# NetProber: a Component for Enhancing Efficiency of Overlay Networks in P2P Systems*

Luc Onana Alima
*Dept. of Microelectronics and Information Technology*
*Royal Institute of Technology*
*Kista, Sweden*
onana@it.kth.se

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

Peter Van Roy
*Dept. of Computing Science and Engineering*
*Université catholique de Louvain*
*Louvain-la-Neuve, Belgium*
pvr@info.ucl.ac.be

Seif Haridi
*Dept. 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.*

*This research is partially funded by the PIRATES project of the Walloon Region (Belgium) and the PEPITO project of the European Union.

## 1. Introduction

The client/server paradigm has established itself as the most popular paradigm in building distributed applications. 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 the client/server architecture 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 and 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 some preliminary definitions.

Throughout this paper, we represent a network by an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where each node is represented by a vertex $u \in \mathcal{V}$, and an edge $(u, v) \in \mathcal{E}$ represents a bidirectional communication link between $u$ and $v$. A node $u \in \mathcal{E}$ is said to be a *neighbor* of $v \in \mathcal{V}$ if there is an edge $(u, v)$ in $\mathcal{G}$. We represent the set of neighbors of a node $v$ by $N(v)$ and we use $n_v$ to denote the size of $N(v)$.

In the rest of this paper, we assume *connected networks*, i.e., there is a path from each node to any other node of the network. We call *path* of $\mathcal{G}$, a sequence of nodes $v_0, v_1, \cdots, v_k$ in which no node appears more than once and for $1 \leq i \leq k$, $(v_{i-1}, v_i) \in \mathcal{E}$. Given a path $p$, we write $nodes(p)$ to denote all the nodes that appear in $p$. We represent a path which first and second nodes are respectively $u$ and $v$ by $p_{uv}$ and we use $\Gamma_{uv}$ to denote all such paths. We shall write $Nodes(\Gamma_{uv})$ for $\cup_{p \in \Gamma_{uv}} nodes(p)$

**Definition 1.1** *Let $\mathcal{A} = (\mathcal{V}, \mathcal{E})$ be an overlay network. Let $d$ be some metric*[1] *(e.g. bandwidth, latency, number of IP hops) over $\mathcal{A}$. An edge $(u, v) \in \mathcal{E}$ is said to be a good link at $u$, with respect to $d$ if and only if there is no path $p_{uv}$ that contains a node $w \notin N(u)$ such that $d(u, w) < d(u, v)$. A link $(u, v)$ is said to be bad at $u$, with respect to $d$ iff $(u, v)$ is not a good link at $u$.*

[1] We assume that $d$ is symetric and satisfies the triangle inequality.

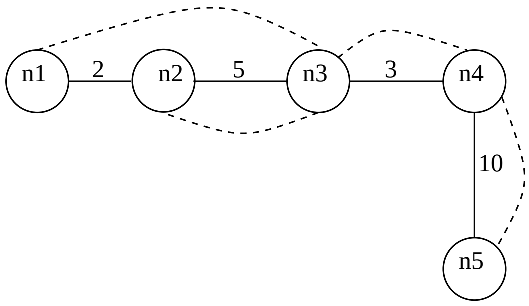


Figure 1: Good/bad links illustrated. Solid lines represent the physical network. The dashed lines represent the overlay network. The numbers between two nodes represent the distance between the two nodes.

Notice that a link $(u, v)$ can be good at node $u$ while being bad at node $v$. Figure 1 illustrates the idea of good (respectively bad) links. In this figure, each circle represents a machine hosting an overlay node, solid lines represent a physical network, and the dashed lines represent the overlay network. Each number between two nodes represents the distance between the two nodes. According to Definition 1.1, the link $(n_1, n_3)$, for example, is a good link at $n_3$. But the same link is a bad link at node $n_1$; the link $(n_4, n_5)$ is a good link both at $n_4$ and at $n_5$.

**Definition 1.2** *Let $\mathcal{A} = (\mathcal{V}, \mathcal{E})$ be an overlay network. Let $d$ be some metric over $\mathcal{A}$. Let $u \in \mathcal{V}$ and $v \in N(u)$. We call the closest node to $u$ via $v$, denoted $closest(u, v)$, a node $w \neq u$ such that*

$$d(u, w) = \min_{x \in Nodes(\Gamma_{uv}) \wedge x \notin N(u)} \{d(u, x)\}$$

*If there is no node $w$ that satisfies the above condition, then $closest(u, v) = v$.*

We can now introduce the notion of goodness of an overlay node.

**Definition 1.3** *Let $\mathcal{A} = (\mathcal{V}, \mathcal{E})$ be an overlay network. Let $d$ be a metric over $\mathcal{A}$. Let $u \in \mathcal{V}$. The goodness of $u$ with respect to $d$ is*

$$gness(u, d) = \frac{\sum_{v \in N(u)} d(u, closest(u, v))}{\sum_{x \in N(u)} d(u, x)}$$

From Definition 1.2 and Definition 1.3, the following lemma is derived.

**Lemma 1.1** *Let $\mathcal{A} = (\mathcal{V}, \mathcal{E})$ be an overlay network. Let $d$ be a metric over $\mathcal{A}$. For every $u \in \mathcal{V}$, we have*

$$0 < gness(u, d) \leq 1$$

From Definition 1.3 and Definition 1.1, the following lemma follows.

**Lemma 1.2** *Let $\mathcal{A} = (\mathcal{V}, \mathcal{E})$ be an overlay network. Let $d$ be a metric over $\mathcal{A}$. Let $u \in \mathcal{V}$. If all links $(u, v)$, $v \in N(u)$, are good at $u$ with respect to $d$, then $gness(u, d) = 1$*

We now give our formal definition of the goodness of an overlay network given a metric $d$.

**Definition 1.4** *Let $\mathcal{A} = (\mathcal{V}, \mathcal{E})$ be an overlay network. Let $d$ be a metric over $\mathcal{A}$. The goodness of $\mathcal{A}$ with respect to $d$ is*

$$Gness(\mathcal{A}, d) = \frac{\sum_{u \in \mathcal{V}} gness(u, d)}{\mid \mathcal{V} \mid}$$

From Lemma 1.1 and Definition 1.4, the following Lemma follows.

**Lemma 1.3** *Let $\mathcal{A} = (\mathcal{V}, \mathcal{E})$ be an overlay network. Let $d$ be a metric over $\mathcal{A}$. We have that*

$$0 < Gness(\mathcal{A}, d) \leq 1$$

We say that an overlay network is good with respect to a given metric $d$, when its godness tends to one.

Our investigation of the goodness of overlay networks is a work in progress. However, it is worth mentioning that 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 an overlay network $\mathcal{A} = (\mathcal{V}, \mathcal{E})$, where each node $u \in \mathcal{V}$ obtains its neighbors randomly, and a metric $d$ over $\mathcal{A}$, how could we let, in a distributed and scalable manner, $\mathcal{A}$ adjust itself such that within a finite time, $\mathcal{A}$ becomes good with respect to 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 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 randomly built overlay network in order to let the network become good with respect to a given metric $d$, within a finite time.

An interesting property of the proposed component is that the component requires only very few changes of any overlay 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 one which is close to the underlying IP network. This results in the problem mentioned by M. Ripeanu in [11], where the author 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 servents (i.e., overlay nodes) to drop or add connections as necessary to keep the overlay network efficient. 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 these ideas.

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 communication network. Each message added to the 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 taken by the communication network to forward a message to its destination is arbitrary but finite.

(ii) *Reliability.* Every message injected into a communication network is eventually delivered to its destination provided that the latter remains connected to the communication network.

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

### 1.5.3. Processes

Each process (or node) 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 $R$ of a process $p$ is said to be *enabled* at a configuration $\beta_i$ of the system if and only if the CONDITION of $R_p$ evaluates to *true* at $\beta_i$. The action of a rule can be executed at a configuration $\beta_i$ only if that rule is enabled at $\beta_i$.

**System computations.** A *computation* $\beta$ of a distributed system is a *nonempty, fair* and *maximal* sequence of configurations $\beta_0, \beta_1, \beta_2, \cdots$ such that for each $i \geq 0$, $\beta_{i+1}$ is obtained from $\beta_i$ by an atomic execution of an enabled rule. If in $\beta_i$ 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.

### 1.6. Organization of the paper

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

## 2. NetProber design

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 informal description of how a **NetProber**-based overlay system works.

### 2.1. Architecture of a NetProber-based overlay system

Figure 2 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 $u$, we write $np_u$ to denote the **NetProber** element associated with $u$.

The bidirectional arrows between an overlay node and its associated **NetProber** element represents the fact both components interact. An overlay node $u$ invokes its **NetProber** element $np_u$ when it wishes to switch a neighbor $v$. In response to such an invocation, $np_u$ sends a suggestion to $u$. This suggestion contains the identifier of an overlay node which is closer (for a given metric) to $u$ than $v$.

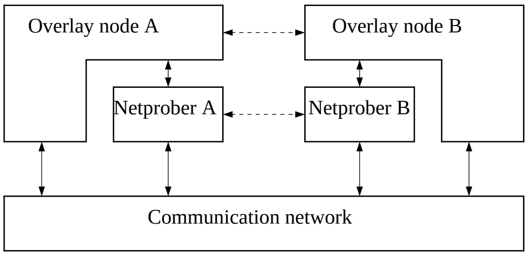


Figure 2: Structure of a NetProber-based overlay network. Messages between overlay nodes are forwarded using the overlay routing mechanism. Messages between **NetProber** elements are routed using the underlying IP network.

### 2.2. Informal description of the NetProber algorithm

In brief, a **NetProber**-based overlay system works as follows: when an overlay node $u$ wishes to adjust its neighborhood with respect to one of its neighbors, say $v$, node $u$ invokes its **NetProber** element, $np_u$. This invocation is done by a message of type FINDBETTERNEIGHBOR, which takes two parameters: the first parameter is the identifier, $v$ of the neighbor that $u$ 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 $v$. We regard this metric as a quality of service (QoS) parameter.

When $np_u$ receives FINDBETTERNEIGHBOR$(v, d)$, $np_u$ sends a message of type NETPROBEREQUEST$(L, v)$ to $np_v$, where the first parameter, $L$, is called the level of the request and is set to 0 when $np_u$ sends the request to $v$. Setting the level of the request to 0 is an indication that $np_u$ sends to $np_v$ asking for the set of pairs $P_v := \{(w, np_w) \mid (w \in N(v) \ \wedge \ (w \neq u))\}$. The second parameter is simply the identifier of $v$, which is merely used for matching responds to requests.

When $np_v$ receives a NETPROBEREQUEST$(L, v)$ with $L = 0$ from $np_u$, $np_v$ responds by sending a message of type NETPROBEREPLY$(L, v, P_v)$ to $np_u$. This response carries the set of pairs $P_v := \{(w, np_w) \mid (w \in N(v)) \ \wedge \ (w \neq u)\}$ and $L$ is the level of the reply. In addition, this message conveys relevant information that $np_u$ uses in order to determine a better neighbor (if any) for $u$, with respect to $v$.

When $np_u$ receives NETPROBEREPLY from $np_v$, $np_u$

mainly does two things. First, 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 $np_u$ 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 *IP hop count*, thus the relevant information that $np_u$ retrieves from the NetProbeReply message is the number of IP hops. Second, $np_u$ sends a NetProbeRequest message to each $np_w$ such that $(w, np_w) \in P_v$. The first parameter of the NetProbeRequest message that $np_u$ sends to each $np_w$ is of level 1.

Each **NetProber** element $np_w$ associated with $w \in N(v)$ that receives a NetProbeRequest of level 1 from $np_u$ responds by sending a NetProbeReply message that carries the singleton set $\{(w, np_w)\}$.

Hence, when $np_u$ has received a NetProbeReply (and retrieved relevant information) from each $np_w$ such that $(w, np_w) \in P_v := \{(w, np_w) \mid (w \in N(v)) \wedge (w \neq u)\}$, $np_u$ computes the "better" candidate that is returned to $u$ in a message of type Suggestion, which gives just an indication to $u$. It is the responsibility of $u$ to decide whether or not the connection with $v$ is to be put off.

The better candidate is an overlay node $w$ that satisfies the following three conditions:

(i) $w$ is a neighbor of $v$;

(ii) the distance (in terms of the given metric) between $u$ an $w$ is smaller than that between $u$ and $v$;

(iii) $w$ is not a neighbor of $u$.

If such a node $w$ does not exist, then the better candidate is $v$ 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 the system of three nodes given in Figure 3. In this figure, assume that $d(u, w) < d(u, v)$ and $d(u, w) < d(w, v)$. Now, if by running the **NetProber** algorithm, nodes $u$ and $w$ discover by the same time that they should each switch their connection with $v$, node $v$ might become isolated. To overcome this problem, we propose that the disconnection be done in an "agreed transaction", i.e. when a node $u$ decides to change its neighbor $v$ by a better

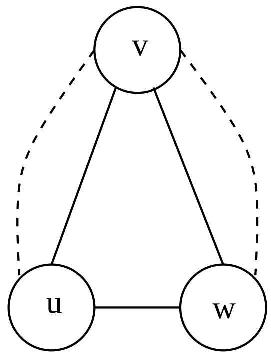


Figure 3: Necessity of coordinated disconnection illustrated. Solid lines represent the physical network. Dashed lines represent the overlay network.

candidate neighbor $w$, node $u$ asks $v$ if it accepts this change. Node $v$ will accept the change only if it is still connected to $w$. Hence, if a request from $w$ arrives at $v$ asking if $v$ accepts to let $w$ change its connection from $v$ in order to connect to $u$, node $v$ will reject this request from $w$, because $v$ is no longer connected to $u$. In this way, we maintain the connectivity of the overlay network.

## 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 give the rules specifying the behavior of a **NetProber** element upon receiving each of the messages involved in a **NetProber**-based system.

### 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 an arbitrary **NetProber** element $np_u$ has access to or maintains the following components:

- *Neighbors* : *OverlayId* $\rightarrow$ *NetProberId*

  This mapping is provided by the overlay layer.

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

  Let $d$ be the metric used. At a **NetProber** element $np_u$, this variable serves to store for an overlay node $v \in N(u)$, the set $\{(v, d(u,v)\} \cup \{(w, d(u,w)) \mid w \in N(u) \wedge w \neq u\}$. Initially, this mapping is empty.

- $WaitSet : OverlayId \rightarrow 2^{NetProbeId}$

  This variable serves to store the identifiers of the **NetProber** elements from which a NetProbeReply message is expected. Initially, this mapping is empty.

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

  This variable serves to store the identifiers of the overlay neighbors of $u$ that $u$ is currently trying to replace. Initially, this set is empty.

- $OtherNeighbors : OverlayId \rightarrow 2^{OverlayId}$

  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. Formal description of the NetProber algorithm

The **NetProber** algorithm has mainly three rules. Rule (1) describes the reaction of a **NetProber** element $np_u$ upon receiving a FindBetterNeighbor message from its overlay node $u$. In this rule, $d$ denotes the metric used to determine better neighbors.

**receive**($u : np_u$ : FindBetterNeighbor$(v, d)$) $\wedge$ $v \notin WorkingSet$

---

$np_v := Neighbors(v)$
$WorkingSet := WorkingSet \cup \{v\}$
**send**($np_u : np_v$ : NetProbeRequest$(0, v)$)

(1)

Rule (2) specifies the reaction of a **NetProber** element $np_v$ upon receiving a NetProbeRequest message from another **NetProber** element $np_u$. The reaction of $np_v$ depends upon the level of the NetProbeRequest.

**receive**($np_u : np_v$ : NetProbeRequest$(L, v)$)

---

```
if L = 0 then
    SetOfPairs := Neighbors \ {(u, np_u)}
    send(np_v : np_u : NetProbeReply(0, v, SetOfPairs))
else
    SetOfPairs := {(v, np_v)}
    send(np_v : np_u : NetProbeReply(1, v, SetOfPairs))
fi
```

(2)

The third rule describes the reaction of a **NetProber** element $np_u$ upon receiving a NetProbeReply message from another **NetProber** element $np_w$.

**receive**($np_w : np_u$ : NetProbeReply$(L, v, SetOfPairs)$) $\wedge$ $v \in WorkingSet$

---

```
α := getDistance(NetProbeReply, d)
if L = 0 then
    (DistanceTable(v))(v) := α
    for every (a, np_a) ∈ SetOfPairs do
        send(np_u : np_a : NetProbeRequest(1, v))
    WaitSet(v) := SetOfPairs_|2
    OtherNeighbors(v) := SetOfPairs_|1
else
    Let {x} = SetOfPairs_|1;
    (DistanceTable(v))(x) := α;
    WaitSet(v) := WaitSet(v) \ {np_x}
    if WaitSet(v) = ∅ then
        Γ := {v} ∪ {s : s ∈ OtherNeighbors(v) ∧
                        Neighbors(s) = ⊥}
        m := min{(DistanceTable(v))(y) : y ∈ Γ}
        cn := selection(Γ, m)
        send(np_u : u : Suggestion(v, cn))
        WorkingSet := WorkingSet \ {v}
        OtherNeighbors(v) := ⊥
    fi
fi
```

(3)

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 $\Gamma$.

**Remark** *The algorithm presented here explores only the neighbors of a neighbor. The change required to enable the probing of nodes at a certain depth is obvious and is ommitted here for simplicity.*

## 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 progessively 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 and Thomas Sjöland, and Sameh El-Ansary from IMIT/LECS/DCS for their contribution.


## 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 Kubiatowicz, 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. Kubiatowicz, 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.