UNIVERSITÉ de Liège

Design of a Peer-to-Peer System for Network Measurement

by
ADRIEN KUNYSZ

August 2007

Advisor: PROF. GUY LEDUC

Acknowledgements

I would like to thank Prof. Guy Leduc, Mr. Emmanuel Tychon and Teaching Assistant François Cantin for their valuable insights and suggestions. I also thank all the reviewers for their helpful comments, especially Xavier Dalem who spontaneously volunteered to proofread parts of this paper.

Contents

Chapter 1

Introduction

While networks are growing faster and becoming more valuable every day, managing them efficiently becomes both harder and more important. There are of course technical problems involved in monitoring ever expanding networks. Performances and reliability become serious issues when centralized services outgrow their bottlenecks. But there are also organizational concerns. Large interconnected networks with separated administrative entities can not be managed efficiently by a central authority forever. The various operators need to cooperate to ensure the network keeps working correctly.

In this context the use of a distributed monitoring system seems useful to address both problems. On the technical side, adding peers to a properly designed peer-to-peer system make it more useful and reliable instead of less useable and more prone to failure. On the administrative side, a decentralized monitoring system to globally share measurements may let operators anticipate and diagnose problems with minimal administrative overhead and maximum efficiency. Furthermore a peer-to-peer system without single point of failure is naturally more resilient to significant networks problems thus ensuring it stays available when it is most needed.

In this work we first analyze the existing tools that can be used to design an efficient distributed monitoring system in chapter 2. We then describe the implementation, advantages and limits of a peer-to-peer system for network measurements in chapter 3. This is followed by a review of related work by other authors in chapter 4. We finally conclude with some words summarizing what has been done and how this work can be used in chapter 5.

Chapter 2

Overview of Peer-to-peer Systems and Network Metrics

This chapter describes the concepts that will be used in chapter 3 as well as some alternatives.

2.1 Peer-to-peer Architectures

The traditional paradigm in network systems is the client-server architecture in which several clients connect to one or more servers which provide a service. There is no direct connection between client hosts. When two or more clients have to communicate with each other they have to make all their messages go through a common server. Some typical client-server protocols are HTTP, Telnet and SSH.

The "peer-to-peer" term is used to name a lot of different things but in this document it refers to a computer network in which hosts (which are called *nodes* or *peers*) can act both as servers and clients. That is, each host can connect directly to any other host. These networks have been popularized by Napster, KaZaA and more recently by eDonkey and BitTorrent. A peer-to-peer system is usually considered more robust (since there is no single point of failure[1]) and scalable (since adding nodes is not supposed to increase load on a single server) but slower than a client-server architecture under normal load. A peer-to-peer network is usually built on top of an existing network which serves as the transport layer for the overlay network. That is, an arbitrary subset of the hosts of the physical network are nodes of the peer-to-peer network.

What follows is a summary of the main criteria used to classify peer-to-peer network architectures (sections 2.1.1 and 2.1.2) as well as an overview of a common problem in such networks (section 2.1.3).

2.1.1 Centralization

A peer-to-peer network is said to be *centralized* when the only part of the network that is distributed is the data itself while the network management stays centralized. In such a network, all peers have to contact a common master host that will tell them where to find the data they seek. These data move from peer to peer directly but the way to find which host has it is centralized. BitTorrent, eDonkey and Napster (in their original form) are popular examples of such peer-to-peer networks.

On the other hand, all nodes of a decentralized network are equal. They all run the same algorithm in such a way that the overlay network management is fully decentralized. Overlay networks of this type are naturally a lot more robust since there is no single point of failure. Any node can crash without disturbing significantly the network.

Besides, some peer-to-peer architectures are hybrid : some nodes (usually called *supernodes*) act like in a decentralized network while others, which form the bulk of the network, are like normal peers in a centralized network with the supernodes acting as masters for them. It means an hybrid network is a decentralized network of centralized networks. A typical example of this kind of peer-to-peer architecture is the FastTrack protocol powering the KaZaA software.

[1]This is not necessarily true because some so-called peer-to-peer networks are highly centralized while a client-server system may be built in such a way that it is so redundant it will be essentially without a single point of failure.

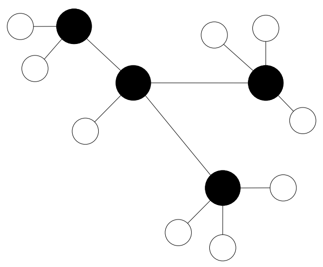


Figure 2.1: Example of a small hybrid peer-to-peer network. Black nodes are *supernodes* acting as *masters* for the other nodes. Edges represent the communications concerning network management while communications conveying data can happen between any two nodes.

In practice, most peer-to-peer networks require some kind of centralization at least to bootstrap nodes (see section 2.1.3).

2.1.2 Structure

In a structured peer-to-peer network, each node is responsible for a clearly defined part of the shared data set. Given a key, there is an algorithm to find a node who has the corresponding data efficiently.

Conversely, in an unstructured network, any node can have any data and there is no guarantee that the sought value will be found even if it exists somewhere on the network since there is no way to query every node efficiently. Moreover, this approach will usually generate significantly more traffic than the structured way that will try to exploit the overlay network more efficiently.

2.1.3 The Bootstrap Problem

A problem common to all decentralized peer-to-peer architectures[2] is that in order to join the network a node needs to know the address of at least one other available node. There are several ways to solve that problem but most of them add some kind of centralization.

An easy way to bootstrap a decentralized network is to make all nodes register to a centralized directory server. When a node wants to join the network it just has to ask the server for one or more entry point. Of course this adds an important bottleneck and single point of failure which makes the network significantly less robust and scalable.

The problems of this solution can be mitigated by using a hierarchy of directory servers. However that hierarchy itself relies on a centralized mechanism. This still seems a good pragmatic solution in most cases because such a hierarchy is already in place in most computer networks through the use of domain name servers.

The DNS system makes it possible to bind any kind of data to a *domain* and to have that information easily replicated to a lot of intermediary servers. The use of DNS can make the distribution of entry points for a peer-to-peer network a lot more scalable and reliable than with a simple centralized server but the modification of the addresses of these starting points may take several hours to propagate to all servers that have cached the information. In this case, the risks would be to give addresses of nodes that are not part of the overlay anymore and to always give the addresses of the same nodes which would overload these nodes while other ones stay idle. The *time to live* of each address may be set so low that no dead node stays for long in the list and addresses keep changing but it would defeat the purpose of DNS in this case, which is to avoid reaching the master server for each request.

A really decentralized bootstrapping could be implemented by broadcasting an "hello" message to all hosts on the node's subnetwork, then to hosts

[2] Decentralized networks are not the only field in which bootstrap problems arise. In computer science the term is most often used when talking about *booting* (hence the name) a computer or to refer to the process of writing a compiler for a language in which it is written but it may also apply when speaking about establishing a trust relation. Moreover it is used in other fields like statistics, physics, linguistics or finance to refer to a solution to some kind of chicken-and-egg problem.

that are one hop further, and so on. In practice such a ring search is rarely effective. First, it requires a relatively high proportion of hosts to be part of the overlay to avoid wasting a lot of bandwidth. Secondly, most routers on the internet today don't forward packets with arbitrary broadcast destinations precisely to avoid saturating bandwidth.

An alternative would be to scan (sequentially or randomly) addresses to find nodes but it still requires a high node density and wastes bandwidth. Moreover the upcoming of IPv6 and its 128 bits addresses (versus 32 bits for IPv4) makes such bootstrapping scheme a lot less efficient that it already is since most addresses won't be in use by any host before long.

2.2 Distributed Hash Table Algorithms

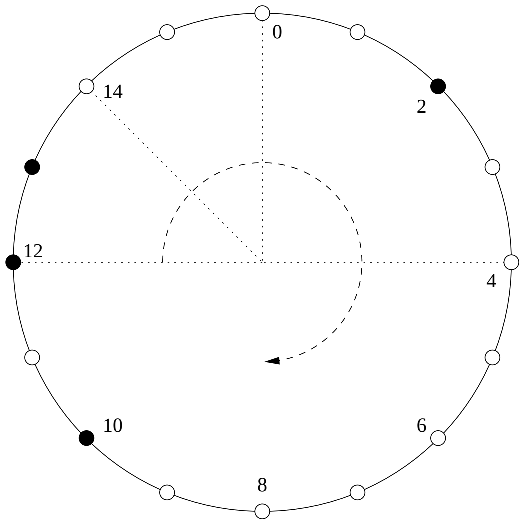
Distributed hash table algorithms are at the hearth of modern structured peer-to-peer systems. Their role is to distribute data in such a way that they are reachable by any node that knows their key in an efficient manner. It means it has to make sure that the addresses of the nodes that have it can be derived from the corresponding key as fast as possible.

Here is an overview of the four original DHT algorithms as they were first described in 2001 : Chord [28] (section 2.2.1), Pastry [24] (section 2.2.2), CAN [22] (section 2.2.3) and Tapestry [33] (section 2.2.4).

2.2.1 Chord

Each Chord node has an unique m bits identifier and all keys are encoded on m bits. Conceptually, all the identifiers are ordered numerically on a circle. Usually, only some of the identifiers on the circle have an associated node since there are less than 2^m nodes. The successor of an identifier *id* is defined as the identifier to which a node is associated that is equal to or follows *id*.

Each node knows its s successor nodes and its s predecessor nodes on the circle by their identifier and address. Each node also maintains a *finger table* with m rows indexed from 1 to m. The ith row of node n contains the identifier and address of the first node $r = \mathit{successor}((n + 2^{i-1}) \bmod 2^m)$.



i	start	successor
1	13	13
2	14	2
3	0	2
4	4	10

Figure 2.2: A simple four nodes Chord circle with $m = 4$ and the finger table for node 12. The black dots are nodes while white ones are identifier to which no node is associated. The circle is divided in m areas representing the m starting points from the finger table.

A value with key k is stored on the node with identifier $successor(k)$. This is achieved by finding the numerically closest but lower than k node identifier and address by using the informations in the finger table and the successor table. If keys and nodes are uniformly distributed on the key space, finding the successor of a given key generally takes $O(\log N)$ steps (with N the number of nodes) because the distance to the destination is roughly halved on each step.

Adding a node to the network means it has to find its successor and insert itself between it and its predecessor. Since any node can crash at any time, successors, predecessors and finger tables have to be checked frequently to be kept up to date. The network is supposed to withstand the crash of up to $s - 1$ consecutive nodes since it can always communicate with the next or previous one.

2.2.2 Pastry

Like with Chord, Pastry nodes are arranged on a circle which can contains 2^m hosts. However, routing in Pastry occurs differently than in Chord. Nodes

02212102	1	**2**2301203	**3**1203203
0	1**1**301233	1**2**230203	1**3**021022
10**0**31203	10**1**32102	2	10**3**23302
102**0**0230	102**1**1302	102**2**2302	3
1023**0**322	1023**1**000	1023**2**121	3
10233**0**01	1	10233**2**32	
0		102331**2**0	
		2	

Table 2.1: A Pastry routing table for node with identifier 10233102 and $b = 2$ as seen in [24]. The top row is row zero. All numbers are in base 4. The underlined parts are the common prefixes while the bold digits are the *next digit*. Cells with a single digit correspond to the current node (it's the ith digit of the node identifier with i the row number) while empty cells represents unknown nodes. Nodes addresses are not depicted.

identifiers are considered as numbers in base 2^b. Each node maintains a routing table with $\lceil log_{2^b} N \rceil$ rows with N the number of nodes in the network. Each row contains $2^b - 1$ nodes identifiers and their associated addresses. The nth row contains references to nodes that have the same identifier than the current node up to the $n-1$th digit but for which the nth digit is the column identifier. The entry in row i and column j is empty either if the ith digit of the current node identifier is j or when no node is known that has the same prefix than the current node identifier with ith digit equal to j.

Besides the routing table, each Pastry node maintains two other data structures: a *leaf set* and a *neighborhood set*. The leaf set contains the $|L|/2$ nodes identifiers and addresses that are closest but follow the current node on the circle as well as the $|L|/2$ nodes identifiers and addresses that are closest but precede the current node on the circle. The leaf set is used in the last part of the routing procedure.

The neighborhood set contains the identifiers and addresses of the $|M|$ nearest nodes according to some proximity metrics independent of the distances on the circle (see section 2.3.1 for some possible metrics). This set is not directly used in the routing algorithm. It exists to try to keep all entries in the routing table near (according to the proximity metrics) the current node. This lets the overlay network adapts to the real network in such a way that communication between two nodes that are far apart according to the proximity metrics is discouraged.

Forwarding a message is simply a matter of finding a node with an identifier that is nearer to the message identifier than the current node identifier. This is achieved by looking up the leaf set first, then the routing table. If no node in the routing table share a prefix with the message identifier that is longer than the prefix of the current node, we try to find a node with the same prefix but that is numerically closer.

PAST, the original Pastry implementation, doesn't store each value in only one node but on the k closest nodes to the key on the circle. This allow for node failure and "faster" (depending of the proximity metrics) look up of values since the client only has to reach the node among k that is nearest to him.

2.2.3 Content Addressable Network

CAN diverges from Chord and Pastry in that it doesn't put nodes on a virtual circle. Rather, values to store are given a key that represents the coordinates of the value in a n dimensions space. Each node is responsible for the values in a n dimensions hypercube in the data space. Each node knows the limits of the adjacent hypercubes and the addresses of the nodes in charge of them. Routing a message simply means to pass it to the neighbor that is in the direction of the destination point.

A new node a willing to join the CAN overlay has to choose a random point in the space and ask the node b responsible for that point to give it some space. The node b then split its hypercube in two and delegates one of them to node a with all the existing $(key, value)$ pairs in that volume. Once this is done, a and b notify their new neighbors of the change.

To allow for nodes failure, each node periodically sends its zone coordinates and its neighbors addresses to its neighbors. When a node a doesn't receive any keep alive message from one of its neighbors b for some time, it decides to initiate a takeover of the zone of b. This is done by sending an appropriate message to b's neighbors. On reception of takeover request, the node cancel the takeover it initiated itself if its new zone would be larger than the one that is being created by the sender of the takeover message.

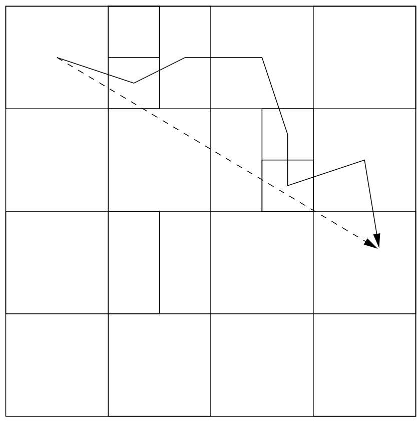


Figure 2.3: Example of the routing of a message in a two-dimensions space with 21 nodes. Each rectangle represents a single node and the message is passed between adjacent nodes based on the destination coordinates and the knowledge of the limits of the current rectangle.

Redundancy may be achieved by establishing several ”parallel” spaces in which each node is responsible for different zones. This can speed up routing by always using the space in which the destination is closer. Moreover this lets nodes fail without loosing data.

2.2.4 Tapestry

Analogously to Pastry, Tapestry routing is done through longest suffix match and the overlay is organized according to a specified proximity metrics (network latency) to minimize some lower level resource usage. However, the overlay topology is not a circle but rather a tree or more precisely, a different tree for each stored object.

Each object stored in the DHT maps to an unique root node which maps the object’s identifier to the identifier of the nodes that store it. A node wishing to make an object available to the overlay has to send a *publish* message to the corresponding root. When passing the message, all the intermediate nodes store an object location pointer mapping the key to the originating node. This means that whenever a query for a given object crosses the publication path, it is directly forwarded to the appropriate node without having to go all the way to the root. This mechanism is illustrated on figure 2.4. The

0642	*042	**02	***0
1642	*142	**12	***1
2642	*242	**22	***2
3642	*342	**32	***3
4642	*442	**42	***4
5642	*542	**52	***5
6642	*642	**62	***6
7642	*742	**72	***7

Table 2.2: A simplified view from [33] of a Tapestry neighbor map for node 0642 when using four-digits octal identifiers ($8^4 = 4096$ nodes in namespace). Each number in the table maps to a (possibly empty) set of nodes matching the corresponding suffix.

location pointer also contains a timestamp for versioning as well as a pointer to the upstream node in the publish path. This pointer allows objects to be moved from node to node by deleting the old path when a new publication message is sent.

Besides a neighbor map (as shown in table 2.2) and location pointers, each Tapestry node maintains backpointers to nodes that have it in their neighbor map. These backpointers are used both to maintain the neighbor map and to to integrate the node in the overlay.

2.2.5 Recursive Routing vs Iterative Routing

All DHT routing algorithms can be implemented either by using recursive routing or iterative routing. Recursive routing let the message ”naturally” travel through the nodes: the originating node contacts only one node. In iterative routing on the other hand, the originating node is the one doing all the work by asking each node on the path for routing information and ultimately by delivering the message to the destination node itself.

As observed in [23], the advantage of iterative routing is that it allows for easy parallelization. This is very useful when nodes failure are common (which is the case for any sufficiently large system) because it can dramatically reduce time lost waiting for crashed nodes. However, iterative routing

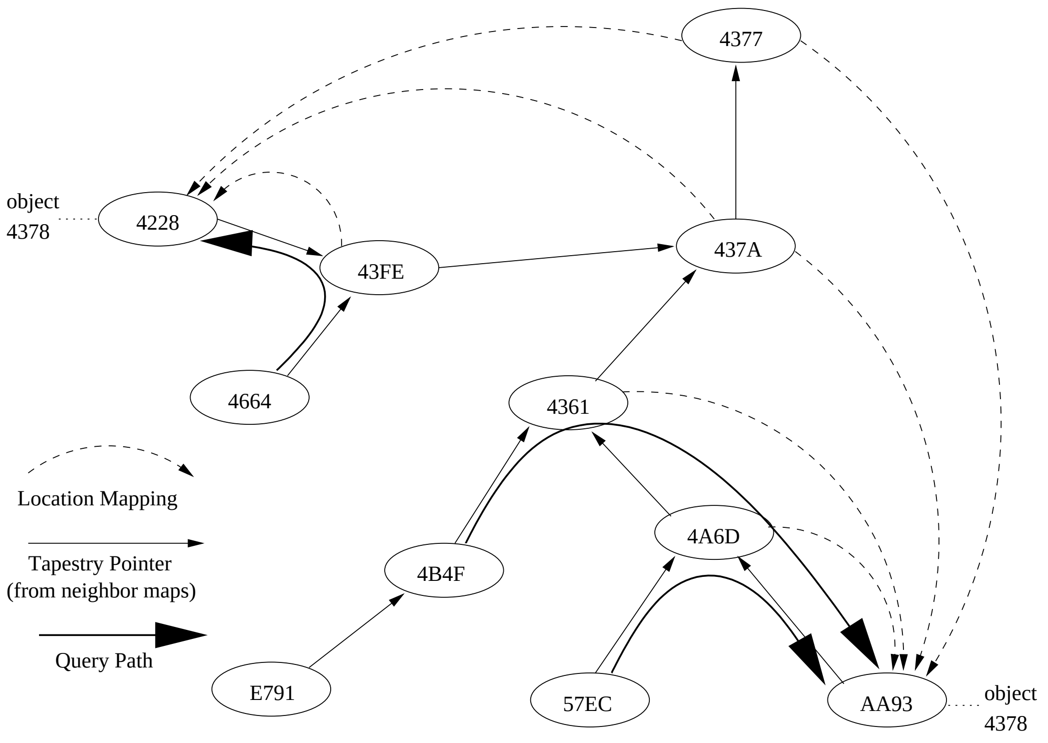


Figure 2.4: Example of a Tapestry route as seen in [32]. The object 4378 is published by nodes 4228 and $AA93$ to root node 4377. When a message sent to object 4378 crosses the publish path, it follows the location pointer to the nearest copy of the object.

makes it more difficult to estimate timeouts because it makes nodes contact a lot of different nodes that are not their neighbors. Contrarily to recursive routing where nodes only communicate with their neighbors (which means they can easily measure normal response time and thus maintain a good timeout estimations). This problem may be resolved by using RTT-based virtual coordinates estimation systems which let a node estimate its distance to any remote node without taking any direct measurement. Some virtual coordinates estimation systems are described in section 2.4.

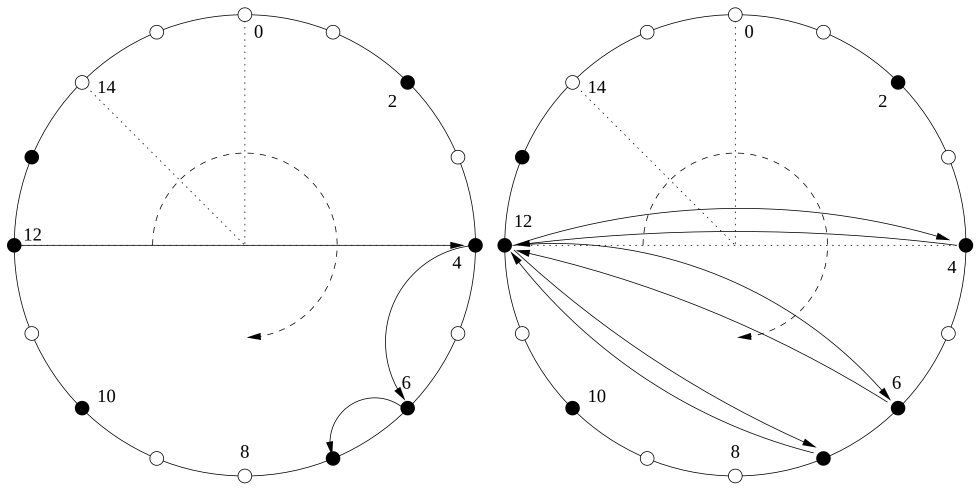


Figure 2.5: Examples of recursive (on the left) and iterative (on the right) routing in a Chord circle ($m = 4$). Notice that in practice, a good implementation will maintain two finger tables per node to allow circling both ways. In this case these figures are not accurate since reaching node 7 from node 12 would be faster going counterclockwise.

2.3 Network Measurement

Network measurements are useful in assessing the state of (part of) a network. Either for operational (detecting problems before the users do,...) or strategic reasons (how many additional customers can the system withstand,...).

All networks measurements can be divided in two classes: active or passive. Active measurements are those for which hosts must explicitly send probes. These probes may or may not be subject to the same network conditions than actual traffic depending of the circumstances. Moreover this additional traffic may cause congestion when generated inappropriately. On the other hand passive measurements only use existing traffic. It means that it can not take measurement about links through which no packet is passing. While this may not seem to be a problem at first sight, some rarely used links have to be closely monitored to make sure they are available (like backup or emergency lines,...).

Beside the metrics themselves one must choose which hosts, links of traffic classes must be measured. There seems to be no fixed rule for that since it depends on the kind of network, available equipment and the network topology itself. For example administrators of an enterprise network with a single ISP and some VPN connections might be more interested in making sure that all user can access a predefined set of servers while internet backbone operators will find more useful to monitor latency from one end to the other of their network.

2.3.1 Metrics

There are several metrics that can be used to measure network performances. Let us consider the most common ones.

Round Trip Time

Round trip time or RTT between two hosts A and B is the time it takes for a packet to go from A to B and back. This measurement is usually done by sending an *ICMP echo* packet (as described in [20]) and measuring the time it takes to receive the corresponding *ICMP echo reply* message. It can also be done passively by timing existing data flows when an immediate reply is expected. RTT is by far the most basic and common metric used in this context. It is easy to interpret and doesn't require any special software on the target host since virtually all IP-enabled equipments can speak ICMP. A significant increase in RTT value usually means that packets are delayed because of congestion on the path between the two hosts. It may also be caused by a change in routing topology. The distinction can often be made by examining the time-to-live field in the IP packet. RTT is the natural metric for timeout estimation as described in section 2.2.5. Of course it is necessary to account for a safety margin to reduce risks of false timeouts.

IP Hops count

IP hops count between two hosts is the number of intermediate routers that lie between those two hosts. IP hops count measures more topological values which don't have much to do with instantaneous network performances *a priori*. However a sudden change of topology usually indicates some serious network problems that have consequences on performances and availability of service. IP hops count measurements are usually taken by sending UDP packets with increasing TTL. First packet is sent with a TTL of one hop which means an *ICMP TTL exceeded* message will be sent by the first router on the path. Second packet is sent with a TTL of two and so on until a packet reaches its destination (and an *ICMP port unreachable* packet is received). It means it takes at least twice as many packet as there are routers on the path to take a measurement. Moreover, there is a large proportion of hosts on the internet that don't reply at all to UDP packets on closed ports because of very defensive firewall configurations (usually to slow down rogue port scans). This can be worked around in most cases by using *ICMP echo* packets (assuming *ICMP TTL exceeded* messages are not blocked and no router on the path enforces packet normalization with minimum TTL[3]) but it still uses a relatively large number of packets.

Packet Loss

Packet loss measurements are useful to detect congestion, overloaded routers and excessive link noise. However it is hard to measure accurately and efficiently since such event are relatively rare on typical networks. Sending too much probes will skew the results while sending not enough probes will render the measurement useless. Packet loss measurement is usually combined with RTT measurement. Passive measurement is also possible but it is not always possible to determine if a packet was really lost on the network or if no response was required from remote host without deep knowledge of the upper-layer protocols (and thus possibly costly analysis of the packets payloads).

Jitter

Jitter is the variation of delay between packets of the same stream. It is usually caused by the variation of routers load. While jitter doesn't matter much for "traditional" IP network services (HTTP, FTP, SMTP,. . .) it becomes critical when dealing with applications that have real-time constraints like live voice and video streaming. Proper jitter measurement is not always possible because both endpoints need to actively cooperate by measuring time between received packets.

[3] Some firewalls can be configured to enforce a minimum TTL for specific packets. This is mostly used to prevent ambiguities in packet interpretation in such a way that network intrusion detection systems see the same thing than the destination host.

Other metrics, namely autonomous systems hops count and geographical distance have been investigated by Huffaker, Fomenkov, Plummer, Moore and Claffy [14] but they are revealed to reflect network conditions poorly and are thus not very useful in assessing networks performances.

2.4 Virtual Coordinates Estimation Systems

As seen in section 2.2.5, an efficient DHT algorithm for a large network of unreliable nodes needs a way to accurately estimate response times between any pair of nodes. This can be done by assigning network coordinates to each node which let us easily estimate *distance* between any two nodes. Moreover, this will allow us to estimate measurements between any pair of hosts assuming the distance metric is the same than the metric we want to measure.

The field of network coordinates estimation is somewhat new but there are already several working solutions among which GNP [18] (section 2.4.1), PIC [10] (section 2.4.2) and Vivaldi [11] (section 2.4.3). Moreover, assigning coordinates implies we need to settle on a geometric space in which we will use them. This is discussed in section 2.4.4.

2.4.1 Global Network Positioning

The Global Network Positioning system and its sequel, the Network Positioning System described in [19], are based on a set of *landmark nodes* from which all the coordinates of other hosts are derived. It assumes a n dimensions Euclidean geometric space in which the triangle inequality holds[4]. The $n+1$ landmark nodes $\mathcal{L}_1$ to $\mathcal{L}_{n+1}$ determine their coordinates in such a way that they minimize the error between the virtual distances and the measured distances. Let $d_{x,y}$ and $\hat{d}_{x,y}$ be the measured and computed distances between x and y respectively and $c_{L_1}, \ldots, c_{L_{n+1}}$ the sought coordinates, we need to minimize the objective function f_{obj}:

$$f_{obj}(c_{\mathcal{L}_1}, \ldots, c_{\mathcal{L}_{n+1}}) = \sum_{\mathcal{L}_i, \mathcal{L}_j \in \{\mathcal{L}_1, \ldots, \mathcal{L}_{n+1}\} | i > j} g(d_{\mathcal{L}_i, \mathcal{L}_j}, \hat{d}_{\mathcal{L}_i, \mathcal{L}_j}) \tag{2.1}$$

[4] The triangle inequality states that for any three points A, B, C in an Euclidean space $|AB| \leq |AC| + |CB|$.

with g being an error measurement function that can be as simple as

$$g(d_{x,y}, \hat{d}_{x,y}) = (d_{x,y} - \hat{d}_{x,y})^2 \tag{2.2}$$

This optimization problem can be easily resolved using any non-linear programming optimization algorithm (the original GNP system uses the downhill simplex method [17]). Once the landmarks coordinates are known, the coordinates h of any host can be computed similarly by measuring the distance to each landmark and minimizing the function h_{obj}:

$$h_{obj}(h) = \sum_{\mathcal{L}_i \in \{\mathcal{L}_1, \ldots, \mathcal{L}_{n+1}\}} g(d_{h,\mathcal{L}_i}, \hat{d}_{h,\mathcal{L}_i}) \tag{2.3}$$

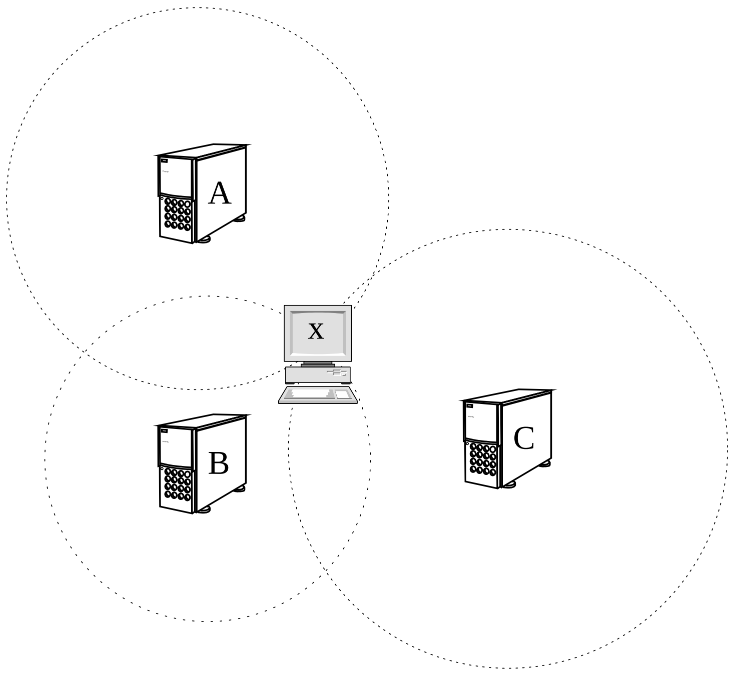


Figure 2.6: Example of GNP and PIC coordinates calculation in a two dimensions space. Coordinates of host x are inferred by measuring its distance to each of the landmark nodes A, B and C.

The most obvious problem with this scheme is that it is heavily centralized: landmarks can get overloaded by measurement requests and a bad selection of these master nodes will make coordinates estimation a lot less accurate. In order to mitigate the centralization problem and make the system more scalable, NPS introduces a landmarks hierarchy in such a way that nodes at level L only contact nodes at level L and $L - 1$.

2.4.2 Practical Internet Coordinates

Like GNP, PIC assumes a n dimensions Euclidean space and each node requires at least $n+1$ landmark nodes to compute its coordinates. However, there are no central landmarks. Each node knows $N \geq n+1$ neighbors from which it chooses its landmarks. Choosing nodes at random from the neighbors set gives better results when estimating relatively long distances while choosing closest nodes makes short distances estimation more accurate. A compromise that seems to give go results overall is an hybrid set of 25% of close nodes and 75% of random nodes.

Finding the closest nodes is done by using PIC itself. When starting, a node first computes its coordinates using a random set of nodes then ask them for addresses of nodes that are close to itself. When it knows enough close nodes, it probes then and recomputes its coordinates using the new measurements. A ring search is also considered but as stated in section 2.1.3 it will work well only if the nodes density is sufficiently high.

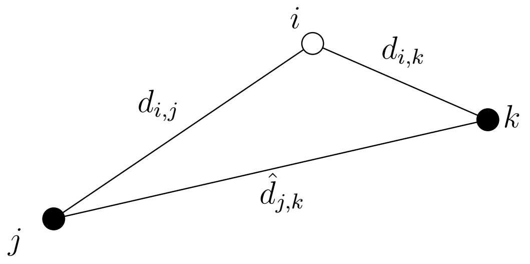


Figure 2.7: Node i measures distances $d_{i,j}$ and $d_{i,k}$ while asking nodes j and k for their coordinates. This lets it compute the distance $\hat{d}_{j,k}$ then remove the nodes that most violate the triangle inequality by applying equations 2.4 and 2.5.

The PIC paper also describes a way to alleviate the effect of malicious nodes that lie about their coordinates. To do so, the node computing its coordinates simply ignores the nodes that most violate the triangle inequality using distances measured to neighbors and coordinates given by neighbors. For each node j in its landmark set, node i computes the corresponding upper and lower bounds u_i and l_i as defined by equations 2.4 and 2.5. The nodes with highest value for these metrics are then removed from the landmark set.

$$u_i = \sum_{k|d_{i,j} \leq d_{i,k} + \hat{d}_{j,k}} d_{i,j} - (d_{i,k} + \hat{d}_{j,k}) \tag{2.4}$$

$$l_i = \sum_{k|d_{i,j} \geq |d_{i,k} - \hat{d}_{j,k}|} |d_{i,k} - \hat{d}_{j,k}| - d_{i,j} \tag{2.5}$$

2.4.3 Vivaldi

The idea between Vivaldi is that each node is virtually linked to some other nodes by springs. Measurements between linked nodes and comparison with current coordinates tells in which direction and how much the node should be moved in the virtual space to minimize effort on the springs. Each node i starts at a small distance from the origin in a random direction. The new coordinates x_i are computed every time a new measurement is taken:

$$x_i = x_i' + \delta \times (d_{i,j} - \hat{d}_{i,j}) \times \frac{x_i - x_j}{\|x_i - x_j\|} \tag{2.6}$$

with δ the displacement timestep and x_i' the old coordinates of node i. The δ factor determines how fast the coordinates will converge and how much they will oscillate around their point of equilibrium. Vivaldi makes the displacement timestep a function of the estimated error of the coordinates of the local and remote nodes:

$$\delta = c \times \frac{e_i}{e_i + e_j} \tag{2.7}$$

with e_i the estimated error on coordinates of node i and $c < 1$ a constant factor. This means that a node that knows that its coordinates are accurate enough won't move a lot on each step and thus won't oscillate much around its real coordinates while nodes with large error will be moved quickly to a position that reflects more precisely their actual coordinates. Moreover, if the remote node doesn't know its coordinates accurately, it won't affect the local node much. The coordinates accuracy is simply derived from the error between the latest measurement and the latest distance estimation.

Vivaldi doesn't require an Euclidean space because it doesn't rely on the triangle inequality. It allows for a better choice of geometric space to model the network as discussed in section 2.4.4. Another possible advantage of Vivaldi over GNP and PIC is that since it doesn't need to run an relatively CPU intensive optimization algorithm, it might be better suited for embedded devices with limited resources.

2.4.4 Choice of the Geometric Space

Assigning coordinates to hosts requires to choose a geometric space in which to embed the network.

Euclidean Space

The most obvious space on which to model networks is a two or three dimensions Euclidean space. However it may not be the most appropriate one. Adding dimensions make all coordinate estimation systems better but empirical study by Dabek, Cox, Kaasshoek and Morris [11] suggests that there is little gain to using more than 2 to 3 dimensions to model the internet accurately while Tang and Crovella [29] which use more diverse datasets show that there is no point in using more than 7 to 9 dimensions. The same paper suggests that despite the potential for triangle inequality violations on the internet it is generally respected when considering RTT metric. For any three hosts A, B, C in the used dataset, there are less than 7% of cases in which $\|AC\| + \|CB\| < \|AB\|$. Moreover less than 10% of all the hosts pairs A, B have an alternate path going trough a third host C that is more than 20% shorter.

Spherical Space

After observing that ‘the distances we are attempting to model are drawn from paths along the surface of a sphere (namely the Earth)’ Dabek *et al.* [11] tried to embed the internet into a spherical space. Results show that a two-dimension Euclidean space performs better. This is because internet paths are not always following the shortest physical path even at the Earth level. That is most routes between Europe and Asia go through America.

Hyperbolic Space

Work done by Shavitt and Tankel [27] suggests that the internet is well modeled by an hyperbolic space. This is explained by the fact that the internet is mostly a core with remote nodes attached to it through links of various lengths. This means that any path between any two hosts will be "bent" to the core. Experimental results show that such a space gives more accurate coordinates estimations than traditional Euclidean models excepts for very short distances that are overestimated.

Height-Vector Space

A simpler to manipulate but less natural geometric space has been devised by Dabek *et al.* [11] as an Euclidean space augmented with a height. To go from point X to point Y, one has first to cover the height from X to the Euclidean space, go to the projection of Y then travel along the height of y. This can be formalized as such:

$$\begin{aligned} [x, x_h] - [y, y_h] &= [(x - y), x_h + y_h] && (2.8)\\ \|[x, x_h]\| &= \|x\| + x_h && (2.9)\\ \alpha \times [x, x_h] &= [\alpha x, \alpha x_h] && (2.10) \end{aligned}$$

with x_h the height of X and x its coordinates in the associated Euclidean space. This model is a rough approximation of an hyperbolic space. Instead of being simply bend to the core, all paths have to go straight to the Euclidean space and back. Experiments show that the height vector space base on a two dimensions Euclidean space is more accurate than a simple three dimensions Euclidean space.

It should be noted that most of the cited studies in this section use dataset derived from the internet. It means that as well as a model fits the internet, it doesn't necessarily represent other networks with the same accuracy. For example, *ad hoc* wireless networks may be highly decentralized to the point of not having any significant backbone. This is important, because this is precisely that kind of collaborative networks that could get the most benefit from a decentralized monitoring system (see section 3.7). Moreover the prominent metric used for coordinates and distances estimations is the round trip time to the detriment of other ones. This is not necessarily a problem as most network metrics are linked in some way to latency between hosts.

Chapter 3

Implementation

This chapter describes the design of a distributed monitoring system, the choices that have been made, possible improvements that have not been implemented (section 3.6) and test results (section 3.5).

3.1 Which Metric to Use for What?

There are two network metrics to choose for this system. There is the metric that will be measured by the system of course but there is also the proximity metric to use in the DHT algorithm to optimize performances (as seen in sections 2.2.2, 2.2.4 and 2.2.5). While the former should be replaceable by any measurable metric the later should be carefully chosen to reflect network performances in order to estimate timeouts accurately.

Most proximity metrics used by DHT implementations and virtual coordinates estimation systems are based on RTT measurements. This seems to be a sensible choice for timeout estimation and to try to keep the best possible performances. As it is also one of the easiest metric to measure both passively or actively we will keep this choice for the implementation.

The metric measured by the system could be any of those cited in section 2.3.1 but I chose RTT for simplicity's sake. Moreover it allows us to reuse the virtual coordinates estimation system for more than just performances.

It permit indirect measurements as described in section 3.4. However to be extended to other metrics this requires a parallel coordinates estimation system for each metric.

3.2 Modifying an Existing Implementation

Instead of writing a complete DHT and virtual coordinates estimation system from scratch, I chose to modify an existing implementation. It means I had to write less code than what would have been required to build a completely new implementation but it still required significant work since understanding an existing application deep enough to modify it is not always an easy task. Moreover, modifying an existing implementation is generally more useful for the original author and users because, even if changes are not directly assimilated, they usually reveal some (potential) bugs that wouldn't have been found and fixed otherwise.

There are numerous implementations of the distributed hash table algorithms described in section 2.2 but there are very few open source[1] DHT that incorporate a network coordinates estimation system. In fact, the only one I was able to find is Bamboo [6], developed by Sean Rhea for is PHD thesis [23]. An alternative to modifying Bamboo would have been to incorporate an existing network coordinates estimation system into an existing DHT but this is the kind of software integration that may prove time expensive while modifying an existing software including both systems is easier.

Bamboo is an open source[2] Pastry DHT written in Java that uses iterative routing and the Vivaldi [11] coordinates estimation system. It has been designed to be especially resilient to slow nodes and rapid nodes joins and failures. To achieve this goal it performs periodic recovery by replicating data to a configurable number of nodes. Objects stored into Bamboo are in fact quadruplets $< key, value, h(secret), ttl >$. The *key* and *value* are self explanatory but the last two fields are worth an explanation. The *ttl* (short for "time to live") is the time remaining before the object is deleted from

[1] Open source as in "the source can be freely modified for academic purpose".

[2] Open Source as in the Open Source Initiative definition since it is licensed under the three-clauses BSD license.

the node if it isn't refreshed. The $h(secret)$ is a cryptographic hash[3] of a secret that may be used to remove the object from the DHT. When a node receive a remove request containing triplet $< key, h(value), secret, ttl >$, it discards the corresponding object and propagates this value to replicas while TTL isn't exhausted. Objects are uniquely identified by the triplet $< key, h(secret), h(value) >$ which means that it is possible to store several values with the same *key* and $h(secret)$.

Bamboo is currently deployed on PlanetLab as part of the OpenDHT project [2]. PlanetLab [3] is an overlay network of nearly 800 nodes distributed on over 400 sites. It is used as a testbed and deployment platform for a variety of research projects involving distributed networks. It is notoriously unreliable[4] and thus is the perfect environment to test a distributed application that tries to be resilient to nodes failure. Anybody connected to the internet can store and retrieve data on the OpenDHT network.

Bamboo doesn't really try to solve the bootstrap problem (see 2.1.3): node operator needs to explicitly specify at least one existing node address in the configuration file. However addresses of OpenDHT nodes are available through DNS requests which effectively alleviate the problem since these addresses don't change often.

3.3 Bamboo Stages

Bamboo is based on a *staged event-driven architecture* (SEDA) [30] which divides the application into *stages* which communicate by sending *events* to each other through *queues*. The Bamboo architecture make it relatively easy to add features taking advantage of the DHT and Vivaldi stages. One just has to write a new stage which registers events types it should receive and sends events to other stages whenever needed. Once the stage is written and compiled, it must be added to the Bamboo configuration file (which is also

[3] A hash is a fixed length "summary" of an arbitrary length string. A hash function h is said to be cryptographic if (a) given a hash h' it is "hard" to find a m such as $h' = h(m)$ and (b) it is "hard" to find two different messages m and m' such as $h(m) = h(m')$. Popular cryptographic hash function are MD5 and SHA-1.

[4] An amusing example comes from [26] where the author says 'We were thus tempted to blame our performance woes on PlanetLab (a popular pastime in distributed systems these days)'.

used to pass parameters and options to stages) so it is loaded when Bamboo is started.

Another advantage of the use of SEDA in Bamboo is that it is possible to set up nodes that don't run all the stages just by modifying a configuration file before starting the application. An example would be a node running the DHT stage (and thus storing the data) but not the monitoring part (and thus not probing any host outside the overlay).

What follows is a short description of the main Bamboo stages. They are presented roughly from lower to higher level.

bamboo.lss.Network
The `Network` stage is the one responsible for most of the low level network communication using UDP packets. It receives `NetworkLatencyReq` and `NetworkMessage` events that are handled by sending the corresponding packets on the network and emits `NetworkLatencyResp` and `NetworkMessageResult` events. Stages willing to receive network messages have to register their handler using the `Network.registerReceiver()` method. A network simulator is also available using the `bamboo.sim.Network` stage that inherits the `bamboo.lss.Network` stage and redefines the appropriate methods to simulate a whole network of Bamboo nodes using only one computer.

bamboo.router.Router
The `Router` stage is responsible for mapping the node identifier to the corresponding IP address, maintaining the Pastry circle and routing messages to nodes. This is basically the hearth of the Pastry implementation. The main event type it has to handle is `BambooRouteInit` which is a a request to route a message in the overlay. Stages wishing to receive messages received through Pastry have to register with the `Router` stage by sending it a `BambooRouterAppRegReq` event.

bamboo.db.StorageManager
The `StorageManager` is responsible for storing data on disk using Berkeley DB files as backend. The main events it receives are `PutReq` for storing new objects, `DiscardReq` to discard existing objects, `GetByKeyReq` to retrieve stored objects and `IterateByGuidReq` to retrieve objects spanning a specified key range. `IterateByGuidResp` objects that are sent in re-

sponse to `IterateByGuidReq` events may contains a handler to include into `IterateByGuidCont` events in order to continue an iteration that hasn't been completed in one step. The `StorageManager` stage also makes sure that all data is written to disk before Bamboo shuts down so it can load it back when it started again.

bamboo.dmgr.DataManager

The `DataManager` stage is the one in charge for replicating objects to other nodes as well as transmitting remove requests. It essentially handles `PutOrRemoveMsg`, `FetchDataReq` and `FetchMerkleTreeNodeReq` events. In order to keep synchronized data over all the nodes replicating a specified key with minimum network overhead, the `DataManager` uses Merkle trees. Merkle trees (or hash trees) are data structures that are used to divide arbitrary data into smaller chunks that can be checked for integrity individually in any order. They are used by `DataManager` to efficiently find which objects are not consistent between two replicating nodes.

bamboo.dht.Dht

As the name suggests, the `Dht` stage is the interface to the DHT itself. It receives `PutReq` and `GetReq` events whose content is forwarded either to remote nodes or to the local `StorageManager` stage. Requests to remove objects are in fact `PutReq` events with the `put` flag set to false. After handling a `PutReq` or `GetReq` event, the `Dht` stage sends back a `PutResp` or `GetResp` event telling whether the request succeeded and the requested object encapsulated in a `GetValue` object when appropriate. Figure 3.1 details the main events received and sent by the `Dht` stage.

bamboo.dht.Gateway

The `Gateway` stage is responsible for handling communications with hosts outside the overlay either through a Sun RPC or XML-RPC interface. There are five procedures that clients can call:

put which takes three self explaining arguments: `key`, `value` and `ttl`. There are three possible return values which may mean the operation succeeded, that there are not enough available nodes for replication or that a transient failure occurred.

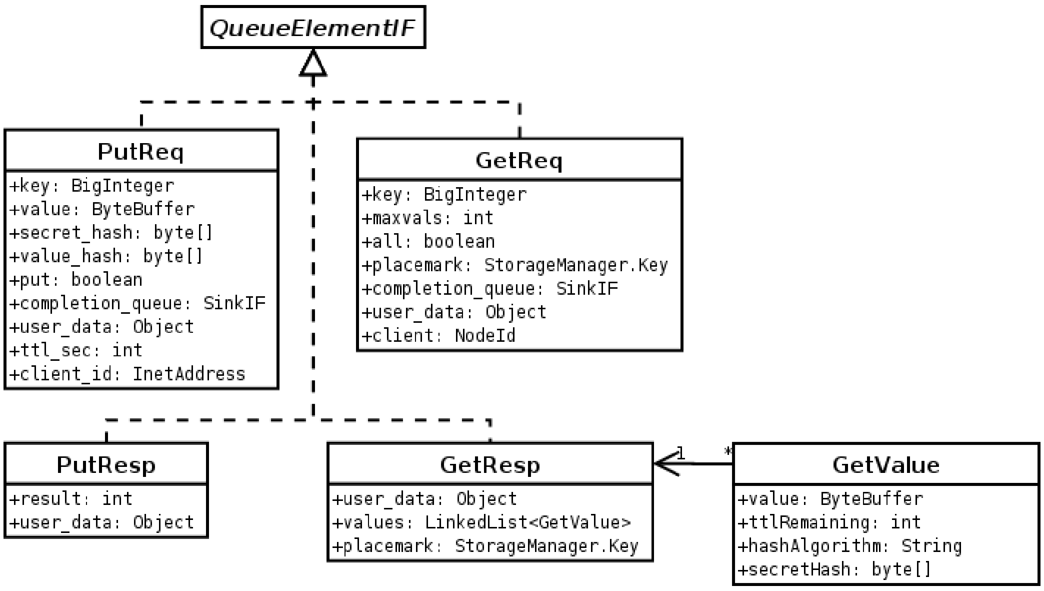


Figure 3.1: Simplified class diagram of the event classes associated to the `Dht` stage. The `GetValue` class is not an event (it doesn't implement `QueueElementIF`). It is used to transmit the value itself from the `Dht` stage to other ones through a `GetResp` event. The `placemark` field found in `GetResp` and `GetReq` events is a handler to allow iterating over objects.

get has three arguments too: `key`, `maxvals` which indicates the maximum number of values to return and `placemark` which is a handler that is used to continue iterating on values returned by a previous `get`. The return value contains a set of `values` and a `placemark`.

put_removable is like `put` but with two additional arguments: `secret hash` which has been described in section 3.2 and `hash type` which is supposed to let the user choose what hash to use for the `secret hash`. The only valid hash types are "SHA" and the empty string to indicate that no secret hash is used.

get_details is like `get` but with more detailed return value. For each returned value, the corresponding ttl, hash type and secret hash is also returned.

rm takes five arguments: `key`, `value hash`, `ttl`, `secret` and `hash type` whose purposes are obvious. The return value is the same as for `put`.

All these procedures take two additional arguments: `client library` and `application` which are analog to the "User-Agent" HTTP header (in fact, when using the XML-RPC interface, the `client library` field is filled with the content of the "User-Agent" HTTP header). These requests are converted into `GetReq` and `PutReq` objects that are then dispatched to the `Dht` stage which sends back `GetResp` and `PutResp` events that makes the `Gateway` stage respond accordingly to client. I modified this stage in the following ways:

- A client can now ask a node to start probing a specified host. This was done by patching the XML-RPC `get` callback so that it sends a `ProbeReq` event when the `maxvals` argument is one and the `key` is a valid IP address.
- A client can now ask a node to stop probing a specified host by calling the `rm` procedure with the IP address of the probee as the `key`.
- A client can now ask a node its coordinates in the virtual coordinates system by calling the `get` procedure with the `key` argument being the IP address of the gateway and a `maxvals` value of one. This makes the client able to estimate indirect measurements as described in section 3.4.

bamboo.vivaldi.Vivaldi

The `Vivaldi` stage is a simplified implementation of the Vivaldi algorithm described in section 2.4.3. Instead of being calculated after the estimated accuracy of the remote node, the δ factor is constantly decreased before each coordinates update until it reaches a specific threshold (fixed at 0.05). Three geometric spaces are available: two Euclidean ones (three and five dimensions) and a height-vector one based on a two dimensions Euclidean plane. Latency samples are taken either by piggybacking on `NetworkLatencyResp` events that are used by the `Router` stage to maintain the Pastry circle or by sending `PingMsg` (which inherit from `NetworkMessage`) to random nodes. A problem that may arise when using `NetworkLatencyResp` events to update coordinates is that the sample will be naturally biased since they concern only nodes that are neighbors on the Pastry circle. On the other hand, sending explicit pings to random nodes implies an overhead first to locate the random node, second to ping it.

bamboo.prober.Prober
The `Prober` stage is a completely new stage. It is responsible for taking the measurements themselves and passing them to the `Dht` stage for storage. For performance sake, measurements are also stored locally. Probees can be added either by specifying them on startup in configuration file or by sending a `ProbeReq` event containing the address of the host to probe and the address of the local node. When receiving a `ProbeReq`, the `Prober` stage simply create a `Probee` object with no measurements and add it both to its local cache and to the DHT by sending a `PutReq` event. To stop probing an host, one has to send a `ProbeReq` event with the host address and a `null` prober.

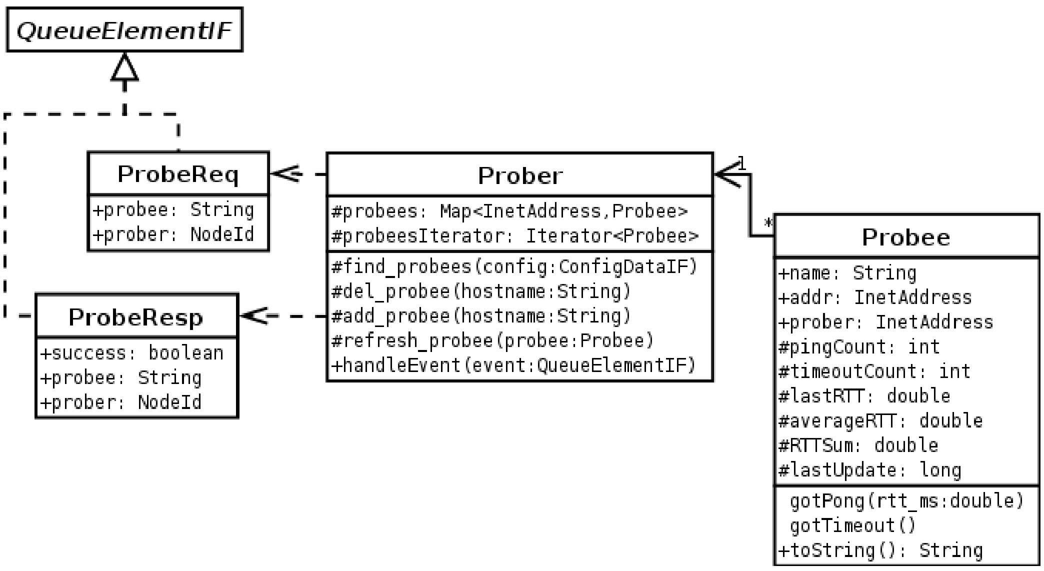


Figure 3.2: Simplified class diagram of the `Prober` stage and its associated classes.

Every 42 seconds (the frequency can be changed in the configuration file by setting the `probes_period` variable), the `Prober` stage receives a `ProbeAlarm` event. This triggers a measurement regarding the next `Probee`. The measurement is done by sending a `PingReq` event addressed to the `PingStage` that is part of the `ostore.network` package (which underlies the `Network` stage). The `PingStage` then sends back a `PingSuccess` or `PingFailure` event which is used to update the corresponding `Probee` object accordingly (both locally and in the DHT). The `Network` stage is not used for probing because it only handles UDP messages while we have to send ICMP echo packets to take RTT measurements with hosts outside the overlay as described in section 2.3.1. This means we can not use the `bamboo.sim.Network` simulator stage without modification either.

3.4 Storing and Retrieving Measurements

After taking a measurement, the `Prober` stage puts a serialized `Probee` in the DHT. Initially they were simply raw Java objects but since they don't really have to be read back by Bamboo and to facilitate debugging they are now human-readable strings summarizing the measurements. The key with which is stored the `Probee` is the IP address of the probed host while the secret hash is the hash of the IP address of the prober node. This lets the client quickly retrieve all the measurements regarding a given host while retaining the capability of removing measurements for a specific $(prober, probee)$ pair. Since the `Probee` stored in the DHT contains the IP address of the prober there is no problem to find the secret corresponding to the hash and thus remove the object from the DHT when needed. However no measurement is ever removed explicitly from the DHT by Bamboo. They are kept until their time to live is over. Since different values having the same key don't overwrite each other, it means there will nearly always be a constant number of outdated measurements stored in the DHT. The number of these outdated values could be reduced by tweaking the TTL. On the other hand, this can be used as an history of all the measurements for the last n hours.

Instead of reusing Bamboo as a client, I wrote an independent client to retrieve measurements from the DHT. It is written in Perl and based on a modified version of the `Net::OpenDHT` library [8] which was originally written to access OpenDHT through the XML-RPC interface. The library was modified in the following ways:

- it now allows the user to choose the `maxvals` argument for `get` requests;
- the `rm` procedure has been implemented;
- gateway address and port can now be defined by user instead of being hardcoded to use OpenDHT.

The client itself handles seven commands:

probe which takes two arguments: the prober and the probee addresses. It connects to the prober which must be running the `Gateway` stage and ask it to start probing the specified host.

rmprobe which have the same arguments than `probe` and ask the prober to stop probing the specified probee.

measure which takes two arguments: a gateway and a probee address. It retrieves all the stored measurements regarding the probee, parses them and prints them.

getcoord whose only argument is a node address which must be running the `Gateway` stage. It just asks it its coordinates and prints them.

getdist which takes two nodes addresses as arguments, ask them their coordinates and prints the estimated distance between them.

infercoord which takes a gateway and a probee addresses as arguments, retrieves coordinates of $N+1$ probers (assuming a N dimensions space) of the probee as well as measurements they took, infers coordinates of the probee from these data then prints them. The algorithm for inferring host coordinates from probers data is detailed below.

inferdist which takes either a gateway and a probee addresses or a gateway and two probees addresses as arguments. It infers coordinates of the probee or probees and prints the distance between the hosts specified by the last two arguments.

Since Vivaldi requires a relatively large measurements set before converging and because we want to be able to estimate coordinates for an host that is probed by a minimum number of nodes the GNP/PIC algorithm is used to estimate coordinates of hosts outside the overlay. It means that an Euclidean space an Euclidean space with a limited dimensions count has to be used. In order to estimate the host coordinates h in the N dimensions Euclidean space we need $N+1$ prober nodes from which we retrieve their coordinates $\mathcal{L}_1, \ldots, \mathcal{L}_{N+1}$ and the measurements $\hat{d}_{h,\mathcal{L}_1}, \ldots, \hat{d}_{h,\mathcal{L}_{N+1a}}$ they have taken regarding the host (in fact all the measurements are retrieved with one request to any node, each prober is contacted only to retrieve the coordinates). We then need to minimize the objective function 2.3. This is done using Matlab and the following error measurement function:

$$g(d_{x,y}, \hat{d_{x,y}}) = (d_{x,y}^2 - \hat{d}_{x,y}^2)^2 \tag{3.1}$$

The distances are squared simply to avoid computing a square root for every evaluation of the function. Once the coordinates of the host are known it is trivial to estimates the distance between that host and any other host for which the coordinates are known.

As stated before, the Vivaldi stage only estimates coordinates regarding the network latency which means only latency can be estimated between two hosts. Other metrics would require to set up parallel Vivaldi stages that would take corresponding measurements.

3.5 Testing

Continuous testing was performed during development. The application was semi automatically deployed and tested on four to twenty nodes before each commit. Using a simple script for deployment through SSH saved a lot of time for test runs. Debugging was done through heavy use of log messages using the log4j framework [1] already in use in Bamboo and the jdb debugger.

As stated earlier, Bamboo provides a network simulator that can substitute itself to the `Network` stage. However the `PingStage` bypasses the `Network` stage which means the simulator can't be used as is to test indirect measurements. Instead four real nodes were deployed on five European hosts with the purpose of monitoring ten probees. Each probee was pinged by four different nodes. The distance to the fifth node was then estimated using the algorithm described in section 3.4. An effective RTT measurement was then performed and the relative error was finally calculated. Among the ten estimations, seven are correct within 10% while the three others show aberrant values with relative error between 150 and 200%. This seems to be due to the fact that these hosts are far away from the other ones while the nodes are all relatively close from each others.

3.6 Possible Improvements

This section describes ideas of future directions that have not been implemented in this work but that seem promising.

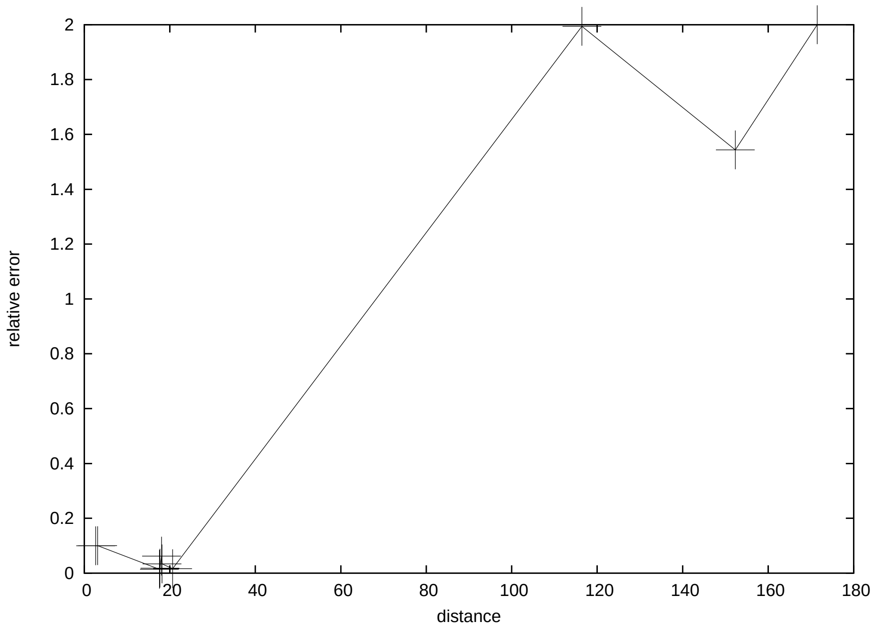


Figure 3.3: Distance estimation shows an aberrant behavior when the probee-prober distance is significantly higher than the prober-prober distances.

3.6.1 Automatically Finding Probees

Instead of letting the user choose which node should monitor which host it would be possible to let the system decide by itself. However it is not clear how this should be done. Discovering potential probees is easy but deciding which nodes should be responsible for probing them isn't. In fact, different networks have different needs and measurements that would be relevant in one might be useless in another.

Most of the ways to find hosts addresses can be divided into two classes: local (or passive) discovery which doesn't need to actively send packets on the network and remote (or active) discovery which requires the prober to actively access the network. Depending on the type of computer on which the software is running, there are various places to scour to find hosts addresses without accessing the network. Among these we can cite:

- the IP routing table;
- the ARP table;
- the DNS cache;
- the connections table (netstat,. . .);
- the headers of inbound and outbound packets;
- the DHCP leases database;
- various configuration files depending on the platform.

Notice that the headers of inbound and outbound packets contain almost all the addresses that can be found at other places but their inspection usually requires special privileges. Moreover, examining them all may be expensive depending on the computer and its use.

Finding new hosts on the network may be done through several techniques depending on the expected hosts density on the current subnetwork:

- ping broadcast;
- traceroute;
- webspider;
- random or sequential addresses scan.

It seems obvious that these discovery attempts should be throttled to avoid any kind of denial of service.

3.6.2 Security Considerations

There is currently no way to restrict access to the DHT short of firewalling the RPC interface. It means that anybody able to retrieve measurements can also put or remove values from the DHT as well as ordering a node to start or stop probing an arbitrary host. This could be used by malicious users to introduce erroneous measurements, delete stored data, initiate denial of service attacks or possibly bypass firewalls. To avoid this problem, an authentication scheme could be integrated in the `Gateway` stage but this would significantly complicate the RPC interface.

Another problem regarding malicious users is the possible presence of rogue nodes especially set up to disrupt the overlay and measurements. As explained in section 2.4.2 PIC includes a mechanism to ignore misbehaving nodes for coordinates calculation. It is adaptable to Vivaldi as long as an Euclidean space is used. However it doesn't solve the problem of nodes manipulating data in the DHT in some pernicious way. The Pastry-based pMeasure DHT [15] (see chapter 4) uses a public key infrastructure to avoid this kind of problem. However authenticating nodes and the messages they send doesn't make them automatically thrustable. The solution may lie in the introduction of a currency system as described by Garcia and Hoepman [12].

3.6.3 Implementation Improvements

Rather than modifying semantic of existing XML-RPC commands it would have been cleaner to add new procedures. However the `Gateway` stage is a 1500+ lines class and the best way to modify it wasn't obvious at first sight. Adding procedures with their own arguments would allow for more flexible measurement requests. For example it would be possible to set probe frequency for each probee instead of probing hosts following a fixed rate round-robin pattern. It would also make alternative measurements easier. Instead of being limited to RTT, any metric cited in section 2.3.1 could be used. However it would require an additional Vivaldi stage for each new metric we want to be able to estimate between any two hosts.

Accuracy of distance estimation could probably by improved by tweaking the error function in equation 3.1 as well as the minimization algorithm used to find the coordinates.

3.7 Use Cases

Such a distributed monitoring network system could be used in various network environments. A deployment at the internet-level could allow services providers to cooperate more efficiently by sharing their measurements without the need for a centralized or hierarchical storage infrastructure. Since all the measurements history taken with this system are available to any network administrator, it could dramatically reduce time needed to diagnose and locate the origin of a lot of network problems.

Another place where such a tool seems to have a lot of potential is in city-wide networks operated by individuals like ReseauCitoyen [5] in Brussels, Île Sans Fil [7] in Montréal or P***s Sans Fil [4] in Paris[5]. Given that each "operator" of these networks is typically an end user without enough technical expertise to set up complex monitoring softwares, that he controls only one router and that there is rarely a robust centralized infrastructure deployed, a distributed monitoring system could be very valuable to network administrators in order to diagnose problems that are not necessarily caused by their own routers.

Of course, this system could also be used in more traditional networks in large companies or universities but these networks usually already have an established centralized administration able to configure each router that may make a distributed monitoring system less useful.

A possible drawback of this implementation is that Bamboo was not designed to run on hardware with limited computing power and memory space. Now, routers (be it "professional" or home-made routers) are typically dedicated machines with relatively small RAM space. Which means this implementation will probably not run properly on most of the currently used routers. However, it is still possible to run it on hosts that are not routers.

Another obstacle that this system would have to overcome for adoption in private networks is the use of firewalls. Obviously nodes must be able to communicate with each other as well as to send probes to probees. This may be a problem on heavily compartmentalized networks.

[5] Paris Sans-Fil is not the correct denomination anymore since July 2007 after a justice decision that this name infringe on the *Paris* name owned by the city. The new name of the association is not known at the time of this writing.

Chapter 4

Related Work

The closest work to this one seems to be pMeasure by Liu, Boutaba and Hong [15]. The architecture is similar with Pastry nodes performing on-demand measurements but it doesn't allow indirect measurements without explicit probing. However it provides a public key based authentication and reputation system that let nodes refuse to take measurement on behalf of the client if it hasn't performed enough measurements itself.

The Scalable Sensing Service (S^3) developed by Yalagandula, Sharma, Banerjee, Lee and Basu is composed of a set of *sensor pods* that take measurements and aggregate them on-demand. Clients may tell pods how the data should be aggregated and subscribe to existing aggregated feeds. Aggregation is done through a distributed information management system named SDIMS [31] based on an arbitrary DHT algorithm. It tries to estimate all the measurable metrics in the network based on the actual measurements and a proximity estimation system namely Netvigator [21]. Unlike Vivaldi, Netvigator doesn't try to estimate global coordinates but only local distances between nodes using a set of landmarks and intermediate routers that are not part of the overlay.

DipZoom by Rabinovich, Triukose, Wen and Wang [16] is a centralized marketplace for buying and selling measurements. It is targeted at companies that want to monitor their services from clients' perspective. However matching the measurement requests to the offers is done only by geographic location.

More abstract papers are the Network Oracle by Hellerstein, Paxson, Peterson, Roscoe, Shenker and Wetherall [13] and the Knowledge Plane by Clark, Partridge, Ramming and Wroclawski [9] which push for more self-managed networks by putting back intelligence in the core. The knowledge plane suggests that measurements should be shared globally to allow the network to heal itself using an integrated approach[1]. Distributed monitoring systems are a step in this direction.

[1] That paper is somewhat reminiscent to the seminal ”End-to-end argument” [25] and indeed they share a common author.

Chapter 5

Conclusion

The goal of this work was to design a scalable, automated and robust monitoring system using a peer-to-peer network. We started by investigating distributed hash table algorithms and their characteristics. We then examined common network metrics, how to measure them and how they can be interpreted. That was followed by a study of some virtual coordinate estimation systems that can be used to model large networks.

We then modified an existing DHT implementation including a virtual coordinates estimation system to turn it into a network measurement tool. This was done by adding a simple probing functionality that can be extended to cover nearly any metric and storing the measurements in the DHT itself. The next step was to modify the DHT interface so that it can control the monitoring tool by telling it which hosts to probe as well as to be able to retrieve the coordinates of the node in the virtual space.

Finally we implemented a client application to query the monitoring system. This tools lets the user control which host should be monitored by which node as well as estimates measurement between virtually any pair of hosts that are probed by enough different nodes (assuming probers have enough distance between them).

This application may prove to be useful to diagnose problems in large collaborative networks with fractioned administration by allowing any operator to quickly find the measurements he needs thanks to the globally shared data. Furthermore the peer-to-peer nature of this system makes it relatively easy to manage since there is no central server to set up and backup since the data are automatically replicated through the network.

Bibliography

[1] log4j: a java logging framework. [Online; accessed 10-August-2007]. http://logging.apache.org/log4j.

[2] OpenDHT: A Publicly Accessible DHT Service. [Online; accessed 23-July-2007].
http://www.opendht.org/.

[3] PlanetLab, An open platform for developing, deploying, and accessing planetary-scale services. [Online; accessed 9-August-2007].
http://www.planet-lab.org/.

[4] P***s Sans-Fil. [Online; accessed 11-August-2007].
http://paris-sansfil.fr/.

[5] ReseauCitoyen. [Online; accessed 11-August-2007].
http://reseaucitoyen.be/.

[6] The Bamboo Distributed Hash Table. [Online; accessed 8-April-2007].
http://www.bamboo-dht.org/.

[7] Île Sans Fil. [Online; accessed 11-August-2007].
http://www.ilesansfil.org/.

[8] Leon Brocard. Net::OpenDHT. [Online; accessed 21-August-2007].
http://search.cpan.org/~lbrocard/Net-OpenDHT-0.33/lib/Net/OpenDHT.pm.

[9] David D. Clark, Craig Partridge, J. Christopher Ramming, and John Wroclawski. A knowledge plane for the internet. In Anja Feldmann, Martina Zitterbart, Jon Crowcroft, and David Wetherall, editors, *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 3–10, Karlsruhe, Germany, August 2003. ACM.

[10] Manuel Costa, Miguel Castro, Antony Rowstron, and Peter Key. PIC: Practical internet coordinates for distance estimation. Technical Report MSR-TR-2003-53, Microsoft Research (MSR), September 2003.

[11] Frank Dabek, Russ Cox, M. Frans Kaashoek, and Robert Morris. Vivaldi: a decentralized network coordinate system. In Raj Yavatkar, Ellen W. Zegura, and Jennifer Rexford, editors, *Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 15–26, Portland, Oregon, USA, 2004. ACM.

[12] Flavio D. Garcia and Jaap-Henk Hoepman. Off-line karma: A decentralized currency for peer-to-peer and grid applications. In John Ioannidis, Angelos D. Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security, Third International Conference, Proceedings*, volume 3531 of *Lecture Notes in Computer Science*, pages 364–377, New York, USA, 2005.

[13] Joseph M. Hellerstein, Vern Paxson, Larry L. Peterson, Timothy Roscoe, Scott Shenker, and David Wetherall. The network oracle. *Bulletin of the IEEE Computer Society Technical Comitee on Data Engineering*, 28(1):3–10, 2005.

[14] Bradley Huffaker, Marina Fomenkov, Daniel J. Plummer, David Moore, and K Claffy. Distance metrics in the internet. July 11 2002.

[15] Wenli Liu, Raouf Boutaba, and James Won ki Hong. pmeasure: A tool for measuring the internet, September 06 2004. `http://bcr2.uwaterloo.ca/~w7liu/pmeasure.html`.

[16] Michael Rabinovich, Sipat Triukose, Zhihua Wen, and Limin Wang. DipZoom: The Internet Measurements Marketplace. In *9th IEEE Global Internet Symposium 2006*, 2006.

[17] J.A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[18] T. S. Eugene Ng and Hui Zhang. Predicting internet network distance with coordinates-based approaches. 2002.

[19] T. S. Eugene Ng and Hui Zhang. A network positioning system for the internet. In *USENIX Annual Technical Conference, General Track*, pages 141–154. USENIX, 2004.

[20] J. Postel. Internet Control Message Protocol. RFC 792 (Standard), September 1981. Updated by RFC 950.
`http://www.ietf.org/rfc/rfc792.txt`.

[21] Puneet Sharma, Zhichen Xu, Sujata Banerjee, and Sung-Ju Lee. Estimating network proximity and latency. Another version is "Netvigator: Scalable Network Proximity Estimation", 2005.

[22] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. (TR-00-010), October 2000.

[23] Sean Rhea. *OpenDHT: A Public DHT Service*. PhD thesis, University of California, Berkeley, August 2005.

[24] Antony Rowstron and Peter Druschel. Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems. November 2001.

[25] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions in Computer Systems*, 2(4):277–288, November 1984.

[26] John Kubiatowicz Sean Rhea, Byung-Gon Chun and Scott Shenker. Fixing the embarrassing slowness of opendht on planetlab. In *Proceedings of the Second Workshop on Real, Large Distributed Systems (WORLDS '05)*, 2005.

[27] Yuval Shavitt and Tomer Tankel. On the curvature of the internet and its usage for overlay construction and distance estimation. In *INFOCOM: The Conference on Computer Communications, joint conference of the IEEE Computer and Communications Societies*, 2004.

[28] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. 2001.

[29] Liying Tang and Mark Crovella. Virtual landmarks for the internet. In *Proceedings of the 2003 ACM SIGCOMM conference on Internet measurement (IMC-03)*, pages 143–152, New York, October 27–29 2003. ACM Press.

[30] Matt Welsh. SEDA: An Architecture for Highly Concurrent Server Applications. [Online; accessed 8-April-2007].
`http://www.eecs.harvard.edu/~mdw/proj/seda/`.

[31] Praveen Yalagandula and Mike Dahlin. A scalable distributed information management system. Technical Report CS-TR-03-47, The University of Texas at Austin, Department of Computer Sciences, November 1 2003. Tue, 19 Jun 107 19:21:48 GMT.

[32] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1), January 2004.

[33] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: an infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, University of California at Berkeley, April 2001.

Appendix A

Source Code

The full source code for this project is available at `http://ms800.montefiore.ulg.ac.be/~kunysz/bamboov-final.tar.gz`. The SHA-1 hash for the file is `4b38914737e67c81ec639e5d0700c6c32407d60a`. Here is the list of files that have been created or modified compared to the original Bamboo implementation:

```
test/prober.sh
test/prober.pl
doc/prober.cfg
lib/lib/perl5/site_perl/5.8.6/Net/OpenDHT.pm
src/bamboo/dht/Gateway.java
src/bamboo/prober/Prober.java
src/bamboo/prober/Probee.java
src/bamboo/prober/Makefile
src/bamboo/vis/FetchNodeInfoThread.java
src/bamboo/util/GuidTools.java
src/bamboo/Makefile
bin/prober.sh
bin/run-java
```

The client is `test/prober.sh` (which is a wrapper for `test/prober.pl`). Nodes are started with the `bin/prober.sh` script. Logs are written in the `log/` directory. The configuration file is generated by `bin/prober.sh` based on the `doc/prober.cfg` template.