University of Würzburg
Institute of Computer Science
Research Report Series

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

Andreas Binzenhöfer[1], Gerald Kunzmann[2], and Robert Henjes[3]

Report No. 373 November 2005

[1] Department of Distributed Systems
Institute of Computer Science
University of Würzburg, Am Hubland, 97074 Würzburg, Germany
binzenhoefer@informatik.uni-wuerzburg.de

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

[3] Department of Distributed Systems
Institute of Computer Science
University of Würzburg, Am Hubland, 97074 Würzburg, Germany
henjes@informatik.uni-wuerzburg.de

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

**Andreas Binzenhöfer**
Department of Distributed Systems
Institute of Computer Science
University of Würzburg, Am Hubland, 97074 Würzburg, Germany
binzenhoefer@informatik.uni-wuerzburg.de

**Gerald Kunzmann**
Technische Universitaet Muenchen
Institute of Communication Networks
80290 Munich, Germany
gerald.kunzmann@tum.de

**Robert Henjes**
Department of Distributed Systems
Institute of Computer Science
University of Würzburg, Am Hubland, 97074 Würzburg, Germany
henjes@informatik.uni-wuerzburg.de

**Abstract**

Peer-to-peer (p2p) systems are a highly decentralized, fault tolerant, and cost effective alternative to the classical 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 [1]. Compared to the classical 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 [2], a distributed VoIP solution, oceanstore [3], a global persistent data store, and even p2p based network management [4, 5].

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 [6]. 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 [7]. In general, DHTs are theoretically understood in depth and proved to be a scalable and robust basis for distributed applications [8]. 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 collecting unit.

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. An analysis of the algorithm can be found in Section 4, while Section 5 verifies its functionality by simulation. Section 6 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 [7].

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 problems 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 [9] or MD5 [10]. 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 peers finger table contains the identity of the first peer that succeeds $z$'s own *id* by at least

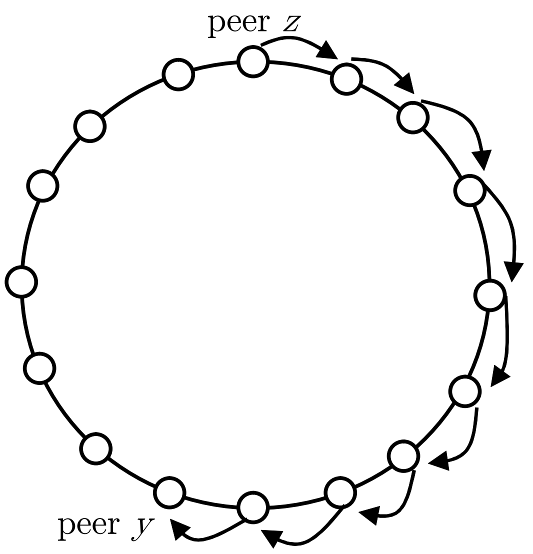

Figure 1: A simple search.

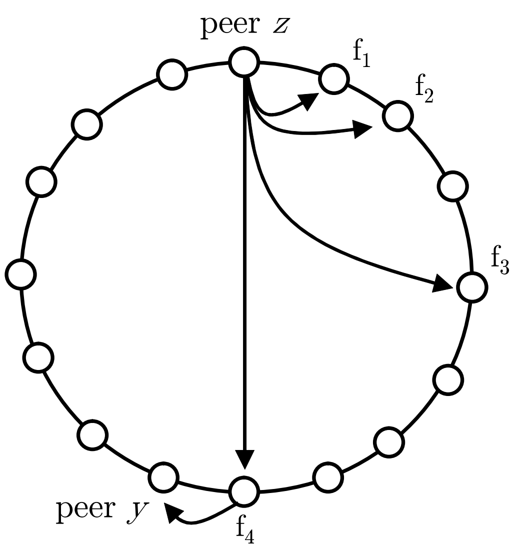

Figure 2: Search using the fingers.

$2^{i-1}$ 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 [11]. 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 One: Divide the Overlay into Subparts

The algorithm to divide 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 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 yellow 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.

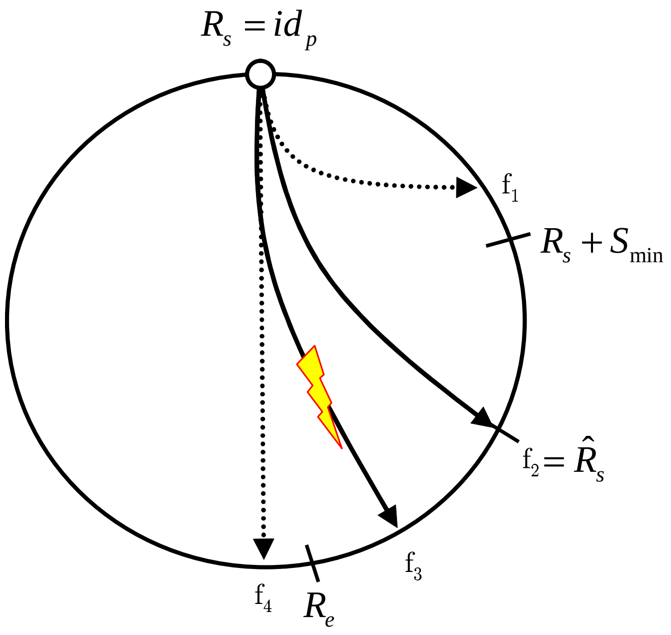

Figure 3: Visualization of the snapshot algorithm

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 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 contact the closest finger to $R_e$ until it does not

**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
$id_{fmax} = max(\{id_{finger} | id_{finger} \in \text{fingerlist} \wedge id_{finger} < R_e\})$
while $id_{fmax} > R_s + S_{min}$ do
  send snapshot($id_{fmax}$, $R_e$, $S_{min}$, $CP$) request to peer $id_{fmax}$
  if acknowledgment from $id_{fmax}$ then
    call snapshot($id_p$, $id_{fmax} - 1$, $S_{min}$, $CP$) at local peer
    return {exit the function}
  else
    remove $id_{fmax}$ from fingerlist
    $id_{fmax} = max(\{id_{finger} | id_{finger} \in \text{fingerlist} \wedge id_{finger} < R_e\})$
  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
```

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 Two: 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.

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

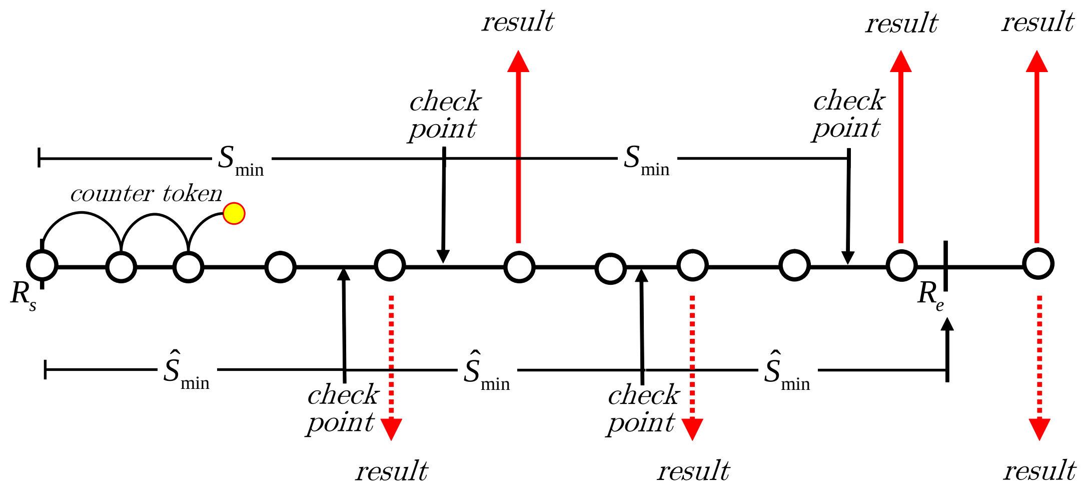

Figure 4: Results are sent back to the $CP$ after each checkpoint

to the following equation:

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

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 Section 4.

**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** $R_s \leq id_p \leq R_e$ **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**
```

## 3.3 What to monitor?

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

| Statistic | Information gained |
|---|---|
| Number of hops per token | Size of the network, Distribution of the identifiers |
| Mean search delay | Performance of the algorithm |
| Sender $\overset{?}{==}$ 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 |

Table 1: Possible statistics gathered during snapshot

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. 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 Analysis and Optimizations

From an analytical point of view there are two interesting aspects of the algorithm, which should be considered in greater detail. At first, we will regard the overhead produced during a snapshot. Independent of the size of the overlay, the number of messages sent by the individual peers during the snapshot is about two (one intermediate and the final result). The bandwidth required at the central collecting point ($CP$), however, strongly depends on the size of the overlay and $S_{min}$ (cf. Subsection 4.1). Secondly, depending on the application the duration of a snapshot (cf. Subsection 4.2) and the temporal distribution of the token arrival times at the $CP$ (cf. Subsection 4.3) might be of equal importance. Last but not least, we will go into the issue of losing tokens while creating a snapshot of the running system in Subsection 4.4.

## 4.1 Required Bandwidth at the Monitor Station

The snapshot algorithm takes only one single parameter $S_{min}$. It basically determines the number of areas $N_r$ into which the Chord ring is divided during a snapshot:

$$N_r = \left\lceil \frac{S_{id}}{S_{min}} \right\rceil \tag{1}$$

Independent of the current size of the overlay network at least one result per region is sent back to the $CP$. A closer look at step one of the snapshot algorithm yields the following bounds for $N_c$, the number of counter tokens sent to the $CP$:

$$2 \cdot N_r > N_c \geq N_r \tag{2}$$

The equation can be explained as follows: According to the second step of the algorithm, a counter token sends an intermediate result every $\widehat{S}_{min}$ and an additional result at the end of the region. Obviously, this way, at least on result is sent per region. In the worst case, however, the region is slightly larger than the original $S_{min}$ in which case an intermediate checkpoint is created and the number of tokens is thus doubled.

As can be seen in Equation 1, $S_{min}$ can be used to adjust the trade-off between the duration of a snapshot and the number of tokens, which arrive at the $CP$. The larger $S_{min}$, the more hops per counter token and the longer the snapshot will take. The smaller $S_{min}$, the less hops per counter token but the more tokens arrive at the $CP$ in total. Also note, that if $S_{min}$ is slightly larger than a power of two, it is very likely that the corresponding peer has a finger just before the end of the minimum measurement region. Since the distance to the theoretical finger pointers is always a power of two, the next finger sits at a distance of about twice $S_{min}$ from the peer. The resulting counter token will therefore travel a distance of about twice $S_{min}$ as well.

## 4.2 Duration of a Snapshot

Besides $S_{min}$, the duration of a snapshot mainly depends on two things, the current churn rate and the implementation of the Chord algorithm. The more sophisticated the implementation and the less churn there is in the network, the less timeouts will occur during the snapshot. In Section 5 we therefore simulate different scenarios with a varying churn rate and also give details about the implementation of the Chord algorithm in our simulator. To be able to calculate an estimate of the duration of a snapshot, we assume a scenario without any peers joining or leaving the network in this subsection.

In this case, it is quite straightforward to estimate the duration of step one, the signaling step. The last region, in which a counter token will be started is the one directly following the initiating peer. This is due to the fact, that the initiating peer will start its counter token no sooner than it divided the ring into separate regions. Before it starts the counter token, it contacts its fingers until the first finger is closer to itself than $S_{min}$. The initiating peer has at most $\log_2(n)$ fingers and each of the fingers sends an acknowledgment, before the peer can go on with the algorithm. If $T_O$ is the random variable describing one overlay hop, then the duration of step one of the algorithm is at most

$$D_{step1} = 2 \cdot \log_2(n) \cdot E[T_O] \tag{3}$$

The worst case for step two would be that the initiating peer did not know any fingers and directly sends the counter token. This would take $n \cdot E[T_O]$, but is very unlikely to happen. An easy solution to this problem would be to pass the responsibility of dividing the ring to the direct successor in case the counter token region becomes too large. In general, however, it is possible to estimate the average number of peers per region. If there are $n$ peers in the overlay, there are roughly $P_r$ peers per region:

$$P_r = \frac{n}{N_r} \tag{4}$$

As stated in the last subsection, if $S_{min}$ is slightly larger than a power of two, the counter token region becomes almost twice as large. Therefore a good estimate for the duration of the counting step of the algorithm is:

$$D_{step2} = 2 \cdot P_r \cdot E[T_O] \tag{5}$$

This results in the following total duration of a snapshot:

$$D = \left( \log(n) + \frac{n \cdot S_{min}}{S_{id}} \right) \cdot 2 \cdot E[T_O] \tag{6}$$

To estimate the required bandwidth at the central $CP$, however, we also need the distribution of the arrival times of the tokens at the $CP$. This aspect is covered in the next subsection.

### 4.3 Token Arrival Time Distribution

To get a rough estimate for the distribution of the arrival times of the counting tokens at the $CP$, we consider the special case where the size of the overlay $n = 2^g$ is a power of two and $S_{min}$ is such that $N_r = 2^h$ with $h < g$. Furthermore, we assume that the individual peers sit at equal distances on the ring as shown in Figure 5.

It can be shown, that in this case $h = \log_2(N_r)$ is the number of overlay hops it takes until the first counting token is started during a snapshot. Similarly, it takes $2 \cdot h$ hops until the last counting token is started according to our assumptions. The probability $p_i$ that a counting token is started after exactly $i$ hops for $i = h, h+1, ..., 2 \cdot h$ can also be calculated as:

$$p_i = \frac{\binom{h}{i-h}}{\sum_{x=h}^{2 \cdot h} \binom{h}{x-h}} \tag{7}$$

The above considerations are nontrivial, but can nicely be explained using the simple example shown in Figure 5, where $g = 4$, $h = 2$ and therefore $n = 2^4$ and $N_r = 2^2$. The solid arrows in the figure show the messages of the signaling step, the dotted arrows the corresponding acknowledgments. The numbers next to the arrows represent the number of overlay hops, which have passed since the beginning of the snapshot.

In the example, peer A starts a snapshot of the entire ring. It sends a request to B to cover the region between B and A. Peer B sends an acknowledgment back to A and a simultaneous request to C to cover the region from C to A. C has no fingers outside its minimum measurement region and starts the first counting token after $h = 2$ overlay hops. Simultaneously it sends an acknowledgment back to B. Peer B then starts its counting token after 3 overlay hops. In the meantime A signals D to cover the region from D to B. Peer D immediately starts its counting token after a total of 3 overlay hops. Peer A waits for the final acknowledgment and starts its counting token after $4 = 2 \cdot h$ overlay hops. Summarizing the above, there are four counting tokens started after 2, 3, 3, and 4 overlay hops respectively.

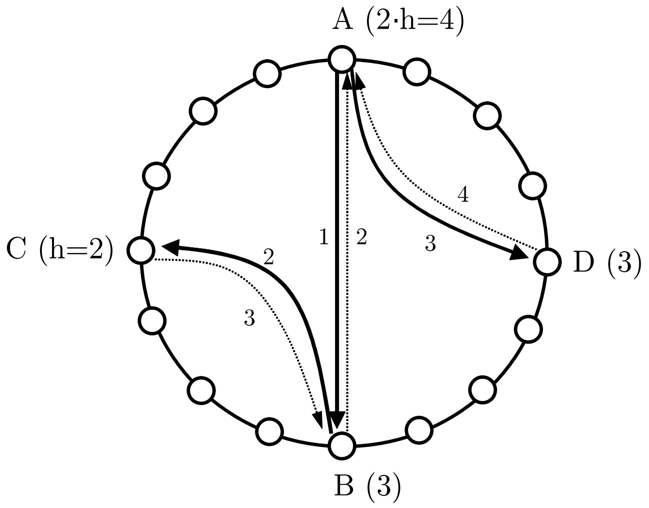

Figure 5: Starting times of the counting tokens for $N_r = 2^2$ and $n = 2^4$

According to our assumptions, each counting token needs exactly $P_r = 4$ hops to travel the corresponding region and one final hop to arrive at the $CP$. A rough estimate for the distribution of the arrival times of the counting tokens at the $CP$ is therefore given by the phase diagram shown in Figure 6. The signaling step takes $i$ overlay hops with a probability $p_i$ for $i = h, h + 1, ..., 2 \cdot h$, which is followed by $P_r$ hops of the counting token and the final hop to report the result back to the $CP$.

To validate our analytical results, we simulated a Chord ring of size $n = 2^{15}$ with $S_{min} = 2^9$ according to the above assumptions. Figure 7 shows the probability density function of the token arrival times at the $CP$. The green line corresponds to our analysis, the blue curve represents a histogram of one single simulation run. Obviously, the curves match very well and the binomial distribution of the duration of step one can

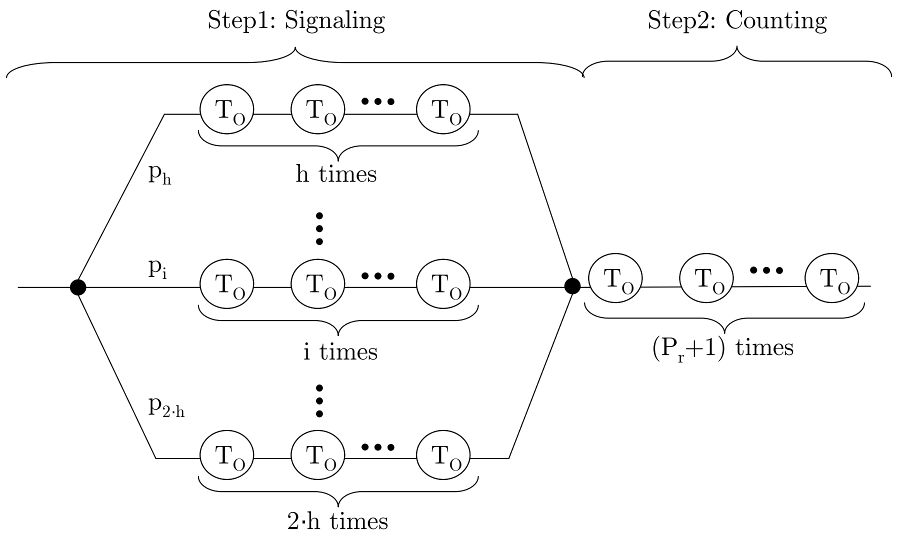

Figure 6: Phase diagram of the token arrival time distribution

be recognized. So far, in our example there is always a peer at exactly distance $S_{min}$ of each peer. In reality, however, the finger would sit at a slightly different position, which again would result in an additional checkpoint at the middle of the region. The red curve in the figure corresponds to a slightly modified phase diagram, which adds an intermediate result in the middle of the measurement region. The first rise of the pdf therefore represents the intermediate results sent back to the $CP$ at the checkpoint. The second rise still represents the regular results at the end of the region.

In the next section we will present some simulations of more realistic scenarios including churn, timeouts and so forth. Fortunately, timeouts, which are caused by churn in the network, tend to spread the results and decrease the spike of counting tokens arriving at the $CP$. This spike could also be minimized using a list of $CP$s and sending each result to a random entry of this list. The same recursion of the snapshot algorithms could also be used backwards to collect the results. The downside of this approach is that it takes longer and is more vulnerable to the loss of a token. The general implications of a lost token are summarized in the next subsection.

## 4.4 Lost Tokens

As in all token based algorithms, there is a certain probability of losing a token or a signaling message. In our case, this is especially true during high churn phases. However, the loss of a token only results in a loss of the measurements of the corresponding region. Fortunately, most statistics do not require all counting tokens in order to produce useful results. In the case of a measurement of the size of the overlay, it is, e.g., possible to extrapolate the received results. In case of a more fault-prone statistic, such a partial loss can also be dealt with very easily. All the operator has to do is to make additional

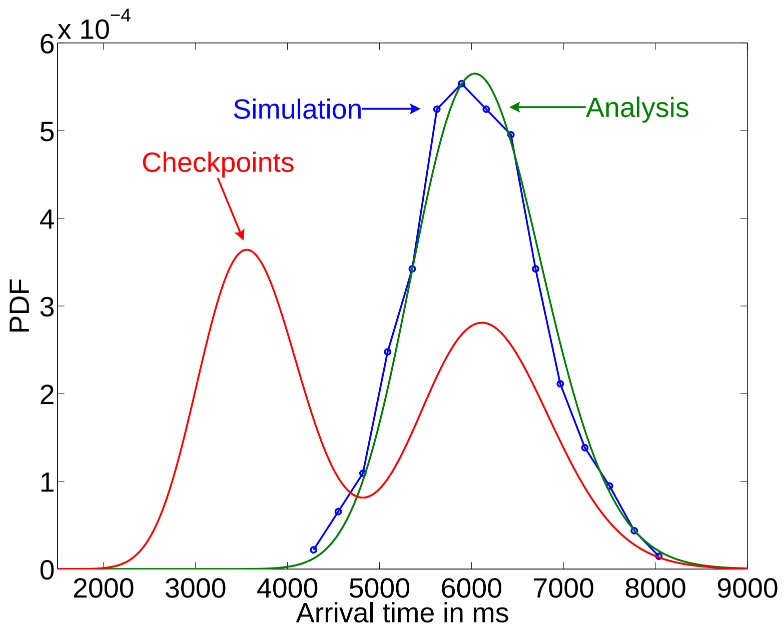

Figure 7: Probability density function of the token arrival time

snapshots of the lost region(s).

Additionally, the probability to lose a token is very small, since there are only two scenarios which result in the loss of a token. First, if a peer goes offline while it still waits for the timeout of a signaling messages. Second, if a peer goes offline while it still waits for the timeout of counting token message. Again both scenarios only involve the loss of the measurements of the corresponding region of the ring. In both cases the probability for the loss of the region is

$$p_{loss} = P(A \leq timeout) \tag{8}$$

where $A$ is the random variable describing the session duration of a peer and *timeout* is the value of a timeout in the overlay network.

## 5 Results

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 [12, 13]. 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 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 8 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. 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 blue curve correspond to the intermediate results (first elevation) and the results at the end of the measured subpart (second elevation). Note that the final results arrive about twice as late as the intermediate results.

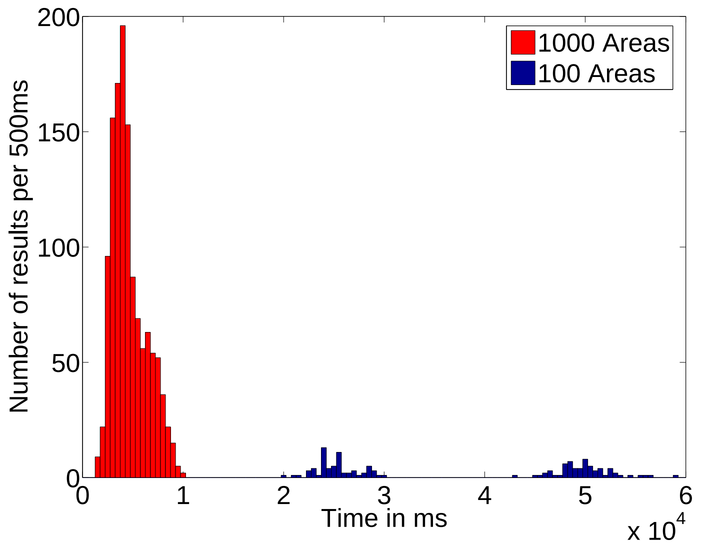

Figure 8: Arrival times of the results for 40000 peers without churn.

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 $x$th 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 9 shows the different arrival times of the results for $N_r = 512$ and $N_r = 500$ in an overlay of 20000 peers. The skewed shape of the red curve 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.

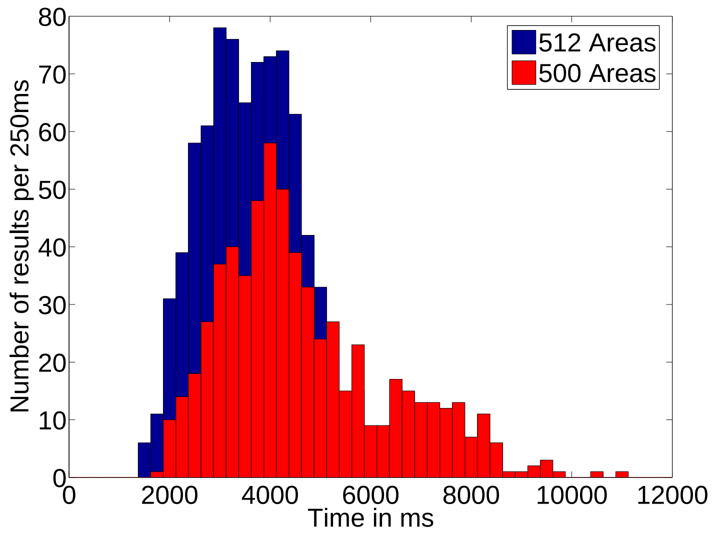

Figure 9: Arrival times of the results for 20000 peers without churn.

A more detailed analysis of the influence of $N_r$ can be found in Figure 10, which shows the number of results received at the $CP$ in dependence of $N_r$. As mentioned in Section 4.1 the number of counting tokens sent to the $CP$ is limited by $2 \cdot N_r > N_c \geq N_r$. The blue lines in the figure show the corresponding limits. The black and red 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 countingtoken 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 countingtoken, 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 countingtoken. 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 countingtoken does not send an intermediate result, since the first checkpoint is behind the end of the measured region. As long as $S_{min}$ 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.

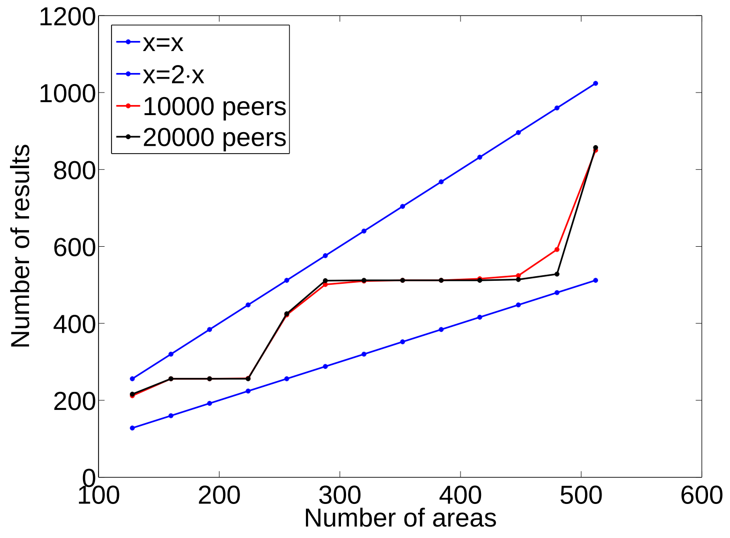

Figure 10: Number of results received at the $CP$ in dependence of $N_r$

However, the more peers are within one region, the more hops each countingtoken has to make, before it can send its results back to the CP. Figure 11 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 countingtoken. As mentioned above, the average countingtoken sends 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 in the blue histogram in Figure 11. 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 countingtokens can be started. In practice the current size of the overlay can be estimated [14] to be able to choose an appropriate value for $N_r$.

So far, we did not consider the influence of churn in our simulations. However, 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.

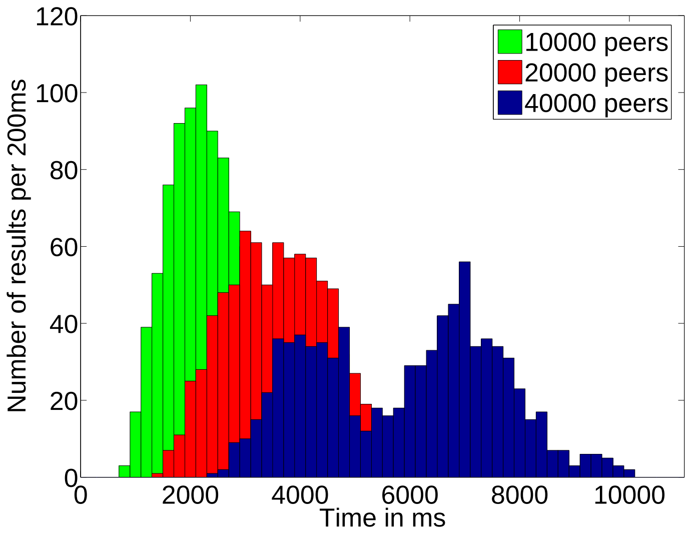

Figure 11: Arrival times of the results at the $CP$

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 12 shows the corresponding histograms. As a result of churn in the system, the two elevations of the

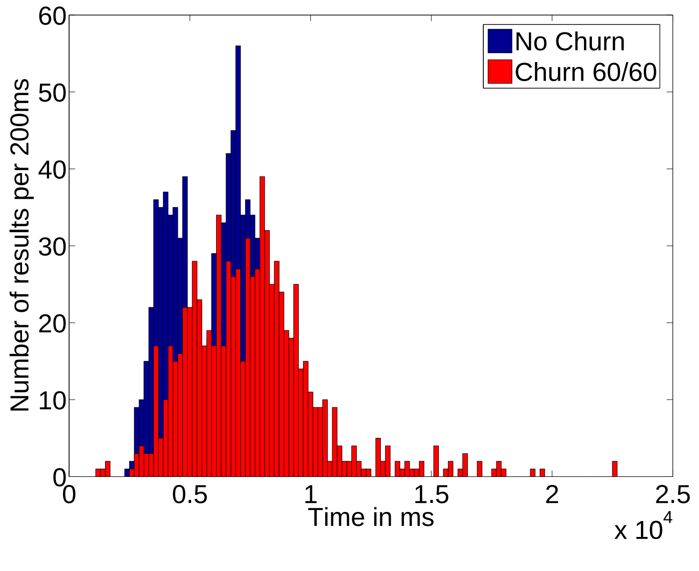

Figure 12: Influence of churn on the pattern traffic at the $CP$

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

As already shown in Section 4.4, the probability to lose a token is almost negligible. 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 13 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 represent the mean of several simulation runs,

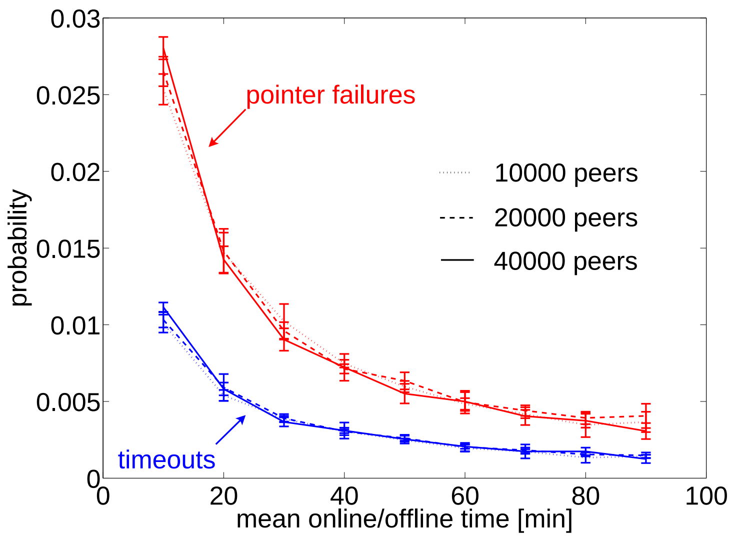

Figure 13: Relative frequency of timeouts and pointer failures.

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 the curve 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 using the global view of our discrete event simulator (c.f. Figure 14). The two curves are almost

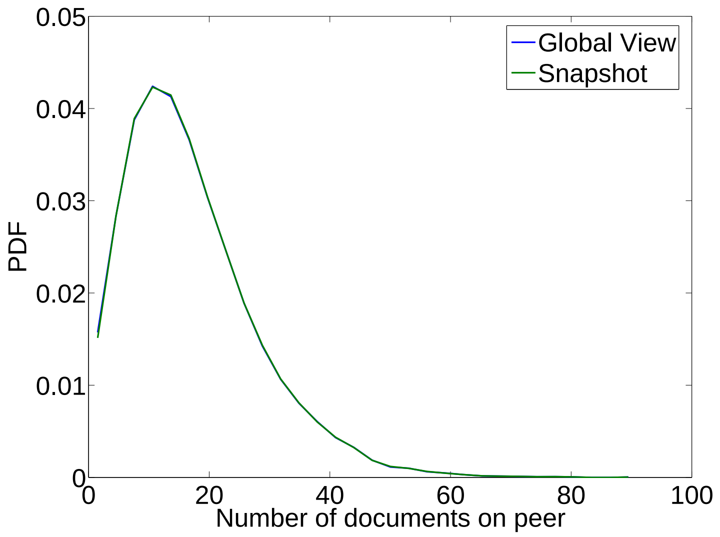

Figure 14: Results of a snapshot compared to the global view.

indistinguishable from each other. 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.

## 6 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] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu., "P2P Computing," Tech. Rep. HPL-2002-57, Hewlett Packard Lab, 2002.

[2] S. Technologies, "Skype." URL: http://www.skype.com.

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

[4] A. Binzenhöfer, K. Tutschku, and B. auf dem Graben, "DNA – A P2P-based Framework for Distributed Network Management," in *Peer-to-Peer-Systeme und -Anwendungen*, (GI/ITG Work-In-Progress Workshop in Cooperation with KiVS 2005, Kaiserslautern), March 2005.

[5] 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.

[6] 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*, vol. cs.DC/0408035, 2004.

[7] 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.

[8] 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.

[9] FIPS PUB 180-1, "Secure hash standard." Federal Information Processing Standards Publication 180-1, April 1995.

[10] R. Rivest, "RFC 1321 - The MD5 Message-Digest Algorithm," April 1992.

[11] 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.

[12] 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.

[13] 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.

[14] 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.