To determine the optimum they highlight how the join and leave rate of peers in the network must first be measured. They also hypothesize whether these measurements can be observed from the behaviour of neighbours.

Ginita and Teo in [22] address this question in the Chord DHT by allowing each peer to collect data from the network and adjust it's own stabilization rate accordingly. They assume nodes joining the DHT overlay are uniformly distributed over the identifier space and that each routing table entry i is accurately positioned at time T^i_{pin}. By modeling nodes joining the network as a Poisson process with rate λ, the probability of a routing entry becoming inaccurate at time t is dependent upon the distance of the current routing entry i from its ideal position $distance(i_{current}, i_{ideal})$, the size of the identifier space S_{size} and the time since the routing entry was known to be correct.

$$P^i_{inacc} = \frac{distance(i_{current}, i_{ideal})}{S_{size}} \cdot \lambda(t - T^i_{pin}) \tag{3.1}$$

Ghinita and Teo [22] also propose an adaptive keep-alive mechanism. Using an artificial exponential distribution to model node failure, their algorithm allows each node to measures the rate of churn in the network and adjusts the frequency of maintenance accordingly. In an exponential distribution the probability of a node being offline $P_{offline}$ given it was successfully contacted T_{since} seconds ago is given by (3.2) where μ is the rate of churn.

$$P_{offline}(T_{since}) = 1 - e^{-\mu(T_{since})} \tag{3.2}$$

$P_{offline}$ is used to estimate the probability of a lookup timing out and a keep-alive is scheduled to ensure this probability does not fall below a certain threshold P^{Thr}_{tout}. As exponential distributions are memoryless [58], $P_{offline}$ is

not dependent on T_{alive} the time a node has already spent online. As as a result all nodes will be probed with the same interval, dependant on the rate of churn μ and threshold P_{tout}^{Thr}, with the expected lifetime of each node being $1/\mu$.

Based upon their preliminary work in [40]; Castro et al. in [11] examine a self-tuning failure detection algorithm as part of the Microsoft Research implementation of Pastry. By estimating the number of nodes in the network and their failure rate μ, suitable probing periods can be set to meet a target failure detection delay. Once again however, the failure rate of nodes is modelled by a memoryless exponential distribution.

In [60], So and Sirer formally analyse the tradeoff between resource consumption and detection latency when creating multi-node failure detectors. They produce two optimal algorithms, given that the average session time of each neighbour is known. Their first algorithm $\sqrt{s} - LM$ minimises the latency or failure detection delay. Given a target bandwidth budget β, $\sqrt{s} - LM$ achieves the smallest average delay between failures occurring and their subsequent detection, giving an optimal probing period T_i^* as shown (3.3):

$$T_i^* = \frac{pq}{\beta}\sqrt{S_i}(\sum_{j=1}^{N}\frac{1}{S_j}), \forall i, 1 \leq i \leq N \tag{3.3}$$

Where p is the packet size of a keep-alive message, q the estimated number of packets needed to be sent and S_i the estimated session time (or lifetime time) of node i. Their second bandwidth minimising algorithm $\sqrt{s} - BM$ will ensure a specified target latency T_L is reached whilst consuming as small amount of bandwidth as possible. The optimal probing period T_i^* is given by:

$$T_i^* = \frac{2(T_L - r\triangle)(\sum_{j=1}^{N} \frac{1}{S_j})\sqrt{S_i}}{\sum_{j=1}^{N} \frac{1}{S_j}}, \forall i, 1 \leq i \leq N \tag{3.4}$$

Where r is the number of retries sent after a keep-alive message goes unacknowledged.

The key parameter the $\sqrt{s} - LM$ and $\sqrt{s} - BM$ algorithms need to estimate is each individual node's *session time* S_i, the time a node spends online in the network before leaving, which is referred to as *current lifetime* in [60]. The prediction mechanism in [60] estimates each node's current session time based upon it's previous session time(s). Knowledge of each individual node's previous session(s) may only be available in networks where nodes that reappear are likely to reconnect to previous neighbours. In networks such as Gnutella, node's often leave and reappear as new nodes [56]. Furthermore, as we investigate in the next chapter, while studies have shown that there exists a strong correlation between a node's previous and subsequent session times [66, 61], these correlations only exist when individual session times are less than one day and no correlation exists beyond this point [61].

3.5 Summary

This Chapter has surveyed several existing failure detection techniques. The predominant mechanism used within peer-to-peer network overlays is the standard keep-alive algorithm which causes a node to probe each connection it maintains at fixed intervals. Improvements to the SKA approach fall into two categories: predictive and gossip-based mechanisms. Predictive mechanisms attempt to observe the behaviour of peers in the network and adjust the rate of keep-alive messages accordingly. Whereas gossip-based approaches inform the neighbours of a failed node once its failure has been

detected.

It is clear that predictive mechanisms and gossip-based approaches are complimentary to one another. Predictive mechanisms attempt to detect failures more efficiently, either by reducing the bandwidth incurred by maintenance or detecting failures sooner. Once detected, knowledge of these failures can then be distributed by gossip-based approaches to other nodes within the network overlay. The next chapter surveys several measurement studies which seek to quantify and analyse the behaviour of peers in a range of overlay networks.

4

Characterising the Session Times of Nodes in P2P Networks

Understanding the characteristics of peer session times is important in at least two respects. Firstly, it allows researchers to better understand the complex process of individual nodes joining and leaving the network, known as *churn*. Knowing the rate at which nodes join the network and how long nodes tend to remain for; enables researchers to accurately model and therefore understand P2P networks. Secondly, being able to model how long peers remain in a network also enables us to describe when peers tend to leave. If

we are to improve failure detection in P2P networks beyond a fixed periodic approach, understanding when peers are most likely to leave a network is essential.

This chapter reviews several studies of P2P networks - focusing upon measuring the session times of peers - before explaining how information on how long a peer has spent in the network has been proposed as a heuristic for selecting routing table entries in existing work. We do not describe other aspects and features of P2P networks such as traffic patterns, the available content, the geographical distribution of peers, the latency between peers, bandwidth capacity and availability.

As an individual peer's *session* can be defined as the process of joining, participating and eventually leaving a network, a peer's *session time* is the elapsed time between a peer joining and subsequently departing a network [66].

4.1 Measurement Studies

Methods of measuring deployed P2P networks can be divided into at least two categories. The first is the active probing method which pro-actively contacts a subset of nodes within the network [55, 5, 9, 67, 66]. The second is the passive packet filtering method, which gathers data by intercepting messages or by peers voluntarily reporting their status [30, 57].

Active probing or crawling a network involves periodically polling peers within a P2P network and recording their responses. Crawlers either probe a fixed subset of peers or progressively crawl the network, contacting peers as they are discovered. By actively polling peers and learning their neighbours an entire network can be progressively crawled. However, this method of

measuring an overlay can be viewed as intrusive by clients who may receive many unsolicited messages. Whereas passive monitoring may involve intercepting data or studying network logs, which does not interfere or impact the peers themselves.

While both techniques allow us to determine which nodes were present within a network at a particular time; passive monitoring generally does not report the neighbours of nodes within the network. Therefore through actively probing peers and recording their neighbours a *snapshot* of the system at a particular time can be created. This snapshot can then be presented as a graph of peers as vertices and connections as edges as illustrated by Figure. 4.1. However, passive monitoring techniques may be viewed as less intrusive than active probing.

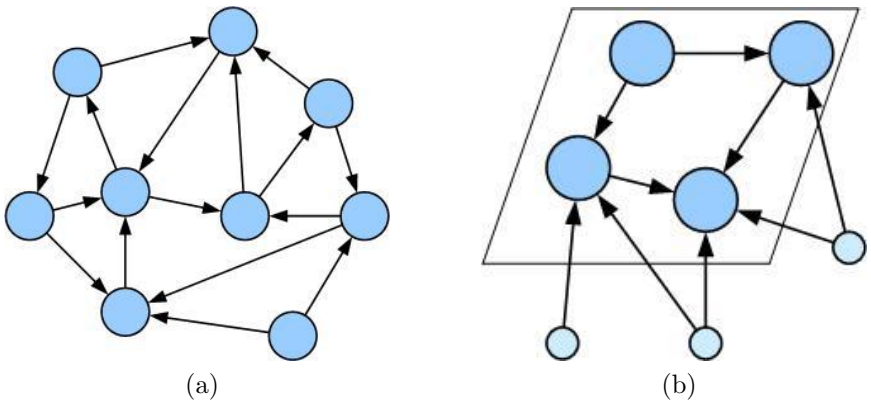


Figure 4.1: Unstructured and Semi-structured overlay topologies with peers represented as vertices/nodes and connections represented as edges.

A snapshot reveals the status of a network at a point in time, but capturing a snapshot is not an instantaneous process. As nodes can join and leave within a single snapshot, slow crawlers may inaccurately capture the network.

Saroiu et al. [55] in 2003, conducted one of the first studies reporting

upon peer availability in the Napster and Gnutella P2P networks. Measuring the lifetime of 7,000 Napster peers over 25 hours, probing them every 2 minutes and 17,125 Gnutella peers probed every 7 minutes over a period of 60 hours. They found that peer participation is similar in both Gnutella and Napster networks, with 50% of the session times of peers never exceeding an hour. The median session time for peers in both networks is also reported as approximately one hour.

Importantly, Saroiu et al in [55] identify an inherent limitation when measuring P2P networks for a finite length of time T. If we consider all the observed session times in a finite measurement window of any size we may bias our observations towards short-lived peers. While there are T opportunities to measure a session of single unit length, a session lasting T units must begin exactly at the start of the window and finish as the window ends. Furthermore, as only sessions that begin and end within the measurement window can be accurately accounted for, sessions longer than T can be counted but cannot be measured. If we were to include every observed node within the entire measurement window when calculating session lengths, we would create a bias toward short lived peers as we simply have more opportunities to observe them.

Saroiu et al, in [55], were the first to address this problem in a P2P network measurement context, employed the *create-based method*. By dividing the measurement window into two equal halves and only considering sessions that begin within the first half of the window; any session that begins and ends in the second half of the window is ignored. This results in all sessions less that $T/2$ being considered with equal opportunity as illustrated in Figure 4.2. Subsequent studies measuring session times in P2P networks have all utilised this method.

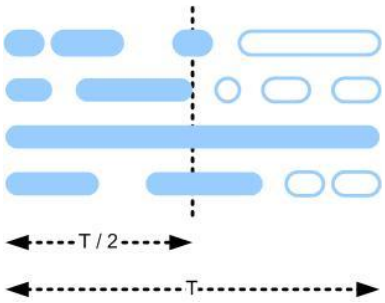


Figure 4.2: The create-based method only considers sessions that begin within the first half of a finite measurement window T, as indicated by the shaded bars.

Bhagwan et al. in [5], employed an active prober to investigate peer availability in the Overnet DHT network over the course of a seven day period. The *availability* of a peer is the percentage of time it is online and responding to traffic. They show that peer availability is largely interdependent between different peers. As a result, a small subset of peers are highly unlikely to be dependent on one another and therefore unlikely to all fail together. This is despite a diurnal pattern existing in the number of available peers and each peer's availability showing a strong correlation with the time of day.

However, availability is a fairly coarse grain measurement for the behaviour of different peers. For example a node with 50% availability during one day, may appear for a single twelve hour session or three separate times for four hours at a time. As Figure 4.3 illustrates a number of peers with varying behaviours can all have the same observed availability.

Sen and Wang [57] use passive monitoring to measure the FastTrack, Gnutella and DirectConnect P2P networks in 2001 at a single ISP. Although using this approach they were only able to track individual users by their IP address, they observe that 60% of IP addresses stayed in FastTrack for 10 minutes or less per day.

In [30] the author's analyse the BitTorrent tracker log of the Linux Red-

Figure 4.3: Five individual nodes are connected to a network during the time represented by the shaded areas. Each node is available 50% of the time despite having highly varying behaviour.

Hat9 distribution, which has now been made publicly available [29]. The RedHat9 tracker log contains statistics for a total of over 180,000 individual peers, with 51,000 of these appearing in the first five days. Although the trace data was recorded over a period of 5 months it contains a number of large gaps which go unreported in [30]. The largest continuous measured period is over 79 days.

A BitTorrent tracker acts as a centralised rendezvous point for a BitTorrent network. A tracker therefore passively monitors a BitTorrent network logging all this data, providing us with the arrival and departure times of peers to the nearest second. Each time a peer joins a tracker it is given a unique session id that allows individual sessions to be easily identified.

Although, the study presented in [30] does not solely focus upon characterising the distribution of session lengths. The study does report that clients which download the file in a single session take on average 8.1 hours, whereas multi-session downloads take 19.2 hours. The authors also report that 90% of non-completed sessions stay in the network less than 10,000 seconds.

As part of [9], Bustamante and Qiao measure the lifespan of over 500,000 Gnutella peers for 7 consecutive days every 21 minutes in 2003. They show that the distribution of session times can be modeled by a Pareto distribution. The authors highlight that the Pareto distribution belongs to the

used-better-than-new (UBNE) class of distributions, where the expected remaining lifetime grows as peers age. However, due to relatively high granularity of their measurements they could only observe peer's that remain in the system longer than 21 minutes.

Published in 2005, [67] is the first in a series of important measurement studies conducted by the authors Stutzbach and Rajaie, although earlier components of their work did appear in [65, 63, 64]. Their work in [67] presents *Cruiser* a fast and accurate network crawler which is used to trawl and then analyse the Gnutella network [23]. Cruiser can capture a complete snapshot of the Gnutella network, containing over a million peers, in just a few minutes by leveraging the two-tier topology of the Gnutella network. As shown in Figure 4.1b, all leaf peers must be connected to at least one ultrapeer. By only probing ultrapeers, the number of peers that need to be contacted to ensure a complete snapshot of the network is substantially reduced. To further improve speed, Cruiser also crawls hundreds of peers in parallel reducing the duration of a full network crawl.

Stutzbach and Rajaie extend their work in [66], to analyse the arrival and departure times of peers in the Gnutella network [23], Kademlia network [42] captured by their Cruiser network crawler [67], whilst also analysing multiple BitTorrent tracker logs [13]. First they examine inter-arrival time, the time that passes from the start of one peer's session to the start of the next session. Measuring the inter-arrival time of peers allows us to describe the rate at which peers join the network. Stutzbach and Rajaie highlight that it is not possible to accurately analyse inter-arrival time using an active probing techniques such as Cruiser. Despite Cruiser minimising the time between consecutive snapshots several minutes may have elapsed, during this time tens of thousands of new peers can join the network causing the granularity

of measurements to be quite coarse.

To overcome this deficiency Stutzbach et al. utilise the publicly available tracker logs from the RedHat9 BitTorrent distribution [29], previous studied in [30]. In this respect, the granularity of BitTorrent tracker logs often compares favorably to active probing based techniques. As crawlers must progressively probe the network, the more nodes a crawler incorporates into a single snapshot the larger the interval between successive snapshots becomes. This interval currently ranges anywhere between four and thirty minutes [66, 40, 67, 55]. By analysing the tracker logs, the authors find that peer arrivals are not completely independent, with peers more likely to be active during certain times of the day. As a result, Weibull distributions were found to provide a better fit when modeling the inter-arrival time than the typically used exponential distributions.

Active probing methods however are suited to measuring the session times of peers, it is upon this feature of peer behaviour that Stutzbach and Rajaie in [66] highlight several important characteristics. Firstly the authors compare the distribution of session lengths captured at different times from within a single system, showing that these distributions do not change significantly over time. They also show that these distributions are similar across different systems. This indicates that user-behaviour - the driving factor behind the session length distributions - is consistent across several P2P systems. Critically, the authors highlight that the distribution of session lengths clearly does not follow a exponential distribution as commonly used to model P2P networks, nor are the distributions heavily tailed as previous studies report [9, 34]. Instead the authors explain how two alternative models better describe their observations:

"In summary, while most sessions are short (minutes), some ses-

> sions are very long (days or weeks). This differs from exponential distributions, which exhibit values closer together in length, and heavy-tailed distributions, which have more pronounced extremes (years). The data is better described by Weibull or log-normal distributions."
>
> Understanding churn in Peer-to-Peer Networks
> Daniel Stutzbach and Reza Rajaie.

Weibull distributions are a more flexible alternative to exponential distributions. Stutzbach and Rajaie explore the consequences of these findings by examining the distribution of peer uptime and uptime predictability. Their findings show that at any single point in the captured network traces the majority of peers in the system are long-lived peers. However, as short-lived peers join and leave the system so frequently they constitute a relatively large portion of the overall number of sessions.

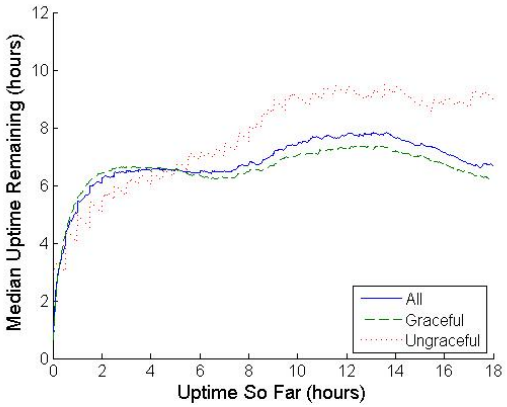


Figure 4.4: Remaining Uptime.

Importantly for our study, the authors then analyse whether a peer's *current uptime*, the time it has spent in the network already, is a good predictor of it's remaining uptime. As the BitTorrent RedHat trace dataset used in [66]

is currently available from [29] we were able to recreate some of their observations. Figure 4.4 shows the complementary cumulative distribution (CCDF) function of the median remaining uptime as a function of a peer's current uptime. As it is not possible to calculate the length of very long sessions that either begin and/or end outside of measurement window, we cannot compute the mean session length. However as it is possible to count these sessions we are able accurately calculate the median session length. From Figure 4.4, we can observe that the median uptime remaining increases quickly as a peer's uptime reaches one hour, but beyond this point once peers have been online for two hours or more all have approximately the same remaining uptime. Stutzbach and Rajaie show that while uptime is in general a good predictor of remaining uptime, its strength varies across different systems, they summarise:

> "Our results show that while uptime is on average a good indicator of remaining uptime, it exhibits high variance. Therefore it should only be used when a bad prediction does not have a major cost but making better choices on average improves overall performance."
>
> Understanding churn in Peer-to-Peer Networks
> Daniel Stutzbach and Reza Rajaie.

Finally, Stutzbach and Rajaie turn their attention to whether the session times of peer's that appear multiple times are correlated. Although the authors show that past session length is a good indicator of a peer's next session time in both Kad and Gnutella, their observations only consider sessions that last for less than a single day.

Recent work by Steiner et al. in [61], continues in the same vein of research as Stutzbach and Rajaie in [66, 67] and significantly extends their findings.

Steiner et al. crawl part of the Kademila based DHT known as KAD every five minutes for six consecutive months. KAD is part of the much larger P2P file sharing system eDonkey, which served several million users. Steiner et al. observe the KAD network in a well established steady state, with an equal amount of peers arriving and departing.

Interestingly, they observe that KAD identifiers, once presumed to be persistent to a single user, are in fact frequently changed by some users who wish to improve their anonymity. This issue added to the fact that peers may also regularly change their IP addresses makes tracking individual peers over multiple sessions particularly difficult. Furthermore, they observe $250,000$ KAD instances with identifiers systematically covering the entire identifier space all belonging to the company Media Defender. Allowing the company to monitor the activities of all peers within the KAD network.

Steiner et al. examine if the consecutive sessions of individual peers are correlated in length. In other words, does the length of a peer's previous sessions have a bearing on the length of future sessions. Crucially, the authors find that when taking all session length samples almost no correlation exists between consecutive sessions. However, when only considering sessions that last up to a single day then a considerable positive correlation emerges, as previously reported in [66]. These observations therefore suggest that failure detection mechanisms, such as [60], that depend upon previous session lengths correlating with future session lengths are not only difficult to track, due to peers wishing to retain their anonymity, but also are limited in their application to sessions that last a single day.

Previously Stutzbach and Rajaie in [66] found that the distribution of session lengths exhibits a considerable tail, to which a Weibull distribution proves a very good fit. Steiner et al. in [61] whose measurement window ex-

tends to six months confirm this point. The authors explain the consequences of this finding:

> "The Weibull distribution has two parameters $\alpha > 0(shape)$ and $\lambda > 0(scale)$. The Weibull distribution with $\alpha < 1$ is part of the class of the so-called sub-exponential distributions, for which the tail decreases more slowly than any exponential tail [25]. Sub-exponential distributions are a subclass of the class of heavy-tailed distributions [7]. This implies that knowing the past (uptime) of a peer allows us to predict the future (residual uptime)."
>
> Long Term Study of Peer Behaviour in the KAD DHT
> Moritz Steiner, Taoufix En-Najjary and Ernst W. Biersack.

In other words, the longer a node has been online in the system the more likely it is to remain online in the future. Such distributions can be characterised as UBNE (used-better-than-new-in-expectation) as older peers tend to remain longer in the network than their younger counterparts. From these findings, the authors show that the a peer that has been online for a time of length T_{alive}, can be expected to remain online for a duration that is in order of $O(T_{alive}^{1-\alpha})$.

While knowledge that peers who have been in the network for some time are more likely to remain than peer's who have just joined, has only been recently described formally by studies such as [66, 61]; it has been generally accepted for some time. In fact, eMule chooses to only publish content on peers that have been online for at least two hours for this reason. Indeed, current uptime has become a common heuristic to select routing table entries upon. The next section reviews several pieces of existing work that utilise a peer's current uptime to improve some aspect of network performance.

4.2 Existing Uses of Current Uptime within P2P Networks

As outlined in the previous section, Bustamante and Qiao [9] fit their observations of the session times of over 500,000 Gnutella peers, over the course of 7 days to a Pareto distribution. They use this observation to justify protocols that select the oldest available nodes as neighbours, Bustamante and Qiao [9] show that these connections tend to last longer and therefore need to be replaced less often, producing more robust networks. Using trace-driven simulations they show simple lifespan-based protocols can reduce the number of connection failures by over 42%, when compared to the standard random neighbour selection scheme. Extending their own work, Bustamante and Qiao in [10] also show the performance advantages of session-time based query related strategies.

As part of [33], Ledlie et al. also state that current uptime is an effective predictor of remaining uptime. Again using trace-drive simulations, this time with the data gathered by Sariou et al in [55], the authors compare neighbour selection strategies based upon proximity (i.e latency) and current uptime. They report selecting the oldest available peers as neighbours reduces maintenance messages by 42% at the expense of increasing latency by 50% when compared to a proximity-based strategy.

In [24] Godfrey et al. show selecting peers on the basis of longest uptime offers better performance than random selections and pre-compiled preference lists by reducing the effects of churn. As node's with larger current uptimes can be expected to remain in routing tables longer and therefore be replaced less often, the effects of churn are minimised. Interestingly however, the authors also highlight that random replacement strategies perform

surprising well and generally outperform strategies that use pre-compiled preference lists. This can be explained by random selection eventually selecting long-lived neighbours, as there are more opportunities to do so, and thereby acting like schemes that select neighbours on the basis of current uptime. Whereas schemes that select neighbours based upon other metrics such as proximity or from an overly restrictive identifier-space may actively exchange long-lived neighbours for shorter-lived replacements and as a result increase the effects of churn.

Work by Li et al in [34], proposed a DHT based protocol called Accordion which allows each node to expand and contract it's routing table dependent on a internal bandwidth budget. Accordion nodes attempt to find a balance between maintaining large and expensive routing tables that quickly resolve lookups and smaller cheaper routing tables that resolve lookups less efficiently. Although Accordion is not presented as a failure detection algorithm, it does control the size of a node's routing table by evicting neighbours when the estimated probability of a node being online P_{online}, based upon a node's current uptime, falls below than some threshold P_{thresh}.

Accordion calculates the probability of a node being online explicitly by assuming each node's session time is drawn from a heavy-tailed Pareto distribution as reported by [55] and shown in (4.1). In such a distribution the probability of a node leaving a network before time t is given by:

$$P(session < t) = 1 - (\frac{\lambda}{t})^{\alpha} \tag{4.1}$$

Where α is the shape parameter and λ is the scale parameter. Li et al in [34] highlight that P_{online} is dependent not only T_{since}, but also conditional upon T_{alive}, the time a node has already spent alive in the network referred to as a node's *current uptime*. Li et al continue and explicitly calculate P_{online}

as:

$$P_{online} = (\frac{T_{alive}}{T_{alive} + T_{since}})^{\alpha} \tag{4.2}$$

Equation (4.2) shows that if session times of nodes are drawn from a Pareto distribution as described by equation (5.7) then the probability, P_{online}, that a node is still online after a certain duration, is dependent upon the time a node has already been in the system T_{alive}. Essentially the longer a node has been online the more likely it will still be online some time in the future.

4.3 Summary

This Chapter has explained how several studies have highlighted how current uptime can be used to estimate the likelihood of nodes leaving a P2P network. Although it's widely accepted that long-lived peers are more likely to remain in a network longer than their short-lived counterparts; up until now, current uptime has only been used as a heuristic to select routing table entries. While this has proven to create more robust networks, it's not the only potential application of current uptime. The next Chapter details how current uptime can be used to measure the likelihood of a peer being online and how this prediction can be implemented within three alternative failure detection algorithms.

5

Using Current Uptime to Predict a Peer's Online Status

In any reasonable node session time distribution; as the amount of time since a node has been last observed T_{since} increases, the probability P_{online} that the node is still online decreases. In other words, the longer it has been since we last contacted a node the more likely it is to have left the network.

Early studies of popular P2P networks reported node session times could be modeled by an exponential distribution. Importantly exponential distributions are well-known to be *memoryless*, meaning the probability of a node

being online is not dependent on the time a node has already spent in the network T_{alive}. We can show this by calculating the conditional probability of a node being online P_{online} given it has already spent time in the network T_{alive} and we last observed it online T_{since} seconds ago, as shown in (5.1):

$$P_{online} = P(lifetime > (T_{alive} + T_{since}) | lifetime > T_{alive}) \tag{5.1}$$

Using an exponential distribution (5.2) with rate μ and Bayes theorem (5.3)

$$P(lifetime > t) = 1 - P(lifetime \leq t) = 1 - (1 - e^{-\mu t}) \tag{5.2}$$

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B|A)P(A)}{P(B)} \tag{5.3}$$

Clearly, the probability of a node being online at least T_{alive} given the node has been alive at least $T_{alive} + T_{since}$, $P(B|A)$, is equal to one. Using (5.2) to produce (5.4), (5.5) and (5.3) we find:

$$P(A) = 1 - P(lifetime \leq (T_{alive} + T_{since})) = 1 - (1 - e^{-\mu(T_{alive}+T_{since})}) \tag{5.4}$$

$$P(B) = 1 - P(lifetime \leq T_{alive}) = 1 - (1 - e^{-\mu(T_{alive})}) \tag{5.5}$$

$$
\begin{aligned}
P_{online} &= \frac{P(B|A)P(A)}{P(B)} \\
&= \frac{1 * (1 - (1 - e^{-\mu(T_{alive}+T_{since})}))}{1 - (1 - e^{-\mu(T_{alive})})} \\
&= \frac{e^{-\mu(T_{alive}+T_{since})})}{e^{-\mu(T_{alive})}} \\
&= \frac{e^{-\mu T_{alive} - \mu T_{since}}}{e^{-\mu T_{alive}}} \\
&= \frac{e^{-\mu T_{alive}} \cdot e^{-\mu T_{since}}}{e^{-\mu T_{alive}}} \\
&= e^{-\mu T_{since}} \qquad (5.6)
\end{aligned}
$$

However, more recent studies show [9, 66, 67] that the distribution of session times can be more accurately described by Pareto or Weibull distributions. As Li et al. in [34] use a Pareto distribution to highlight, P_{online} is dependent not only on T_{since}, but also upon T_{alive}, the time a node has already spent alive in the network.

$$P(session < t) = 1 - (\frac{\lambda}{t})^{\alpha} \qquad (5.7)$$

However, a non-shifted Pareto distribution such as (5.7) cannot account for samples less than λ. While a shifted Pareto distribution does not have this limitation, we instead focus on Weibull distributions which accurately describe peer session times as shown independently by both Stutzbach and Rajaie and Steiner et al, in [66] and [61] respectively.

A Weibull distribution, as shown in (5.8), is commonly used to model lifetimes in reliability engineering due to their flexibility and versatility. The shape α and and scale λ parameters can be used to describe exponential distributions when $\alpha = 1$.

$$P(lifetime \leq t) = 1 - e^{-(t/\lambda)^{\alpha}} \tag{5.8}$$

Using equation 5.8 to produce equation 5.9 and equation 5.10 we use equation 5.3 to calculate:

$$P(A) = 1 - P(lifetime \leq (T_{alive} + T_{since})) = 1 - (1 - e^{-((T_{alive}+T_{since})/\lambda)^{\alpha}})) \tag{5.9}$$

$$P(B) = 1 - P(lifetime \leq T_{alive}) = 1 - (1 - e^{-((T_{alive})/\lambda)^{\alpha}}) \tag{5.10}$$

$$\begin{aligned} P_{online} &= \frac{P(B|A)P(A)}{P(B)} \\ &= \frac{1 * (1 - (1 - e^{-((T_{alive}+T_{since})/\lambda)^{\alpha}}))}{1 - (1 - e^{-((T_{alive})/\lambda)^{\alpha}})} \\ &= \frac{e^{-((T_{alive}+T_{since})/\lambda)^{\alpha}}}{e^{-((T_{alive})/\lambda)^{\alpha}}} \end{aligned} \tag{5.11}$$

The basic idea of this work is to regularly examine each connection a node maintains and only send keep-alive messages once these connections are likely to have failed. As these equations show, failure detection mechanisms based upon exponential distributions only send keep-alive messages according to the time since a peer was last contacted, T_{since} and the rate of churn μ. This in essence causes all peers to be probed at the same periodic interval [22, 11].

However, as Weibull distributions more accurately characterise the observed distributions of peer session times, the probability of a node being

online has been shown to be also dependent on the time it has spent online T_{alive}. We therefore propose three adaptive failure detection algorithms which each send keep-alive messages based upon the probability of a node having failed dependent on the time it has spent in the network. While we utilise a Weibull distribution to calculate the likelihood of a node being online, as this distribution has been shown to accurately describe node session times, in essence any suitable distribution could be used for our algorithms to function.

5.1 The ProbKA Algorithm

Similar to the standard keep-alive algorithm, the Probabilistic Keep-Alive (ProbKA) algorithm specifies a regular interval k for all connections maintained by a single node. Although the interval k is the same size for all a node's connections, each routing table entry is updated independently.

Once this interval has expired the ProbKA algorithm examines the individual connection and determines the likelihood that the corresponding neighbour has failed, for the sake of clarity the pseudo-code is shown in Algorithm 2. With probability $P_{offline} = 1 - P_{online}$, where P_{online} is given by (5.11), a keep-alive message is sent to the corresponding node to determine it's online status.

Algorithm 2 Prob_Offline(T_{alive},T_{since})

return $1 - ((e^{-((T_{alive}+T_{since})/\lambda)^{\alpha}})/(e^{-((T_{alive})/\lambda)^{\alpha}}))$

The ProbKA algorithm has several advantages over the standard keep-alive algorithm. Firstly it is adaptive; as nodes spend more time in the network the likelihood of them remaining in the network gradually increases. Keep-alive messages are sent stochastically causing fewer to be sent as nodes

age and are perceived to become more reliable. Secondly it can be easily tuned to network conditions by using suitable parameters λ and α which define the distribution of node session times. The pseudo-code for the ProbKA algorithm is given in Algorithm 3.

Algorithm 3 ProbKA()

```
for all i in routing_table do
  if T^i_since ≥ k then
    P_offline = Prob_Offline(T^i_alive,T^i_since)
    {generate random number rand between [0 1]}
    if P_offline ≥ rand or (1 − P_offline) > P_thresh then
      T^i_since = 0
      probe i
    end if
  end if
end for
```

However by extending the intervals between keep-alive messages we are also potentially extending the delay between a failure occurring and it's subsequent detection. While the SKA algorithm can guarantee that all failures will be detected after at most k seconds, the ProbKA algorithm as it stands does not. As T_{since} grows the probability of not sending a keep-alive message grows increasingly small but is always non-zero. To ensure the failures do not go undetected a maximum interval should be set after which a keep-alive message must be sent.

Furthermore, the ProbKA algorithm in it's simplest form has no lower bound on the probability that keep-alive messages should be sent. It's likely that application designers may wish to specify a minimum likelihood of a neighbour being online by setting a threshold P_{thresh}. Once the probability that a node remains online drops below P_{thresh} a keep-alive message is then sent to ensure it's still alive.

5.2 The PredKA Algorithm

The second proposed algorithm, the Predictive Keep-alive (PredKA), gradually increases the size of keep alive period based upon the likelihood of nodes being online in the system increasing as nodes age. While the ProbKA algorithm examines each connection at regular intervals and determines whether a keep-alive should be sent, the PredKA algorithm defines the size of the next interval after which a keep-alive must be sent.

An advantage of the PredKA algorithm over the ProbKA approach is that it does not rely on a stochastic process to determine when the next keep-alive message is sent. As the PredKA algorithm schedules the next keep-alive message to be sent at some point in the future we know precisely when it will occur, whereas the ProbKA algorithm may or may not send a keep-alive message at the next opportunity to do so. The disadvantage of this is once a connection has failed the interval until the next keep-alive message may be very large, while the ProbKA algorithm has regular intervals at which a keep-alive message may be sent.

Using the Weibull distribution as described by (5.8) and probability of nodes remaining online given by (5.11) as a basis; we can ensure that any network message sent between two connected nodes will be delivered with probability of at least P_{target}, whilst adjusting the keep alive period according to value of T_{since} as given by:

$$
\begin{aligned}
P_{target} &= \frac{e^{-((T_{alive}+T_{since})/\lambda)^{\alpha}}}{e^{-((T_{alive})/\lambda)^{\alpha}}} \\
logP_{target} &= log[\frac{e^{-((T_{alive}+T_{since})/\lambda)^{\alpha}}}{e^{-((T_{alive})/\lambda)^{\alpha}}}] \\
&= -(\frac{T_{alive}+T_{since}}{\lambda})^{\alpha} + (\frac{T_{alive}}{\lambda})^{\alpha} \\
(\frac{T_{alive}+T_{since}}{\lambda})^{\alpha} &= logP_{target} - (\frac{T_{alive}}{\lambda})^{\alpha} \\
(\frac{T_{alive}+T_{since}}{\lambda}) &= [logP_{target} - (\frac{T_{alive}}{\lambda})^{\alpha}]^{\frac{1}{\alpha}} \\
&= [logP_{target} - (\frac{T^{\alpha}_{alive}}{\lambda^{\alpha}})]^{\frac{1}{\alpha}} \\
&= [\frac{\lambda^{\alpha} \cdot logP_{target} - T^{\alpha}_{alive}}{\lambda^{\alpha}}]^{\frac{1}{\alpha}} \\
&= \frac{1}{\lambda}[\lambda^{\alpha} \cdot logP_{target} - T^{\alpha}_{alive}]^{\frac{1}{\alpha}} \\
T_{alive}+T_{since} &= [\lambda^{\alpha} \cdot logP_{target} - T^{\alpha}_{alive}]^{\frac{1}{\alpha}} \\
T_{since} &= [\lambda^{\alpha} \cdot logP_{target} - T^{\alpha}_{alive}]^{\frac{1}{\alpha}} - T_{alive} \qquad (5.12)
\end{aligned}
$$

Using (5.12), the PredKA algorithm defines a time in the future, T_{since} seconds away, to send the next keep-alive message. The PredKA algorithm explicitly calculates the size of the interval until the next keep-alive message based upon the time a node has spent in the network T_{alive} and the target likelihood that the corresponding node will remain in the network P_{target}. As this likelihood increases, as T_{alive} increases, nodes send fewer keep-alive messages to their longer-lived neighbours. Algorithm 4 details the pseudo-code for the PredKA algorithm.

Algorithm 4 PredKA()

if keep_alive message acknowledged from i **then**
  $k = [\lambda^{\alpha} \cdot logP_{target} - T^{\alpha}_{alive}]^{\frac{1}{\alpha}} - T_{alive}$
end if

However, the PredKA algorithm also suffers from the same limitation as the ProbKA algorithm described earlier. To ensure failures do not go undetected beyond a certain acceptable threshold a maximum interval size needs to be defined.

5.3 The BudgetProb Algorithm

The third algorithm we propose is the Budget probabilistic algorithm (BudgetProb), which maintains the connections of each node according to a bandwidth budget β. While the SKA algorithm can be thought of as dividing bandwidth equally amongst all connections, the BudgetProb algorithm divides the bandwidth budget proportionality to each connection, dependant on the likelihood of individual connections failing. The BudgetProb algorithm allocates connections that are likely to fail more of the overall bandwidth budget. These connections therefore benefit by having smaller keep-alive intervals resulting in a reduction in the average failure detection delay.

In this thesis we experiment with several values of the bandwidth budget β, equivalent to the bandwidth consumed by the standard periodic approach using a range of keep-alive intervals.

At a regular interval r each node calculates for each neighbour $P^i_{offline}$ the probability of neighbour i failing after the next r seconds. We experimented with several values of r and found $r = 120$ seconds to perform well. The BudgetProb algorithm specifies the interval for each connection according to (5.13), where p is the packet size of keep-alive message and the pseudo-code for which is shown in Algorithm 5.

$$k = \frac{(p * 2)}{\beta} / \frac{P^i_{offline}}{\sum_{j=1}^{N} P^j_{offline}} \forall i, 1 \leq i \leq N \tag{5.13}$$

Algorithm 5 BudgetProb()

```
Norm = 0
for all j in routing_table do
  Norm = Norm + Prob_Offline(T^j_alive,T^j_since)
end for
for all i in routing_table do
  k = ((p * 2)/β) / (Prob_Offline(T^i_alive, T^i_since) / Norm)
end for
```

5.4 Gossiping Failures

To further investigate our predictive algorithms we also implemented a simple gossip mechanism to further reduce the incurred failure detection delay. This mechanism, which is also applied to the SKA approach, shares information regarding node failures with their mutual neighbours.

As illustrated in Figure 5.1 each time that node X sends a probe to node Y it also learns of all Y's neighbours $n(Y)$. If node X discovers Y has failed it then informs other neighbours in $n(Y)$ of this event. These mutual neighbours then immediately probe Y themselves to ensure news of the failure is correct without informing others of the outcome. Although node X may have an outdated view of $n(Y)$, nodes that are not informed of Y's failure will eventually detect it themselves. This simplistic gossip-based mechanism shouldn't be considered novel, further examples of alternative and more complex sharing-based, gossip-based and flooding-based mechanisms are evaluated in [70, 16].

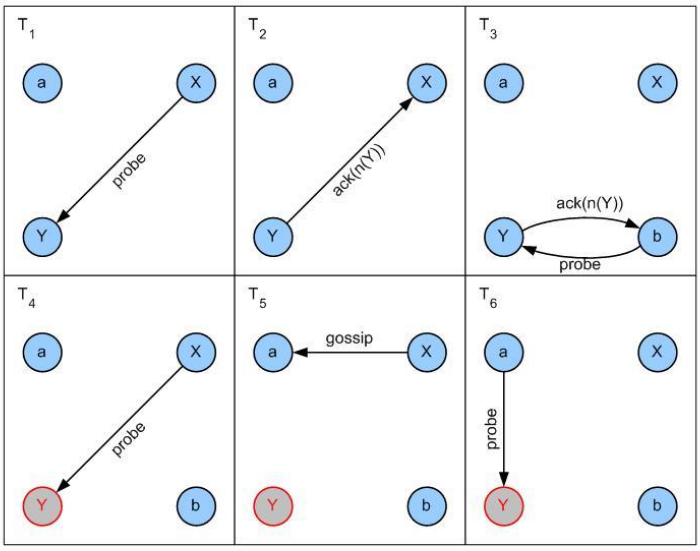


Figure 5.1: Gossiping failures to known neighbours as they are detected.

While gossip-based and similar mechanisms reduce the failure detection delay this comes at the cost of increased control overhead. Additional messages are required to inform mutual neighbours, who themselves check a node has failed. The next section details our experimental methodology which we use to analyse and compare the algorithms described above.

6

Extending Keep-Alive Intervals in Unstructured P2P Overlay Networks

Typically P2P networks predefine a fixed keep-alive period k, a maximum interval in which connected nodes must exchange messages. If no other message has been sent within this interval then keep-alive messages are exchanged to ensure the corresponding node has not left the system.

P2P systems maintain links according to a fixed periodic interval to detect node failures in a predictable and timely fashion. Keep-alive messages act as a effective and proactive recovery mechanism replacing broken connec-

tions before they are needed by the network. However, defining a suitable interval between keep-alive messages is dependant upon the rate of *churn*, the collective effect of many nodes joining and leaving a network in parallel. Churn itself is a poorly understood process resulting in the interval between keep-alive messages often being determined by rules of thumb. Although each keep-alive message is relatively small, around 40 bytes, they may be sent as frequently as once every 30 seconds for each connection a node maintains. Keep-alive messages can be seen as the cost of connections whilst they are inactive.

We propose extending these intervals gradually as connections between nodes age. Studies have shown the more time a node has spent in the network the more likely it is to remain in the system in the future. Therefore as the estimated reliability of nodes increases we seek to reduce the traffic overhead of each individual connection.

However, extending the intervals between maintenance messages alone may causes failures to mount up, reducing the efficiency of the overall network and increasing the chance of partitioning. When failures do occur, connections need to be replaced in a timely fashion to avoid being forcefully disconnected from the network. We investigate these trade-offs by comparing three alternative maintenance strategies using simulations based upon measured network data.

6.1 Experimental Methodology

In order to accurately evaluate and compare the mechanisms described above our simulations are based on real data from a P2P network. Using a publicly available [29] and well researched BitTorrent tracker log [66, 30], we simulate

the peers of the RedHat9 BitTorrent network as they appear during a portion of the five month logged period. Furthermore we also use real network data covering a thirteen day period in March 2009 from LegalTorrents.com.

At the time the RedHat9 torrent was released BitTorrent clients only supported single file downloads, which may suggest that node session times would be similar across all nodes, with each node downloading the same file and leaving the network shortly afterwards. However this is not the case, the RedHat9 tracker log contains statistics for over a $180,000$ individual peers in total, with a large proportion of these being short-lived sessions with just 19% of sessions eventually completing the file transfer [30]. Many studies have shown short-lived peers tend to make up a large proportion of sessions in many different P2P networks [66, 55]. To further avoid any bias we also utilise the LegalTorrents data that includes nearly 3,000 file distributions with over 100,000 individual sessions using almost 50 unique clients.

Stutzbach et al. in [66] highlight that while current uptime is on average a good predictor of remaining uptime it exhibits high variance. The consequences of inaccurately predicting a node's remaining session time in this context may result in an increased delay in detecting a failed node. The cost of this increased failure detection delay is application-specific. Applications that cannot afford messages to timeout, due to latency requirements or the shear size of messages, should not purely use uptime as the basis of their maintenance algorithms. However, applications that are more resilient to latency and churn can afford to increase the potential failure detection delay and thereby reduce bandwidth spent on maintaining inactive connections.

A BitTorrent tracker acts as a centralised rendezvous point for a BitTorrent network, with nodes contacting the tracker upon joining, sending periodic updates and, if they leave gracefully, informing the tracker upon

their departure. As the tracker logs all this data it provides us with the arrival and departure times of actual peers to the nearest second. This enables us to model the process of churn both accurately and realistically.

By processing the tracker logs we can determine when any graceful node joins and subsequently leaves the network during the torrent's lifetime. Although the tracker cannot detail the departure time of ungraceful nodes, as all nodes update their progress periodically at thirty minute intervals we can safely assume they leave at most thirty minutes after their last update. We do not exclude ungraceful nodes from our simulation, instead we add a uniformly random time of at most thirty minutes since they last updated their progress in order to determine when they leave the simulated network. We also utilise a fail-stop model in which all nodes graceful and ungraceful do not inform their neighbours upon departing the network.

As this work solely focuses upon maintenance messages the model does not replicate any P2P lookup, routing or file distribution algorithms. Instead we simulate an unstructured network containing the nodes as they appear in the Redhat9 BitTorrent and LegalTorrents networks during a portion of their logged period. We use the tracker log solely to specify when each node joins the simulated network and subsequently leaves.

We also do not simulate latency, the elapsed time from a message between two peers being sent and subsequently received. Firstly as we have no information regarding the actual latency between peers within the tracker logs, which would mean any simulated latency would have to be artificially generated. Secondly any simulated latency only add a fractional amount to the time taken to detect failures by all the simulated algorithms. As this thesis seeks to analyse and compare several failure detection algorithms, we focus upon features of P2P networks that may differentiate the alternative

approaches.

The tracker log does not provide us with the connections each node creates and maintains while part of the network. Whilst online each node creates and maintains a fixed number of connections D with other existing nodes selected at random, all our experiments set $D = 30$. As we only simulate maintenance messages we believe this work is general enough to be applied to any type of P2P network overlay.

Our experiments simulate the first five days of the LegalTorrents data and the largest continuous measured period of RedHat9 BitTorrent network. The simulation begins cold, i.e without any peers. The first twelve hours of the network then allows nodes to populate and leave the network according the events given by the trace. Once this period is finished each node then creates D connections with existing nodes and we report the maintenance of these connections over the subsequent four and a half simulated days. The results we present below are averaged over a series of ten experiments each. Furthermore, all our algorithms P_{online} is estimated by $P_{online} = \frac{e^{-((T_{alive}+T_{since})/\lambda)^{\alpha}}}{e^{-((T_{alive})/\lambda)^{\alpha}}}$.

We acquire the parameter values of α and λ by fitting the observed session times from the BitTorrent tracker logs to a Weibull distribution. After parsing the largest continuous stretch of the tracker log we find the session lengths, down to the nearest second, of each node. Then using the create-based method, as described in Chapter 4, we select only the nodes that begin in the first half of the measurement window. We also dismiss all the ungraceful peers and peers that only appear once, i.e with a session length equal to zero. By discarding the sessions that begin in the second half of the window, we avoid biasing our measurements with too many short-lived peers. Ungraceful peers are discarded as we cannot accurately measure their session times. The Perl script created to parse the BitTorrent tracker logs is

availabile from [46].

$$f(x|\alpha,\lambda) = \lambda\alpha^{-\lambda}x^{\lambda-1}e^{-\frac{x}{\alpha}^{\lambda}} \quad I_{1,\infty}(x) \tag{6.1}$$

Finally we search the landscape of potential values of α and λ, using maximum likelihood estimation by calculating the probability of each data point x given each possible pair of parameter values as shown in (6.1). Selecting the α and λ values that maximise the likelihood of all the data points. This provides us with the model's parameters of $\alpha = 0.39$ and $\lambda = 3962$ for the RedHat network data and $\alpha = 0.41$ and $\lambda = 2632.25$ for the LegalTorrents data. A table of experimental parameters used in this Chapter is provided below:

Parameter	Value
α	0.39, 0.41
β	20, 10, 9, 8, 7, 6, 5, 2.5
Degree	30
δ	1, 2, 3
k	120, 240, 480, 960, 1920
λ	3962, 2632.25
p	40
P_{thresh}	0.99
P_{target}	0.99, 0.98, 0.97

6.2 Results

We evaluate each mechanism based upon two main criteria:

- **Cost:** The average bandwidth consumed per node per second online. Formally the cost C is equal to $\frac{(s+a)\cdot p}{T}$; where s and a are the number

of keep-alive messages sent and acknowledged respectively, p is the size of a keep-alive message and T the sum of all node session times.

- **Failure detection delay:** The mean and median time that elapses between a failure occurring and subsequently being detected.

These performance metrics have also been used in other evaluations of failure detection algorithms including [60, 70, 12]. We set the cost of a single keep-alive message to be 40 bytes, which is equivalent to header of a IP packet containing a TCP segment of size 0. The higher the cost value the more bandwidth each node is consuming while part of the network. As in [35] we ignore the cost of storing each node's routing table as communication is typically considered to be far more expensive than the storage.

Unlike in [70], but as in [35], we do not examine the issue of packet loss and as a result we do not examine the probability of detecting false positive node failures. In practice lost keep-alive messages are simply resent a number of times at short intervals once the round trip timeout (RTO) has expired. Typically after three resent probes a node can safely be considered to have failed. Therefore the issue of packet loss simply adds a small but constant amount of time to the incurred detection delay of all failure detection algorithms. Accordingly the simulator used in this thesis does not model packet loss and nodes are considered to have failed after the first keep-alive message goes unacknowledged. Unless gossip mechanisms are being investigated; packet loss should not present a significant problem, as the number of times a probe is resent can be tuned to effectively eliminate the probability of falsely detecting node failures. Gossip mechanisms add the further complication of when to inform the other neighbours of a node's failure [70].

Figure 6.1a compares the cost incurred by our probabilistic keep-algorithm (ProbKA) with a probability threshold $P_{thresh} = 99\%$ and the widely used

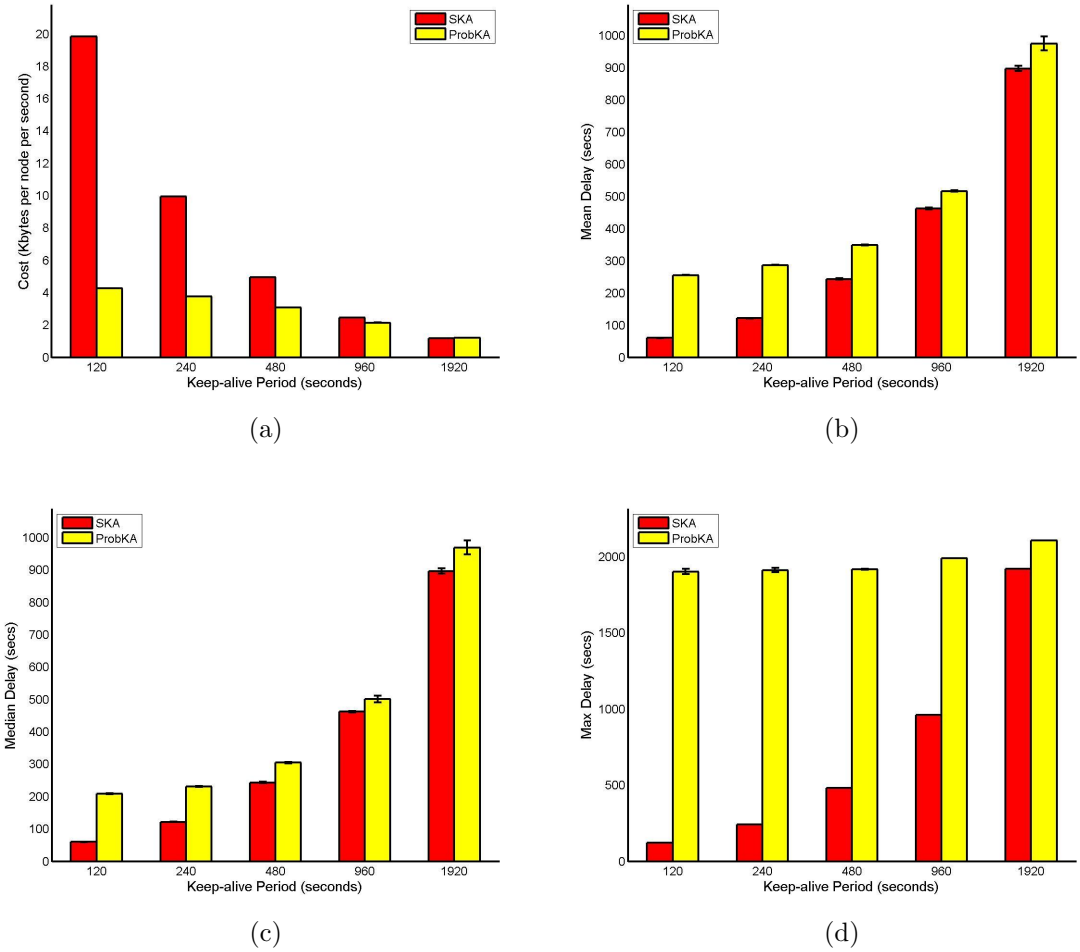


Figure 6.1: Performance comparison of the SKA and ProbKA algorithms in terms of cost and mean, median and maximum delay.

standard keep-algorithm (SKA). The results shows our ProbKA algorithm significantly reduces the cost in terms of bandwidth each node incurs while part of the network. By only sending a keep-alive message when a node is likely to be offline the ProbKA algorithm reduces the number of messages sent and therefore acknowledged. As the PredKA algorithm has no keep-alive period k parameter it cannot be compared side-by-side to the SKA algorithm in this fashion.

Nodes actually gain very little information from acknowledged keep-alive messages. They only serve to inform a node that it's neighbour was online at the time of receipt. It does not guarantee that node will remain online or that the connection will still be active if data is sent along it. An acknowledged keep-alive message merely informs the SKA algorithm that a connection does not currently need to be replaced. While an acknowledged keep-alive message also updates the information regarding the time a node has been online, T_{alive}, and resets the time since we last observed a node, T_{since} for the ProbKA and PredKA algorithms. This information is then used to calculate the probability of a node still being alive in the future and reduce the number of keep-alive messages sent.

As would be expected, by doubling the size of the keep-alive interval the cost incurred by the SKA algorithm is reduced by half. As the ProbKA algorithm does not send a keep-alive period unless a node is likely to have left the network, increasing the keep-alive interval does not have a dramatic effect on reducing the number of messages sent. By not sending keep-alive messages the ProbKA also reduces the number of messages each peer has to respond to, thereby reducing the number of required acknowledgment messages.

However, by extending the interval between successive keep-alive messages we are making an inherent trade-off between the cost of maintenance and the potential delay between a failure occurring and it's subsequent detection. To investigate this trade-off further we measured the mean, median and maximum delay using both the SKA and ProbKA algorithms with a range of interval sizes. The mean delay is simply the average time it takes for a failed connection to be detected, whilst the maximum delay is the longest time it takes for a failed connection to be detected during the entire simulation. The

latter can be seen as the worst case scenario.

Figure 6.1b shows the mean failure detection delay incurred by the SKA and ProbKA algorithms. The SKA algorithm by regularly checking each connection ensures node failures are detected and replaced consistently. As explained earlier the mean delay is very close to $k/2$ as node failures can occur uniformly at random within the interval of k. The ProbKA algorithm by extending these intervals also extends the average delay. The inclusion of a P_{thresh} parameter proved to be essential, as without a minimum threshold the ProbKA algorithm sends very few keep-alive messages at all. Lower values of P_{thresh} where also tested, resulting in lower costs and higher delays. The median failure detection delay, shown in Figure 6.1c, incurred by the ProbKA algorithm is significantly lower than the mean failure detection delay. This indicates a skewed distribution of failure detection times with a few large detection times raising the overall average.

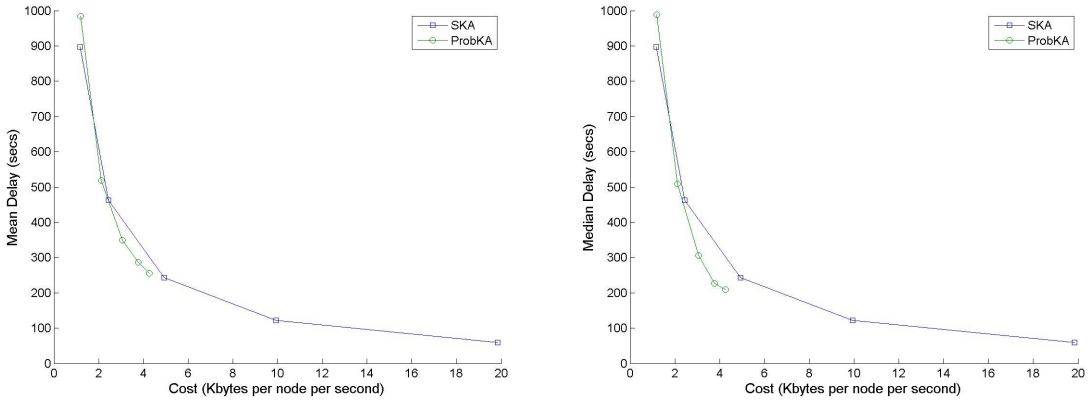
Figure 6.1d shows the maximum delay, the longest time it takes for a failed connection to be detected during the entire simulation. This can be used as an worst case scenario, the SKA algorithm performs particular well and should always detect a node failure within k time steps. With no upper bound on the interval between keep-alive messages the ProbKA algorithm can incur a significantly higher maximum delay in comparison, although such large delays are rare.

While the ProbKA algorithms increase the interval between successive keep-alive messages, the self-organising nature of the network further facilitates the extension of these intervals. Stutzbach et al showed [67] that as networks age, long-lived peers tend to become connected to one another. This forms a stable core of long-lived peers in unstructured networks. This stable-core occurs via self-organization, peers only replace connections upon

failure and connections with short-lived peers are replaced relatively quickly compared to connections with long-lived peers. Therefore long-lived peers by simply replacing failed connections, by forming new connections with existing peers selected at random, will eventually find other long-lived peers. Long-lived peers in our trace-driven experiments will also tend to eventually connect with other long-lived peers. The ProbKA and PredKA algorithms subsequently send fewer keep-alive messages as these nodes are likely to remain online for longer. However, when long-lived peers eventually fail the time until the next keep-alive message is likely to be an extended interval which reduces the incurred cost but also causes the average delay to be increased.

As detailed earlier, several studies have shown that as nodes spend more time in the network they are more likely to remain in the network longer. This can be explained intuitively, a node that has spent ten hours in the network is more likely to remain in the network for an additional hour than a node that has only been in the network five minutes. Our adaptive algorithms exploit this behaviour by extending the interval between successive keep-alive messages as nodes age. As the simulation progresses and the network ages, the connections between nodes also age; causing fewer and fewer keep-alive messages to be sent and as a result needing to be acknowledged. The SKA algorithm however has a fixed periodic interval and does not adapt it's behavior in an aging network resulting in the average cost per node remaining constant.

However Figure 6.1 does not clearly illustrate the how the ProbKA and SKA algorithms compare against one another. Figure 6.2 shows a cost versus performance comparison of the SKA and ProbKA algorithms allowing a direct analysis to be made.

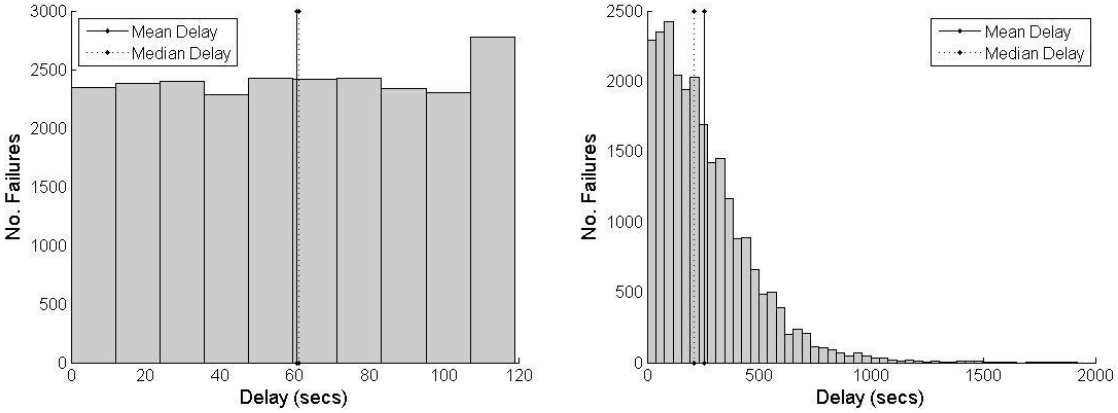


(a) (b)

Figure 6.2: Mean and median performance comparison of the SKA and ProbKA algorithms

This form of analysis is similar to the performance versus cost framework presented in [35], which analysed and compared the Chord, Kademlia, Kelips, OneHop and Tapestry protocols by viewing them as consuming bandwidth in order to achieve a certain lookup latency. By systematically searching through many tunable parameters the study highlighted important features of each DHT. While this thesis does not compare alternative overlay networks it does highlight the performance/cost tradeoff in terms of extending the intervals between keep-alives and the incurred failure detection delay. Furthermore as in [35] we clearly separate the detection of a failed node from recovering from failures during lookup.

Figure 6.2a shows that ProbKA algorithm consistently reduces the average failure detection delay when compared to the SKA algorithm at similar cost levels. Figure 6.2b also shows the median delay incurred by the ProbKA algorithm is even further reduced when compared to the SKA algorithm. As the failure detection delay is uniformly distributed within the keep-alive period when using the SKA algorithm the mean and median de-



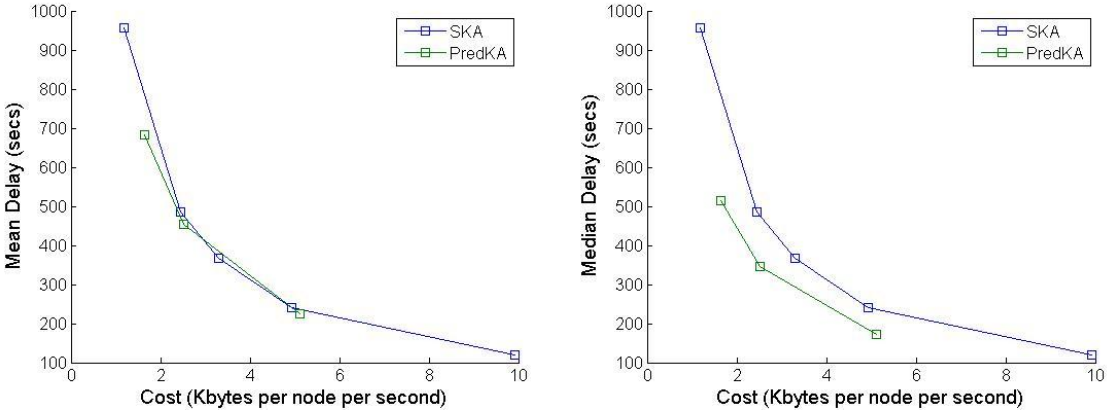
(a) (b)

Figure 6.3: Histogram of the SKA and ProbKA incurred failure detection delay with $k = 120$ and $P_{thresh} = 0.99$.

lays are the same. This indicates the mean failure detection delay incurred by the ProbKA algorithm is skewed by a small number of relatively large failure detection delays as shown in Figure 6.1d.

Figure 6.3 emphasises the importance of the median incurred failure detection delay as a performance metric, as it is stable with respect to outliers. Figure 6.3a shows the distribution of failure detection times as a histogram for the SKA algorithm with $k = 120$, with the failure detection delay being uniformly distributed throughout the keep-alive period. As a result the mean and median failure detection times are not significantly different. However, Figure 6.3b shows the distribution of failure detection times for the ProbKA algorithm with $k = 120$ and $P_{thresh} = 0.99$, showing the mean detection time is skewed by a few significantly large failures detection delays. Whereas the median incurred detection delay is more representative of the overall distribution.

Undetected failed connections may incur expensive timeouts as lookups are forwarded through them, reducing the network's efficiency. Furthermore,



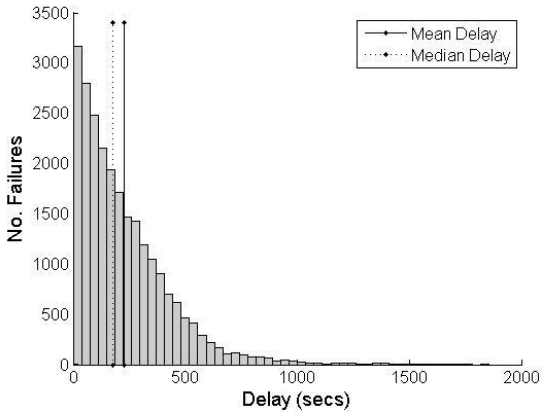
(a) (b)

Figure 6.4: Mean and median performance comparison of the SKA and PredKA algorithms

the larger the delay between failures occurring and being detected increases the likelihood of a node being forcefully disconnected from the network. A forced disconnect occurs when all a node's neighbours fail without being replaced. A node that is no longer connected to any other online node is effectively disconnected from the network. There are a number of approaches that can be taken to reduce the number of forced disconnections. In order to reduce the likelihood of all a node's neighbours leaving the network we could simply increase the number of connections each node maintains. Alternatively, node's can maintain each connection more frequently to ensure any node failures that do occur are then detected and replaced with as small a delay as possible.

Figure 6.4 shows the performance of the PredKA algorithm is very sensitive to adjustments to the P_{target} parameter. The results shown set $P_{target} = 0.97, 0.98$ and 0.99 from left to right respectively. As the PredKA algorithm is deterministic it will always send a keep-alive message at the end of each keep-alive period. Whereas the ProbKA algorithm is stochastic, it may send

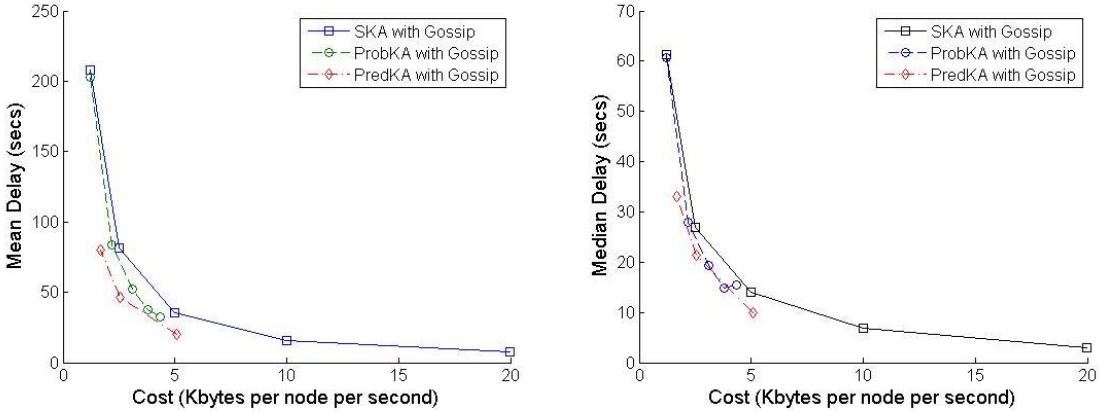
a keep-alive message at the end of each keep-alive period based upon the likelihood of a having failing occurred. Furthermore, the keep-alive period is fixed for the ProbKA algorithm. Despite these differences the ProbKA and PredKA algorithm perform around a similar level with the ProbKA algorithm being slightly more flexible. Figure 6.5 shows a histogram of the failure detection delay times incurred by the PredKA algorithm with $P_{target} = 0.99$, once again showing how the mean failure detection delay is skewed by a small number of failures with large detection times.



(a)

Figure 6.5: Histogram of the incurred failure detection delay by the PredKA with $P_{target} = 0.99$.

Figure 6.6 shows that augmenting the SKA, ProbKA and PredKA algorithms with a gossip mechanism causes the mean and median detection delay to be significantly reduced without significantly increasing the cost. As nodes inform mutual neighbours of any failures that are detected news of a failed node travels fast. Furthermore, the increased overhead of gossip mechanism is relatively small, typically just 0.01 bytes per node per second in all experiments. This includes the cost of the additional messages triggered by the gossip mechanism. When failures are detected, by using a simple gossiping



(a) (b)

Figure 6.6: Mean and median performance comparison of the SKA, ProbKA and PredKA algorithms with Gossip

mechanism as described earlier, nodes cooperate and quickly inform their mutual neighbours who then detect it for themselves. The vast majority of used bandwidth is spent sending and successfully acknowledging keep-alive messages, relatively few failures are detected compared to the number of keep-alive messages sent and acknowledged. As the additional overhead of the simple gossip mechanism is so low we retain the overall reduction in terms of cost and delay of ProbKA and PredKA algorithms when compared with the SKA algorithm. Furthermore, as our adaptive algorithms tend to detect failures sooner than the SKA approach, adding a gossip mechanism further augments the reduction in delay. By detecting failures earlier, these results show our predictive algorithms can complement other optimisations, such as gossip mechanisms, to the standard to the standard keep-alive algorithm [70, 16].

However, even with the gossip mechanism the maximum delay by our probabilistic mechanisms is relatively high. As the gossip mechanism allows nodes to inform a potentially outdated set of mutual neighbours nodes that

are interested in a failure may not be informed. These nodes have to discover the failure for themselves which may incur long detection delays. More advanced gossip mechanisms such as a flooding mechanism used in [16] or epidemic based approaches studied in [70] could be used to minimise, but not eliminate, the likelihood of outdated neighbourhood sets. An alternative, effective and simple response is to define a maximum interval size after which a keep-alive message must be sent.

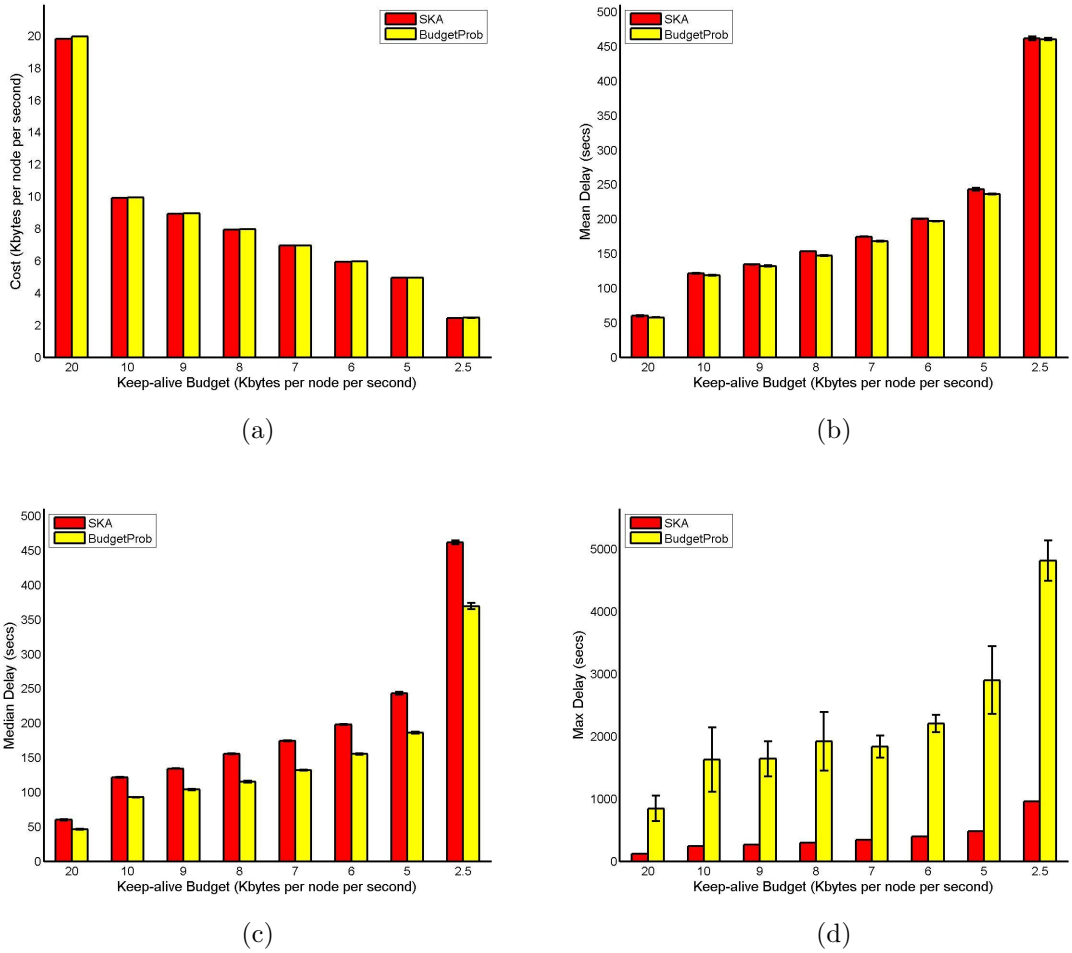
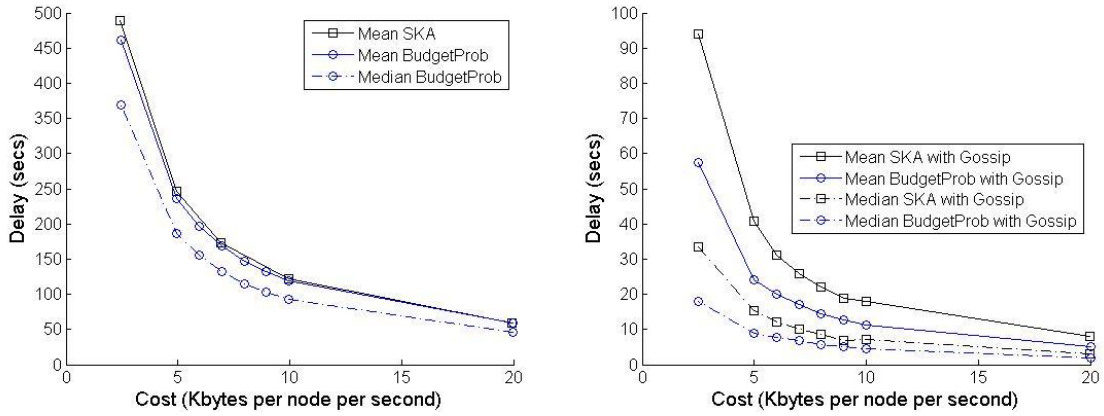


Figure 6.7: Performance comparison of the SKA and BudgetProb strategies in terms of cost and mean, median and maximum delay.

Figure 6.7 compares the BudgetProb and SKA algorithms. The main advantage of the BudgetProb algorithm is that it allows and keeps the cost of maintenance within a the bandwidth budget. Again our adaptive approach consistently reduces the mean and median failure detection delay without increasing the maintenance cost. Figure 6.7c shows our adaptive BudgetProb approach reduces the median delay just under 24% on average. By prioritising connections that are more likely to fail our budget based approach shortens the keep-alive interval for younger nodes whilst extending the intervals for older nodes. Shorter keep-alive intervals for younger peers results in more failures being detected earlier, whereas the standard keep-alive algorithm sets all intervals to a uniform length.

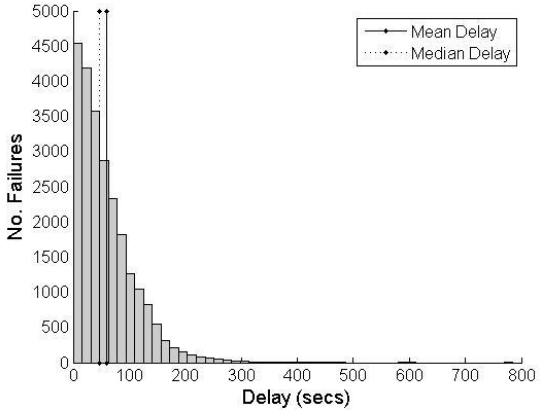


(a) (b)

Figure 6.8: Performance comparison of BudgetProb and SKA algorithms.

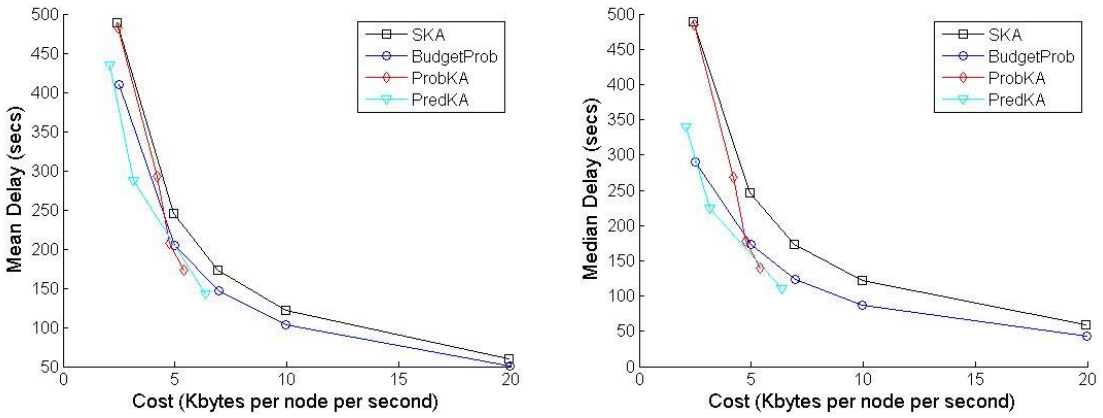
When combined with the simple gossip mechanism our BudgetProb algorithm performance increases, Figure 6.8b shows the mean and median delay is reduced by 35% on average.

Finally, Figure 6.10 shows the performance of all three adaptive algorithms upon the LegalTorrents network data which contains the session data



(a)

Figure 6.9: Histogram of the BudgetProb incurred failure detection delay with $\beta = 20$.



(a) (b)

Figure 6.10: Mean and median performance comparison of the SKA, ProbKA and PredKA algorithms using network data from LegalTorrents.

of nodes over numerous file distributions. The results not only show our approach is applicable to other network data but also that the mean and median failure detection delay is even further reduced in the LegalTorrents data than in RedHat9 distribution. The BudgetProb algorithm reduces mean and median failure detection delay by 14% and 30% respectively. Whereas the

ProbKA algorithm performs well at higher cost levels but it's performance degrades as the bandwidth consumed is reduced. Nodes from LegalTorrents data appear to remain longer in the network when compared with the RedHat9 session times. One possible explanation for this behaviour is some BitTorrent clients now allow downloads to be automated via RSS (Really Simple Syndication) feeds. Nodes controlled by such automated clients will generally remain in the network until a pre-defined upload to download ratio has been reached and therefore may stay longer than user-controlled clients.

6.3 Learning of Current Uptime

A critical component of each of the three adaptive algorithms examined above is that they each require knowledge of how long a node has already spent in the network to function. The previous experiments, up until this point, assumed that neighbours would honestly report their current uptime. However, in a real network it is often unsafe to assume such altruistic behaviour especially when nodes could gain some advantage by misreporting their own age. For example, a node could receive and have to respond to fewer keep-alive messages by reporting it's age as significantly higher than it actually is, thereby conserving it's own bandwidth.

We investigate three alternative approaches to estimating the current age of individual nodes within the network:

1. **Ultra**: The first method of making our adaptive algorithms more robust to dishonest nodes, would be to not ask node's their age directly. The most basic and most conservative way of ruling out dishonesty would be to assume any new neighbour has spent no time in the network beforehand. We call this the *ultra* conservative approach, as it starts

each connections age at zero. While this would, potentially severely, underestimate a neighbours age it would ensure neighbouring nodes could not capitalise on the adaptive elements of our algorithms.

2. **Moderate**: Alternatively a more balanced approach would be to allow nodes to approximate a neighbours age by asking one of it's neighbour, selected at random, for the time it has known the node in question. By learning from an independent source a node can initialise a connection with an approximation of a neighbour's age. While this may allow an element of collusion between neighbours it does allow a node to at least estimate a neighbours current age.

3. **Liberal:** Finally, our original method, as studied previously in this chapter, of assuming each node honestly reports it's own age we now refer to as the *liberal* approach. The liberal approach could at worst be described as naïve, gullible and open to abuse but in an altruistic environment it provides the simplest and most accurate method of learning a neighbours age.

Figure 6.11 shows the moderate approach very closely matches the performance of the more trusting liberal approach, whereas the ultra conservative approach underestimates the current uptime of neighbours resulting in more keep-alive messages being sent.

As incoming connections are established over time, the moderately conservative approach of asking a node's neighbour for its age approximates a node's age reasonably well. For example, under the moderate approach if node Y is the first node to establish an incoming connection towards node X, Y assumes X has just joined the system and initialises X's age as $T_{alive} = 0$. Should a second incoming connection to X be subsequently established by

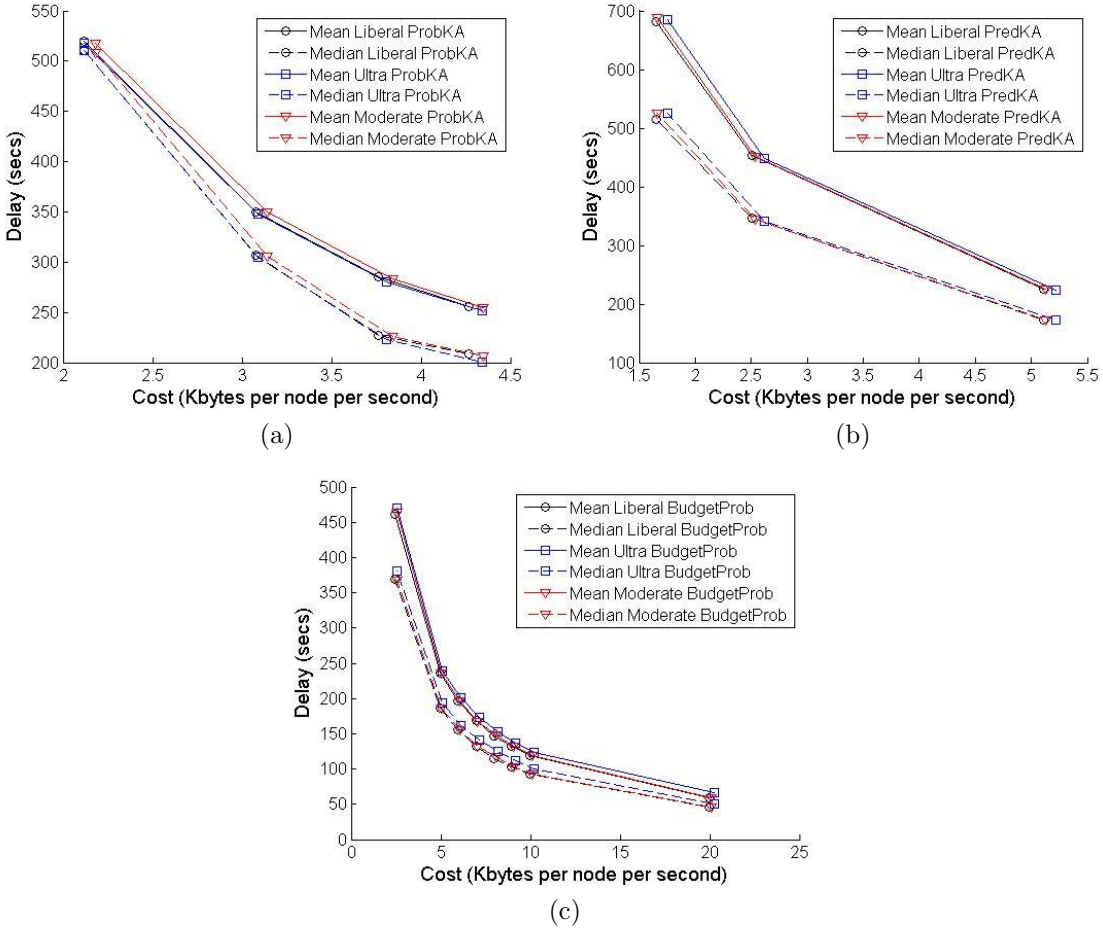


Figure 6.11: Comparison of liberal, moderate and ultra conservative approaches to the ProbKA, PredKA and BudgetProb algorithms.

node Z, under the moderately conservative approach Z then asks Y for the age of it's connection with X. Node Z then uses this value to initialise it's own connection. Any further connections that are established will randomly select one of the existing connections, which have all initialised their own connection either directly or indirectly based upon the age of Y's connection with X. The additional overhead of asking a another node for it's neighbours age is also minimal. For these reasons the moderate conservative approach is a robust and efficient method of ascertaining a node's age, as long as a

node's neighbours can be trusted. Otherwise, the ultra conservative approach of assuming all new neighbours have just joined the network still performs comparably well.

In [10], Bustamante and Qiao describe a three phase protocol to determine the age of peers. This protocol involves collecting a subset of a node's neighbours, sampling the ages they report and trimming suspiciously large ages before determining the new neighbours established age. Although the protocol does not determine the "real" age of the peer it does produce a safe estimation. However for our purposes contacting multiple peers proved prohibitively expensive, caused the additional overhead in terms of cost to outweigh the benefit in terms of reducing the failure detection delay. Whereas the moderate approach proved a good balance between low overhead and a good estimation of a peer's current uptime.

6.4 The Influence of Dishonest Nodes

While the experiments presented in Figure 6.11 investigate alternative methods of discovering a node's age, they still assume that nodes do not act in a purposefully dishonest manner. The next set of experiments investigate the performance of our adaptive algorithms when node's deliberately misreport their own age in order to conserve their own bandwidth. Each experiment sets a dishonesty parameter δ, each time a new connection is established the receiving node misreports it's actual age T_{alive} as T_δ which is uniformly randomly generated within the interval $T_{alive} \leq T_\delta \leq \delta \cdot T_{alive}$. Essentially δ determines the level of dishonesty in a experiment, the greater the value of δ the greater each node's capacity for dishonesty. When $\delta = 1$ there is no dishonest behaviour amongst nodes and they report their actual age as

assumed in all the previous experiments. We continue to assume however that we have accurate and truthful knowledge of the underlying distribution of session times. Furthermore, these experiments use the liberal approach of asking a node directly for it's own age and naïvely assume it reports truthfully. The results are shown in Figure 6.12.

As Figure 6.12 illustrates the influences of dishonesty upon our adaptive algorithms, clearly the fixed SKA approach is unaffected as it does not take into account the reported current uptime of neighbours. However, the dishonest nature of neighbours affects the ProbKA and PredKA algorithms by increasing the incurred failure detection delay while decreasing the bandwidth spent on maintenance. By increasing the value of their own reported age, neighbours essentially delay the sending of keep-alives to themselves. This benefits the dishonest nodes as they receive and therefore have to respond to fewer keep-alive messages. Importantly, our adaptive algorithms are not radically influenced by this type of malicious behaviour. Nodes still send and receive keep-alive messages according to the underlying distribution of peer session times, but at a decreased rate as nodes believe their neighbours are older than they actually are. The higher value of δ, the more the ProbKA and PredKA algorithms are affected.

Perhaps surprisingly the BudgetProb algorithm seems to be unaffected by nodes dishonestly reporting their age. The BudgetProb algorithm assigns a fixed amount of bandwidth to be spent on maintenance, therefore clients using this algorithm are essentially committed to sending a fixed amount of keep-alive messages. As in our experiments all nodes behave as dishonestly as one another, the BudgetProb algorithm isn't influenced at all by their behaviour. Assigning each neighbour a proportion of it's bandwidth budget according to the likelihood it is online in relation to the other neighbours.

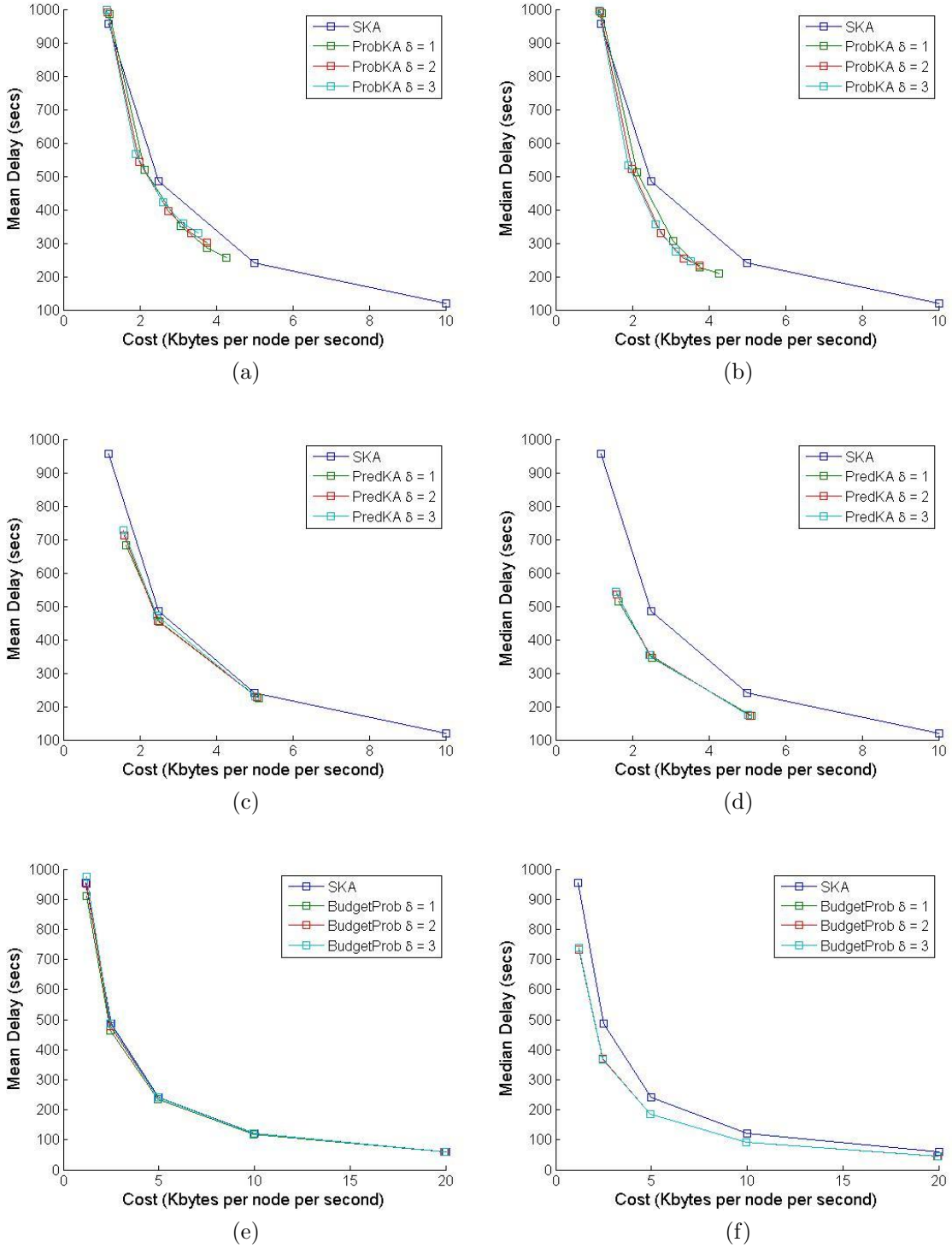


Figure 6.12: Affects of dishonesty on the ProbKA, PredKA and BudgetProb algorithms with $\delta = 1, 2$ and 3

Of course, the BudgetProb algorithm would be more affected in a network where just a few nodes dishonestly reported their age. In such an environment, we could expect similar results as seen with the ProbKA and PredKA algorithms, where dishonest nodes receive fewer keep-alives increasing the incurred failure detection delay.

Furthermore, these experiments have assumed that nodes behave dishonestly in order to gain some benefit, for example reducing the bandwidth spent on maintenance. As Figure 6.12 showed this type of behaviour generally results in fewer keep-alive messages being sent but overall the algorithms perform well. However, it is possible that some nodes may behave dishonestly with the motivation of crippling the underlying network. In such an environment, the adaptive algorithms presented in this section could be severely influenced. Under such circumstances the safest way to apply our algorithms would be to use the ultra conservative approach when establishing connections. This approach does not depend directly on the honesty of neighbours reporting their age but does perform reasonably well reducing the mean and median incurred failure detection delay.

6.5 Summary

This Chapter presented three new algorithms based on the principle that nodes become more reliable as they age, these algorithms reduce the average failure detection delay when compared directly to the widely deployed standard periodic approach. In doing so they reduce the mean and median failure detection delay by as much as 35% while operating at a similar level of cost.

Using a trace-driven simulation based upon measured network data we

empirically evaluated both of the proposed algorithms against the widely deployed standard keep-alive algorithm. With a BitTorrent tracker log as the basis of the simulation platform we ensured the complex process of churn was modelled both accurately and realistically.

We also showed, our approach can complement other keep-alive mechanisms. By adding a simple gossip mechanism the average failure detection delay can be further reduced without expending substantial additional bandwidth. Although the ProbKA, PredKA and BudgetProb algorithms reduce the mean and median delay the maximum failure detection delay is potentially increased. However, by defining a maximum interval size the failure detection delay can be limited to a suitable upper bound.

Furthermore, we investigated more robust ways of ascertaining a node's age when establishing a connection with a new neighbour. The performance of the adaptive algorithms did not degrade significantly when nodes found out their neighbour's ages indirectly or when they simply assumed all new neighbours have just joined the system. The influence of dishonesty amongst nodes was also investigated, showing that node's that deliberately increased their own age affect the rate of keep-alive messages sent causing the an increase in the failure detection delay. However, the BudgetProb algorithm was shown to be significantly resilient to this type of behaviour and remained largely unaffected by increasing levels of dishonesty in the network.

Overall, setting an appropriate keep-alive period is a trade-off between the incurred bandwidth and the failure detection delay. All of our adaptive algorithms increase the interval between successive keep-alives as nodes age and their estimated reliability increases. As short-lived peers constitute a large proportion of sessions and by prioritising connections that are more likely to fail the average failure detection delay is reduced. Furthermore, as a

side-effect of our adaptive algorithms nodes that remain in the network longer receive and have to respond to fewer keep-alive messages. In conclusion this Chapter has shown that predictive mechanisms can be successfully used to reduce the average failure detection delay whilst limiting traffic overhead of maintenance protocols in unstructured networks.

7

Extending Keep-Alive Intervals in Structured P2P Overlay Networks

This Chapter investigates the application of current uptime based failure detection algorithms on structured networks, or more specifically how the stabilization process implemented by many structured networks may affect the adaptive algorithms presented in this thesis.

As introduced in Chapter 2, while structured P2P networks have many of the same properties as their unstructured counterparts, there are a few important differences. Firstly DHT-based structured networks, such as Chord

[62], Pastry [54] and Bamboo [51], position nodes within an overlay deterministically according to their identifiers. Although hashing ensures identifiers are uniformly distributed across the identifier space, hashing a node's key will always produce the same identifier resulting in a node always occupying that same position within the virtual overlay. Generally all positions in a structured network are considered to be equal, the only differences being which keys a node is responsible for and whom a node maintains connections with.

To ensure efficient routing, that typically scales logarithmically with the number of nodes, each node within a structured network maintains connections with the nodes that are the closest to predefined points in the network. For example the prototypical DHT-based structured network Chord, specifies each node maintains a finger table, with the i^{th} entry of node n's finger table containing the first node that succeeds n by at least 2^{i-1} for $1 \leq i \leq m$.

To maintain the structure of the network it is insufficient to just send keep-alive messages and replace finger table entries when they fail. Instead nodes must also periodically ensure each finger table entry is pointing towards the node closest to it's target position, a process known as *stabilization* but sometimes referred to as *re-pinning* [22]. In Chord the procedure responsible for this process is called fix_fingers(), the pseudo code for which is given in Algorithm 6:

Algorithm 6 fix_fingers()

$next = next + 1$
if $next > m$ **then**
  next = 1
end if
$finger[next] = find.successor(n + 2^{next-1})$;

Unless a significant number of nodes have joined or left the network, en-

tries pointing toward a close but not the closest node to a target identifer do not have a significant impact on the routing efficiency of the overall network [62]. The purpose of the fix_fingers() procedure is simply to retain the structure of the network as nodes join and leave. As a node's routing table entry is initially pointed to the closest node available to a certain target position in the network, new nodes may appear that are closer to the target position of that routing table entry. The stabilization procedure therefore, replaces the established connection with a new connection to a younger node. This may have a significant impact on our adaptive algorithms which rely on maintaining connections with neighbours based upon their age.

7.1 Experimental Methodology

As before, to evaluate and compare our adaptive algorithms in structured networks we use the RedHat9 BitTorrent network data as made available from [29]. Using the largest continuous period of this data, our experiments once again begin cold, allowing nodes to populate the network for the first simulated twelve hours. Once this period is finished each node then creates connections with existing nodes and we report the maintenance of these connections over the subsequent four and half simulated days. Each of the results presented below are averaged over a series of ten experiments.

Unlike the previous Chapter which examined unstructured networks and created $D = 30$ connections to node's selected at random, this section's experiments create connections according to the rules utilised by the structured network Chord [62]. Firstly, each node is first given a unique key, selected uniformly at random from the list of available keys, which determines it's position in the overlay. Each node then maintains m routing table entries

with the i^{th} entry of node n's finger table containing the first node that succeeds n by at least 2^{i-1} for $1 \leq i \leq m$. As only $O(logN)$ are routing table entries are distinct; more than one routing table entry may point towards the same node, in this case a node only maintains one connection to any other individual node.

Again, we use our the standard keep-alive (SKA) algorithm as a comparison to performance of the ProbKA, PredKA and BudgetProb algorithms. Costs are defined in terms of bandwidth and incurred failure detection delay. Each of the failure detection algorithms govern the sending of keep-alive messages as detailed in Chapter 5. Each time a failure is detected, that routing table entry is replaced with a new connection using Chord's deterministic rules. Finally, we run the stabilization procedure, fix_fingers() as given in Algorithm 6, at varying rates to examine it's affects on the SKA algorithm and our adaptive approaches.

7.2 Results

Figure 7.1 compares the affects of stabilization on the fixed periodic algorithm and our adaptive BudgetProb approach, where each node runs the fix_fingers() procedure on a entry in it's routing table every 240, 480, 960, 1920 seconds and not at all when stabilization = 0. We should highlight that we are not investigating how often the stabilization process should be run, as in [22]. Figure 7.1a shows the SKA approach is largely unaffected by the stabilization process and simply incorporates the replaced connections into it's fixed periodic cycle. The BudgetProb algorithm is affected however, with smaller rates of stabilization causing the incurred median failure detection delay to be significantly lower than when stabilization is run less

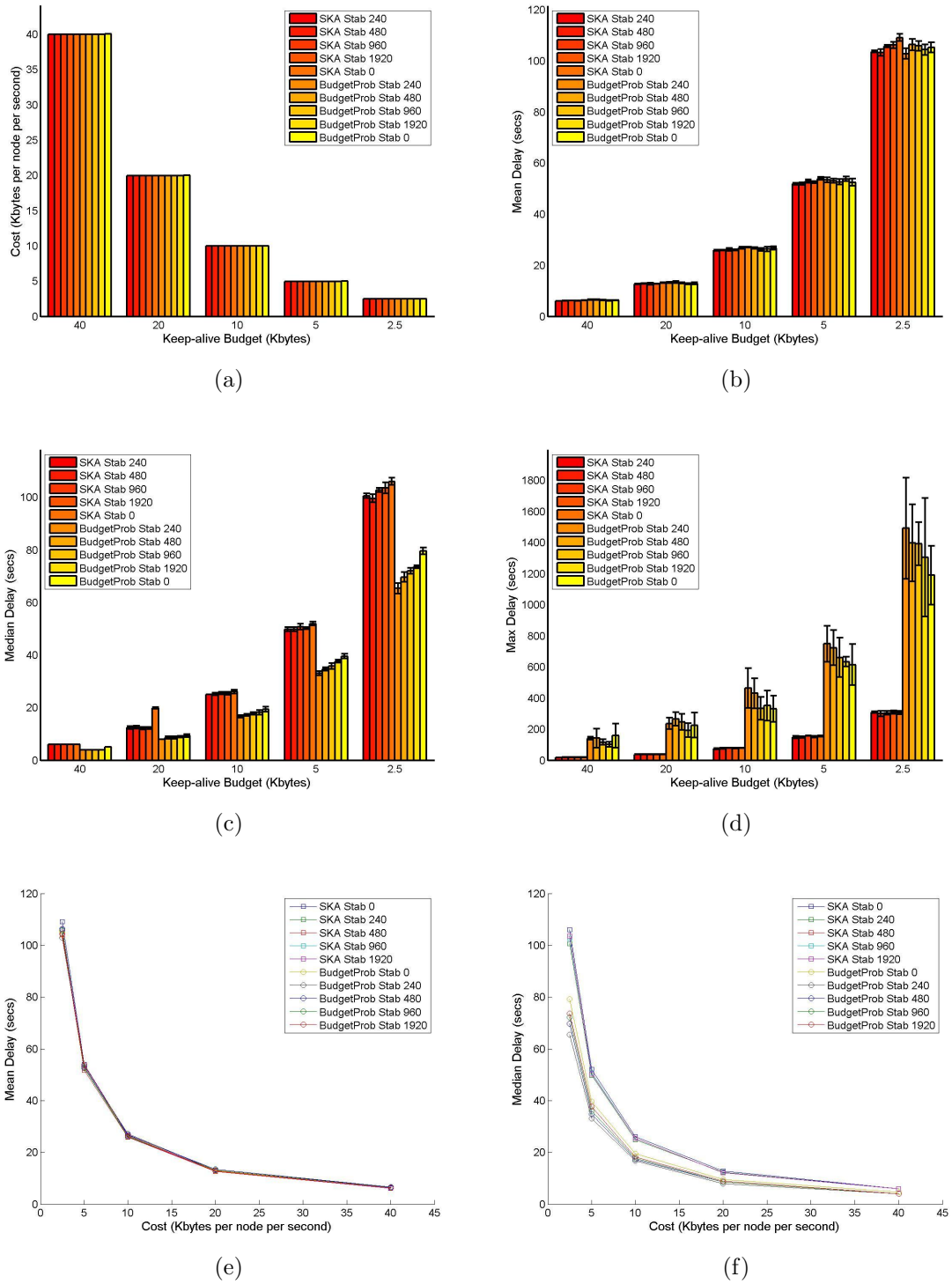


Figure 7.1: Performance of BudgetProb in a Deterministic Chord Network.

often, as clearly shown in Figure 7.1c. Figure 7.1b shows once again the mean failure detection delay, skewed by a few relatively large failure detection times, is larger than the observed median failure detection delay and does not as clearly show the affects of stabilization. The maximum incurred failure time, as shown in 7.1d, despite exhibiting a high standard deviation, is also noticeably lower with reduced rates of stabilization.

By replacing older connections with more recently arrived nodes the stabilization process causes the BudgetProb algorithm to send keep-alive messages to these routing table entries at shorter intervals. As these connections to younger nodes are more likely to fail; the shorter keep-alive intervals result in more failures being detected earlier, which explains the lower incurred median failure detection times at higher rates of stabilization. The more often stabilization is performed the more often connections are replaced by younger nodes. Whereas the larger maximum incurred failures times are caused by the BudgetProb algorithm allocating less of the overall bandwidth budget to relatively older connections as newer nodes are incorporated into routing tables. As a result, the keep-alive periods of older neighbours are extended allowing the potential maximum time until a failure is detected to grow.

These results are further emphasised in Figure 7.2 and Figure 7.3, which show the ProbKA and PredKA algorithms respectively incurring lower failure detection delays when stabilization is run more often. Figure 7.2a and Figure 7.3a shows increasing the rate of stabilization also causes the ProbKA and PredKA algorithms to consume more bandwidth. The ProbKA and PredKA algorithms react to younger neighbours found by the stabilization process by generating keep-alive messages earlier than they otherwise would, increasing the number of messages sent. Whereas the BudgetProb algorithm responds to newly found young neighbours by not only shortening their keep-alive in-

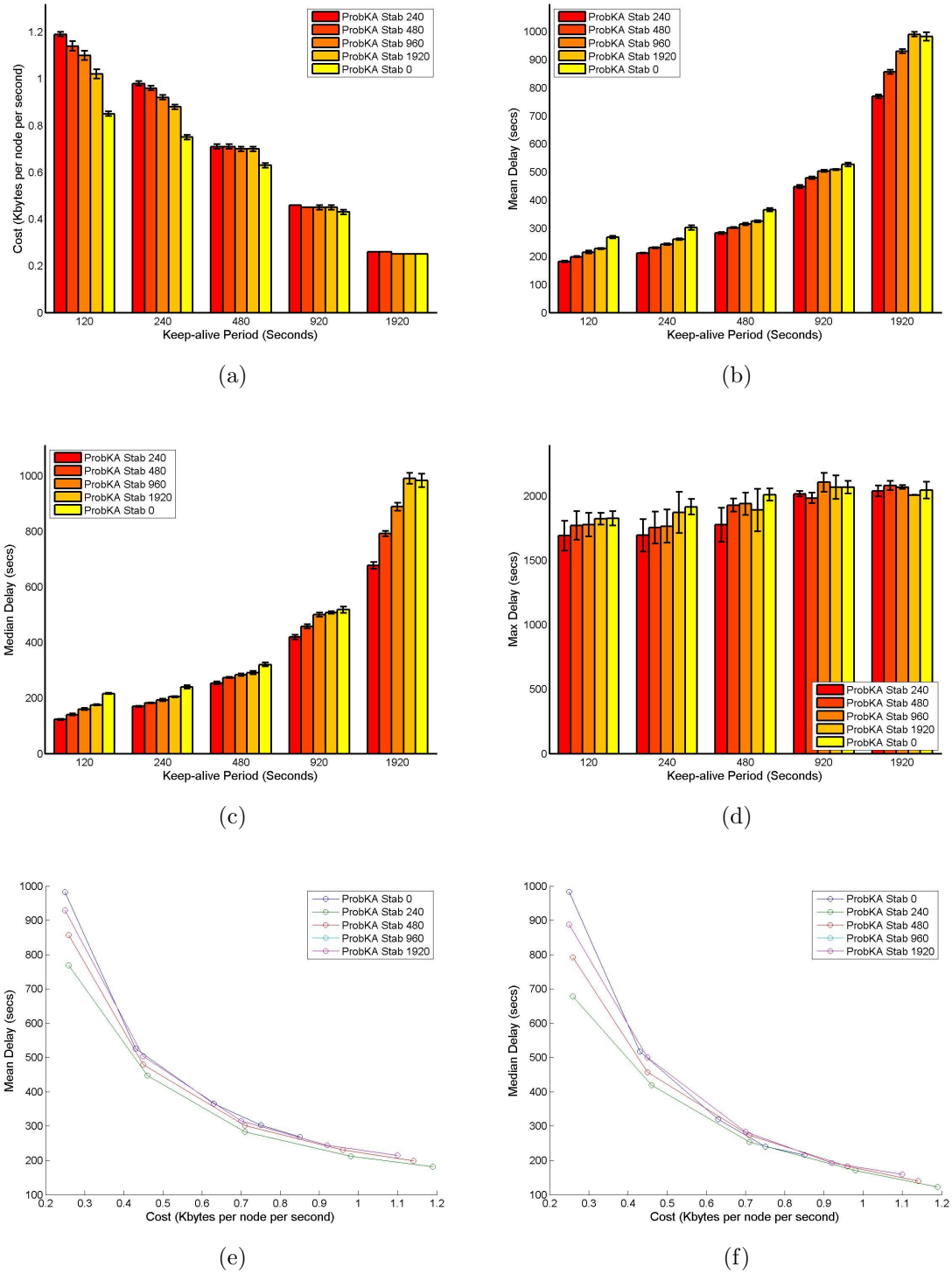


Figure 7.2: Performance of ProbKA in a Deterministic Chord Network.

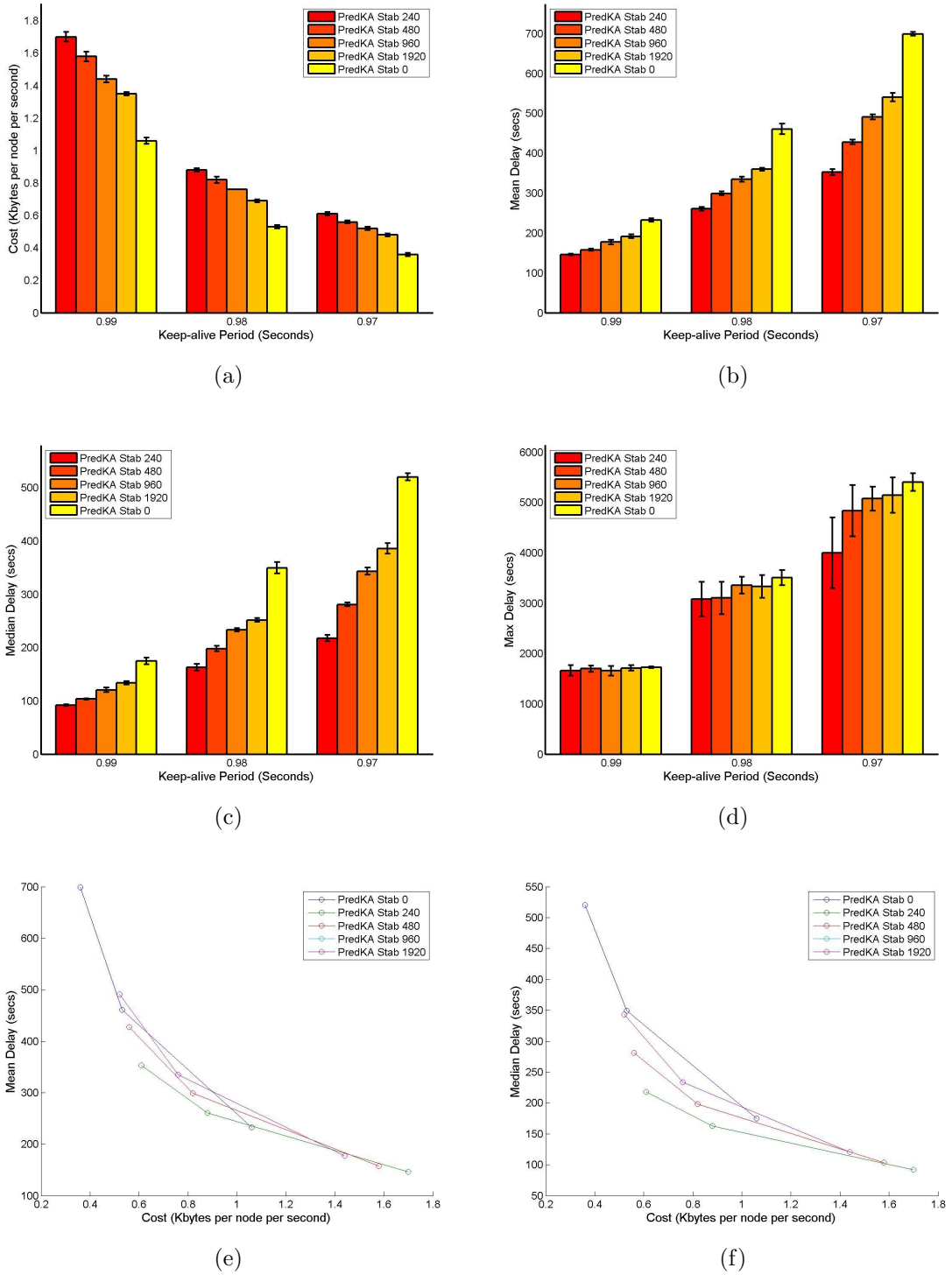


Figure 7.3: Performance of PredKA in a Deterministic Chord Network.

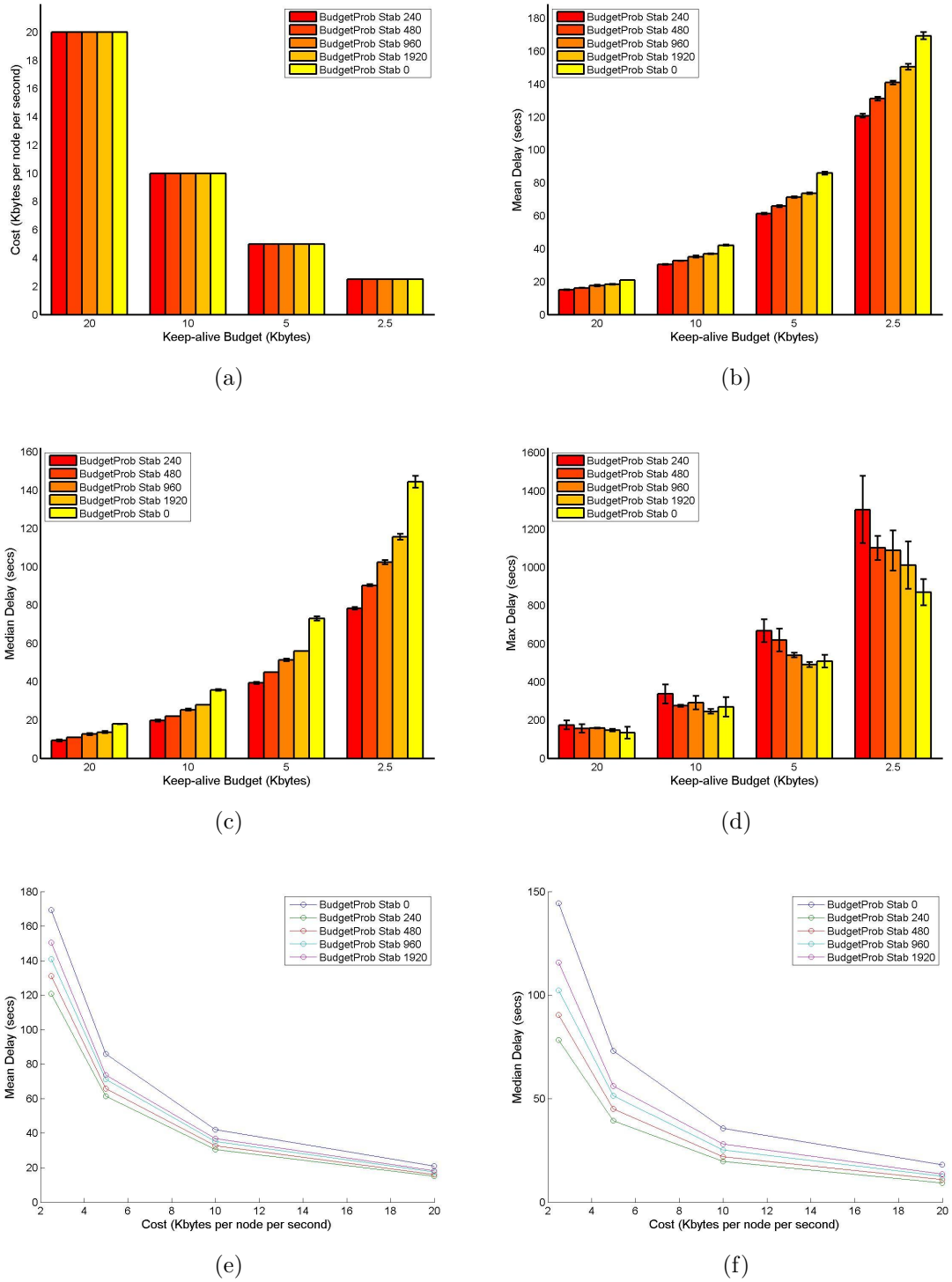
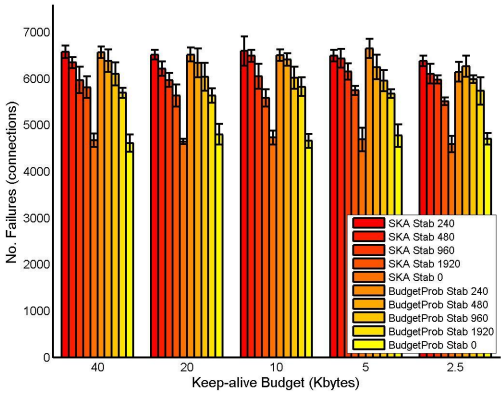


Figure 7.4: Performance of BudgetProb in a Deterministic Chord Network using the LegalTorrents network trace.

tervals but by also extending the keep-alive intervals of older connections, which causes the bandwidth to be consumed at a constant rate. While this results in the maximum failure detection delay incurred by the BudgetProb algorithm to grow as the stabilization procedure is run more often, the maximum failure detection delay is largely unaffected within the ProbKA and PredKA algorithms, as Figure 7.2d and Figure 7.3d show. Using the LegalTorrents network data the affects of stabilization on the performance of the BudgetProb algorithm is even more pronounced as Figure 7.4 illustrates.

The stabilization procedure implemented by structured networks essentially prevents older peers eventually forming connections with one another, as observed in unstructured networks. Analysis by Stutzbach et al. in [67] revealed that a stable and layered core exists within overlay topology providing an efficient backbone in Gnutella. The authors rationalise that the stable onion-like layered core is formed primarily due to user-driven dynamics. They show that each peer may establish and destroys many connections to other peers during the first 100 minutes of it's uptime. However peers that remain in the system beyond this point maintain their connections with long-lived peers and only add a connection once an existing connection has been dropped.

The implication for our adaptive algorithms is that stabilization causes peers to establish connections with newly arrived peers more often. This in turn causes our algorithms to send more keep-alive messages to these younger peers, detecting failures earlier on average. The downside of this process is that stabilization exacerbates the disruption caused by churn, stabilization causes connections to be made with younger nodes who tend to spend a short time in the network. Stabilization does not cause more node failures but causes connections to be made with nodes that fail more often. Figure 7.5



(a)

Figure 7.5: Number of failures detected by the SKA and BudgetProb Algorithms under Stabilization

shows the number of failures detected is not dependent on using either the SKA or the BudgetProb algorithm or the bandwidth consumed by either algorithm but that the number of failures increases as the rate of stabilization increases. These connections then when they fail, have to be replaced. In a structured network replacing a failed connection requires finding the node responsible for a particular key within the overlay, the same process as the $lookup(key)$ operation which typically takes $O(logN)$ messages in many DHT-based networks [62, 54, 51].

Alternatives to the rigid routing tables rules are available, such as Pastry which allows any node that fulfills a specific node identifier prefix to be fill a routing table entry [54]. Furthermore, more flexible alternatives to neighbour selection specifically within Chord have been developed, which allow a node to fill a routing table entry i that succeeds itself within the range a node n within the range $[2^{i-1}, ..., 2^i]$ [24, 41, 28]. Figure 7.6 shows the results of our experiments using this more flexible neighbour selection scheme. In such a network, our algorithms perform much as they would in a unstructured

network as no stabilization process needs to occur.

7.3 Summary

This Chapter has highlighted unexpected issues when implementing adaptive failure detection algorithms upon structured networks. As these types of networks often maintain their structure as well as their routing state this affects the performance of adaptive failure detection algorithms. As the stabilization process of Chord [62] replaces established connections with newly arrived nodes, our current uptime-based algorithms increase the number of keep-alive messages sent to these nodes. Newly arrived nodes also tend to leave earlier, with shorter keep-alive intervals resulting in the incurred failure detection delay being reduced. Overall while stabilization doesn't cause more node failures to occur it does cause more node failures to be observed. The resulting efficiency of more accurate routing table entries may not be worth the increased overhead of an aggressive stabilization process. While our adaptive algorithms may seem to benefit from the stabilization process they have also been shown to perform well in systems without stabilization proving them to be a general failure detection framework that can be applied to most types of overlay networks.

So far, we have examined one purpose of keep-alive messages; to check if the connection between two peers is still alive. However, keep-alive messages are also used to minimise the risk of external devices such as a routers, NATs and firewalls dropping connections due to extended periods of inactivity. The next Chapter investigates how failure detection algorithms can be designed with the limitations of NAT devices in mind.

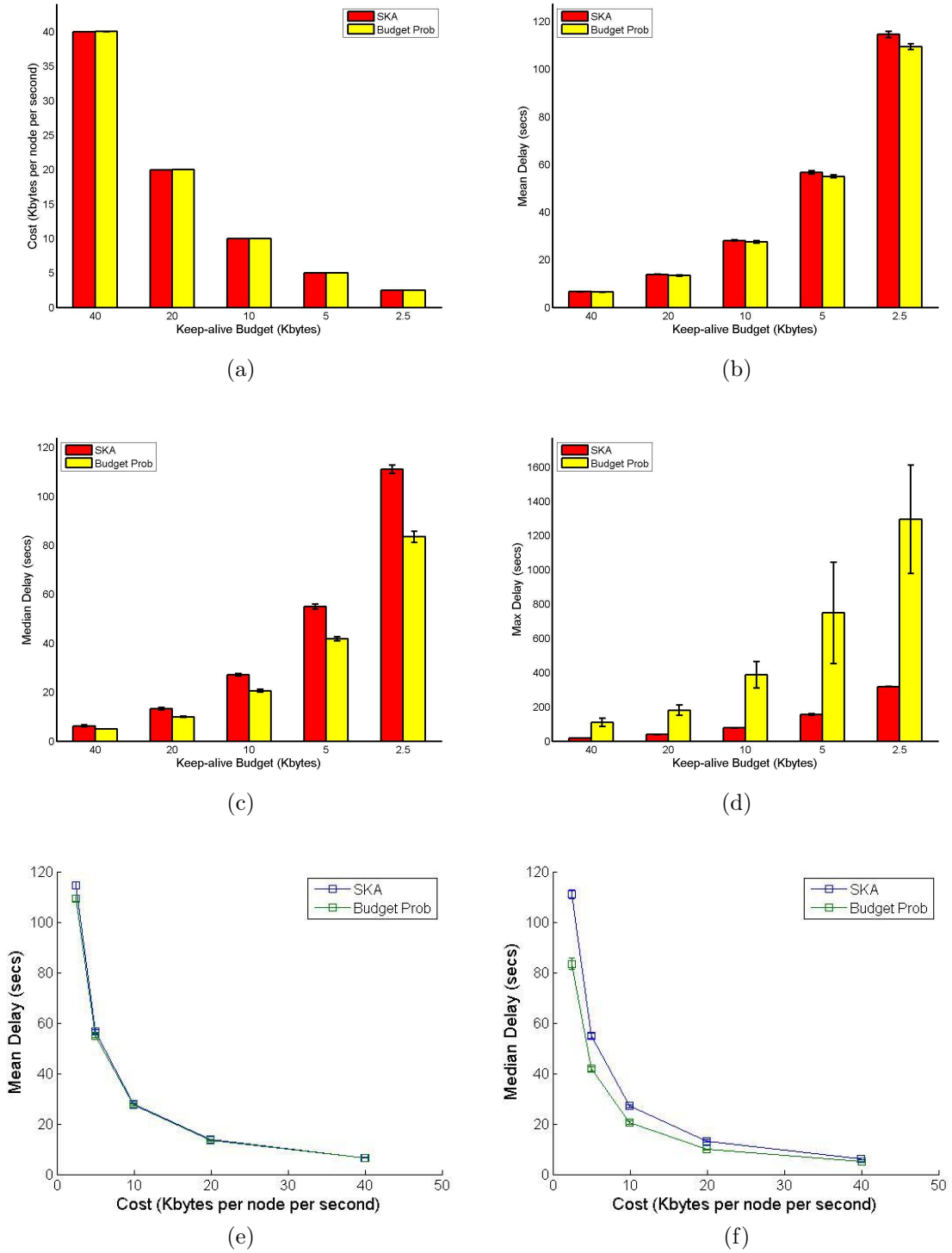


Figure 7.6: Comparison of BudgetProb and SKA algorithm in a Flexible Chord Network.

8

Adapting to NAT timeout values in P2P Overlay Networks

As we've already seen, nodes within existing P2P networks typically exchange periodic keep-alive messages in order to maintain network connections between neighbours. However, keep-alive messages actually serve a dual purpose, they are not only used to detect node failures but they are also used to prevent idle connections from being expired by Network Address Translation (NAT) devices. Despite being widely used the interval between messages is typically fixed below the timeout value of most NAT devices based upon

crude rules of thumb.

Although many studies have been conducted to traverse NAT devices [53, 27, 6, 21], and other studies - as reviewed in Chapter 3 - seek to improve failure detection in P2P overlay networks; the limitations of NAT devices have received little research attention. This Chapter explores algorithms which allow nodes to adapt to the timeout values of individual NAT devices and investigates the resulting trade-offs.

8.1 Network Address Translation Devices:

Using stateful translation tables that map multiple private addresses onto a single public address, Network Address Translation (NAT) devices allow several nodes within a private network to share a single public IP address. This allows private home networks and business intranets to interface with public networks such as the Internet. The widespread use of NATs has helped to alleviate the IPv4 network address shortage problem.

However due to limited resources and the vulnerability to denial of service attacks, NAT devices cannot indefinitely hold the state of their translation tables. As a result, idle connections are eventually expired and connection states removed after a NAT *timeout period*.

To avoid connections becoming idle, connected nodes must periodically exchange *keep-alive messages* at an interval shorter than the timeout period. As a result, keep-alive messages are widely used throughout all types of networks to maintain connections between nodes. The keep-alive period k defines the maximum interval that a connection between two nodes can remain inactive. Keep-alive messages are exchanged if no message has been sent within a keep-alive period to ensure the corresponding node is both still

online and to avoid the connection being removed by a NAT device.

Keep-alive messages therefore serve a dual purpose, firstly keep-alive messages are used to detect the departure of ungraceful nodes. When a node that is part of the network receives a keep-alive message it responds by returning an acknowledgement message. Nodes that have left the network do not respond allowing failed connections to be detected and replaced. By proactively replacing failed connections nodes can ensure they remain well connected to a network overlay.

The second purpose of keep-alive messages is to prevent connections from becoming inactive and, as a result, being removed by NAT devices. Every time a packet is sent through a connection the NAT device at the other end restarts the timeout period. Keep-alive messages therefore serve as artificial packets, forcing NAT devices into resetting the timeout period and keeping the connection alive.

The keep-alive period is typically a fixed periodic interval uniform across all nodes in the network. The size of this keep-alive period interval is often determined by hand and is selected to fall within the timeout values of most NAT devices. The designers of BitTorrent [13] for example have set the default keep-alive period to $k = 120$ seconds. However RFC 1122 [8] recommends TCP stacks should wait for at least 2 hours between sending TCP keep-alive packets, accordingly NAT devices should not expire a TCP connection within this time. As a result a node in a BitTorrent network may be sending sixty times more keep-alive messages than is strictly necessary.

In this Chapter, we explore and empirically analyse algorithms that can efficiently adapt keep-alive intervals to match the timeout values of NAT devices and investigate the trade-offs of extending these intervals. More specifically:

- We formally explain and analyse iGlance an existing algorithm that roughly estimates the timeout values of NAT devices. Accordingly we propose a more accurate algorithm based upon traditional binary search that can efficiently find the timeout value of NAT devices.

- We evaluate the proposed algorithms using real network data from the RedHat9 BitTorrent distribution. Using our trace driven simulation platform we compare the adaptive algorithms to the standard periodic approach commonly used throughout P2P networks.

- Using distinct evaluation metrics we identify the trade-offs in terms of bandwidth and the incurred failure detection delay when aligning keep-alive intervals to timeout values.

- Finally we show by augmenting all the algorithms with a simple gossip mechanism it is possible to extend keep-alive intervals, thereby reducing bandwidth spent on maintenance without timing out connections while retaining a reasonable average failure detection delay.

8.2 iGlance:

The closest work to our own is iGlance [2], authored by David Barrett. iGlance an open-source Voice-Over-IP (VOIP) application that allows clients to adapt the size of the keep-alive interval to an approximate NAT timeout period. Each client creates two connections with the server, one for live traffic and the other as a test connection. The live traffic connection always operates on a known safe keep-alive period k_{live}, while the test connection experiments with an alternative keep-alive interval k_{test}. If a corresponding node acknowledges a keep-alive message through the test connection, k_{live} is

updated to reflect the new safe keep-alive period k_{test}, if k_{test} is larger than the current value of k_{live}. Furthermore the test connection interval k_{test} is doubled.

Should the corresponding node fail to acknowledge the keep-alive message, it is presumed to have timed out by the NAT device and the interval is halved. We implement a slight modification of this approach, by sending a keep-alive message through the live connection as well as the test connection when k_{test} period expires. Should only the test connection fail to respond we can safely assume a NAT device has expired the connection, whereas should both connections fail it is likely that the corresponding node has left the network.

Algorithm 7 iGlance_search()

if $ack.received$ **then**
  {ack.received indicates $t \leq k_{test}$}
  if $k_{test} > k_{live}$ **then**
    $k_{live} = k_{test}$
  end if
  $k_{test} = k_{test} \cdot 2$
else
  $k_{test} = (\frac{k_{test}}{2})$
end if

By experimenting with increasingly large keep-alive intervals iGlance can approximate the NAT timeout period and reduce the number of keep-alive messages sent, whilst also allowing the live connection to operate within a safe interval. However, the simple process of doubling and halving the keep-alive interval may never find the exact timeout value of a particular NAT device. Furthermore, iGlance's adaptive algorithm is only a small part of much larger overall VOIP application. The results of adapting to the timeout values of NAT devices, as far as we know, have never been empirically tested. In this Chapter, we implement and evaluate the iGlance algorithm comparing

it against the standard periodic approach and our own algorithm[1].

8.3 Approach

This section first describes the standard keep-alive algorithm before describing our Binary search based approach that is able to accurately and efficiently find the timeout value of NAT devices.

As described in Chapter 3, the Standard Keep-Alive (SKA) algorithm widely used to detect the departure of ungraceful nodes is also used to prevent connections being removed by NAT devices. A node assumes an entry in it's routing table to be online in the network for a duration of time defined by the keep-alive period k. Therefore, the time that a connection between two nodes can remain inactive is defined by the keep-alive period. If no message has been exchanged within a keep-alive period, keep-alive messages are exchanged to ensure the corresponding node is still online.

8.3.1 Binary Search

To improve on the iGlance algorithm that doubles and halves the test-interval we implement a Binary Search algorithm. As in iGlance the Binary search approach creates a live and a test connection, and updates the k_{live} and k_{test} intervals through positive and negative feedback. The binary search approach locates the NAT timeout period t by finding upper and lower bounds of t and selecting the middle value of these two points to progressively divide the search space in half.

In our context, the test connection sets a keep-alive period of k_{test} sec-

[1]We would like to thank David Barrett for providing us with extensive details of the iGlance adaptive algorithm.

onds. Each time the corresponding node successfully responds through the test connection, we learn the NAT timeout period t is greater than k_{test}. Accordingly we set k_{live} and min equal to current value of k_{test}. To begin with there currently is no upper bound on t, so the interval k_{test} is doubled. When the k_{test} interval exceeds t the NAT device removes the connection and the keep-alive message eventually fails, at this point we know t is lower than the current value of k_{test} so the upper bound max is set to k_{test}.

At this point the NAT timeout interval t must between lower and upper bounds min and max respectively, the binary search approach is to set k_{test} to the the middle of these two points $\frac{min+max}{2}$ and repeat the process. If the keep-alive message is acknowledged after k_{test} seconds min and k_{live} are updated, otherwise the test connection has timed out and upper bound max is updated. Finally k_{test} is set to $\frac{min+max}{2}$ and the process is repeated. This implementation of Binary search should find a NAT timeout period t in at most $2[log_2 t]$ steps.

Algorithm 8 binary_search()

if $ack.received$ **then**
  {ack.received indicates $t \leq k_{test}$}
  $min = k_{test}$
  $k_{live} = k_{test}$
  if $max = 0$ **then**
    $k_{test} = k_{test} \cdot 2$
  else
    $k_{test} = floor(\frac{min+max}{2})$
  end if
else
  $max = k_{test}$
  $k_{test} = floor(\frac{min+max}{2})$
end if

8.3.2 Gossiping Failures

To further reduce the failure detection delay we augmented all algorithms with a simple gossip mechanism as described in Chapter 5. This mechanism shares information regarding node failures with their mutual neighbours.

While gossip-based and similar mechanisms reduce the failure detection delay this comes at the cost of increased control overhead. Additional messages are required to inform mutual neighbours, who themselves check a node has failed. However as previous studies have shown this additional overhead is relatively small as the majority of maintenance consists of keep-alive messages with gossip messages only being sent when a failure is detected. The next section details our experimental methodology which we use to analyse and compare the algorithms described above.

8.4 Experimental Methodology

As throughout this thesis, to evaluate and compare the above algorithms our simulations are based on a publicly available BitTorrent tracker log [29]. Our simulations are trace-driven based on a portion of the five month logged period of the RedHat9 BitTorrent network. Enabling us to model the complex process of churn accurately and realistically.

As before, the arrival and departure time of any graceful node can be accurately determined by processing a tracker log. Although the departure time of ungraceful nodes are not listed within tracker logs, as all nodes periodically update their progress at thirty minute intervals we can assume they leave at most thirty minutes after their last update. Accordingly we can add a uniformly random time of at most thirty minutes to the last progress update of ungraceful nodes to determine their simulated departure time. Further-

more, all our experiments utilise a fail-stop model in which all nodes graceful and ungraceful do not inform their neighbours upon departing the network.

We simulate an unstructured network, trace-driven upon the RedHat9 BitTorrent tracker log data. Whilst online each simulated node creates and maintains a fixed number of connections D with other existing nodes selected at random, we experiment with a range of node degree values setting $D = 20, 30$ and 40. Again to avoid unnecessary confusion, we only simulate maintenance messages so that this work is general enough to be applied to any type of P2P network overlay regardless of it's structure.

In addition we also simulate the expiration of connections via NAT device timeouts. Each node n has a individual NAT timeout period t, if any connection to n is idle for longer than t seconds that connection is expired. In order to accurately simulate the NAT timeout values we generate our simulated timeout values according to [26] with a minimum imposed timeout value of two minutes. Guha and Francis in [26], find that only 35.6% of NAT devices keep an idle connection open for at least two hours. Furthermore, 21.8% of NATs expire idle connections after less than fifteen minutes, with the remaining 42.6% of NATs having a timeout period in somewhere between two hours and fifteen minutes. While connections in a unstructured network are unidirectional, data may flow both ways as a result we expire a connection according to the lower of the two timeout values of two connected nodes.

For each experiment the simulation begins cold, i.e without any peers. The first twelve hours of the network then act as a warm-up period as nodes populate and leave the network according the events given by the trace. Once the warm-up period is finished each node then creates D connections with existing nodes and we report the maintenance of these connections over the subsequent twelve simulated hours. Each experiment is repeated five times

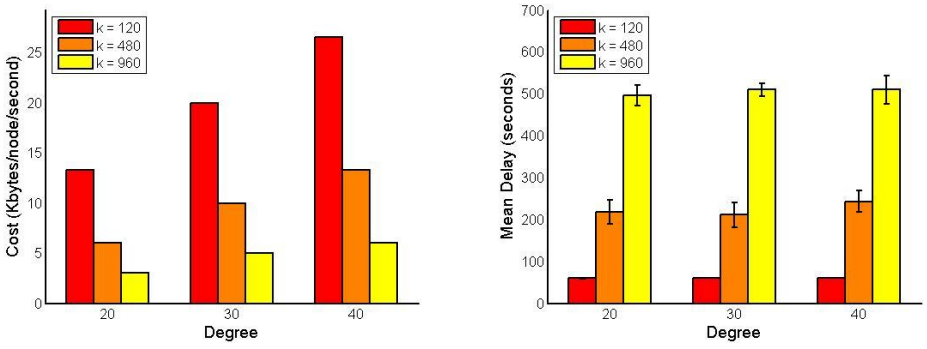


Figure 8.1: The performance of the SKA algorithm in terms of cost and mean failure detection delay whilst varying the number of connections each node maintains.

with the results averaged over these runs and standard deviation shown in the next section.

8.5 Results

Once more, we evaluate each mechanism based upon the the average bandwidth consumed per node per second online and the time that elapses between a failure occurring and subsequently being detected.

In this Chapter we compare the standard keep-alive (SKA), iGlance and Binary search algorithms against one another using the cost and failure detection delay as metrics. Furthermore we also implement an imaginary "optimal" algorithm called the Omni approach that has global knowledge of the timeout interval of NAT devices and matches the keep-alive period to that value. As we assume NAT devices do not know their own timeout period, clearly the global knowledge possessed by the Omni algorithm would be impossible to obtain in reality. The purpose of the Omni algorithm is to establish the baseline minimum amount of traffic incurred when extending

the keep-alive period as far as possible without expiring any connection.

Figure 8.1, shows how the Binary search approach reduces the cost of maintenance to approximately the same level as the standard periodic algorithm with a keep-alive interval of $k = 960$. Despite the additional overhead incurred by using two connections per neighbour, the average number of keep-alive messages sent and received is around five times lower than the default parameter of BitTorrent [13]. By adapting to NAT timeout values the live connection is able to use increasingly large keep-alive intervals that are also safe. As the live connection always uses the largest known safe keep-alive interval there is no risk of a NAT device expiring the connection before it is needed.

Figure 8.2a illustrates the iGlance algorithm by only doubling and halving the k_{test} interval it cannot accurately approximate many NAT timeout values and cannot achieve the lower cost values of the more accurate Binary Search approach.

Figure 8.2b and 8.2c highlight the inherent tradeoff, by extending the keep-alive intervals we unavoidably increase the average failure detection delay. The only consideration of the adaptive iGlance and Binary search approaches is to find the NAT timeout values of nodes present in the network. As these values may be as large as two hours the resulting failure detection delay is also large. Periodic keep-alive algorithms generally set the keep-alive interval to be lower than the majority of NAT device's timeout values as a consequence the average failure detection delay is also very low. By finding these timeout values the iGlance, Binary Search and Omni algorithms incur large failure detection delays when nodes eventually leave the network. From Figure 8.2 we can observe that the Omni approach is only optimal in terms of minimising bandwidth. By matching the timeout values immediately the

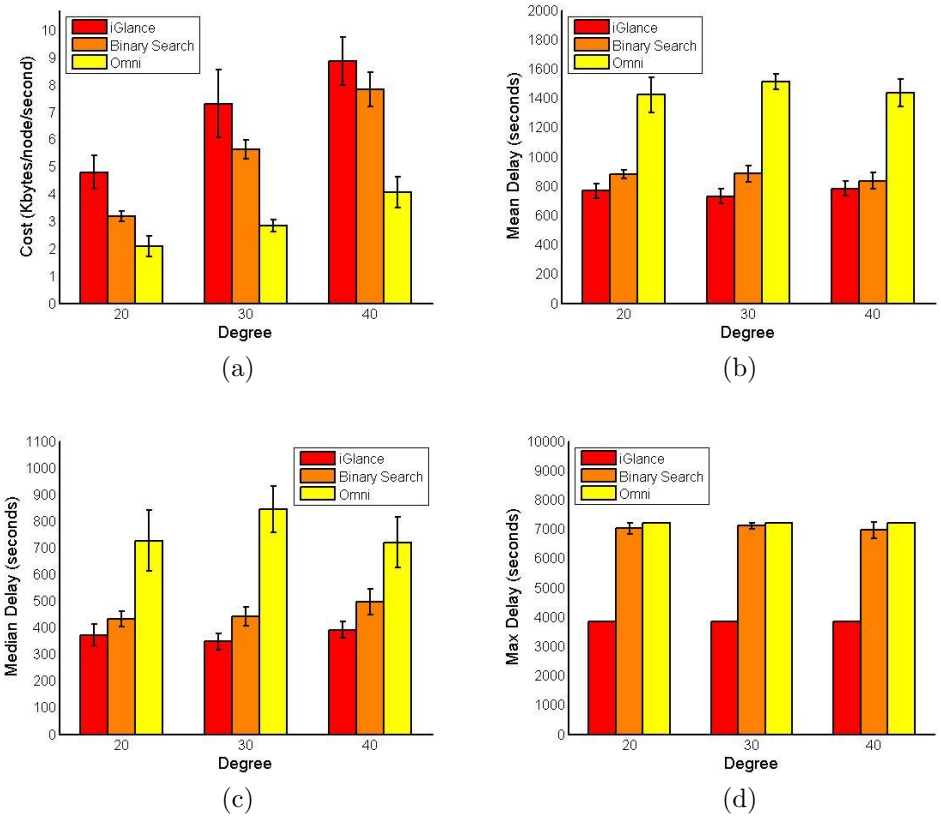


Figure 8.2: Comparison of the iGlance, Binary Search and Omni approaches in terms of cost, mean, median and maximum failure detection delay.

Omni approach incurs the largest failure detection delays as node must wait until the end of the next keep-alive period to detect a failure.

The largest delay incurred by the adaptive strategies is illustrated in Figure 8.2d. Strikingly the iGlance approach only incurs approximately half of the maximum delay incurred by the other adaptive approaches. The iGlance algorithm beginning at 120 seconds can at most double the test connection interval until it reaches 7680 seconds, a period just over two hours and therefore greater than the largest possible NAT timeout period. As a result the maximum safe live connection interval it can find is half that amount, 3840 seconds. Whereas the other Binary Search and Omni approaches can ex-

tend their live connection intervals further, lowering the cost but potentially incurring much larger delays.

Whereas the mean and median detection delay are equivalent for the periodic approach, the median delay is significantly lower for the adaptive algorithms than the incurred mean failure detection delay. This is due to some timeout values being low, resulting in a short keep-alive interval being found and also nodes failing while the live connection is using relatively small keep-alive intervals. While application designers wanting to create low overhead networks may find these delays acceptable, designers of high churn networks should be wary of incurring such high failure detection delays. Undetected failed neighbours could potentially occupy routing table entries for long periods of time, potentially incurring expensive message delays.

However, it should be highlighted that our experiments assume a fail-stop model, meaning all nodes leave the network ungracefully. In real networks the majority of nodes will leave gracefully, informing their neighbours upon departure and incurring no failure detection delay as a result. Our experiments therefore simulate the worst-case scenario maximising the possible incurred failure detection delay.

To rectify these large failure detection delays, further experiments shown in Figure 8.3, show a simple gossip mechanism as described in Chapter 5 and discussed in [16, 70] can be used to significantly reduce the mean and median failure detection delay of all the algorithms with little additional bandwidth cost. As the degree of node is increases the more mutual neighbours nodes share, further reducing the incurred failure detection delay. However gossip mechanisms are not designed to prevent connections expiring, only to inform mutual neighbours of the departure of neighboring nodes. As a result, the adaptive algorithms with gossip prevent connections becoming idle and

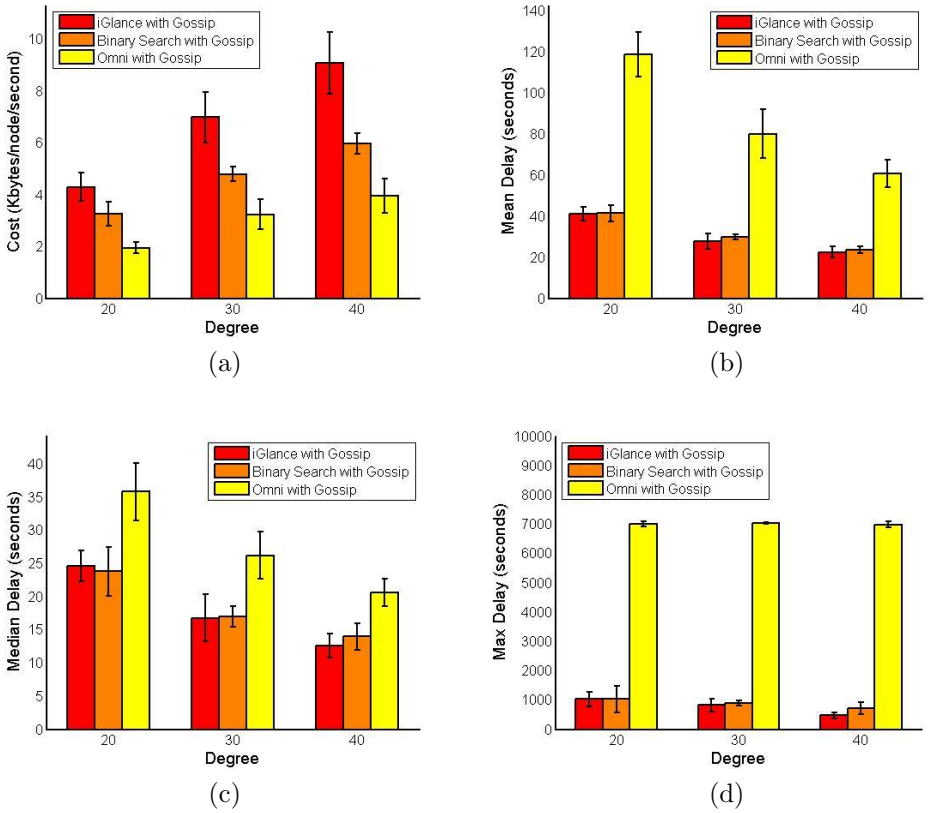


Figure 8.3: Comparison of the iGlance, Binary Search and Omni approaches with Gossip in terms of cost, mean, median and maximum failure detection delay.

keep the mean failure detection delay acceptably low. Of course the standard periodic approach can also be augmented with gossip further reducing the average failure detection delay, as shown in [70], but cannot ensure connections do not expire.

Figure 8.3 also shows that the delay incurred by the Omni approach is reduced the least by adding a gossip mechanism. As the Omni approach immediately extends the keep-alive period to the NAT timeout period it incurs the highest possible delays without expiring the connection. As all nodes also operate using these maximum safe intervals they also all incur the

same high delays, causing failures to being detected and thereby triggering gossip messages later. The relatively high mean and median delays incurred by the Omni approach can also be explained by the maximum delay being being seven orders of magnitude higher than the other adaptive approaches as shown in Figure 8.3d.

8.6 Summary

This Chapter presented a simple yet novel algorithm based upon Binary Search that adapts itself to the timeout period of NAT devices to ensure the connection for live traffic does not become idle. When compared to the existing iGlance algorithm and the widely deployed standard periodic approach, the Binary Search algorithm reduces the cost of maintenance incurred by nodes while preventing live connections expiring. In doing so, the failure detection delay is significantly increased as keep-alive intervals match the potentially large timeout values. However by augmenting these algorithms with a simple gossip mechanism the failure detection delay can be reduced to more acceptable levels.

Overall, this Chapter has shown it is possible to improve performance, in terms of bandwidth, by adapting to the timeout value of NAT devices. However tuning keep-alive intervals presents an inherent trade-off between the bandwidth spent and the incurred failure detection delay. All of the adaptive algorithms presented here reduce the cost of maintenance at the expense of increased failure detection delays. However, by adapting to NAT timeout values these algorithms ensure keep-alive messages are only sent when strictly necessary and no live connections are expired due to idleness.

9

Conclusions and Future Work

Failure detection in Peer-to-Peer networks aims to discover neighbours that have left the network ungracefully efficiently. Each entry in a routing table specifies a connection to another node, should a node leave the network without informing others, the neighbours of that node are left ignorant to the change in network topology. Once a failed routing table entry has been detected it can then be replaced.

This thesis presented three new algorithms - ProbKA, PredKA and BudgetProb - based on the principle that nodes become more reliable as they age, these algorithms reduce the failure detection delay when compared di-

rectly to the widely deployed standard periodic approach. In unstructured networks, they have been shown to reduce the median failure detection delay by as much as 24% on average and when combined with a complimentary gossip mechanism they further reduce the failure detection delay to 65% of the levels incurred by the SKA approach when also aided by gossip and similar levels of cost. Our results show whereas the mean delay is skewed by a small number of relatively large detection delays, the median detection delay is a more representative metric of the performance of our algorithms.

In structured networks, the stabilization process as featured in Chord affects our adaptive algorithms by replacing connections with older nodes with connections to newly appeared nodes. As our algorithms shorten the keep-alive period to these younger nodes they tend to incur significantly lower failure detection delays. We also showed in a more flexible version of Chord that our algorithms performed as they would do in a unstructured network.

All of our empirical analysis was done using trace-driven simulations based upon measured network data. With two independent BitTorrent tracker logs forming the basis of our simulation platform we ensured the complex process of churn was modeled both accurately and realistically.

Investigations into more robust ways of ascertaining a node's age highlighted the performance of our adaptive algorithms did not degrade significantly. When either indirectly estimating a node's age or when simply assuming a node had just appeared the performance of our adaptive algorithms remained fairly consistent. Dishonesty amongst nodes was shown to affect our algorithms by increasing the incurred failure detection delay, with the BudgetProb algorithm proving to be the most resilient against this type of behaviour. The algorithms presented in this thesis would however be prone to node's that maliciously misreported their own age without seeking benefit

- in terms of bandwidth - for themselves.

While the majority of this thesis focussed upon the failure detection aspect of keep-alive messages, we also examined their secondary purpose of preventing connections becoming idle and being expired by NAT devices. Chapter 8 presented and evaluated the simple yet novel Binary Search based algorithm which can quickly and accurately adapt itself to the timeout period of NAT devices, minimising the cost of maintenance but significantly increasing the delay between a failure occurring and it subsequently being detected. By augmenting adaptive algorithms with a simple gossip mechanism the failure detection delay was reduced to more acceptable levels.

Overall an failure detectors face an inherent trade-off between the bandwidth consumed and the incurred detection delay. The adaptive failure detection algorithms introduced in this thesis, increase the interval between success keep-alives as nodes age. By prioritising connections that are more likely to fail, the time between a failure occurring and being detected is reduced for most connections. In conclusion this thesis has shown that predictive and adaptive failure detection mechanisms that utilise current uptime can be successfully applied to limit the traffic overhead and improve the performance of maintenance protocols in unstructured and structured networks peer-to-peer networks.

9.1 Future Work

This section presents potential further investigations into the systems covered by this thesis.

- **Real-world Systems:** The comparison of the algorithms presented in this thesis has been conducted solely through the use of simulation.

Although this is the only method of evaluation available to us at this time, building a real-world implementation of such a system would certainly be a desirable undertaking. Having access to all the nodes and their clients within even a reasonably sized peer-to-peer network is not feasible. Instead, it may be possible to implement our algorithms by modifying individual clients, inserting them into a existing network and recording their performance. While measuring the bandwidth a client consumes is a fairly straight-forward process, accurately measuring the delay between the ungraceful failure of another node and its subsequent detection is not possible in a distributed system. Instead as graceful nodes do inform their neighbours upon leaving the network, we could estimate the incurred failure delay by taking the time remaining until the next keep-alive message would have occurred for each graceful departure.

- **Self-Learning:** The creation of an adaptive keep-alive algorithm which requires no prior knowledge also remains a priority. Although the PredKA, ProbKA and BudgetProb algorithms can be adapted to specific node session time distributions by adjusting the α and λ parameters, an algorithm that learns these parameters whilst in the network could potentially be deployed in any P2P overlay network to reduce the overhead of maintenance. Of course, as Chapter 4 described, accurately measuring the session times of nodes in a peer-to-peer network is a significant and contemporary problem even when the entire network can be observed for large periods of time. Therefore, measuring session times locally from a individual peer's perspective, who also may depart the network, may not be possible.

- **Massive Node Failure:** Further investigation is also needed to fully understand how strategies can best cope with massive node failure that occur suddenly. Designing effective responses to sudden widespread failure may lead to interesting and new network recovery mechanisms. We would also like to more thoroughly investigate the self-organising behaviour of P2P networks and specifically focus on how this affects their performance across several distributions of node session times.

- **Collaborative NAT Adaption:** It would also be possible to augment the binary search algorithm, presented in Chapter 8, by adding collaborative and randomized features. As many nodes may be connected to a single node x, the mutual neighbours of x could work together to find x's NAT timeout value. One possible drawback of a collaborative feature is neighbours could experiment with the same k_{test} values in parallel, to avoid this we could also implement a version of the collaborative algorithm that randomises the k_{test} interval uniformly within the range min and max.

Glossary:

Availability: The time, expressed as a percentage, that a node is online in a network during its entire lifetime.

Churn: The process of individual nodes joining and leaving the network.

Current Uptime: The current amount of time that has elapsed since a node joined the network.

Degree: The number of connections a node maintains.

Forced Disconnect: If all a node's neighbours leave the network without being replaced, that node is no longer connected to the network.

Latency: The elapsed time from a message between two peers being sent and subsequently received.

Lifetime: The total time between a node first joining a network and finally leaving. A node's lifetime can encapsulate several sessions.

Neighbours: The nodes a node is connected to.

Packet Loss: The probability of a packet being lost somehow in the underlying network.

Routing Table: The list of a node's connections.

Session Time: The elapsed time between a node joining a network and leaving.

Ungraceful Node: A node that leaves the network without informing its neighbours. Often referred to as a *failed node*. As opposed to a graceful node.

Bibliography

[1] L.O. Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS (N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications. In *Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 344. IEEE Computer Society, 2003.

[2] D. Barrett. iGlance. In *Presented at CodeCon*, 2006.

[3] S. A. Baset and H. G. Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–11, 2006.

[4] Bearshare. [Online], March 2010. http://www.bearshare.com.

[5] Ranjita Bhagwan, Stefan Savage, and Geoffrey M. Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on*, volume 2735 of *Lecture Notes in Computer Science*, pages 256–267. Springer Berlin / Heidelberg, February 2003.

[6] A. Biggadike, D. Ferullo, G. Wilson, and A. Perrig. NATBLASTER: Establishing TCP connections between hosts behind NATs. In *ACM SIGCOMM Asia Workshop*, volume 5, 2005.

[7] O. Boxma and B. Zwart. Tails in scheduling. *ACM SIGMETRICS Performance Evaluation Review*, 34(4):20, 2007.

[8] R. Braden. RFC 1122: Requirements for Internet Hosts Communication Layers, OCT 1989.

[9] Fabian E. Bustamante and Yi Qiao. Friendships that last: peer lifespan and its role in p2p protocols. In *Web content caching and distribution: proceedings of the 8th international workshop*, pages 233–246, Norwell, MA, USA, 2004. Kluwer Academic Publishers.

[10] Fabián E. Bustamante and Yi Qiao. Designing less-structured p2p systems for the expected high churn. *IEEE/ACM Trans. Netw.*, 16(3):617–627, 2008.

[11] Miguel Castro, Manuel Costa, and Antony Rowstron. Performance and dependability of structured peer-to-peer overlays. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 9, Washington, DC, USA, 2004. IEEE Computer Society.

[12] W. Chen, S. Toueg, and M.K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on computers*, pages 13–32, 2002.

[13] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, 2003.

[14] Russ Cox, Athicha Muthitacharoen, and Robert Morris. Serving dns using chord. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, March 2002.

[15] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. *ACM SIGOPS Operating Systems Review*, 35(5):202–215, 2001.

[16] Ivan Dedinski, Alexander Hofmann, and Bernhard Sick. Cooperative keep-alives: An efficient outage detection algorithm for p2p overlay networks. In *P2P '07: Proceedings of the Seventh IEEE International Conference on Peer-to-Peer Computing*, pages 140–150, Washington, DC, USA, 2007. IEEE Computer Society.

[17] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, New York, NY, USA, 1987. ACM.

[18] P. Druschel, E. Engineer, R. Gil, Y.C. Hu, S. Iyer, A. Ladd, et al. FreePastry. *Software available at http://www.cs.rice.edu/CS/Systems/Pastry/FreePastry*, March 2010.

[19] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1), 2001.

[20] eMule. [Online], March 2010. http://www.emule-project.net/.

[21] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-peer communication across network address translators. In *USENIX Annual Technical Conference*, pages 179–192, 2005.

[22] G. Ghinita and Yong Meng Teo. An adaptive stabilization framework for distributed hash tables. page 10 pp., april 2006.

[23] Gnutella. Gnutella protocol development wiki. [Online], March 2010. http://wiki.limewire.org/index.php?title=GDF.

[24] P. Godfrey, S. Shenker, and I. Stoica. Minimizing churn in distributed systems. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, page 158. ACM, 2006.

[25] C.M. Goldie and C. Klüppelberg. Subexponential distributions. *A Practical Guide to Heavy Tails: Statistical Techniques and Applications*, pages 435–459, 1998.

[26] S. Guha and P. Francis. Characterization and measurement of TCP traversal through NATs and firewalls. In *Proceedings of the Internet Measurement Conference (Berkeley, CA), October*, pages 49–681, 2005.

[27] S. Guha, Y. Takeda, and P. Francis. NUTSS: A SIP-based approach to UDP and TCP network connectivity. In *Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, page 48. ACM, 2004.

[28] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of dht routing geometry on resilience and proximity. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 381–394, New York, NY, USA, 2003. ACM.

[29] M. Izal. Bittorrent traces and tools. World Wide Web, March 2010. http://mikel.tlm.unavarra.es/ mikel/bt_pam2004/.

[30] M. Izal, Guillaume Urvoy-Keller, Ernst W. Biersack, P.A. Felber, A. Al Hamra, and L. Garcs-Erice. Dissecting bittorrent: Five months in a torrents lifetime. In Chadi Barakat and Ian Pratt, editors, *Passive and Active Network Measurement*, volume 3015 of *Lecture Notes in Computer Science*, pages 1–11. Springer Berlin / Heidelberg, 2004.

[31] D. Katz and D.D. Ward. Bidirectional forwarding detection, July 14 2009. US Patent 7,561,527.

[32] S. Krishnamurthy, S. El-Ansary, E. Aurell, and S. Haridi. Comparing Maintenance Strategies for Overlays. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 473–482. IEEE Computer Society, 2008.

[33] J. Ledlie, J. Shneidman, M. Amis, M. Mitzenmacher, and M. Seltzer. Reliability-and capacity-based selection in distributed hash tables. *Harvard University Computer Science, Tech. Rep*, 2003.

[34] Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek. Bandwidth-efficient management of dht routing tables. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 99–114, Berkeley, CA, USA, 2005. USENIX Association.

[35] Jinyang Li, Jeremy Stribling, Robert Morris, M. Frans Kaashoek, and Thomer M. Gil. A performance vs. cost framework for evaluating DHT design tradeoffs under churn. In *Proc. of the 24th Infocom*, March 2005.

[36] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the twenty-first annual*

symposium on Principles of distributed computing, pages 233–242. ACM New York, NY, USA, 2002.

[37] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Observations on the dynamic evolution of peer-to-peer networks. In *First International Workshop on Peer-to-Peer Systems*. Springer, 2002.

[38] Limewire. [Online], March 2010. http://www.limewire.com.

[39] Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys & Tutorials, IEEE*, pages 72–93, 2005.

[40] Ratul Mahajan, Miguel Castro, and Antony Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *In IPTPS*, 2003.

[41] G.S. Manku, M. Naor, and U. Wieder. Know thy neighbor's neighbor: the power of lookahead in randomized P2P networks. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 54–63. ACM New York, NY, USA, 2004.

[42] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. *Proceedings of IPTPS02, Cambridge, USA*, 1:2–2, 2002.

[43] μTorrent. [Online], March 2010. http://www.utorrent.com/.

[44] Napster. [Online], March 2010. http://www.napster.com.

[45] Slyck News. eDonkey2000 Nearly Double the Size of FastTrack, June 2005. http://www.slyck.com/news.php?story=814.

[46] R. Price. BitTorrent Tracker Parser. [Online], March 2010. http://www.cs.bham.ac.uk/∼rmp/files/trackerParserPerl.cgi.

[47] R. Price, T.T.A. Dinh, and Theodoropoulos G. Analysis of a self-organizing maintenance algorithm under constant churn. In *Proceedings of the 2008 International Symposium on Applications and the Internet-Volume 00*, pages 209–212. IEEE Computer Society Washington, DC, USA, 2008.

[48] R. Price and P. Tino. Still alive: Extending keep-alive intervals in p2p overlay networks. In *Collaborative Computing: Networking, Applications and Worksharing, 2009. CollaborateCom 2009.*, pages 1 –10, Nov. 2009.

[49] R. Price and P. Tino. Adapting to NAT timeout values in P2P Overlay Networks. In *Proceedings of the 2010 The Ninth International Workshop on Performance Modeling, Evaluation, and Optimization of Ubiquitous Computing and Networked Systems*. IEEE Computer Society Washington, DC, USA, 2010.

[50] Sylvia Ratnasamy, Paul Francis, Scott Shenker, and Mark Handley. A scalable content-addressable network. In *In Proceedings of ACM SIGCOMM*, pages 161–172, 2001.

[51] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a dht. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[52] J. Ritter. Why gnutella cant scale. no, really. *Preprint, http://www.darkridge.com/ jpr5/doc/gnutella.html*, 2001.

[53] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN-simple traversal of user datagram protocol (UDP) through network address translators (NATs). 1993.

[54] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proc. of the IFIP/ACM Int. Conf. on Distributed Systems Platforms Heidelberg*, pages 329–350. Springer-Verlag, 2001.

[55] S. Saroiu, P.K. Gummadi, S.D. Gribble, et al. A measurement study of peer-to-peer file sharing systems. In *proceedings of Multimedia Computing and Networking*, volume 2002, page 152, 2002.

[56] Stefan Saroiu, Krishna P. Gummadi, and Steven D. Gribble. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia Syst.*, 9(2):170–184, 2003.

[57] S. Sen and J. Wang. Analyzing peer-to-peer traffic across large networks. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 137–150. ACM New York, NY, USA, 2002.

[58] R. Shimizu. On a lack of memory property of the exponential distribution. *Annals of the Institute of Statistical Mathematics*, 31(1):309–313, 1979.

[59] A. Singla and Ch Rohrs. Ultrapeers: Another step towards gnutella scalability, November 2002.

[60] Kelvin C. W. So and Emin Gün Sirer. Latency and bandwidth-minimizing failure detectors. *SIGOPS Oper. Syst. Rev.*, 41(3):89–99, 2007.

[61] Moritz Steiner, Taoufik En-Najjary, and Ernst W. Biersack. Long term study of peer behavior in the kad dht. *IEEE/ACM Trans. Netw.*, 17(5):1371–1384, 2009.

[62] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[63] D. Stutzbach and R. Rejaie. Capturing accurate snapshots of the Gnutella network. In *Proceedings IEEE INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 4, 2005.

[64] D. Stutzbach and R. Rejaie. Characterizing the two-tier Gnutella topology. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, page 403. ACM, 2005.

[65] Daniel Stutzbach and Reza Rejaie. Evaluating the accuracy of captured snapshots by peer-to-peer crawlers. In *Passive and Active Measurement Workshop, Extended Abstract*, pages 353–357, 2005.

[66] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *IMC '06: Proceedings of the 6th ACM SIGCOMM on Internet measurement*, pages 189–202, New York, NY, USA, 2006. ACM Press.

[67] Daniel Stutzbach, Reza Rejaie, and Subhabrata Sen. Characterizing unstructured overlay topologies in modern p2p file-sharing systems. In *Proc. of Internet Measurement Conference (IMC)*, 2005.

[68] R. Van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proceedings of Middleware*, volume 98, pages 55–70, 1998.

[69] Vuze. [Online], March 2010. http://www.vuze.com/.

[70] SQ Zhuang, D. Geels, I. Stoica, and RH Katz. On failure detection algorithms in overlay networks. In *Proceedings IEEE INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, 2005.