©2006 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. The definitive version of this paper has been published in Third International Workshop on Hot Topics in Peer-to-Peer Systems (Hot-P2P) in conjunction with the IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006), 2006, 10.1109\/ipdps.2006.1639710.

A Scalable Algorithm to Monitor Chord-based P2P Systems at Runtime

Andreas Binzenhöfer[1], Gerald Kunzmann[2], and Robert Henjes[1]

[1]University of Würzburg
Institute of Computer Science
Am Hubland, 97074 Würzburg, Germany
{binzenhoefer, henjes}@informatik.uni-wuerzburg.de

[2]Technische Universitaet Muenchen
Institute of Communication Networks
80290 Munich, Germany
gerald.kunzmann@tum.de

Abstract

Peer-to-peer (p2p) systems are a highly decentralized, fault tolerant, and cost effective alternative to the classic client-server architecture. Yet companies hesitate to use p2p algorithms to build new applications. Due to the decentralized nature of such a p2p system the carrier does not know anything about the current size, performance, and stability of its application. In this paper we present an entirely distributed and scalable algorithm to monitor a running p2p network. The snapshot of the system enables a telecommunication carrier to gather information about the current performance parameters of the running system as well as to react to discovered errors.

1 Introduction

In recent years peer-to-peer (p2p) algorithms have been widely used throughout the Internet. So far, the success of p2p paradigms was mainly driven by file sharing applications. Despite their reputation, however, p2p mechanisms also offer the solution to many problems faced by telecommunication carriers today [8]. Compared to the classic client-server architecture they are decentralized, fault tolerant, and cost effective alternatives. Those systems are highly scalable, do not suffer from a single point of failure, and require less administration overhead than existing solutions. In fact there are more and more successful p2p based applications like Skype [12], a distributed VoIP solution, Oceanstore [4], a global persistent data store, and even p2p based network management [10].

One of the main reasons why telecommunication carriers are still hesitant to build p2p applications is the lack of control a provider has over the running system. At first, the system appears as a black box to its operator. The carrier does not know anything about the current size, performance, and stability of its application. The decentralized nature of such a system makes it hard to find a scalable way to gather information about the running system at a central unit. Operators, however, do not want to lose control over their systems. They want to know what their systems look like right now and where problems occur at the moment. The first problems already occur when testing and debugging a distributed application. Finding implementation errors in a highly distributed system is a very complex and time consuming process [9]. A provider also needs to know whether his newly deployed application can truly handle the task it was designed for.

The latest generation of p2p algorithms is based on distributed hash tables (DHTs). The algorithm that currently attracts the most attention is Chord, which uses a ring topology to realize the underlying DHT [11]. In general, DHTs are theoretically understood in depth and proved to be a scalable and robust basis for distributed applications [7]. However, the problem of monitoring such a system from a central position is far from being solved. In this paper we therefore present a novel and scalable approach to create a snapshot of a running Chord based application. Using our algorithm a provider can either monitor the entire system or just survey a specific part of the system he is currently interested in. This way, he is able to react to errors more quickly and can verify if the taken countermeasures are successful. On the basis of the gathered information it is, e.g., possible to take appropriate action to relief a hotspot or to pinpoint the cause of a loss of the overlay ring structure. The overhead involved in creating the snapshot is evenly distributed to the participating peers so that each peer only has to contribute a

negligible amount of bandwidth. Most important, the snapshot algorithm is very easy to use for a provider. It only takes one parameter to adjust the trade off between the duration of the snapshot and the bandwidth needed at the central unit, which collects the measurements.

The remainder of this paper is structured as follows. Section 2 gives a brief overview of Chord with a focus on aspects relevant to this paper. The snapshot algorithm as well as some areas of application are described in Section 3. The functionality of the algorithm is verified by simulation in Section 4. Section 5 finally concludes this paper.

2 Chord Basics

This section gives a brief overview of Chord with a focus on aspects relevant to this paper. A more detailed description can be found in [11].

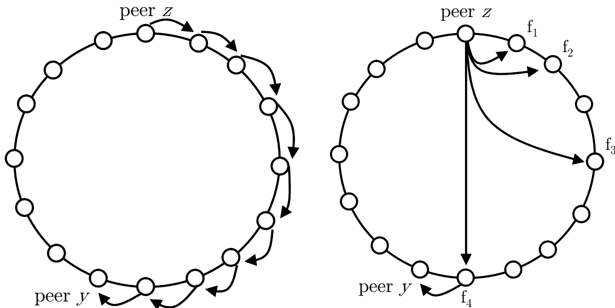

Figure 1. A simple search.

Figure 2. Search using the fingers.

The main purpose of p2p networks is to store data in a decentralized overlay network. Participating peers will then be able to retrieve this data using some sort of search algorithm. The Chord algorithm solves this problem by arranging the participating peers on a ring topology. The position id_z of a peer z on this overlay ring is determined by an m-bit identifier using a hash function such as SHA-1 or MD5. In a Chord ring each peer knows at least the *id* of its immediate successor in a clockwise direction on the ring. This way, a peer looking up another peer or a resource is able to pass the query around the circle using its successor pointers. Figure 1 illustrates a simple search of peer z for another peer y using only the immediate successor. The search has to be forwarded half-way around the ring. Obviously, the average search would require $\frac{n}{2}$ overlay hops, where n is the current size of the Chord ring. To speed up searches a peer z in a Chord ring also maintains pointers to other peers, which are used as shortcuts through the ring. Those pointers are called fingers, whereby the *i-th* finger in a peer's finger table contains the identity of the first peer that succeeds z's own *id* by at least 2^{i-1} on the Chord ring. That is, peer z with hash value id_z has its fingers pointing to the first peers that succeed $id_z + 2^{i-1}$ for $i = 1$ to m, where 2^m is the size of the identifier space.

Figure 2 shows the fingers f_1 to f_4 for the same peer z of the last figure. Using this finger pointers, a search does only take two overlay hops. For the first hop peer z uses its finger f_4. Peer y can then directly be reached using the successor of f_4 as indicated by the little arrow. This way, a search only requires $\frac{1}{2}\log_2(n)$ overlay hops on average. A detailed mathematical analysis of the search delay in Chord rings can be found in [3]. The snapshot algorithm presented in Section 3 makes use of the finger tables of the peers.

3 The Snapshot Algorithm

In this section we introduce a scalable and distributed algorithm to create a snapshot of a running Chord system. The algorithm is based on a very simple two step approach. In step one, the overlay is recursively divided into subparts of a predefined size. In step two, the desired measurement is done for each of these subparts and sent back to a central collecting point (CP). In the following, we describe both steps in detail.

3.1 Step 1: Divide Overlay into Subparts

The algorithm to divide a specific region of the overlay into subparts is called *snapshot*(R_s, R_e, S_{min}, CP). This function is called at an arbitrary peer p with id_p. The peer then tries to divide the region from $R_s = id_p$ to R_e into contiguous subparts using its fingers. The exact procedure is illustrated in Figure 3. In this example peer p has four fingers f_1 to f_4. It sends a request to the finger closest to R_e within $[R_s; R_e]$. Finger f_4 is neglected since it does not fall into the region between R_s and R_e. This makes finger f_3 the closest finger to R_e in our example. If this finger does not respond to the request as illustrated by the bolt, it is removed from the peer's finger list and the peer tries to contact the next closest finger f_2. If this finger acknowledges the request, peer p recursively tries to divide the region from $R_s = id_p$ to $\widehat{R}_e = id_{f_2} - 1$ into contiguous subparts. Finger f_2 partitions the region from $\widehat{R}_s = id_{f_2}$ to R_e accordingly.

As soon as a peer does not know any more fingers in the region between the current R_s and the current R_e, the recursion is stopped. The peer changes into step two of the algorithm and starts a measurement of this specific region. In this context, the parameter S_{min} can be used to determine the minimum size of the regions, which will be measured in step two. Taking S_{min} into

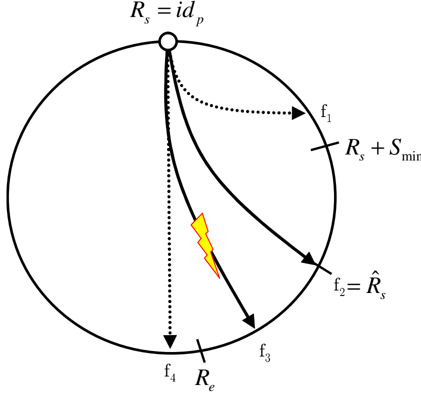

Figure 3. Visualization of the algorithm

account, a peer will already start the measurement if it does not know any more fingers in the region from the current $R_s + S_{min}$ to the current R_e. In this case finger f_1 would be disregarded, as illustrated by the dotted line in Figure 3, since it points into the minimum measurement region. The parameter S_{min} is designed to adjust the trade off between the duration of the snapshot and the bandwidth needed at the collecting point. The larger the regions in step two, the longer the measurement will take. The smaller the regions, the more results are sent back to the CP.

A detailed technical description of the procedure is given in Algorithm 1. Note, that a snapshot of the entire system can be created calling snapshot(id_p, $id_p - 1$, S_{min}, CP) at peer p with $id = id_p$. Peer p will

---

Algorithm 1
The snapshot algorithm (first call $R_s = id_p$)

---

snapshot(R_s, R_e, S_{min}, CP)
send acknowledgment to the sender of the request
$id_{fm} = max(\{id_f | id_f \in \text{fingerlist} \wedge id_f < R_e\})$
while $id_{fm} > R_s + S_{min}$ **do**
  send snapshot(id_{fm}, R_e, S_{min}, CP) request to peer id_{fm}
  if acknowledgment from id_{fm} **then**
    call snapshot(id_p, $id_{fm} - 1$, S_{min}, CP) at local peer
    return {exit the function}
  else
    remove id_{fm} from fingerlist
    $id_{fm} = max(\{id_f | id_f \in \text{fingerlist} \wedge id_f < R_e\})$
  end if
end while
$\widehat{S} = \frac{R_e - R_s}{\lceil \frac{R_e - R_s}{S_{min}} \rceil}$ {explanation see step two}
$result = 0$
call countingtoken(id_p, R_e, S_{min}, CP, $result$) at local peer

---

contact the closest finger to R_e until it does not know any more fingers in between $R_s + S_{min}$ and R_e. If so, it changes into step two and starts a measurement of this region calling countingtoken(id_p, R_e, S_{min}, CP, $result$) at the local peer. A detailed description of this function is given in the next subsection.

3.2 Step 2: Measure a Specific Subpart

The basic idea behind the measurement of a specific subpart from R_s to R_e is very simple. The first peer creates a token, adds its local statistics and passes the token to its immediate successor. The successor adds its statistics and recursively passes the token to its immediate successor and so on. The first peer with an $id > R_e$ sends the token back to the collecting point, whose IP is given in the parameter CP.

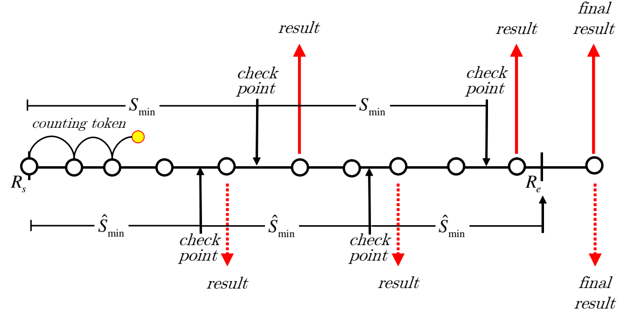

Figure 4. Results sent after each checkpoint

Ideally, each of the regions measured in step two would be of size S_{min} as specified by the user. The problem, however, is that the region from R_s to R_e is slightly larger than S_{min} according to step one of the algorithm. In fact, if the responsible peer did not know enough fingers, the region might even be significantly larger than S_{min}. The solution to this problem is to introduce checkpoints with a distance of S_{min} in the corresponding region. Results are sent to the CP every time the token passes a checkpoint instead of sending only one answer at the end of the region. This is illustrated in the upper part of Figure 4. The counting token is started at R_s. The first peer behind each checkpoint sends a $result$ back to the CP as illustrated by the red arrows pointing upwards. The final $result$ is still sent by the first peer with $id > R_e$.

A drawback of this solution is that the checkpoints might not be equally distributed in the region. In particular, the last two checkpoints might be very close to each other as shown in the figure. We therefore recalculate the positions of the checkpoints according to the following equation:

$$\widehat{S}_{min} = \frac{R_e - R_s}{\lceil \frac{R_e - R_s}{S_{min}} \rceil}$$

The new checkpoints can be seen in the lower part of Figure 4. Again, the first peer behind each checkpoint sends a *result* back to the CP as illustrated by the dotted arrows pointing downwards. As before the last *result* is sent by the first peer with $id > R_e$. Note, that the number of checkpoints remains the same, while their positions are moved in such a way, that the results are now sent at equal distance.

As can be seen at the end of Algorithm 1, the recalculation of S_{min} is already done in the first step, just before the counting token is started. A detailed description of the counting token mechanism is given in Algorithm 2. If a peer p receives a counting token it makes sure that its identifier is still within the measured region, i.e. $R_s \leq id_p \leq R_e$. If not, it sends a *result* back to the CP and stops the token. Otherwise it adds its local measurement to the token and tries to pass the token to its immediate successor. Additionally, if it is the first peer behind one of the checkpoints, it sends an intermediate result back to the CP and resets the token.

As mentioned above the parameter S_{min} roughly determines the minimum size of the regions measured in step two. If S_{id} is the total size of the identifier space, there will be:

$$N_c \geq \frac{S_{id}}{S_{min}}$$

counting tokens arriving at the CP. A more detailed analysis of the snapshot algorithm is given in [1].

3.3 Collect Statistics

So far we concentrated on the technical aspects of a snapshot and did not give any details about what to monitor at the individual peers. Generally speaking, there are two different kinds of statistics, which can be collected using the counting tokens. Either a simple mean value or a more detailed histogram. In the first case the counting token memorizes two variables, V_a for the accumulated value and V_n for the number of values. Each peer receiving the counting token adds its measured value to V_a and increases V_n by one. The sample mean can then be calculated at the CP as $\frac{\sum V_a}{\sum V_n}$. In the second case of the histogram, the counting token maintains a specific number of bins and their corresponding limits. Each peer simply increases the bin matching its measured value by one. If the measured value is outside the limits of the bins it simply increases the first or the last bin respectively.

There are numerous things that can be measured using the above mentioned methods. Table 1 summarizes some exemplary statistics and the kind of information, which can be gained by them. The most obvious application is to count the number of hops for each counting token. On the one hand, this is a direct measure for the size of the overlay network. On the other hand, it also shows the distribution of the identifiers in the identifier space. If the hash function does not work as expected, this distribution will be skewed and the number of hops per token will vary significantly. To gain information about the performance of the Chord algorithm, the mean search delay or a histogram for the search time distribution can be calculated and compared to expected values. Furthermore, Chord's stability can only be guaranteed as long as the successor and predecessor pointers of the individual peers match each other correspondingly. Since each peer should receive the counting token from its direct predecessor, this invariant can be checked counting the percentage of hops, where the sender of the counting token did not match the predecessor of the receiving peer. Additionally, the number of timeouts per token can be used to measure the current churn rate in the overlay network. The more churn there is, the more timeouts are going to occur due to outdated successor pointers. Similarly, the number of resources stored at each peer is a sign of the fairness of the Chord algorithm. The number of searches answered at each peer can likewise be used to get an idea of the search behavior of the end users. Finally, a peer can keep track of the number of missing resources to verify the integrity of the stored data.

Algorithm 2
The countingtoken algorithm (first call $R_s = id_p$)

```
countingtoken(R_s, R_e, S_min, CP, result)
send acknowledgment to the sender of the request
if R_s ≤ id_p ≤ R_e then
  if id_p > R_s + S_min then
    send result to CP
    result = 0
    R_s = R_s + S_min
  end if
  add local measurement to result
  id_s = id of direct successor
  while 1 do
    send countingtoken(R_s, R_e, S_min, CP, result)
    request to direct successor id_s
    if acknowledgment then
      break
    else
      remove id_s from successor list
      id_s = id of new direct successor
    end if
  end while
else
  send result to CP
end if
```

Table 1. Possible statistics gathered during snapshot

Statistic	Information gained
Number of hops per token	Size of the network, Distribution of the identifiers
Mean search delay	Performance of the algorithm
Sender $\stackrel{?}{==}$ predecessor	Overlay stability
Number of timeouts per token	Churn rate
Number of resources per peer	Fairness of the algorithm
Number of searches answered	User behavior
Bandwidth used per time unit	Maintenance overhead
Missing resources	Data integrity

This can, e.g., be done counting the number of search request, which could not be answered by the peer.

All of the above statistics can be collected periodically to survey the time dependent status of the overlay. Note, that it is also possible to monitor only a specific part of the overlay network. This can, e.g., be helpful if there are problems in a certain region of the overlay network and the operator needs to verify that his countermeasures are successful.

4 Results

A detailed mathematical analysis of the snapshot algorithm in times of no churn can be found in our technical report [1]. Due to the lack of space, we concentrate on simulation results covering more realistic scenarios including churn. In our experience the performance of the Chord algorithm depends on the way the algorithm is implemented. This is especially true for the correctness of the overlay neighbors, i.e. the successors and the fingers of a peer. This section is therefore rather intended to make qualitative than quantitative statements. The results were obtained using our ANSI-C simulator, which incorporates a detailed yet slightly modified Chord implementation [6, 5]. If not stated otherwise an overlay hop is modeled using an exponentially distributed random variable with a mean of 80ms. The results considering churn are generated using peers, which stay online and offline for an exponentially distributed period of time with a mean as indicated in the description of the figures.

The snapshot algorithm takes only one single input argument $N_r = \left\lceil \frac{S_{id}}{S_{min}} \right\rceil$. This parameter influences the duration of the snapshot as well as the number of results arriving at the central collecting point. Figure 5 shows the distribution of the arrival times of the results in an overlay of 40000 peers using $N_r = 1000$ and $N_r = 100$ areas in times of no churn. Obviously, the more areas the overlay is divided into, the faster the snapshot is completed. While the snapshot using 1000 areas was finished after about ten seconds, the snapshot with 100 areas took about one minute. In exchange the latter snapshot produces significantly smaller bandwidth spikes at the CP. The two elevations of the second histogram correspond to the intermediate results (first elevation) and the final results at the end of the measured subpart (second elevation). Note that the final results arrive about twice as late as the intermediate results.

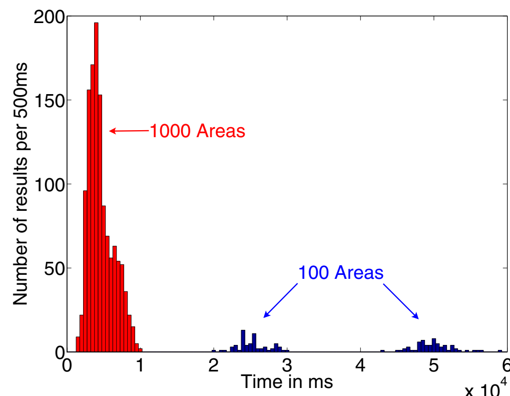

Figure 5. Arrival times of the results

The first step of the algorithm uses the fingers to divide the ring into subparts. Since the distance between a peer and its fingers is always slightly larger than a power of two it is usually not possible to divide the ring exactly into the desired number of areas. In fact it is very likely, that a peer stops the recursion and starts its measurement once it contacted its xth finger, where $2^{x-1} < S_{min} = \frac{S_{id}}{N_r} \leq 2^x$. That is, the recursion stops at finger x with id_{f_x}, whereas the distance between the peer and this specific finger might almost be twice as large as the desired S_{min}. It is therefore advisable to choose N_r as a power of two itself in order to ensure that id_{f_x} is only slightly larger than $id_p + S_{min}$. Figure 6 shows the different arrival times of the results for $N_r = 512$ and $N_r = 500$ in an overlay of 20000 peers without churn. The skewed shape of the histogram in the foreground results from the fact that 500 is slightly smaller than a power of two, which in turn makes S_{min}

slightly larger than a power of two. In this case it is likely that the peer has a finger just before the end of the minimum measurement region $id_p + S_{min}$. Thus, finger x sits at a distance of about twice S_{min} from the peer. The resulting counting token will therefore travel a distance of about twice S_{min} as well.

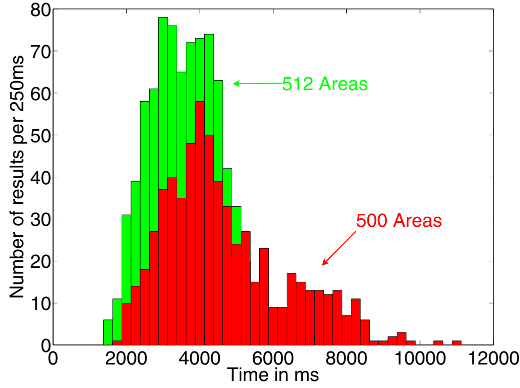

Figure 6. Arrival times of the results for 20000 peers without churn.

A more detailed analysis of the influence of N_r can be found in Figure 7, which shows the number of results received at the CP in dependence of N_r. As shown in [1], N_c, the number of counting tokens sent to the CP, is limited by $2 \cdot N_r > N_c \geq N_r$. The straight lines in the figure show the corresponding limits. The solid and dotted curves represent the results obtained for 20000 and 10000 peers respectively. It can be seen, that the number of results sent to the CP is within the calculated limits and independent of the overlay size. The curves roughly resemble the shape of a staircase, whereas the steps are located at powers of two. There are two main reasons for this behavior. First of all, the average counting token sends two results back to the CP, one intermediate result and the final result at the end of the measurement region. Hence, the smaller the region covered by the average counting token, the more results arrive at the CP. As explained above the closer N_r gets to a power of two, the smaller the region covered by the average counting token. This accounts for the first part of the rise of the number of results received at the CP. The reason, that the curve still rises for a short time once N_r becomes slightly larger than a power of two has a different cause. Due to the fact that the actual finger positions slightly differ from the theoretical finger positions, it is possible that $id_p + S_{min} > R_e$ in the last step of the recursion. In this case the corresponding counting token does not send an intermediate result, since the first checkpoint is behind the end of the measured region. As long as

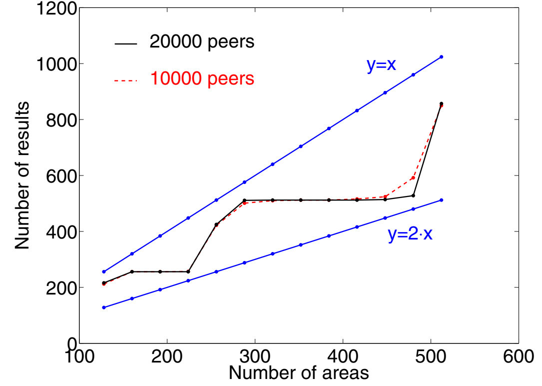

Figure 7. Number of results received at the CP in dependence of N_r

S_{min} is still large enough for this to happen, the curve will slightly rise.

The distribution of the arrival times of the results is also influenced by the current size of the network. The larger the network is, the more peers are within one region. However, the more peers are within one region, the more hops each counting token has to make, before it can send its results back to the CP. Figure 8 shows the token arrival time distribution for three different overlay sizes of 10000, 20000, and 40000 peers respectively. There was no churn in this scenario and N_r was set to 512 areas. As expected, the larger the overlay network, the longer the snapshot is going to take. However, the curves are not only shifted to the right, but also differ in shape. This can again be explained by the increasing number of hops per counting token. As mentioned above, the average counting token sends

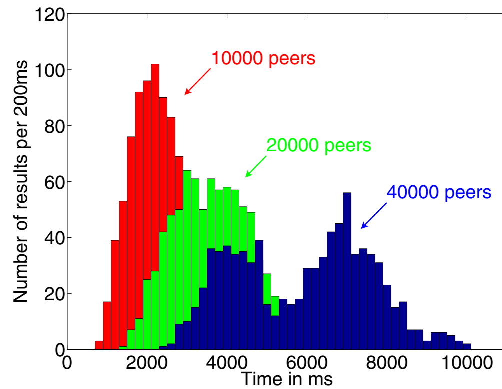

Figure 8. Arrival times of the results at the CP

two results back to the CP, whereas the checkpoints are equally spaced. Thus, the final result takes twice as many hops as the intermediate result. In a network of

10000 peers there are approximately 20 peers in each of the 512 regions. The intermediate results are therefore sent after about 10 hops, the final results after about 20 hops respectively. The two corresponding elevations in the histogram overlap in such a way, that they build a single elevation. In a network of 40000 peers, however, there are approximately 78 peers in each of the 512 regions. The intermediate results are therefore sent after about 39 hops, the final results after about 78 hops respectively. The difference between these two numbers is large enough to account for the two elevations of the histogram in the foreground of Figure 8. Note, that all curves are shifted to the right as compared to the mere hop count since it takes some time for the signaling step until the counting tokens can be started. In practice the current size of the overlay can be estimated to be able to choose an appropriate value for N_r [2].

The arrival time of the results at the CP is also affected by the online/offline behavior of the individual peers. To study the influence of churn we consider 80000 peers with an exponentially distributed online and offline time, each with a mean of 60 minutes. This way, there are 40000 peers online on average, which makes it possible to compare the results to those obtained using 40000 peers without churn. Figure 9 shows the corresponding histograms. As a result of churn in

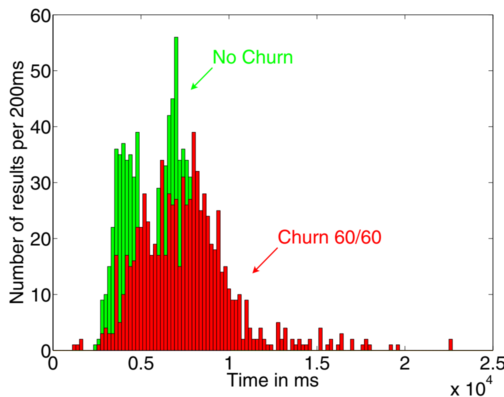

Figure 9. Influence of churn on the pattern traffic at the CP

the system, the two elevations of the original histogram become noticeably blurred and the snapshot is slightly delayed. This is due to the inconsistencies in the successor and finger lists of the peer, as well as the timeouts, which occur during the forwarding of the counting tokens. In return the spike in the diagram and thus the required bandwidth at the CP becomes smaller.

It is easy to show, that the probability to lose a token is almost negligible [1]. Therefore, a more meaningful method to measure the influence of churn is to regard the number of timeouts, which occur during a snapshot. Furthermore, the influence of churn on the stability of the overlay network can be studied looking at the frequency at which the predecessor pointer of a peer's successor does not match the peer itself. Figure 10 plots the relative frequency of timeouts and pointer failures against the mean online/offline time of a peer. The smaller the online/offline time of a peer, the more churn there is in the system. The results

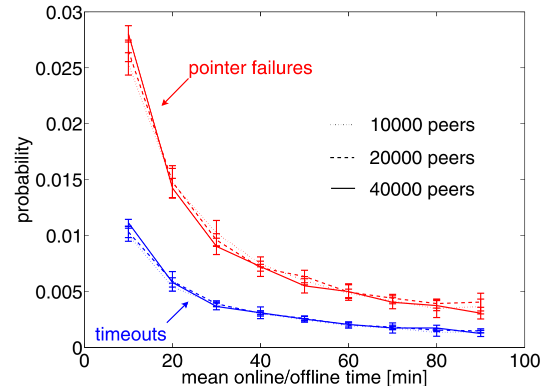

Figure 10. Relative frequency of timeouts and pointer failures.

represent the mean of several simulation runs, whereas the error bars show the 95 percent confidence intervals. The relatively small percentage of both timeouts and failures is to some extent implementation specific. More interesting, however, is the exponential rise of the curves under higher churn rates. The shape of both curves is independent of the size of the overlay and only affected by the current churn rate. The curve can therefore be used to map the frequency of timeouts or failures measured in a running system to a specific churn rate.

Until now, we only regarded the traffic pattern at the central collecting point. From an operator's point of view, however, it is more important to know, whether the snapshot itself is meaningful. To validate the accuracy of the snapshot algorithm, we again simulated an overlay network with 80000 peers, each with a mean online/offline time of 60 minutes. Due to the properties of the hash function and the churn behavior in the system the number of documents on a single peer can be regarded as a random variable. The measurement we are interested in is the corresponding probability density function (pdf) in order to see whether the distribution of the documents among the peers is fair or not. The pdf was measured using our snapshot algorithm as explained in Section 3.3. The result of the snapshot is compared to the actual pdf obtained us-

ing the global view of our discrete event simulator (c.f. Figure 11). The two curves are almost indistinguishable from each other. The same is true for all the other statistics shown in Table 1, like the current size of the system or the average bandwidth used per time unit. That is, the snapshot provides the operator with a very accurate picture of the current state of its system. This nicely demonstrates, that the results obtained by the snapshot can be used to better understand the performance of the running p2p system. The multiple possibilities to interpret the collected data are well beyond the scope of this paper.

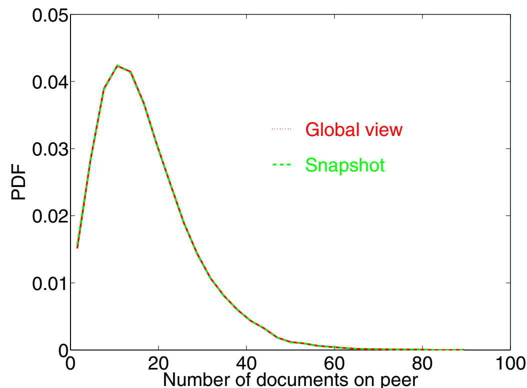

Figure 11. Results of a snapshot compared to the global view.

5 Conclusion

One of the main reasons why telecommunication carriers are still hesitant to build p2p applications is the lack of control a provider has over the running system. In this paper we introduced an entirely distributed and scalable algorithm to monitor a Chord based p2p network at runtime. The load generated during the snapshot is evenly distributed among the peers of the overlay and the algorithm itself is easy to configure. It only takes one input parameter, which influences the trade off between duration of the snapshot and bandwidth required at the central server, which collects the results. In general it takes less than one minute to create a snapshot of a Chord ring consisting of 40000 peers. We performed a mathematical analysis of the basic mechanisms and provided a simulative study considering realistic user behavior.

The algorithm is resistant to dynamic in the overlay network (churn) and provides the operator with a very accurate picture of the current state of its system. It offers the possibility to monitor the entire overlay network or to concentrate on a specific part of the system. The latter is especially useful if a problem occurred in a specific part of the system and the operator wants to assure that his countermeasures are successful.

Acknowledgements

The authors would like to thank Holger Schnabel for the help and the insightful discussions during the course of this work.

References

[1] A. Binzenhöfer, G. Kunzmann, and R. Henjes. A scalable algorithm to monitor chord-based p2p systems at runtime. Technical Report 373, University of Würzburg, November 2005.

[2] A. Binzenhöfer, D. Staehle, and R. Henjes. On the Fly Estimation of the Peer Population in a Chord-based P2P System. In *19th International Teletraffic Congress (ITC19)*, Beijing, China, September 2005.

[3] A. Binzenhöfer and P. Tran-Gia. Delay Analysis of a Chord-based Peer-to-Peer File-Sharing System. In *ATNAC 2004*, Sydney, Australia, December 2004.

[4] U. B. C. S. Division. The oceanstore project. URL: http://oceanstore.cs.berkeley.edu/.

[5] G. Kunzmann, A. Binzenhöfer, and R. Henjes. Analysis of the Stability of the Chord protocol under high Churn Rates. In *6th Malaysia International Conference on Communications (MICC) in conjunction with International Conference on Networks (ICON)*, Kuala Lumpur, Malaysia, November 2005.

[6] G. Kunzmann, R. Nagel, and J. Eberspächer. Increasing the reliability of structured p2p networks. In *5th International Workshop on Design of Reliable Communication Networks*, Island of Ischia, Italy, October 2005.

[7] J. Li, J. Stribling, T. M. Gil, R. Morris, and M. F. Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS04)*, San Diego, CA, February 2004.

[8] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. P2P Computing. Technical Report HPL-2002-57, Hewlett Packard Lab, 2002.

[9] D. L. Oppenheimer, V. Vatkovskiy, H. Weatherspoon, J. Lee, D. A. Patterson, and J. Kubiatowicz. Monitoring, analyzing, and controlling internet-scale systems with acme. *CoRR*, cs.DC/0408035, 2004.

[10] V. N. Padmanabhan, S. Ramabhadran, and J. Padhye. Netprofiler: Profiling wide-area networks using peer cooperation. In *Fourth International Workshop on Peer-to-Peer Systems (IPTPS)*, Ithaca, NY, USA, February 2005.

[11] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001*, San Diego, CA, August 2001.

[12] S. Technologies. Skype. URL: http://www.skype.com.