Still Alive: Extending Keep-Alive Intervals in P2P Overlay Networks

Richard Price and Peter Tino
School of Computer Science
University of Birmingham
Birmingham, United Kingdom
Email: $\{R.M.Price, P.Tino\}@cs.bham.ac.uk$

***Abstract*—Nodes within existing P2P networks typically exchange periodic keep-alive messages in order to maintain network connections between neighbours. This paper investigates a number of algorithms which allow each individual connections to extend the interval between successive keep-alive messages based upon the likelihood that a corresponding node will remain in the system.**

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 that increase the interval between successive keep-alive messages as nodes become more reliable.

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. Failed connections can incur expensive lookup timeouts and increases the network's vulnerability to partitioning. We extensively analyse the properties of these algorithms and compare them to the standard periodic keep-alive mechanism using simulations based upon measured network data.

I. Introduction:

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. Two connected nodes are often referred to as neighbours, with each node's list of neighbours being called it's routing table. 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.

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. Therefore all connections are typically both monitored and maintained in unstructured networks; such as Gnutella [1] and BitTorrent [2] and structured networks such as Chord [3], Pastry [4] and Bamboo [5]. 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 proactive recovery mechanism replacing broken connections 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. This paper investigates these trade-offs by comparing two alternative maintenance strategies using a simulation based upon measured network data.

The contributions of this paper are:

- We analytically derive the probability of nodes remaining online according to the time they have already spent in the network given the distribution of session times.
- Accordingly we propose three alternative algorithms that reduce the number of keep-alive messages sent to nodes as they become more likely to remain in the system.
- We evaluate the proposed algorithms using real network data from the RedHat9 BitTorrent distribution and network data from LegalTorrents[1]. Using our trace driven

[1]We would like to thank Matson Systems for providing us fully anonymized data from the LegalTorrents.com community trackers.

simulation platform we compare our algorithms to the standard periodic approach commonly used throughout P2P networks.

The rest of the paper continues as follows. Section two provides an overview of existing work relating to techniques commonly used to maintain connections within P2P overlays. Section three explains the well established standard keep-alive mechanism, details how we can predict the remaining session time of nodes and we introduce our Probabilistic, Predictive and Budget based keep-alive algorithms. Our experimental methodology is described in section four. Section five presents the results of the simulated experiments, comparing our algorithm with the standard keep-alive algorithm. Finally, we conclude in section seven.

II. Related Work:

While research often focuses upon making P2P network overlay's flexible, efficient and robust [3], [4], [5], the research community has identified reducing the cost of maintenance as an open problem that is important in terms of overall performance [6].

The simplest alternative to periodic maintenance is to not send any keep-alive messages at all and reactively recover from failed connections as network messages timeout. This solution is ineffective as failures are only detected when connections are needed.

The designers of Bamboo [5] highlight the benefits of periodic maintenance, showing reactive recovery may add to network congestion by creating a positive feedback loop and exacerbating existing problems. Bamboo also calculates the expected round-trip timeout (RTTO) for each neighbour based upon previous observations of round-trip times. If an keep-alive acknowledgement is not received once the RTTO has expired a neighbour is considered to have failed.

In [7], the authors propose an adaptive keep-alive mechanism and show how it can be applied to the Chord DHT [3]. 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. However, as exponential distributions are memoryless they cannot be used to predict remaining uptime based upon current uptime as we do in this paper.

Work by Dedinski et al [8], 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 own keep-alive messages so they are sent in sequence by it's common neighbors. Our approach is complimentary to the cooperative-keep-alive algorithm but does not rely on the cooperation of a group of nodes.

In [9] 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 minimises the failure detection delay which given a bandwidth budget achieves the smallest average delay between failures occurring and their subsequent detection. While their second bandwidth minimising algorithm will ensure a specified delay target is reached whilst consuming as small amount of bandwidth as possible.

The prediction mechanism in [9] relies on the strong correlation between a node's previous session time(s) and it's next session time. However knowledge of each individual node's previous session time(s) may not available and may be impossible to learn as nodes may be unlikely to reconnect to one another especially in unstructured networks. Whereas the prediction mechanism used in this paper relies upon knowledge of a population, or subset of the population's, previous session times and current node uptime which are likely to be more readily available in most networks.

The authors of [10] compare the performance of five distributed failure detectors empirically. They show nodes can reduce the average delay between failures occurring and subsequently being detected by sharing information with one another. The study compares four alternative sharing algorithms with the standard keep-alive algorithm which we also use as a baseline. Unlike in this paper, none of the distributed failure detectors examined in [10] are predictive.

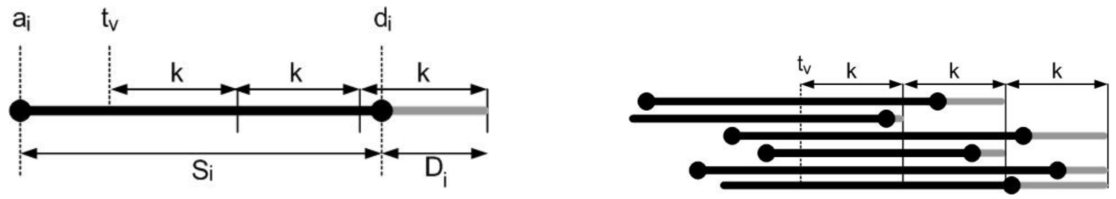
In [11], Castro et al. examine the cost of maintaining structured P2P overlay networks presenting a self-tuning algorithm that adapts itself to the environment. By estimating the number of nodes in the network and the failure rate of these nodes; suitable maintenance rates can be set to meet reliability requirements. However, unlike in [11] our proposed algorithms are not limited to specific topological structures and can be applied to any kind of network.

Past work has investigated selecting neighbours that are likely to be the most reliable to improve the performance of P2P networks [12]. By selecting the oldest available nodes as neighbours, Bustamante and Qiao [12] show that these connections tend to last longer and therefore need to be replaced less often producing more robust networks.

Recently work by Li et al in [13], 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. By learning of new connections via lookups and evicting existing connections that are likely to have failed, the Accordion protocol can find a suitable trade off between the performance of lookups and the incurred bandwidth cost of many connections. Although Accordion does utilise the standard keep-alive mechanism it does adapt it's behaviour according to the probability of nodes being online.

III. Approach:

In this section we first describe the standard approach of detecting failures used by many P2P networks, then analytically show how the distribution of node session times can be used to predict the likelihood of nodes remaining online. We then use this analysis to propose two algorithms that send fewer keep-alive messages as the expected time a node will remain in the network increases as nodes age.



(a) A node's session time $S_i = d_i = a_i$, where D_i is the delay period

(b) The average delay period is $k/2$

Fig. 1. The standard keep-alive algorithm

A. The Standard Keep-Alive Algorithm:

The Standard Keep-Alive (SKA) algorithm is widely used to detect the departure of ungraceful nodes. Ungraceful nodes do not inform their neighbours upon leaving the network, leaving any incoming connections ignorant to the change in network topology. 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 no message has been exchanged within a keep-alive period, keep-alive messages are exchanged to ensure the connection is still alive.

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. In practice unacknowledged keep-alive messages are re-sent multiple times at short intervals. This minimises the risk of false positives where a sent or acknowledged keep-alive message has been somehow lost in the underlying network.

Many existing P2P systems send keep-alive messages according to a 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 [2] for example have set the default keep-alive period to $k = 120$ seconds. Not only is this strategy 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 $(N \cdot D) \cdot 2$ keep-alive messages sent and subsequently acknowledged every k seconds.

Figure 1a illustrates node i arriving in the network at time a_i and leaving at time d_i, it's *session time* $S_i = d_i - a_i$. At time t_v node v connects to node i and begins sending periodic keep-alive messages with an interval of k seconds. 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 period D_i during which node v falsely believes that node i is still present within the network. We call this period the *failure detection delay*.

The size of the failure detection 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 nodes already present within the network as neighbours, the join time t_v can occur at any point during a nodes session time with equal probability. Therefore the leave time d_i of neighbours falls uniformly at random within any single keep-alive interval. The average delay between a node leaving a network and subsequently being detected is $k/2$ using a periodic keep-alive mechanism as illustrated in Figure 1b.

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 many factors such as current network conditions which may be difficult to obtain and furthermore may change rapidly.

B. Predicting a Node's Online Status:

Although early studies of popular P2P networks reported node session times could be modeled by an exponential distribution; more recent studies have shown they can be more accurately described by Pareto or Weibull distributions [14], [15], [12].

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. However studies such as [14], [15], [12] have shown nodes that spend longer in the network become more likely to remain in the network for longer durations. In this paper, we analyse how this widespread property of P2P networks can be used to reduce the number of maintenance messages needed as nodes age.

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 that node has left the network.

Accordion introduced in [13], calculates this probability explicitly by assuming each node's session time is drawn from a heavy-tailed Pareto distribution as reported by [16]. The 'freshness' of each entry in a node's routing table is then estimated, replacing entries that are deemed likely to have failed; thereby expanding and collapsing the size of a node's routing table.

Li et al in [13] 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 as shown in (1):

$$P_{online} = P(session > (T_{alive} + T_{since}) | session > T_{alive}) \tag{1}$$

Although a heavy-tailed Pareto distribution is limited by having a minimum possible value determined by the scale parameter β, these short-comings can be overcome by using a shifted Pareto distribution as highlighted by [17].

Importantly Pareto distributions are heavy-tailed, as a result as T_{alive} grows the less dependent P_{online} becomes on T_{since}. 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.

The recent study by Stutzbach et al in [15], explains how a Weibull distribution can be used to better describe the session times of nodes in multiple BitTorrent networks. Due to their centralised tracker, BitTorrent networks can provide highly accurate data to the nearest second regarding the time that individual nodes join a network, the duration of their stay and when they depart. This gives researchers a valuable insight into the poorly understood process of *churn*; the collective effect of many nodes joining and leaving a network in parallel.

A Weibull distribution, as shown in (2), are 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 $\alpha = 1$.

$$P(session < t) = 1 - e^{-(t/\lambda)^{\alpha}} \tag{2}$$

As nodes spend longer in the network the probability of them remaining online increases allowing us to reduce the number of keep-alive messages being sent. Rather than using the limited Pareto distribution we utilise the more flexible Weibull distribution, consequently the probability that a node remains online, after being online T_{alive} and observed T_{since} seconds ago, is:

$$P_{online} = \frac{e^{-((T_{alive}+T_{since})/\lambda)^{\alpha}}}{e^{-(T_{alive}/\lambda)^{\alpha}}} \tag{3}$$

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. We therefore propose three solutions with this aim in mind.

C. ProbKA: The Probabilistic Keep-alive Algorithm:

Similar to the standard keep-alive algorithm, the Probabilistic Keep-Alive (ProbKA) algorithm specifies a regular interval k for all connections. Although the interval k is the same size for all connections, each connection is maintained independently. Once this interval has expired the ProbKA algorithm examines the individual connection and determines the likelihood that the corresponding neighbour remains online. With probability $P_{offline} = 1 - P_{online}$ (where P_{online} is given by (3)) a keep-alive message is sent to the corresponding node to determine it's online status.

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.

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.

D. PredKA: The Predictive Keep-alive 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. The disadvantage 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 (2) and probability of nodes remaining online given by (3) as a basis; we can ensure that any network message sent between two connected nodes will be delivered with probability of at least P_{online}, whilst adjusting the keep alive period according to value of T_{since} as given by:

$$T_{since} = (T_{alive}^{\alpha} - \lambda^{\alpha} \cdot log(P_{online}))^{1/\alpha} - T_{alive} \tag{4}$$

Using (4), 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_{online}. As this likelihood increases as T_{alive} increases node send fewer keep-alive messages to their longer-lived neighbours.

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.

E. BudgetProb: Probabilistic Keep-alives with a Budget:

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 paper we experiment with several values of the bandwidth budget β, equivalent to the bandwidth consumed by the standard periodic approach using a range keep-alives 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. The BudgetProb algorithm specifies the interval for each connection according to (5), where p is the packet size of keep-alive message:

$$k = \frac{(p * 2)}{\beta} / \frac{P^i_{offline}}{\sum_{j=1}^{N} P^j_{offline}} \forall i, 1 \leq i \leq N \tag{5}$$

F. Gossiping Failures:

To further reduce the failure detection delay we augmented our ProbKA and PredKA algorithms with a simple gossip mechanism. This mechanism shares information regarding node failures with their mutual neighbours.

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. Further examples of alternative and more complex sharing-based, gossip-based and flooding-based mechanisms are evaluated in [10], [8].

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.

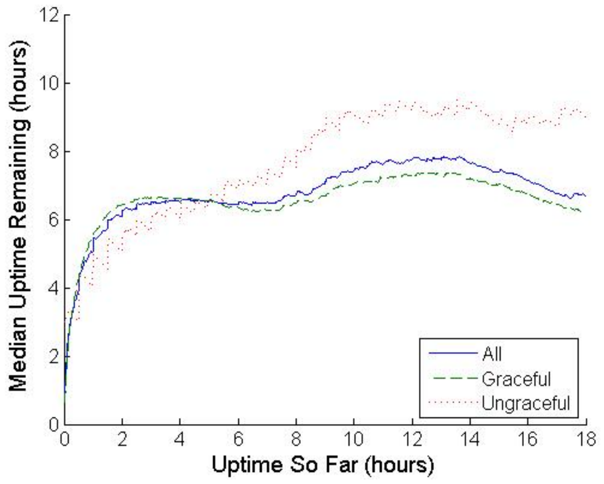


Fig. 2. The median time remaining in the network increases as nodes spend longer in the network.

IV. 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 [18] and well researched BitTorrent tracker log [15], [19], 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 [19]. Many studies have shown short-lived peers tend to make up a large proportion of sessions in many different P2P networks [15], [20], [21]. 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.

Although the trace data contains a small number of gaps greater than four minutes, the largest continuous measured period is over 79 days. Figure 2 shows how the median time remaining in network increases as nodes spend longer in the network during this period.

Stutzbach et al. in [15] 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 message, 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 reduced 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.

The accuracy of BitTorrent tracker logs often compares favorably to other sources of network data, especially crawler based traces that periodically probe a subset of nodes within a network at regular intervals. As crawlers must progressively probe the network; the more nodes a crawler incorporates increases the interval between successive probes, this interval currently ranges anywhere between four and thirty minutes [15], [11], [16]. Crawler based techniques cannot accurately capture session times smaller than the granularity of the network trace. As a result, BitTorrent tracker logs are amongst the most extensive and realistic sources of peer session times currently available.

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 nodes update their progress periodically at thirty minute intervals we can 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 Redhat9 BitTorrent network during a portion of the five month period contained in the tracker log. We use the tracker log solely to specify when each node joins the simulated network and subsequently leaves.

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 regardless of it's structure.

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 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 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 (3) using parameters $\alpha = 0.39$ and $\lambda = 3962$ the RedHat network data and $\alpha = 0.41$ and $lambda = 2632.25$ for the LegalTorrents data.

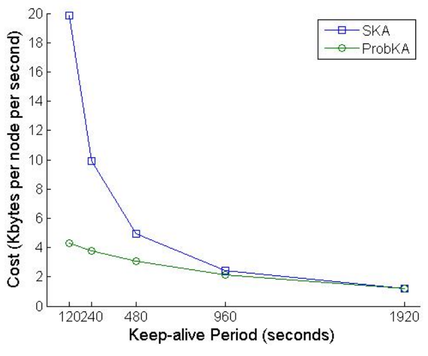
V. Results:

We evaluate each mechanism based upon three main criteria cost, the average failure detection delay and maximum failure detection delay. The failure detection delay is time that elapses between a failure occurring and subsequently being detected. The cost of each strategy equates to the average bandwidth consumed per node per second online. Formally the cost C is; the number of keep-alive messages sent s and acknowledged a, multiplied by the size of each keep-alive message p which is then divided by the sum of all node session times T as given in (6). These performance metrics have also been used in other evaluations of failure detection algorithms including [9], [10]. 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.

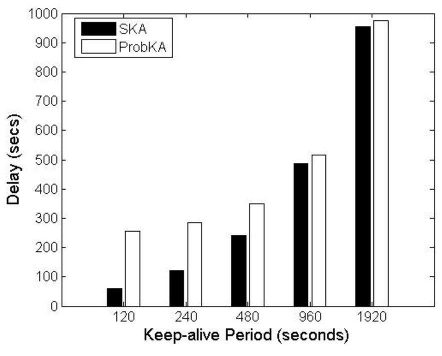
$$C = \frac{(s + a) \cdot p}{T} \tag{6}$$

Figure 3a compares the cost incurred by our probabilistic keep-algorithm (ProbKA) with a probability threshold $P_{thresh} = 99\%$ and the widely used 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.

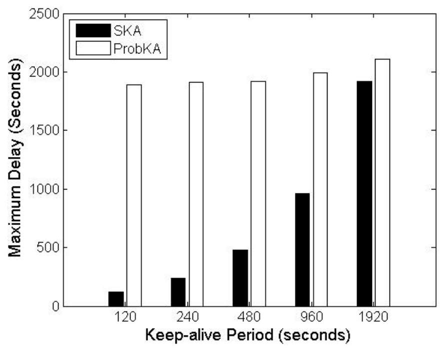
Nodes actually gain very little information from acknowledged keep-alive messages. They only serve to inform a node that it's neighbour was online at the time of receipt. It does not guarantee that node will remain online or that the connection will still be active if data is sent along it. An acknowledged 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



(a)



(b)



(c)

Fig. 3. Performance comparison of strategies in terms of cost and delay.

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 and PredKA algorithm also reduce 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 average and maximum delay using both the SKA and ProbKA algorithms with a range of interval sizes. The average 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.

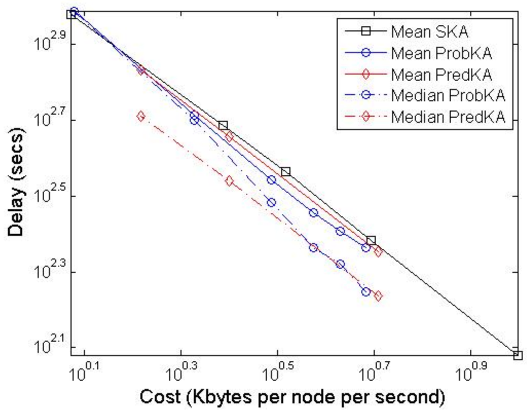
Figure 3b shows the average failure detection delay incurred by the SKA and ProbKA algorithms. The SKA algorithm by regularly checking each connection ensures node failures are detected and replaced consistently. As illustrated earlier in Figure 1b the average delay is very close to $k/2$ as node failures can occur uniformly at random within the interval of k. The ProbKA algorithm by extending these intervals also extends the average delay. Lower values of P_{thresh} where also tested but resulting in lower costs and higher delays.

Figure 3c 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.

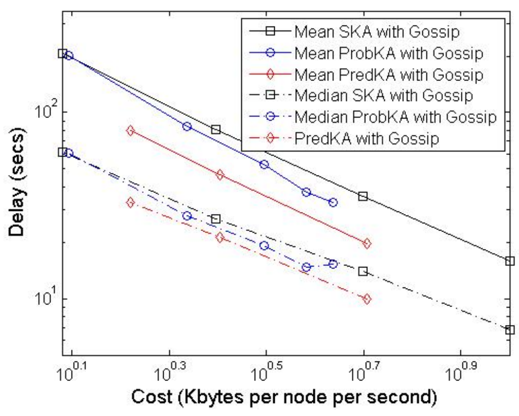
While the ProbKA algorithms increase the interval between successive keep-alive messages, the self-organising nature of the network further facilitates the extension of these intervals. Stutzbach et al showed [14] that as networks age, long-lived peers tend to become connected to one another. This forms 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.

We also examined the behaviour of the ProbKA algorithm as the network ages. Our observations showed as the simulated network begins to age the cost of maintenance per node gradually decreases while the average failure detection delay remains largely the same. Further experiments also show the PredKA algorithm behaves in a similar fashion.

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 ProbKA and PredKA algorithms exploit this behaviour by extending the interval between successive keep-alive messages as nodes, and therefore the connections between nodes, age. As the simulation progresses and the network ages, the connections between nodes also age; causing fewer and fewer keep-alive messages



(a)



(b)

Fig. 4. Mean and median performance versus cost comparison of the SKA, ProbKA and PredKA algorithms

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 3 does not clearly illustrate the how the ProbKA and SKA algorithms compare against one another. Figure 4 shows a cost versus performance comparison in log-log scale of the SKA, ProbKA and PredKA algorithms allowing a direct analysis to be made. Figure 4a shows that ProbKA and PredKA algorithms consistently reduces the average failure detection delay when compared to the SKA algorithm at similar cost levels. Figure 4a also shows the median delay incurred by the ProbKA and PredKA algorithms 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 delays are the same. This indicates the mean failure detection delay incurred by the ProbKA and PredKA algorithms is skewed by a small number of relatively large failure detection delays as shown in Figure 3c.

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

Furthermore, Figure 4 shows the performance of the PredKA algorithm is comparable to ProbKA algorithm but is very sensitive to adjustments to the P_{online} parameter. The results shown set $P_{online} = 0.97, 0.98$ 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 ProbKA algorithm. Despite these differences the ProbKA and PredKA algorithm perform around a similar level with the ProbKA algorithm being slightly more flexible.

Figure 4b shows that augmenting the SKA, ProbKA and PredKA algorithms with a gossip mechanism causes the average 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 gossip mechanism. When failures are detected, by using a simple gossiping 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. These results show that our approach complements other optimisations to the standard keep-alive mechanism [10], [8].

However, even with the gossip mechanism the maximum delay by our probabilistic mechanism 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

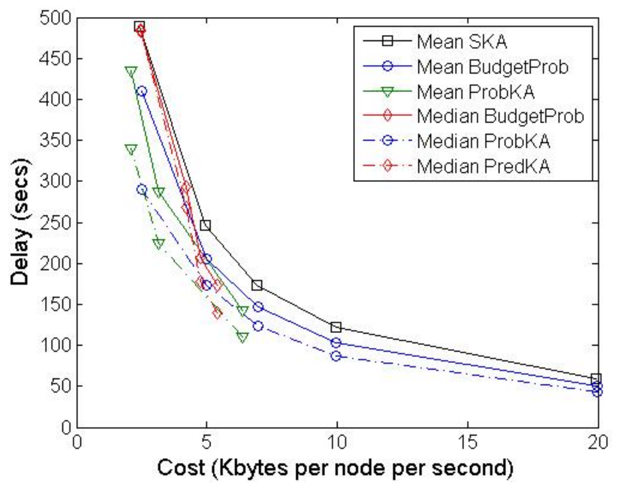


Fig. 6. Mean and median performance versus cost comparison of the SKA, ProbKA and PredKA algorithms using network data from LegalTorrents.

the failure for themselves which may incur long detection delays. More advanced gossip mechanisms such as a flooding mechanism used in [8] or epidemic based approaches studied in [10] 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 5 compares the BudgetProb and SKA algorithms with and without gossip side-by-side. The main advantage of the BudgetProb algorithm is that allows a 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 5a shows our adaptive BudgetProb approach reduces the median delay over 20% on average. When combined with the simple gossip mechanism our BudgetProb algorithm performance increases, Figure 5b shows the mean and median delay is reduced by 35% 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.

Finally, Figure 6 shows the performance of all three adaptive algorithms upon the LegalTorrents network data which contains the session data 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. 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.

VI. Future Work:

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. Such devices may operate with limited resources and often utilise a Least-Recently Used algorithm to ensure idle connections are not kept for prolonged periods. However the behaviour of these devices not only vary greatly from one another but also over time. Current solutions are often ad-hoc depending upon rules of thumb or crude estimations. We are currently exploring alternative keep-alive algorithms that can flexibly handle the behaviour of these devices whilst minimsing the number of keep-alive messages sent.

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 from node failures as they occur could potentially be deployed in any P2P overlay network to reduce the overhead of maintenance.

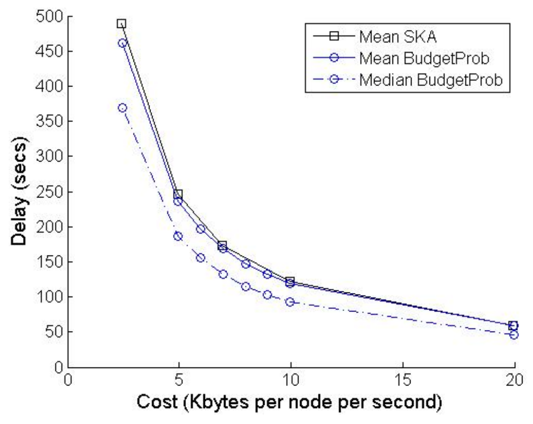
Further investigation is also needed to fully understand how strategies can best cope with massive node failure that occur suddenly. Designing effective responses to sudden widespread failure may lead to interesting and new network recovery mechanisms. We would also like to more thoroughly investigate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

VII. Conclusion:

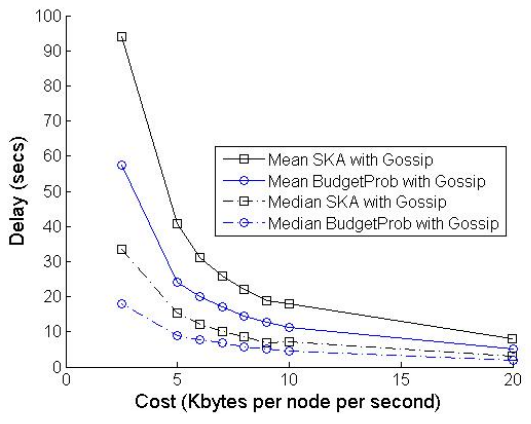
This paper presented three new algorithms based on the principle that nodes become more reliable as they age, these algorithms reduce the average failure detection delay when compared directly to the widely deployed standard periodic approach. In doing so they reduce the mean and median failure detection 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 modeled 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



(a)



(b)

Fig. 5. Performance versus Cost comparison of budget based algorithms.

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.

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 paper has shown that predictive mechanisms can be successfully used to reduce the average failure detection delay whilst limiting traffic overhead of maintenance protocols and there is significant potential for further work.

References

[1] "The gnutella protocol specification v0.4," World Wide Web, http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf.

[2] I. BitTorrent, "Bittorrent," World Wide Web, http://www.bittorrent.com/.

[3] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A scalable Peer-To-Peer lookup service for internet applications," in *Proceedings of the 2001 ACM SIGCOMM Conference*, 2001, pp. 149–160.

[4] A. I. T. Rowstron and P. 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*. Springer-Verlag, 2001, pp. 329–350.

[5] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling churn in a dht," in *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[6] S. El-Ansary and S. Haridi, "An overview of structured overlay networks," in *Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless and Peer-to-Peer Networks*, 2005.

[7] G. Ghinita and Y. M. Teo, "An adaptive stabilization framework for distributed hash tables," in *IPDPS*, 2006.

[8] I. Dedinski, A. Hofmann, and B. 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*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 140–150.

[9] K. C. W. So and E. G. Sirer, "Latency and bandwidth-minimizing failure detectors," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 89–99, 2007.

[10] S. Zhuang, D. Geels, I. Stoica, and R. Katz, "On failure detection algorithms in overlay networks," in *Proceedings IEEE INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 3, 2005.

[11] R. Mahajan, M. Castro, and A. Rowstron, "Controlling the cost of reliability in peer-to-peer overlays," in *In IPTPS*, 2003.

[12] F. Bustamante and Y. Qiao, "Friendships that last: Peer lifespan and its role in P2P protocols," in *Proc. of IWCW*. Springer, 2003.

[13] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek, "Bandwidth-efficient management of dht routing tables," in *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2005, pp. 99–114.

[14] D. Stutzbach, R. Rejaie, and S. Sen, "Characterizing unstructured overlay topologies in modern p2p file-sharing systems," in *In Proc. of Internet Measurement Conference (IMC)*, 2005.

[15] D. Stutzbach and R. Rejaie, "Understanding churn in peer-to-peer networks," in *IMC '06: Proceedings of the 6th ACM SIGCOMM on Internet measurement*. New York, NY, USA: ACM Press, 2006, pp. 189–202.

[16] S. Saroiu, K. P. Gummadi, and S. D. Gribble, "Measuring and analyzing the characteristics of napster and gnutella hosts," *Multimedia Syst.*, vol. 9, no. 2, pp. 170–184, 2003.

[17] D. Leonard, V. Rai, and D. Loguinov, "On lifetime-based node failure and stochastic resilience of decentralized peer-to-peer networks," in *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2005, pp. 26–37.

[18] M. Izal, "Bittorrent traces and tools," World Wide Web, March 2009, http://mikel.tlm.unavarra.es/ mikel/bt_pam2004/.

[19] M. Izal, G. Urvoy-keller, E. W. Biersack, P. A. Felber, and A. A. Hamra, "Dissecting bittorrent: Five months in a torrents lifetime," 2004, pp. 1–11.

[20] S. Saroiu, P. Gummadi, S. Gribble *et al.*, "A measurement study of peer-to-peer file sharing systems," in *Proceedings of Multimedia Computing and Networking*, vol. 2002, 2002, p. 152.

[21] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan, "Measurement, modeling, and analysis of a peer-to-peer file-sharing workload," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, pp. 314–329.