

NetProber: a component for enhancing efficiency of overlay networks in P2P systems

Luc Onana Alima

Dep. of Microelectronics and Information Technology
Royal Institute of Technology
Kista, Sweden
onana@it.kth.se

Valentin Mesaros

Dep. of Computing Science and Engineering
Université catholique de Louvain
Louvain-la-Neuve, Belgium
valentin@info.ucl.ac.be

Peter Van Roy

Dep. of Computing Science and Engineering
Université catholique de Louvain
Louvain-la-Neuve, Belgium
pvr@info.ucl.ac.be

Seif Haridi

Dep. of Microelectronics and Information Technology
Royal Institute of Technology
Kista, Sweden
seif@it.kth.se

Abstract

The Peer-To-Peer (P2P) computing paradigm is an emerging paradigm that aims to overcome most of the main limitations of the traditional client/server architecture. In the P2P setting, individual computers communicate directly with each other in order to share information and resources without relying on any kind of centralized server. To achieve this full decentralization, an application-level (or overlay) network is constructed using, for example, TCP connections.

In most of the existing P2P systems, the overlay network is built in a manner that does not guarantee that the overlay network is efficient with respect to a given metric (e.g. latency, hop count and bandwidth). Hence, an overlay node can be very far away, in terms of a given metric, from its overlay neighbors. This can result in both, an inefficient routing at the overlay network and an ineffective use of the underlying IP network.

*In this paper, we first introduce a new measure, “goodness of overlay networks”, to quantify the quality of an overlay network for a given metric. Then, we propose **NetProber**, a simple, distributed and scalable component that can be combined with any connected overlay network in order to allow the latter to adapt, and to become “good” within a finite amount of time.*

1 Introduction

The client/server paradigm has established itself as the most popular paradigm in building distributed appli-

cations. The main characteristics of this paradigm is that the application is structured in two types of components: clients and servers. In a simple client/server architecture, we have a centralized server and one or more clients that are serviced by the server on demand. Although there are many applications structured this way, it should be mentioned that client/server applications have limited availability and scalability.

The availability of client server is limited, because the server is usually a single point of failure. In the case the server crashes, the clients might need to wait for a significant period of time in order to get access to the service. To overcome this limitation, client/server applications often make use of sophisticated mechanisms for fault-tolerance. Furthermore, these systems require sophisticated mechanisms for load-balancing.

The limitation on scalability of client/server applications is obvious when it comes to the Internet scale. As the number of clients increases, the performance decreases, because the server becomes a bottleneck.

The Peer-to-Peer (*P2P*) computing is an emerging paradigm that enables computers connected through the Internet to act as both clients and servers. In contrast to client/server architectures, in the P2P setting, there is no centralized server to which clients can connect to, but rather the system control is fully decentralized. The principle of these systems lies in the fact that the global properties of the system must emerge from *simple* and *local interactions* of its components (called peers).

Currently, P2P applications include file sharing applications [1, 2, 3], persistent storage services [12, 4] and

distributed lookup services [13, 6, 7].

A common characteristic of most of the existing P2P systems is that they build an application-level or *overlay network* along with its own routing mechanism. In an overlay network, each node can play the role of a client (i.e. generating requests), server (i.e. providing some services) and router (i.e. forwarding requests to their destination).

The structure of the overlay network and the routing used are of paramount importance for the application properties. Achieving good performance of the routing at the overlay network level and/or making better use of the underlying IP network surely depends on how “good” the structure of the overlay network is. At this point, it is worth questioning *what the goodness of an overlay network is*. Currently, we are investigating this question; the next subsection presents our preliminary results in defining this notion.

1.1 Goodness of overlay networks

In order to introduce a formal definition for the notion of goodness of an overlay network, we first give the following preliminary definition.

Definition 1.1 *Let $\mathcal{A} = (\mathcal{V}, \mathcal{E})$ be a connected overlay network where \mathcal{V} is the set of vertices and \mathcal{E} is the set of edges between vertices. Let $a_i \in \mathcal{A}$. Let d be some metric¹ (e.g. bandwidth, latency, number of hops, etc.), we say that a_i is a good node iff*

$$\begin{aligned} (\forall a_j \in \mathcal{V} : a_j \in \text{Neighbors}(a_i) : \\ (\nexists a_k \in \mathcal{V} : d(a_i, a_k) < d(a_i, a_j) \wedge \\ a_k \notin \text{Neighbors}(a_i))) \end{aligned} \quad (1)$$

In words, property (1) says that for any node a_j of \mathcal{A} such that a_j is a neighbor of a_i , there does not exist any node a_k such that $d(a_i, a_k)$ is strictly less than $d(a_i, a_j)$ and a_k is not a neighbor of a_i .

With the above definition, we can now introduce our formal definition of the goodness of an overlay network given a metric d .

Definition 1.2 *Let $\mathcal{A} = (\mathcal{V}, \mathcal{E})$ be a connected overlay network. Let d be a metric. We call goodness of \mathcal{A} for d , and we write, $g(\mathcal{A}, d)$, the ratio $\frac{G}{|\mathcal{V}|}$ where G is the number of nodes of \mathcal{A} that are “good” nodes; $|\mathcal{V}|$ denotes the cardinality of \mathcal{V} .*

As our investigation of the goodness of overlay network is a work in progress, in this paper, we shall use this notion intuitively. However, it is worth mentioning that

¹A metric is merely a function that takes two overlay nodes as arguments and returns a real value. We assume that this function is symmetric, i.e. $d(a_i, a_j) = d(a_j, a_i)$.

having a formal definition of the notion of goodness was a key in deriving the algorithm of the **NetProber**.

1.2 Problem statement

In most of the existing P2P systems, the overlay nodes join the network in a way that *does not* guarantee that the resulting network is *good* (in the sense of the aforementioned goodness). In most of the existing P2P systems [3, 1, 14, 7, 13], a new node joining the overlay network, connects itself to a small subset of known nodes already in the system, no matter how far, in terms of a given metric, the existing nodes reside. This might lead, for example using the *hop count* metric, to an overlay network in which, a node a_1 at UCL (Belgium) has its neighbors in USA and hence an overlay message from node a_1 to a nearby node a_2 , at Amsterdam, will pass through distant nodes in USA in order to reach its destination. This, evidently is an undesirable situation as it could lead to a high latency and/or an inefficient use of the underlying IP network.

The problem we consider in this paper hereupon can be stated as follows:

Given a metric d and a connected overlay network \mathcal{A} where each node obtains its neighbors randomly, how could we let, in a distributed and scalable manner, \mathcal{A} adjust itself such that within a finite time, \mathcal{A} becomes good for the metric d ?

In the sequel of this paper, we refer to this problem as the *mismatching problem*.

1.3 Contribution

The contribution of this paper is twofold. First, we introduce a formal characterization of the the goodness of an overlay network \mathcal{A} given a metric d . Second, we provide a simple, distributed, and scalable component called **NetProber** that can be combined with any connected overlay network in order to let the network become “good” with respect to a given QoS parameter, within a finite time.

An interesting property of the proposed component is that the component requires only very few changes of any overly network that intends to use it.

1.4 Related work

The mismatching problem is considered to be a serious problem in recent P2P designs [11, 13, 9, 10]. Although this has been pointed out by several authors, a formal characterization has been missing, hence making

the problem difficult to understand or solve in a rigorous manner.

When we chose the hop count as metric, it became obvious that a good overlay network is the one which is close to the underlying IP network. This results in the problem mentioned by M. Ripeanu in [11]. In [11], the authors suggested two ideas for solving the mismatching problem in order to improve the performance of Gnutella and similarly built systems. The first idea suggested by Ripeanu in [11] consists of using an agent that constantly monitors the overlay network and intervenes by asking servers (i.e. overlay nodes) to drop or add connections as necessary to keep the overlay network efficient. However, there is no indication of how this idea could be implemented. The second idea in [11] consists of using less expensive routing mechanisms and the abstraction of group communications. However, no mention is made about how to implement this.

In [5], the authors proposed an heuristic that can be used to construct overlay networks with low diameter in enterprise P2P applications. The proposed solution bears some form of centralization and is not suitable for large scale peer-to-peer systems, which are the kind of systems we target.

Using latency as the metric, Ratnasamy et al. [8] propose a scheme called *binning* for constructing CAN (Content Adressable Network) topologies that are congruent with the underlying IP network. The idea builds on the use of a set of well known machines, for example DNS root name servers that act as landmarks in the Internet. Each overlay node (or CAN node, in the terminology of the authors) measures its round-trip-time to each of the landmarks. Using these measurements, each overlay node sorts the landmarks in order of increasing round-trip-time. Each ordering of landmarks corresponds to a portion of the d -dimensional coordinate space on which CAN is based. Thus, in order for a node to join the CAN, it must first bin itself. That is, it must first order the landmarks to determine the portion of the virtual coordinate space through which it must enter the CAN. Assuming that nodes that are close to each other at the underlying physical network will have the same ordering of landmarks, it follows that they will be neighbors of each other at the overlay network level. We believe that this idea might be very difficult to implement, because, for example, of the difficulties that are involved in the partitioning of the virtual coordinate space. Furthermore, this scheme introduces some form of rigid hierarchy, which is not desired in pure P2P systems.

1.5 System model

1.5.1 Distributed system

We consider a distributed system as a set of nodes (or processes) linked together through a communication network.

Nodes communicate by message passing. Each message has the form $(s : d : \text{TYPE}(param))$ where s identifies the sender, d identifies the destination, TYPE denotes the type of the message and $param$ denotes the parameters that depend on the type of the message.

A process s sends a message $(s : d : \text{TYPE}(param))$ to another process d by executing the statement **send** $(s : d : \text{TYPE}(param))$. The effect of executing this statement is to add the message $(s : d : \text{TYPE}(param))$ to the network to which the destination process belongs. It is the responsibility of the network to deliver (that is, to make available) the message $(s : d : \text{TYPE}(param))$ to the destination process p .

Each message added to a communication network remains there until the destination process removes it. How and when a message is removed from a communication network is explained in section 1.5.4.

1.5.2 Communication networks

Topology. In what follows, we assume that communication networks are connected, i.e. there is a path from each node to any other node in the system.

Network behavior. We assume communication networks that satisfy the following properties:

- (i) *Asynchronism.* The time it takes a communication network to forward a message to its destination is arbitrary but finite.
- (ii) *Reliability.* Every message injected to a communication network is eventually delivered to its destination as long this later remains connected to the communication network.

Network states. A state of a communication network consists of the messages currently in it.

1.5.3 Nodes (or Processes)

Each node (or process,) executes an algorithm that consists of a finite set of variables and a finite set of rules. Each variable has a predefined non-empty domain.

Node (or process) states. A *state* of a node n is defined by a value for each variable of n .

Algorithm specification. Each algorithm executed by a node consists of a set of rules. Each rule has the form $\frac{\text{CONDITION}}{\text{ACTION}}$. The **CONDITION** of a rule is a boolean expression over the variables of the node and/or

at most one *receive condition*. A receive condition is of the form **receive**($s : d : \text{TYPE}(param)$). The evaluation of **receive**($s : d : \text{TYPE}(param)$) by node d returns *true* if a message ($s : d : \text{TYPE}(param)$) is available for node d in its communication network; otherwise it returns *false*.

The ACTION part of a rule consists of a sequence of statements, which is executed atomically.

1.5.4 System configurations and computations

System configurations. A *configuration* of a distributed system consists of a state for each node of the system and a state for its communication network.

A rule R_p of a process p is said to be *enabled* at a configuration β_i of the system if and only if the CONDITION of R_p evaluates to *true* at β_i . The action of a rule can be executed at a configuration β_i only if that rule is enabled at β_i .

System computations. A *computation* β of a distributed system is a *nonempty*, *fair* and *maximal* sequence of configurations $\beta_0, \beta_1, \beta_2, \dots$ such that for each $i \geq 0$, β_{i+1} is reached from β_i by an atomic execution of an enabled rule. If in β_i more than one rule is enabled, one is selected non-deterministically. By *maximal*, we mean that the sequence is either finite or infinite. When the sequence is finite, the last configuration of the sequence is a fixed point, i.e. the execution of any rule from that configuration leaves the system in the same configuration. By *fair* we mean that any rule that is continuously enabled is eventually executed. That is, we assume *weak fairness*.

How messages are removed from a communication network. Executing a rule with a receive condition in the CONDITION part removes the corresponding message from the communication network and performs the action part of the rule in an atomic manner.

1.5.5 System dynamism

We assume that nodes can join and leave the system at any time. However, when a node leaves the system, it does not perform any erroneous step.

The remainder of this paper is organized as follows: Section 2 presents the architecture and an overview of a **NetProber**-based system. In Section 3, we give the algorithm of the **NetProber** system. Finally, we conclude in Section 4.

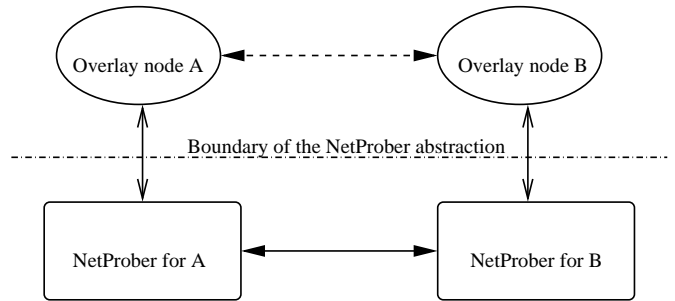


Figure 1: Structure of a **NetProber**-based overlay network. Messages between overlay nodes are forwarded using the overlay routing. Messages between **NetProber** elements are routed using the underlying IP network.

2 Architecture and overview of NetProber

This section describes **NetProber**. In subsection 2.1, we give a conceptual architecture of a **NetProber**-based overlay system. In subsection 2.2, we present an overview of how a **NetProber**-based overlay system works.

2.1 Architecture of a NetProber-based overlay system

Figure 1 shows the architecture of a **NetProber**-based overlay system. Each overlay node has an associated **Netprober** element. Hereupon, we use the following convention: for an overlay node a_i , we write np_i to denote the **NetProber** element associated with a_i .

We assume that each overlay node maintains a *Neighbors* table that contains an entry for each of its overlay neighbors. For two overlay neighbor nodes a_i and a_j , the *Neighbors* table of a_i is such that $Neighbors(a_j)$ is equal to the identifier of the **NetProber** element associated with a_j .

We also assume that the **NetProber** element, np_i , associated with the overlay node a_i , has access to the table *Neighbors* of a_i .

2.2 How NetProber works: an overview

In brief, a **NetProber**-based overlay system works as follows: when an overlay node a_i wishes to adjust its neighborhood with respect to one of its neighbors, say a_j , node a_i invokes its associated **NetProber** element. This invocation is done by a message of type **FINDBETTERNEIGHBOR**, which takes two parameters: the first parameter is the identifier, a_j of the neighbor that a_i

wishes to switch, and the second parameter is a *metric*, which the **NetProber** element uses in order to find a better neighbor (if any), with respect to a_j . We regard this metric as a quality of service (QoS) parameter.

To see the dynamics of the **NetProber** layer upon receiving the above request, let \mathcal{np}_i and \mathcal{np}_j be the **NetProber** elements associated respectively with a_i and a_j .

When \mathcal{np}_i receives `FINDBETTERNEIGHBOR(a_j, M)`, \mathcal{np}_i sends a message of type `NETPROBEREQUEST` to \mathcal{np}_j . This message is just a request that \mathcal{np}_i sends to \mathcal{np}_j in order to obtain the *Neighbors* table of a_j , which \mathcal{np}_j knows.

When \mathcal{np}_j receives `NETPROBEREQUEST` from \mathcal{np}_i , it responds by sending a message of type `NETPROBEREPLY` to \mathcal{np}_i . This response carries the *Neighbors* table of a_j . In addition, this message conveys relevant information that \mathcal{np}_i uses in order to determine a better neighbor (if any).

When \mathcal{np}_i receives `NETPROBEREPLY`, it retrieves the relevant information that will be used for the determination of the better candidate, if any. The relevant information depends on the metric used. For example, if the metric used is *latency*, the relevant information that \mathcal{np}_i retrieves from the received response will be an estimate of the round-trip-time. Another example is the case when the metric used is the *hop count*, thus the relevant information that \mathcal{np}_i retrieves from the `NETPROBEREPLY` message is the number of hops.

Now, since \mathcal{np}_i has the *Neighbors* table of a_j , node a_i probes each neighbor a_k ($a_k \neq a_i$) of a_j by sending a `NETPROBEREQUEST` message. Hence, when \mathcal{np}_i has received a `NETPROBEREPLY` (and retrieved relevant information) from each **NetProber** associated with each neighbor a_k ($a_k \neq a_i$) of a_j , \mathcal{np}_i computes the “better” candidate that is returned to a_i in a message of type `SUGGESTION`, which gives just an indication to a_i . It is the responsibility of a_i to decide whether or not the connection with a_j is to be put off.

The better candidate is a node a_k that satisfies the following three conditions:

- (i) a_k is a neighbor of a_j ;
- (ii) the distance (in terms of the given metric) between a_i and a_k is smaller than that between a_i and a_j ;
- (iii) a_k is not a neighbor of a_i .

If such a node a_k does not exist, then the better candidate is a_j itself.

Switching a connection. We have said that the **NetProber** only suggests “better” candidates to the overlay layer. The important issue to investigate now is the way a disconnection actually takes place. An uncoordinated disconnection might easily lead to system

configuration, where some nodes are isolated. To illustrate the idea, consider a system of three nodes, say a_i , a_j , and a_k such that:

- a_i is neighbor of a_j ;
- a_j is neighbor of a_k ;
- a_i and a_k are not neighbors and
- $d(a_i, a_k) < d(a_i, a_j)$.

Now, if by running the **NetProber** algorithm, nodes a_i and a_k discover by the same time that they should each switch their connection with a_j , node a_j might become isolated.

To overcome this problem, we propose that the disconnection be done in an “agreed transaction”, i.e. when a node a_i decides to change its neighbor a_j by a better candidate neighbor a_k , node a_i asks a_j if it accepts this change. Node a_j will accept the change only if it is still connected to a_k . Hence, if a request from a_k arrives at a_j asking if a_j accepts to let a_k change its connection from a_j in order to connect to a_i , node a_j will reject this request from a_k , because a_j is no longer connected to a_i . In this way, we maintain the connectivity of the overlay network.

In general, if more than one node want to disconnect themselves from a_j in order to get connected to the same candidate a_k , only one node will succeed due to the “agreed transactional disconnection” technique that we use.

3 Algorithm

This section presents a formal description of the **NetProber** algorithm. We begin in subsection 3.1, with a presentation of the key data structures used. Then, in subsection 3.2 we describe the behavior of an initiating **NetProber** element followed by a specification of the behavior of a non-initiating **NetProber** element in subsection 3.3.

3.1 Key data structures

Let consider that we have the following sets:

- *OverlayId*: the set of all possible identifiers of overlay nodes;
- *NetProberId*: the set of all **NetProber** identifiers.

We assume that each **NetProber** element has access to or maintains the following components:

- *Neighbors* : $OverlayId \rightarrow NetProberId$

This mapping is provided by the overlay layer.

- $DistanceTable : OverlayId \rightarrow (OverlayId \rightarrow \mathbb{R})$

Initially, this mapping is empty.

- $WaitSet : OverlayId \rightarrow {}_2NetProbeId$

Initially, this mapping is empty.

- $WorkingSet$: set of identifiers of overlay nodes.

Initially, this set is empty.

- $OtherNeighbors : OverlayId \rightarrow {}_2OverlayId$

Initially, this mapping is empty.

In the sequel, we use the following convention. Given a set P of pairs, we denote by $P_{|1}$ (respectively $P_{|2}$) the set obtained by taking the first (respectively second) component of each pair in P .

Another convention that we use in this paper is the following: Let m be a mapping from a set A to another set B . For $x \in A$, we write $m(x) = \perp$ to mean that the mapping m is undefined for x . We write $m(x) := \perp$ to mean that we remove the entry x from the mapping m .

3.2 Behavior of an initiating NetProber element

A **NetProber** element, np_i , initiates the search for a “better” neighbor upon request from the application (or overlay) layer. To issue a request, the overlay node, a_i associated to np_i sends a **FINDBETTERNEIGHBOR** message to np_i . This message has two parameters, the first one is the identifier, a_j , of the neighbor that a_i wishes to switch. The second parameter is the metric to be used for the selection of the better candidate.

The behavior of an initiating **NetProbe** element is described by rules (2) and (3).

$$\frac{\text{receive}(a_i : np_i : \text{FINDBETTERNEIGHBOR}(a_j, M)) \wedge a_j \notin WorkingTable}{\begin{aligned} np_j &:= Neighbors(a_j) \\ WorkingSet &:= WorkingSet \cup \{a_j\}; \\ \text{send}(np_i : np_j : \text{NETPROBEREQUEST}(0, a_j, np_i)) \end{aligned}} \quad (2)$$

The next rule describes the reaction of the initiating node upon receiving a **NETPROBEREPLY** message. The behavior of the initiating node depends upon the level of the reply, which is either 0 or 1. A **NETPROBEREPLY** of level 0 is sent by the **NetProber** element associated with the node, whose neighborhood is being probed for a better candidate. A **NETPROBEREPLY** of level 1 is sent by a neighbor of a node that we want to switch.

$$\frac{a_j \in WorkingSet \wedge \text{receive}(np_k : np_i : \text{NETPROBEREPLY}(L, a_j, SetOfPairs))}{d := \text{getDistance}(\text{NETPROBEREPLY}, M)} \quad (3)$$

$$\begin{aligned} &\text{if } L = 0 \text{ then} \\ &\quad (DistanceTable(a_j))(a_j) := d \\ &\quad \text{for every } (a_\tau, np_\tau) \in SetOfPairs \text{ do} \\ &\quad \quad \text{send}(np_i : np_\tau : \text{NETPROBEREQUEST}(1, a)) \\ &\quad \quad WaitSet(a_j) := SetOfPairs_{|2} \\ &\quad \quad OtherNeighbors(a_j) := SetOfPairs_{|1} \\ &\text{else} \\ &\quad \text{Let } \{a_k\} = SetOfPairs_{|1}; \\ &\quad (DistanceTable(a_j))(a_k) := d; \\ &\quad WaitSet(a_j) := WaitSet(a_j) \setminus \{np_k\} \\ &\quad \text{if } WaitSet(a_j) = \emptyset \text{ then} \\ &\quad \quad \Gamma := \{a_j\} \cup \{a_\tau : a_\tau \in OtherNeighbors(a_j) \wedge \\ &\quad \quad \quad Neighbors(a_\tau) = \perp\} \\ &\quad \quad m := \min\{(DistanceTable(a_j))(a_\tau) : a_\tau \in \Gamma\} \\ &\quad \quad cn := \text{Selection}(\Gamma, m) \\ &\quad \quad \text{send}(np_i : a_i : \text{SUGGESTION}(a_j, cn)) \\ &\quad \quad WorkingSet := WorkingSet \setminus \{a_j\} \\ &\quad \quad OtherNeighbors(a_j) := \perp \\ &\quad \text{fi} \\ &\text{fi} \end{aligned}$$

Rule (3) deserves some comments. In the action part of this rule, we use some generic functions:

- **getDistance**: this function will return the distance according to the metric used as QoS parameter. It might be the number of hops, the latency, etc..

In the case the metric specified is the number of hops, this function will retrieve the **IP_TTL** value from each message **NETPROBEREPLY** given as argument.

In the case the metric used is the latency, this function will return an estimate of the round-trip-time and requires that we record the (local) time when a **NETPROBEREQUEST** is sent to some other **NetProber** element.

- **Selection**: this function is used to select one of the nodes with minimum distance from the set Γ .

3.3 Behavior of a non-initiating NetProber element

Rule (4) specifies the reaction of a **NetProber** element np_j upon receiving a **NETPROBEREQUEST** message from another **NetProber** element np_i . The reaction of np_j depends upon the level of the **NETPROBEREQUEST**. Level 0 indicates that the receiving netprober element must send the *Neighbors* table of its associated overlay node.

A level 1 indicates that the receiving node needs not send the *Neighbors* table of its associated overlay node.

```

receive( $\mathcal{N}_i : \mathcal{N}_j : \text{NETPROBEREQUEST}(L, a)$ )


---


if  $L = 0$  then
  SetOfPairs := Neighbors \  $\{(a_i, \mathcal{N}_i)\}$ 
  send( $\mathcal{N}_j : \mathcal{N}_i : \text{NETPROBEREPLY}(0, a, \text{SetOfPairs})$ )
else
  SetOfPairs :=  $\{(a_j, \mathcal{N}_j)\}$ 
  send( $\mathcal{N}_j : \mathcal{N}_i : \text{NETPROBEREPLY}(1, a, \text{SetOfPairs})$ )
fi

```

(4)

4 Conclusion

In this paper, we introduced the notion of “goodness”, which can be used to measure the quality of an overlay network given a certain metric. Using this formal characterization, we presented **NetProber**, a simple, fully distributed and scalable algorithm that allows any overlay network that uses it to adapt towards the “goodness” measure.

The QoS parameter that we mainly used in our preliminary simulations was the hop count. In most of the currently obtained results, we observe that the overlay network progressively becomes close to the underlying IP network. These results are encouraging, thus we plan to perform more simulations for an effective evaluation of the **NetProber** algorithm.

Acknowledgements

We would like to thank all the members of the Tuesday meeting at the UCL/FSA/INGI for their valuable comments. We also thank Iyad Al Khatib for his help.

References

- [1] The Gnutella Protocol Specification v0.4. <http://www.clip2.com/GnutellaProtocol04.pdf>.
- [2] Jxta v1.0 protocols specification. <http://www.jxta.org>, June 2001.
- [3] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of ICSI Workshop on Design Issues in Anonymity and Unobservability*, July 2000.
- [4] John Kubiawicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An

architecture for global-scale persistent storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

- [5] Gopal Pandurangan, Prabhakar Raghavan, and Eli Upfal. Building low-diameter p2p networks. In *42th IEEE Symp. on Foundations of Computer Science*, 2001.
- [6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.
- [7] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of ACM SIGCOMM*, August 2001.
- [8] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *Proc. of IEEE INFOCOM*, June 2002. To be published.
- [9] S. Ratnasamy, S. Shenker, and I. Stoica. Routing algorithms for dhts: Some open questions. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems*, Cambridge, MA, USA, June 2002.
- [10] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), January/February 2002.
- [11] Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proceedings of International Conference on Peer-to-peer Computing*, Linkoping, Sweden, August 2001.
- [12] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Middleware*, November 2001.
- [13] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. Technical Report TR-819, MIT, March 2001.
- [14] B. Zhao, J. Kubiawicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, U. C. Berkeley Technical Report, April 2000.