Study of Network Size Measurement Algorithm for P2P System

Bin Zeng and Lu Yao

Center of Information Management, Naval University of Engineering, Wuhan, China
zbtrueice@163.com

Abstract

Peer-to-peer (P2P) networks have recently gained much attention owing to their self-organization characters under dynamic and decentralized environment. It is important for a P2P routing to keep track of the underlying network topology or at least an estimation of the number of nodes in such a network. However, this is quite difficult due to the frequent arrival and leave of nodes in peer-to-peer networks. In order to solve this problem, two network size measurement algorithms are proposed: binary tree and range searching to suit the network fluctuate. The advantages and disadvantages of the two size measurement algorithms are discussed. The results show that both algorithms can efficiently measure the size of P2P networks when topology changes. Simulation results show that both of the proposed approaches can give a good estimate of network size. In addition, comparison of performance indexes such as resilience to Failure, measurement error and workload shows that the binary tree algorithm can give a better estimate at the maintenance cost.

Keywords: peer-to-peer system; network size measurement; distributed hash tables; overlay network

1. Introduction

As network infrastructures and hardware capabilities grow to maturity, P2P (Peer-to-Peer) systems have drastically changed the way we share resources and gather information. Huge benefits have been gained by exchanging various information, instant messaging, and aggregating heterogeneous resources for mass computation and large scale storage services through these P2P systems [1]. In the meantime, the popularity of P2P systems also stimulates their own evolution from primitive systems to nowadays more diversified and advanced systems.

The P2P systems have a logical deterministic overlay network among peers. Object placement is also determined by globally-agreed schemes highly related to overlay networks. Among these systems Distributed Hash Tables (DHTs) have emerged as the most popular scheme in this family [2, 3]. CAN implements a distributed hash table, which provides basic operations such as insertion, lookup, and deletion of some objects [4]. The key idea is to arrange all participants into a d-dimensional virtual coordinate space, where each node takes responsibility for managing a zone. The resource is mapped to a coordinate in the virtual space and then stored, in the form of (key, value) pairs, in the node whose allocated zone covers the coordinate. An object lookup request is propagated and routed to the node which stores the desired key pair by an overlay routing algorithm.

Routing in CAN follows the straight line path through the Cartesian space from source to destination coordinates [5]. Each node maintains a routing table, which contains the IP address and virtual coordinate zone of each of its immediate neighbors in the coordinate space. Note, “Immediate neighbors” means that two nodes’ coordinate spans overlap along d-1 dimensions and abut in one dimension.
A lookup message contains the target coordinate in the CAN virtual space, and the destination node is the one whose zone covers the target coordinate. The routing is performed simply by greedy forwarding the message to its neighbor, whose zone is closest to the destination node. Suppose the nodes are uniformly distributed in the CAN virtual space, the mean routing length would then be $O(n^{1/d})$, where $d$ is the dimension of the CAN virtual space and $n$ is the number of nodes in the overlay network.

Therefore, it is crucial for a DHT routing to be aware of the underlying network topology. It will lead to high lookup latency and unnecessary network traffic if each routing hop takes a message to a node with a random location in the Internet [6]. However, this is quite difficult due to the frequent arrival and departure of nodes in peer-to-peer networks. Instead of the exact network size, Symphony uses a traditional estimation protocol to obtain an approximate value, denoted by $\hat{n}$. The estimation begins with sampling a constant number of distinct nodes. Then $s$ nodes are chosen at random and the segment lengths managed by these nodes are summed up. Finally, we obtain $\hat{n}$ by dividing $s$ by the sum of the segment lengths. A distributed estimation approach based on network flooding called “Extrema Propagation” was proposed too. An efficient methodology based on modeling crawling inaccuracies as a Bernoulli process was proposed to measure the number of active users in the BitTorrent Mainline DHT network [8]. Meantime, a new information propagation based method was presented and evaluated for participants in an unstructured P2P network to establish the size of the overall network [9].

But all of the methods above could not ensure how close the estimate result will be to the real network topology and therefore could not control the estimation error. In order to solve the problem, we propose better two algorithms, binary tree and range searching approach. The former can calculate an unbiased estimation at the same time creating some hot spots which can cause the peer workload unbalanced. The latter can limit the estimation error to a controllable bound and the resulted error is smaller than the former with the cost of maximum network size is limited. The advantages and disadvantages of the two algorithms will be discussed in this paper too.

2. Greedy Routing Algorithm

When the node $v$ joins the overlay network, a probability distribution function is used to generate a random number, $x$, which lies in $[0, 1)$. Then, $v$ finds the manager, $u$, in whose allocated zone the point $(v.id+x)$ lies in, and construct a long-distance link with $u$ through the general lookup routing protocol. For load balancing, we set the upper bound of the number of incoming links per node to be $2k$. Above this number, subsequent requests will be rejected, and the requesting node should re-throw the dice to select another long-distance contact.

However, the construction of long-distance links is nondeterministic. The probability that a link exists between node $u$ and node $v$ is equal to $d(u, v)^r/\sum_u d(u, v)^r$, where $r$ is a parameter controlled by the designer. The geographic interpretation is that individuals live on a grid and know each individual in the neighborhood within $p$ steps. Besides, they also have some acquaintances in faraway places across the grid. Assuming $p$ and $q$ are fixed constants, we could get a family of network models by choosing different value of $r$. Note that if $r=0$, we obtain the uniform distribution used in the scheme proposed by Watts and Strogatz model [10] in which the long-distance contacts are selected independently of their distance from the source. The larger $r$ is, the more clustered the network model will be. Kleinberg suggests that each message holder at any given step on the message routing path has knowledge of:

1. the local contacts of each node on the grid;
2. the location of the target, $t$, in the form of a lattice point;
3. the locations and long-range contacts of all nodes that have handled the message.
Kleinberg proved that, when \( r=0 \), the lower bound of the expected delivery time, the expected number of steps taken by any distributed algorithm is exponential in terms of the expected minimum path length. In other words, when long-distance contacts are chosen uniformly at random, as in the Watts and Strogatz model, short chains of acquaintances do exist but there is no efficient way to find them.

The small-world property can be fulfilled if each pair of nodes, \( u \) and \( v \), has a probability at least \( [O(n)d(u, v)^2]^{-1} \) of having a long-distance connection. The probability distribution function takes the value \( 1/(x\ln n) \) if \( x \) is in the range of \([1/n, 1]\), and 0 otherwise, where \( n \) denotes the network size. It is a challenge to decide efficiently which node should be chosen as the long-distance contact. If we throw the dice one by one for every node in the overlay while a new node joins, the expected construction time would be \( O(n) \); hence, the efficiency would be poor, not to mention the cost of obtaining all other nodes’ distances for the new participant. Instead of a discrete pdf, we use a continuous version. Then the density function can be transformed into a continuous distribution function.

\[
\int_{1/n}^{1} \frac{1}{x \ln n} dx = \frac{\ln x}{\ln n} \bigg|_{1/n}^{1} = 0 - (-1) = 1
\]

The function is slightly modified to be

\[
F(x) = \frac{\ln x}{\ln n} + 1
\]

Thus, \( F(x) \) ranges between 0 and 1. Then an inverse function

\[
x = F^{-1}(y) = \exp(\log n \ast (drand() - 1, 0))
\]

Where \( drand() \) generates a random number between 0 and 1, can be obtained. The designer can exploit this function to generate independent random values which are distributed according to this distribution \( F(x) \). The probabilistic construction protocol needs every node be aware of the exact network size.

3. Binary Tree based Estimation Algorithm

The algorithm adopts the binary tree based routing method in CAN. When a new node joins the CAN overlay network, it is assigned a zone for which it is responsible. The zone allocation is dynamic. A new node gets its zone by an existing node splitting its own space, retaining half and handing the other half to the new node. There are three steps involved in the above procedure:

1. Find a entry node that is already in the overlay network
2. Use routing mechanisms to find the node whose zone will be split
3. Finally, the neighbors of the split space are notified of the change and update their routing table accordingly.

The new node randomly chooses a point, $v$, in the space and sends a JOIN message with the destination $v$. This message can start from any node in the CAN overlay network. The message is forwarded using CAN routing algorithm until it reaches the node in whose allocated zone $v$ lies. The current occupant node then splits its zone in half, keeps one half as its new zone, and gives the other half to the new node. The split must make the node uniformly distributed in the CAN virtual space. For example, in a 2-dimension CAN, a zone is first split along the vertical dimension and then the horizontal dimension and so on. When a node leaves, it gives back its zone to one of its neighbors, which then merges the zone with its own zone. The following scenario is an example of node insertion as shown in Figure 1 and Figure 2.

1. The newly-joined node must find a node already in the CAN network called an “agent”. In this example, the new node first makes contact with node 1.
2. Node 1 maintains a table that contains a subset of all peers in the overlay and picks a point $v$ at random.
3. Then, node 1 invokes the CAN routing procedure to send the JOIN message with the destination $v$.
4. Each node greedily forwards the JOIN message to the neighbor closest to $v$’s coordinate.
5. Finally, the message arrives at node 8, which then splits its own zone in half and assigns one half to the new node.

```
<table>
<thead>
<tr>
<th>4</th>
<th>3</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
```

![Figure 3. Binary tree of CAN. The Manager Node id and Tree Vertex id are displayed in the form of NodeID](image)

Since there is not a binary tree stored in CAN, we have to design an estimation algorithm by creating a binary tree in our algorithm. Each node $P$ uses an internal tree node and its leaf to represent the area controlled by $P$, and each tree node has an ID number, as shown in Figure 3.
In order to keep each node managing a leaf and an internal vertex while inserting a new node, we have to modify the login algorithm of CAN. The algorithms are described as below.

```cpp
class peer {
    Public:
    int CalcNetTopo(float bound);
    void login(node entrypoint) {
        ...
        internal = entrypoint.joinagent();
        leaf = new vertex();
        leaf.parent = internal;
        leaf.vertexid = internal.vertexid + '1';
        internal.rchild = leaf;
    }
    vertex joinagent() {
        ...
        vertex tmp = new vertex();
        tmp.vertexid = leaf.vertexid + '0';
        tmp.parent = leaf;
        leaf.lchild = tmp;
        Vertex internal = leaf;
        leaf = tmp;
        return internal;
    }
    Private:
    ...
    vertex leaf;
    vertex internal;
}

class vertex {
    Public:
    ...
    int estimateNext(float bound);
    int measureNextLeaf();
    binarystring vertexid;
    vertex parent;
    vertex lchild;
    vertex rchild;
}
```

The following theorems can be proven according to Manku’s proposed method [11].

**Theorem 1:** Node $x$ identifies the largest $\ell$ such that $n_{\ell} \geq 16(1+\delta)\delta^2\ln(2^\ell n_{\ell})$. $2^\ell n_{\ell}$ lies in the interval $n/(1 \pm \delta)$ with a probability of at least $1-2/n^2$, where $\delta > 0$ is a user parameter.

Theorem 1 suggests that node $x$ could detect all the nodes whose top $\ell$ bits equal $x$’s. Because the sample size is $1/2^\ell$ of the population, we can derive the network size by scaling the observed density. But, how large a sub-interval suffices so that with a high probability, the actual number of nodes in the sub-interval does not deviate significantly from that expected? Theorem 1 guarantees that if $n_{\ell} \geq 16(1+\delta)\delta^2\ln(2^\ell n_{\ell})$, $2^\ell n_{\ell}$, the estimate $\hat{n}$ would lie in the interval $n/(1 \pm \delta)$ with high probability. The larger $\ell$ is, the smaller the sample size would be. As a result, we choose the largest $\ell$ such that $n_{\ell} \geq 16(1+\delta)\delta^2\ln(2^\ell n_{\ell})$. Theorem 2 could be derived from theorem 1 by summing all nodes.

**Theorem 2:** With a probability of at least $1-2/n$, the estimate of network size made by each node lies in the range $n/(1 \pm \delta)$. 


To find the largest value of $\ell$, $x$ begins at its allocated leaf vertex. $\ell$ is set as the bit length of the leaf id. If $1 \geq 16(1+\delta)\delta^{-2}\ln(2^\ell n)$, we obtain an unbiased estimate $2^\ell n$ for the network size.

### 3. Range Searching based Estimation Algorithm

As mentioned in our work earlier [12], when a peer node joins the network, a unique identification number defined by $CalcID$ will be assigned by mapping the $x$-coordinate and $y$-coordinate of the zone center to a 31-digits binary array. When a node $A$ initiates the calculation step, it would find the largest $\ell$ to ensure $n \geq 16(1+\delta)\delta^{-2}\ln(2^\ell n)$. The process of measuring $n_\ell$ consists of two steps: The first step searches from $\ell = 62$ to 31, and the second step searches from $\ell = 30$ to 0.

**Figure 4. Example of Range Searching Algorithm in the First Step**

The first step measures the number of peers sharing the most significant $i$ bits with $P$, $i \geq 31$. Figure 4 illustrates an example. Node $A$ launches the estimate operation. It first calculates the $y$-dimension boundary $Y_{up}$ and $Y_{down}$ of the search region and then runs $countup$ and $countdown$ operations, which would seek the nodes matching prefix in opposite directions and proceeds until reaching the boundary. In the above example, node $A$ has the $CalcID$ [0111.....1110|0100.....1010] which is divided into two halves of 31 bits. Assuming the $estimateOp$ gets to the 31st iteration, where $\ell = 32$, the search region ranges from [0111.....1110] 0000.....0000 to [0111.....1110]0111.....1111. Node $A$ computes $Y_{up}$ and $Y_{down}$ as follows: $Y_{up} = 0000.....0000(\ell/2^\ell) = 10.0$, $Y_{down} = 0111.....1111(\ell/2^\ell) = 0.499$. Then $countup$ and $countdown$ operations are launched, proceeding along the line $\{x = X_P = 0.4375\}$. Finally, we obtain $n_32 = 3$. If the condition is not satisfied in the first step, the operation will enter into the next step.

**Algorithm of measuring the $n_\ell$-first step**

upon receiving $<measureOp, X_{center}, Y_{up}, Y_{down}, prefix>$:
send $<countupOp, X_{center}, Y_{up}, Y_{down}, prefix>$ and $<countdownOp, X_{center}, Y_{up}, Y_{down}, prefix>$ to local host
upon receiving $<countupOp, n_{up}^\ell>$ and $<countdownOp, n_{down}^\ell>$:
send $<measureOp, n_{up}^\ell + n_{down}^\ell>$ to start node
upon receiving $<countupOp, X_{center}, Y_{up}, CalcIDprefix>$:
if the $y$-dimension of the allocated zone covers $Y_{up}$ then
if the prefix of the node’s `CalcID` matches `CalcIDprefix` then
    send `<countupOp, 1>` to start node
else
    send `<countupOp, 0>` to start node
end if
else
    pick a neighbor \( N \) in the up direction whose allocated zone crosses the line \( \{ x=X_{center} \} \)
    send `<countupOp, X_{center}, Y_{up}, prefix>` to \( N \)
upon receiving `<countupOp, n_{\ell}up>`:
    if the prefix of the node’s `CalcID` matches `CalcIDprefix` then
        send `<countupOp, n_{\ell}up + 1>` to start node
    else
        send `<countupOp, n_{\ell}up>` to start node
    end if
end if

![Figure 5. Example of Range Searching in the Second Phase](image)

The second step needs to search a larger space. In the instance presented in Figure 5, Node \( A \) sends `notifyleftOp` and `notifyrightOp`, requesting node \( B \) and \( C \) to count the number of nodes which matches the prefix. \( B \) and \( C \) then run `countup2Op` and `countdown2Op` to identify the matching nodes and return the results. Finally, the original node, \( A \), would eliminate duplicate results from each agent, \( B \) and \( C \), and obtain \( n_{\ell} \). `countup2Op` and `countdown2Op` are slightly modified from `countupOp` and `countdownOp`, so we omit the details.

**Algorithm of measuring the \( n_{\ell} \)-second phase**

upon receiving `<notifyAgentOp, X_{left}, X_{right}, prefix>`:
    send `<notifyleftOp, X_{left}, X_{right}, prefix>` and `<notifyrightOp, X_{left}, X_{right}, prefix>` to local host
upon receiving `<notifyleftOp, n_{\ell,left}` and `<notifyrightOp, n_{\ell,right}`:
    \[ n_{\ell} = n_{\ell,left} + n_{\ell,right} \]
    send `<notifyAgentOp, n_{\ell}` to start node
upon receiving `<notifyleftOp, X_{left}, X_{right}, prefix>`:
send <countup2Op, X_{left}, X_{right}, prefix> and <countdown2Op, X_{left}, X_{right}, prefix> to local
upon receiving <countup2Op, n_{up}^\ell> and <countdown2Op, n_{down}^\ell>:
if the x-dimension of the allocated zone covers X_{left} then
  send <notifyleftOp, n_{up}^\ell + n_{down}^\ell> to start node
else
  pick a neighbor N in the left direction
  send <notifyleftOp, X_{left}, X_{right}, prefix> to N
upon receiving <notifyleftOp, n_{left}^\ell>:
  send <notifyleftOp, n_{left}^\ell + n_{up}^\ell + n_{down}^\ell> to start node
end if
upon receiving <countup2Op, X_{left}, X_{right}, CalcIDprefix>:
if there is no neighbor in the up direction then
  if the prefix of the node’s CalcID matches CalcIDprefix then
    send <countup2Op, 1> to start node
  else
    send <countup2Op, 0> to start node
  end if
else
  sum:= 0
for all neighbor N in the up direction whose allocated zone overlaps the range
[X_{left}, X_{right}]
  send <countup2Op, X_{left}, X_{right}, CalcIDprefix> to N
upon receiving <countup2Op, n_{up}^\ell>:
  sum:= sum + n_{up}^\ell
end for
if the prefix of the node’s CalcID matches CalcIDprefix then
  send <countupOp, sum+ 1> to start node
else
  send <countupOp, sum> to start node
end if
end if

5. Simulation Results

In Section III and IV, we introduced two different approaches to estimate the network size. Here we analyze the pros and cons of both methods. First we examine the mean estimate network size generated by the following three algorithms: the traditional approach used in Symphony, the binary tree algorithm, and the range searching algorithm. The network size ranges from 1,000 to 10,000 nodes and the user parameter, $\delta$, is set to 2. We plot three values on the chart, the mean estimate network size; the mean estimate network size, plus one standard deviation; and the mean estimate network size, minus one standard deviation. In Figure 6, we observed that the traditional approach, which derives the network size from the sum of the zone areas managed by neighbors, can not achieve a tight estimate. The standard deviation grows rapidly when the network size increases. Moreover, the mean network size deviates from the real network size gradually. On the other hand, the two approaches that we propose achieve similar performances. The mean estimate network size is tight with the real network size, and the dispersion is smaller than that of the traditional approach. However, we will show that the estimate deviation of the range searching algorithm rapidly deteriorates when $\delta >3$. 
We eliminate the traditional approach during our initial assessment. Then we used three measures, the mean load of nodes, the load dispersion, and the estimate dispersion, to compare the performance of the other two approaches. One thousand lookup requests were initiated simultaneously when nodes were inserted, and insertion procedure continued until the network size approached $2^{12}$. The load per node is defined as the number of lookup requests accepted by each node. As shown in Figure 7(a), binary tree approach has a lower mean load than the range searching algorithm. This is exactly as we anticipated, since the former provides an extra distributed data structure, i.e., a binary tree, embedded in the overlay network. It facilitates measuring the number of nodes matching a given prefix by placing nodes with the same prefix in the same branch. In contrast, the range searching algorithm has to search a relatively huge area and hence requires much more time.

Figure 7(b) shows an unexpected result. Even though the binary tree approach is based on a non-symmetrical data structure, it still yields a better load balance than the range searching algorithm, which was designed originally as a balanced scheme. To interpret this result, we measure the mean load of nodes at each level of the binary tree. In our hypothesis, we assumed, incorrectly, that nodes at higher levels receive more requests from their children nodes and hence have a higher probability of forming hotspots. Nevertheless, the result shown in Figure 8 contradicts our hypothesis completely. Take $\delta = 0.5$ for instance, the mean load remains stable in level 0, level 1, and level 2, but thereafter it increases sharply. The reason is that most of the estimating operations do not obtain a qualified estimate to satisfy the condition mentioned in Theorem 1 until level 3 is reached. This conjecture can be further confirmed by the observation that the smaller $\delta$ is, the lower the level that an obvious increase will occur. Even so, the rate of increase is much more gradual. This is a desirable property since most nodes gather at the lower levels, and the load is distributed evenly among these nodes, instead of being concentrated on a small
number of nodes at higher levels. As a result, the binary tree approach is a fairly balanced scheme.

![Graphs showing comparison between binary tree approach and range searching algorithm](image)

Figure 7. Comparison between the Binary Tree Approach and the Range Searching Algorithm in a Network with $2^{12}$ nodes

We now return to Figure 7(c). Here the definition of the estimate deviation is modified from standard deviation by substituting the real network size for the mean estimate network size. In this way, we can measure how much the average of the estimate deviates from the real network size and the degree of dispersion of these estimates in the same time. We see in Figure 7(c) that the performance of the range searching algorithm is similar to the binary tree approach in the early stages, but it deteriorates sharply when $\delta >3$. 
When we measured the mean routing path in Section II, we made an assumption that if the routing hops exceed 9,999, which is obviously larger than the diameter of the network, the lookup message would get stuck in an infinite loop. After that, a timeout exception occurs and the initiating node tries another way to transmit or counts this as a failure. Previously, this never happened because the routing algorithm was guaranteed to find the target within a limited number of hops in a perfectly functioning network. However, for a real network, where nodes are notoriously transient, resilience to repeated and concurrent failures is a very important consideration. Most of peer-to-peer systems exploit a recovery mechanism to guarantee their performance against failure. However, we contend that the system must possess a certain degree of “static resilience”. In other words, the system has to keep functioning properly in the presence of failure before the recovery mechanism has a chance to correct the routing tables. Here we measure the static resilience in terms of the percentage of failed lookups. It is assumed that each link has a certain probability of failure.

Figure 9 shows the percentage of failed lookups experienced by each network of 1,000–20,000 nodes, with a failure ratio between 0 and 0.2. In the original CAN, nearly 90% of the lookup messages are dropped when the failure ratio equals 0.2. It seems that most the lookup failure is due to the timeout caused by an infinite routing loop. In a perfect functioning network without any failure, it is guaranteed that a node \( u \) holding the lookup request can find a next recipient which makes progress to the destination node. However, when some links fail, \( u \) may be closer to the target than all its neighbors. There are two solutions for this problem. First, place the request in the queue and wait for the recovery mechanism fixing the link to the target; second, detour the lookup message by choosing the node \( v \) which is closest to the target among all its neighbors. We use the second solution in our simulation. It may cause an infinite routing loop when \( u \) is also the node closest to the target among all \( v \)’s neighbors.

With additional long-distance links, our small-world CAN reduce the percentage roughly half of lookup failure. Furthermore, the small-world CAN with binary tree based algorithm makes more significant progress. It is because that binary tree based algorithm maintains a look ahead list of its neighbors’ neighbors. When a chosen two-hop-away contact is down, it can instead choose the next recipient from the one-hop-away neighbors. It slows down the efficiency but keep the lookup operation making progress in each routing hop. For this reason, it is quite beneficial to use the binary tree based algorithm in the peer-to-peer network.
Figure 9. Percentage of Lookup Failures experienced by the Respective Network Topologies with 1000–20000 Nodes

7. Conclusion

In this paper we address the network topology aware problem by designing two measurement algorithms to estimate the current size of the network. To summarize, the binary tree approach indeed performs better than the range searching algorithm in many aspects. However, it must pay some price, for example, the extra
maintenance and the fault-tolerant mechanism. In fact, we can regard the binary tree as a virtual space that provides a function to estimate the network size. Combining two different overlays is complicated and may cause quite a few unexpected problems, especially in a rapidly changing environment. We will investigate this area in our future work. To realize a comprehensive system, there are still some problems must be resolved.

Our scheme is guaranteed to recover gracefully from a failure on the assumption that CAN’s recovery mechanism functions properly in the presence of a mutable environment. However, further work is needed to verify the correctness of CAN’s recovery mechanism.

In this paper, we assumed that all nodes have the same capacity. Nevertheless, in practice, current peer-to-peer populations are quite heterogeneous. The original protocol should be modified to exploit this heterogeneity.

References
