ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Semester Thesis

Havelaar – Implementation Of A Peer-to-Peer Reputation System

Dorian Kind
dorian@student.ethz.ch

Prof. Dr. Roger Wattenhofer
Distributed Computing Group

Advisors: Dominik Grolimund, Luzius Meisser and Stefan Schmid

Dept. of Computer Science
Swiss Federal Institute of Technology (ETH) Zürich
Summer 2006

Contents

Havelaar – Implementation Of A Peer-to-Peer Reputation System

Dorian Kind

September 20, 2006

Abstract

In this thesis, we discuss the implementation of a *reputation system* called *Havelaar*. The aim of *Havelaar* is to provide incentives for contributing in a peer-to-peer network, namely the new peer-to-peer overlay network *Kangoo*, which is currently being developed at ETH Zürich. First some general properties and problems of designing such a reputation system are shown, and we discuss related work in this area. Then we take a look at *Havelaar* itself and – more specifically – at its actual implementation into *Kangoo*.

1 Introduction

> *There are two modes of establishing our reputation: to be praised by honest men, and to be abused by rogues. It is best, however, to secure the former, because it will invariably be accompanied by the latter.*
>
> —- Charles Caleb Colton (1780–1832), English cleric and writer

When the first large-scale and publicly visible peer-to-peer networks began to surface (one of the earliest and most notable example being the now infamous and defunct Napster network), little thought was given about providing a mechanism for rewarding contribution (i.e. uploading content to the network). The paradigm change from the traditional server/client model to a flexible, user-powered peer-to-peer network was already great enough for the time being.

As these networks diversified and rapidly gained popularity, however, it became apparent that so-called freeloaders or "leeches" could hamper the development of peer-to-peer technology by driving away other users. Freeloaders are nodes which download regularly, but offer little or no upload capacity. [Free00] has done an early analysis of the Gnutella peer-to-peer network, showing that nearly 70% of all users share no files at all and thus never contribute. If no discrimination is made against this sort of behavior, a network is likely to become slow for all users, first because much of the total bandwidth is consumed by these nodes who offer no upload, second because contributing users are driven off.

Thus techniques that might provide incentives for contributing to a peer-to-peer network gained interest and became the subject of increased research. While the basic requirements of such system are seemingly simple, practical applications have been unsatisfactory so far. Fairness and anonymity are hard to balance, and the inherently anarchistic traits of a true peer-to-peer network make it difficult to establish trust between nodes.

In this article, we wish to give an overview of an efficient reputation system that overcomes these problems and provides each node in the network with a global view of reputation: *Havelaar*. *Havelaar* is implemented as part of the *Kangoo* architecture, a modern peer-to-peer overlay network developed in the Distributed Computing Group at ETH Zürich, that will become available to the general public later in 2006. See [Kang05] for more information regarding *Kangoo*.

The thesis is structured as follows: First, we try to give an overview of related work, namely what kind of reputation systems are already in use in operating peer-to-peer networks, and what theoretical work has been done. The next section introduces *Havelaar* and explains its design. Then we will discuss the actual implementation of *Havelaar*. In "Analysis" we provide some data gained from simulating *Havelaar* in small scale. Finally we draw our conclusions in the last section and try to look at *Havelaar's* future.

2 Related Work

2.1 Deployed Systems

In this section we take a quick glance at what existing peer-to-peer networks have done in the field of providing upload incentives. Of the more popular networks, only few utilize some sort of contribution monitoring, and of those, none is a truly global system – one in which every node has an approximate view of the reputation values of all nodes.

2.1.1 Fasttrack/KaZaA

Fasttrack remains to be one of the most popular peer-to-peer networks. It appeared in early 2001, timed just right to supersede Napster after its forced end. Fasttrack, respective its original client "KaZaA"[1], supported many new features for a peer-to-peer network, most notably the ability to download different segments of a file simultaneously from different nodes.

Kazaa also tried to reward uploaders for the first time. The mechanism used is called "Participation Level" and relies only on a locally computed value. The formula used is as follows:

$$\text{Participation Level} = \frac{\text{Uploads in Megabytes}}{\text{Downloads in Megabytes}} * 100,$$

where the maximum level is 1000. There is also a special provision for so-called "integrity rated" files, which are a kind of user-approved files for ensuring reliable metadata and detecting fake files; "integrity rated" files have their size counted doubly when uploaded. The participation level value is sent to each node that a given node wishes to download from, where it determines its position in the remote node's download queue.

As the participation level is computed and stored locally (although encrypted), and then sent to other nodes, this system is obviously very prone to cheating nodes. One of the first such practices allowed a malicious node to download massive amounts of data from itself, thus quickly increasing its participation level. Next it became possible to save a node's participation level when it was a high value and restore it later on. When the aforementioned encryption was finally broken, clients could set their participation level to whatever value they please[2], rendering the whole system next to useless.

2.1.2 eDonkey/eMule

eDonkey is, like Fasttrack, one of the second generation peer-to-peer networks. It features parallel downloads, a robust hashing system for identifying files, provides upload incentives and is primarily known for its vast selection of files and slow but reliable downloads. It is estimated that several million eDonkey nodes are online at any given time, making it still one of the largest networks to exist. It is not a pure peer-to-peer application, but relies instead on central servers that act as global index of files and negotiate file transfers.

[1] http://www.kazaa.com/

[2] See http://www.khack.com/ for an example.

The reputation mechanism in eDonkey, first implemented in the open-source client eMule[3] (but not supported by all clients), is pretty straightforward. It depends on a value called "Credit Modifier", which is calculated in every interaction between two nodes that exchange data. The value is defined as

$$\text{Ratio1} = \frac{\text{Uploaded Data} * 2}{\text{Downloaded Data}}$$

$$\text{Ratio2} = \sqrt{\text{Uploaded Data} + 2}$$

$$\text{Credit Modifier} = \min\{\text{Ratio1}, \text{Ratio2}\}$$

and may not be lower than 1 or higher than 10. The Credit Modifier is stored for every remote node that a given node has transferred data to or from. It determines how fast a node moves forward in another node's upload queue.

As the Credit Modifier – or reputation – value is not stored locally, but on remote nodes and individually calculated for each two different nodes, it is much more difficult to attack than in the Fasttrack case. The downside of this system is that reputation is only relative to a given pair of nodes. A newly joining node will regard all other nodes as having the same reputation. On the other hand, a node that was a major contributor to the network loses all of it reputation if it installs a new client, as Credit Modifiers are cryptographically bound to a specific client.

2.1.3 BitTorrent

BitTorrent[4] has quickly gained popularity since its introduction a few years ago, and some internet traffic analyzer groups have claimed it is today responsible for quite a considerable amount of all internet traffic – estimates go up to as high as 35%. BitTorrent is not directly comparable to most other peer-to-peer networks, as it offers no search capabilities whatsoever, but is designed to distribute single files, announced by so-called trackers, as quickly as possible. BitTorrent is one of the few peer-to-peer applications that have gained some corporate acceptance, with companies such as RedHat using BitTorrent to ease the load on their download servers.

The reputation system used in BitTorrent has some similarities to the Credit Modifiers of eMule. It is no global system, in the sense that the reputation relationship is unique for every two nodes that interact, and in addition it is unique for every file (or collection of files) that is downloaded. The mechanism is a form of the classic, game-theoretic "tit for tat" strategy, which is believed to be the optimal behavior for its specific domain (the "iterated prisoner's dilemma").

Tit for tat in BitTorrent's sense means for a node to always upload to those remote nodes that provide the best download rates. As all nodes in a BitTorrent swarm compete for the same file, this is a simple and highly efficient method of providing an upload incentive. See [Bit03] for a detailed description of the algorithm.

[Mech05] has done an in-depth analysis of the tit for tat strategy with regard to peer-to-peer networks and BitTorrent in particular.

[3] http://www.emule-project.net/
[4] http://www.bittorrent.com/

2.1.4 Credence

Credence[5] is insofar different from the above examples as that it is not a peer-to-peer network of its own, but instead intended as an add-on to existing ones. At the time of writing, the only supported network was Gnutella. We should also note that the main target of Credence is not to rate nodes, but shared objects inside the network. Nevertheless it has an interesting mechanism to this end. Credence calculates a reputation value for a given object by collecting votes from randomly chosen other nodes which possess the object. It constructs a correlation graph to find nodes that tend to judge objects in a similar fashion as itself. See [Cred05] for the full algorithm.

Credence is an ongoing project and has some promising properties, especially regarding reliability of reputation. Still, the message overhead associated with every lookup (collecting votes from a sizeable part of the network) make Credence probably not very well suited for a node reputation system.

2.2 Theoretical Work

There has already been a considerate amount of research into this newly emerging subject. Some of the approaches taken include:

2.2.1 DHT-based Reputation Systems

As most recent peer-to-peer networks already form some sort of distributed hash table for their overlay structure, it seems natural to store reputation values in it, too. Say a node u makes an observation about another node v. It then stores the appropriate reputation value at the location in the DHT corresponding to v's address. Of course, measures have to be taken to ensure that v cannot set its reputation value itself. This method is very simple and difficult to attack, as a malicious node u would either have to gain control over another node r, which is responsible for u's reputation, or alternatively collude with many different other nodes in order to increase its reputation.

Unfortunately, this system also has its drawbacks. Performing a DHT-lookup every time a node needs the reputation of another node or wants to update it can prove to be quite expensive and a strain on the network [Rep04]. Additionally, we have to think about leaving nodes. A machine that leaves the network has to somehow preserve the reputation values of the node(s) it is responsible for, probably by sending them to its closest neighbor. One could mitigate this problem by storing reputation values at multiple locations, but this brings up the new problem of keeping the reputation values of a single node consistent across the network.

2.2.2 Models Based on Virtual Currency

A somewhat different approach on implementing contribution incentives is not to try and record an individual peer's uploads, but to provide nodes with a form of virtual currency, which they can use to "pay" for their downloads. In this fashion, nodes that upload frequently gain more "money" than nodes which do

[5] http://www.cs.cornell.edu/people/egs/credence/

not. That money can then be used to download more content, or – alternatively – to download *faster* than users with little contribution and money.

[Curr05] proposes such a model, where nodes can produce their own coins by performing a CPU-intensive task (finding collisions on a hash function, to be specific) and then spending them. Coins are signed by every node which possesses them at some point. To prevent double-spending of coins, every coin is "re-minted" once in a while by a central authority. If a coin is spent multiple times by a malicious node, it will also appear multiple times in the re-minting process, where it is easy to identify the cheating peer by looking at the signature history of the offending coins.

The system is elegant and probably well-suited for certain network applications, for example a distributed computing grid, where nodes can pay for utilizing resources of other nodes or for doing task that they wish not to perform themselves. However, it appears less ideal for a reputation system for a number of reasons. First, all payments and thus all file transfers involve public-key cryptographic calculations, which tend to be rather expensive. Second, the size of a coin grows each time it is spent, meaning communication cost can also get quite high if there is no frequent re-minting. The re-minting process itself needs a central, trust-worthy authority which is susceptible to attacks. Finally, as every peer can produce its own coins (providing each node with a fixed starting amount may seem more sensible, but then nodes could just spend that money and rejoin with a new identity), peers with large CPU capacities are preferred.

2.2.3 Other

In general terms, if a node in a peer-to-peer network is not able to make enough direct observations of other nodes' contribution of its own, then it has to somehow take into account *indirect* observations from other peers. Such systems (which the aforementioned DHT-based models are a part of) are well-known today for its usage in auction sites such as eBay, and are called *reciprocity-based* or *reputation* systems. These systems are inherently jeopardized by the phenomenon of *false reports* [Coop03]. Usually one tries to overcome this problem by establishing a *network of trust*, such that observations are weighted by the reputation of the observer. However, there is need for an infrastructure to exchange the second-hand observations, usually done by requesting them immediately before a transaction or by flooding them through the network. Both approaches result in significant communication costs and are ill-suited for a network with lots of transactions. In the *Kangoo* network, storing and retrieving a single file may result in 500 transactions with different peers, so there has to be some other way to provide nodes with reputation values of other nodes.

3 Havelaar

Here we wish to give a short overview of *Havelaar*. For more details, including analysis of the algorithms, please see [Have06] and [Have06b].

Havelaar was designed specifically for *Kangoo*. One of its central design features is that nodes do not poll for the reputation of another node (by looking

it up in the DHT or querying a central authority, for example), but they already are in possession of the reputation of most other nodes at any given moment. *Havelaar* also avoids the problems associated with networks of trust by omitting them altogether. Instead a node u that has benefitted from another node v reports the reputation gained by v to a set of successors that is determined by a hash function over its own address. As the hash function is known, every receiving node can check if the sender was legitimate. To ensure that reputation values are spread through the entire network, a node aggregates all observations from its predecessors before preceding to send the values to its own successors. Thusly, a node (or more specifically, a storage node, as other nodes do not upload) is always in possession of the data it needs to compare the reputation of nodes that want to download from itself and can distribute its upstream bandwidth accordingly.

In more precise terms: A peer u puts all the observations it makes about other nodes into a vector $\vec{o}_0$. An observation is any transaction in which u downloads a fragment form another node, and is calculated as size(fragment) $*$ transfer speed. In each round, u sends a message containing its own observations (i.e. $\vec{o}_0$), the aggregated observations of its k predecessors, which they made during the last round (denoted as $\vec{o}_1$), the aggregated observations of its k predecessors' own k predecessors ($\vec{o}_2$) from two rounds ago, and so forth. The message thus consists of a matrix $O := [\vec{o}_0, ..., \vec{o}_{r-1}]$. When a node receives a matrix $O := [\vec{o}_1, ..., \vec{o}_r]$ (note that the indices have changed because a new round has started) from a predecessor, it aggregates those observations into its own, local reputation matrix and updates its contribution vector $\vec{c}$, which it uses to determine the actual reputation of a remote node. Currently, the contribution vector holds just the sum of all observations for a given node, but one could also envision the introduction of an aging factor, so that observations from previous rounds are weighted less than those of the current round. It is clear that the vectors form previous rounds aggregate an exponentially growing numbers of observations, and that the oldest observations lie r rounds in the past.

It is also worth to note that in *Havelaar*, a node u increases the contribution value, or reputation, of a remote peer v after downloading a fragment from it, but it *does not* decrease that value if u uploads a fragment to v. This may be regarded as not being the fairest solution, but the designers of *Kangoo* explicitly state that they do not wish to provide any disincentives to downloading in the network.

4 Implementation

This section discusses the actual implementation of *Havelaar*. We introduce the different modules and its functionalities and review some of the design choices. *Havelaar* is written in Java (version 1.5), as is *Kangoo*, and consists of about 2'000 source lines of code.

4.1 Nomenclature

A node or peer is a single *Kangoo* node that is uniquely identified by its node id, and is either a storage node (a node that satisfies certain requirements regarding

uptime and free disk space and stores fragments), a super node (a node with good network connectivity that is assigned additional routing tasks) or a client node. There can be multiple nodes running simultaneously on the same machine. A machine is identified by its machine id. A user is the person running one or multiple nodes and is identified by its user id. *Havelaar* tracks the reputation of individual users, so that a user logging in from another node or machine still profits from its gained reputation.

4.2 Composing Classes

4.2.1 ReputationCenter.java

The reputation center is the heart of the *Havelaar* system. It allocates resources, starts the sending rate center and is responsible for keeping the reputation matrix up to date. It also saves its state, including all reputation values, to a preferences file on exiting and reads them back when restarting. As there is no continuous work to do, we chose not to create a separate thread for the reputation center, but to register it with the *Kangoo* scheduler and run every 30 seconds or so.

The reputation center keeps track of its successors and predecessors and allows only matrix transfers from clients who

- have sent a request with a valid certificate,
- are in the set of valid predecessors; i.e. the hash function applied to the sender's machine id returns a list which includes the receiving node's id,
- and have not already sent a matrix in the same round.

At the start of each round, the reputation center serializes its own reputation matrix and sends it to its successors.

4.2.2 ReputationMatrix.java

This is the abstraction of the matrix which consists of the reputation vectors from the current and the last $(r-1)$ rounds. It provides functions to add a new observation, to get the reputation value of a given user id, to merge the received matrices into the local one, to serialize the matrix (which calls the serialize function of all the vectors and concatenates the resulting byte arrays) and to construct one from its serialized form (again by calling the appropriate constructor of the vectors). Also, it contains a *dirty* flag to indicate that the matrix has been modified and that the reputation center should update its contribution vector.

4.2.3 ReputationVector.java

A reputation vector is a vector in which all reputation values of a given round are stored. Internally, the vector is represented as a hash table. As standard Java hash tables can only store objects, not primitive types, much space would be wasted by utilizing them. Instead, we chose to use a hash map implementation

from the *GNU Trove*[6] project, which offers support for primitive types and and claims performance benefits over the Java-supplied classes[7]. We use this hash table to map user ids (which can be represented as ints) to their reputation value.

This class provides functions to add an observation, to get the reputation value of a given user id, to serialize the values to a byte array and construct a new vector from the array. The vector gets locked for the duration of the serialization as observations might be added from a different thread meanwhile.

4.2.4 ReceivedMatrices.java

This is the class which holds the received matrices from a node's predecessors. Every round, it first checks for suspicious entries by comparing all observations for a given user and looking for spikes, which are discarded. It then aggregates the observation values into a single matrix and merges it with the local reputation matrix.

4.2.5 AggregatedVector.java

This class is a subclass of ReputationVector and stores the *aggregated* reputation values, that is the union of all values of the past r rounds. It corresponds to the contribution vector $\vec{c}$ mentioned in the previous description of *Havelaar*. Every time the reputation matrix is marked as dirty (by adding a new observation), the aggregated vector gets updated the next time the reputation center is scheduled to run. AggregatedVector provides additional functionality that divides the reputation values into percentiles, so that every user id can be assigned a percentile his reputation value lies in. This percentile value is then used in the bandwidth distribution mechanism.

4.2.6 SendingRateCenter.java

The sending rate center is the main interface to the rest of *Kangoo*. When an upload of a fragment to another node is initiated, *Kangoo* requests a sending rate from the center, passing the receiving node's user id and its desired rate. If there still is enough upload bandwidth available, a sending rate instance is returned, whose main purpose is to provide a function `getRate()`, which returns the actually assigned rate. As other uploads finish and new ones start, `getRate()` will return different values during the transfer, so it gets called for every packet to send.

4.2.7 Others

There are some other minor and supporting classes, namely the *Havelaar* related messages HavelaarPutRequest.java, HavelaarPutAccept.java and HavelaarPutDeny.java; the handlers for the actual reputation matrix transfer (HavelaarPutHandler.java and HavelaarGetHandler.java) and two different sending

[6] http://trove4j.sourceforge.net/
[7] See http://trove4j.sourceforge.net/performance.shtml for some benchmarks.

rate classes SendingRate.java and SimpleSendingRate.java, instances of which are given out by the sending rate center for each current upload.

4.3 Succesor Selection

The hashing algorithm used to determine the set of successors is a linear congruence PRNG, which is seeded with the machine id x_0, has a modulus of $m = 2^{32}$ (an obvious choice, as Java ignores overflows and silently discards resulting high-order bytes), and a multiplier $a = 663'608'941$, which satisfies Knuth's demand [Knuth98] that $a \bmod 8 = 5$ when m is a power of 2, and that a is between $0.01m$ and $0.99m$.

We also chose to use an increment c that must not be constant (we re-used the seed for that purpose), not primarily to enhance randomness, but because for a constant c, the sets of successors would overlap, resulting in poor diffusion of the reputation values. Say node n_0 calculates its set of successors $\{n_1, ..., n_k\}$, then it sends its reputation matrix to those nodes. n_1 in turn computes its own set of successors, which only differs in one node from that of n_0: $\{n_2, ..., n_{k+1}\}$. As it was decided later on not to use node ids but machine ids as input to the hashing, this problem no longer mattered. The dynamic increment was kept as it increases the randomness of the successor sets.

4.4 Transfer Protocol

When a node has gathered its successor set and is ready to send out its reputation matrix, it initiates the transfer sequence. As a specific – randomly chosen – node from the successor set could be offline[8], it sends a `HAVELAAR_PUT` message containing the given address, but with a flag indicating the message should be routed to the storage node closest to that address. Upon reception of a `HAVELAAR_PUT` message, a node first checks the cryptographically signed machine id of the sender. It then uses that id as input to the hashing function to see whether the original receiver id is correct.

If all those checks succeed and the sending node has not already sent its reputation matrix during the current round, the receiving node answers with a `HAVELAAR_ACCEPT` message and sets up a handler to receive the data. Else it sends a `HAVELAAR_DENY` message back to the sender. Upon reception of the accept message, the sending node assembles the serialized form of its reputation matrix, sets up a handler for the transfer, and sends its observations to the receiving node.

4.5 Serialization

As the designers of the *Havelaar* system note, "the Achilles heel of *Havelaar* is its message size [...]". As every node has to send its entire reputation matrix, which can grow to considerable size[9], to k successors once a round, there is quite

[8] This situation is actually very likely, noticing that we have an address space for about 2^{32} nodes.

[9] The first few vectors are sparsely populated, because they are computed from fewer observations, but the last vector should by design contain as much nodes as possible to provide a global view.

a lot of data to transfer. It is thus imperative to try and reduce this data as much as possible. We do this in two ways:

- First, we decide for each vector whether it is cheaper to send a map or a vector, i.e. whether to send a list of user ids and their corresponding values or send a starting and ending user id and all values in between, some of them probably zero (meaning there have been no observations about this user).
- Second, the data is (optionally) scaled down to a smaller data type, decreasing accuracy but greatly reducing the data size. A reputation vector has facilities to encode the reputation values (which are stored internally as ints) to either ints, shorts, or bytes. This is achieved by finding a scaling factor such that the biggest reputation value of the vector is scaled to just the maximum value of the chosen data type, and sending that scaling factor with the data.

4.6 Distributing the Upload Bandwidth

Havelaar is mainly a system for collecting and distributing reputation values across a peer-to-peer network, but there also has to be a mechanism for utilizing this information. The rewarding scheme has to distribute the upload bandwidth of a storage node among competing peers, that is among peers that wish to download a fragment from the storage node at the same time. There are multiple algorithms that try to do this as fair as possible while preventing starvation of clients with low reputation.

We first evaluated an algorithm discussed in [Game04] which is called "Resource bidding mechanism with incentive and utility feature (RBM-IU)". It introduces the concept of "utility" to represent the degree of satisfaction of a competing node given a certain allocated upload bandwidth. Say for example two nodes v, w compete for downloading a fragment from peer u, which has a maximum upstream bandwidth of 1 Mb/s. v can download with 2 Mb/s, while w has a downstream capacity of 5 Mb/s. If both node have the same reputation, then a straightforward algorithm might allocate a bandwidth of 0.5 Mb/s to each node. It is clear that node v, being able to download with 25% of its maximum rate, is more satisfied than node w, which can only use 10% of its downstream bandwidth. Thus the authors define an utility function for N competing nodes $\mathcal{N}_i,\ i \in \{1, ..., N\}$, where b_i (the *bidding value*) is the desired download rate of a node, and x_i represents its actual allocated rate. The utility function is defined as

$$U_i(x_i) = \log\left(\frac{x_i}{b_i} + 1\right) \qquad \text{where } x_i \in [0, b_i],$$

and the algorithm performs the following constrained optimization – where $\mathcal{W}$ is the maximum upstream bandwidth of the uploading node and $\mathcal{C}_i$ is the reputation value of the node $\mathcal{N}_i$:

$$\max \sum_{i=1}^{N} \mathcal{C}_i U_i(x_i) \qquad \text{s.t. } \sum_{i=1}^{N} x_i \leq \mathcal{W}, \qquad x_i \in [0, b_i]\ \forall\ i.$$

This algorithm has some nice theoretical advantages, however, we found a major weakness: It is possible for a malicious node to exploit the algorithm by probing for the best bidding value so it will gain the maximum download speed possible as determined by its reputation. Consider a source node $\mathcal{N}_0$ with an upstream capacity of 40 kB/s that uploads fragments to 4 competing nodes $\mathcal{N}_{1...4}$ with the same reputation value and desired upload rate (bidding value) 20 kB/s. The resulting bandwidth distribution will be, of course, 10 kB/s for every node. Now a malicious node $\mathcal{N}_5$ with the same reputation enters the picture, which has a possible downstream rate of 100 kB/s. If $\mathcal{N}_5$ were honest, it would report those 100 kB/s as its bidding value and get only the minimum guaranteed rate, as its link's utility is much more expensive to increase than that of the other nodes. But $\mathcal{N}_5$ is not honest, and uses a smaller value for its bidding value, say 10 kB/s. This time, $\mathcal{N}_0$ will set its upload speed to $\mathcal{N}_5$ to the maximum 10 kB/s, much to the disadvantage of the other nodes, which will now only get 7.5 kB/s each. But $\mathcal{N}_5$ does not stop at this point. It quickly sends a new bidding value, slightly higher than the last one, to see if it can get additional downstream bandwidth. For a bidding value of 15 kB/s, $\mathcal{N}_5$ gets a downstream of 12 kB/s. By incrementing the bidding value until the actual provided bandwidth decreases again, $\mathcal{N}_5$ finds the "sweet spot" for its bidding value, which will maximize its download.

This behavior is clearly detrimental to the network, as more malicious nodes will adapt to this probing strategy, rendering the whole concept of desired upstream rate as bidding value useless. For *Havelaar*, we thus chose to implement a more conventional algorithm which allocates upload bandwidth proportionally to a node's reputation. For an example, consider the following distribution for four nodes which download fragments from a source node with upload capacity 40'000 Bytes/s:

```
user 1, reputation 4.0, desired rate 20000, provided rate 17496
user 2, reputation 3.0, desired rate 20000, provided rate 13122
user 3, reputation 2.0, desired rate  5000, provided rate  5000
user 4, reputation 1.0, desired rate  5000, provided rate  4374
```

Note that user[10] 3 is assigned a rate that is not proportional to its reputation, this is because he already reached its desired rate.

We recalculate the bandwidth distribution every time a new node wants to download a fragment, an existing transfer stops, or when a node sends a new bidding message (i.e. its desired download rate has changed). To avoid starvation, every downloading peer is guaranteed a minimum rate, or, more exactly, every node has at least a minimum reputation of 1.0.

5 Analysis

For the analysis and testing purposes, we used a tool called the *Kangoo* Visualizer, which is able to run a simulation of a small *Kangoo* network in real time. The Visualizer is able to create a small set of nodes, which have pre-defined behaviors (such as putting a single or multiple fragments, or getting them back from the network). As every node runs in its separate thread, a true large-scale

[10] We are talking about users and not nodes here because *Havelaar* tracks reputation of users, not nodes. This has no implications for the algorithm, however.

simulation is not possible without writing a new simulation framework that runs synchronously. Still, the visualizer is suited to provide a first evaluation of the general concepts of *Havelaar* and to find bugs in the implementation. Figure 1 shows a screenshot of the Visualizer.

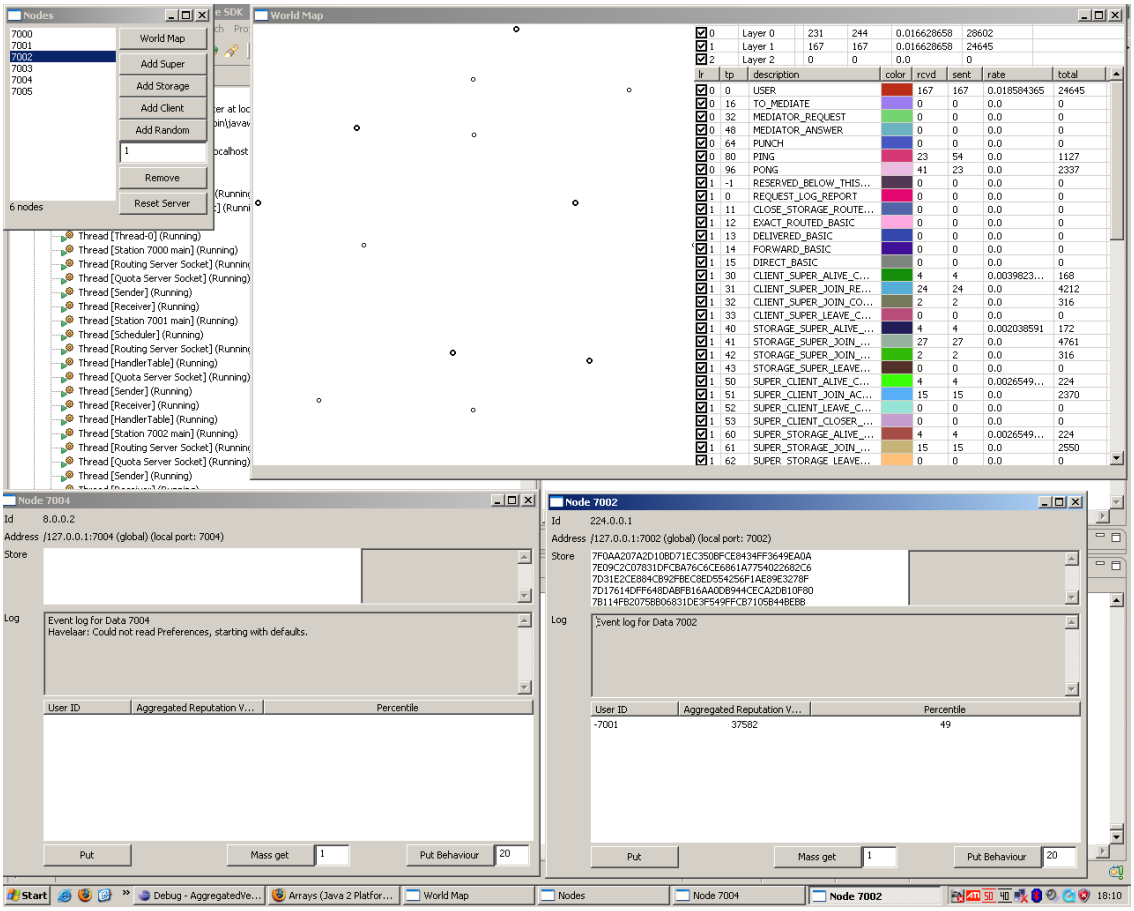

Figure 1: The *Havelaar* Visualizer.

As the *Havelaar* algorithm relies on a large number of interactions and the corresponding observations, a full-scale evaluation of the system is difficult without access to an existing *Kangoo* network. We instead decided to design certain test cases, which examine different key aspects of the algorithm. As the correctness and the efficiency of the algorithm is theoretically proven, this should suffice to preliminary ensure the quality of the implementation. Of course, additional testing will be required in a real-world deployment[11] to further analyze *Havelaar*. The following are the most interesting test cases.

Our first test case is simple enough: We simulate the smallest *Kangoo* network imaginable – consisting of just one super node, storage node and client node each. The client node is configured to regularly put fragments into the network, which – there being only one storage node – all get stored on the same

[11] A closed beta of *Kangoo* is planned for the end of the year.

node. It then downloads those fragments back from the DHT. The purpose of this case is just to check whether *Havelaar* correctly records the downloads from the storage node and updates the reputation matrix on the client node. Figure 2 shows that the reputation matrix of the client node was updated to represent the contribution of the storage node, respectively its user.

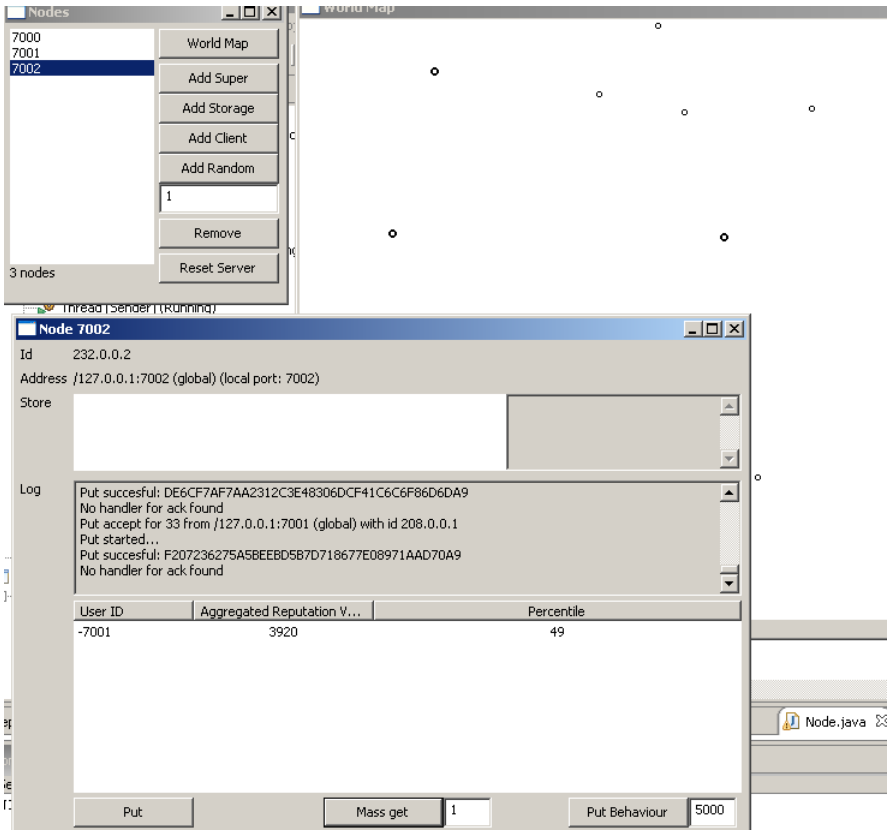

Figure 2: Node 7002 is a client node that has uploaded some fragments into the DHT. When it re-downloaded the fragments, it kept record of the contribution of the user of node 7001, the storage node.

The next test case is similar to the first one, except that we have now a slightly larger network with three storage nodes. The bandwidth of the storage nodes is capped to different levels. As the contribution value of a single transaction is calculated by multiplying the size of the served fragment by the transfer speed, the storage nodes different maximum transfer speeds should be reflected by their users' reputation values at the client nodes. Figure 3 demonstrates that this is indeed the case.

One test case has the goal of checking whether the transfer of a reputation matrix form one node to another works. We again have a small network consisting of a super node, storage node and client node. The client node has already made some observations about the storage node and sends out its reputation matrix. As there is only one storage node, it is the only eligible receiver for the client node's reputation matrix. Figure 4 and 5 show that the transfer of the reputation matrix is successful.

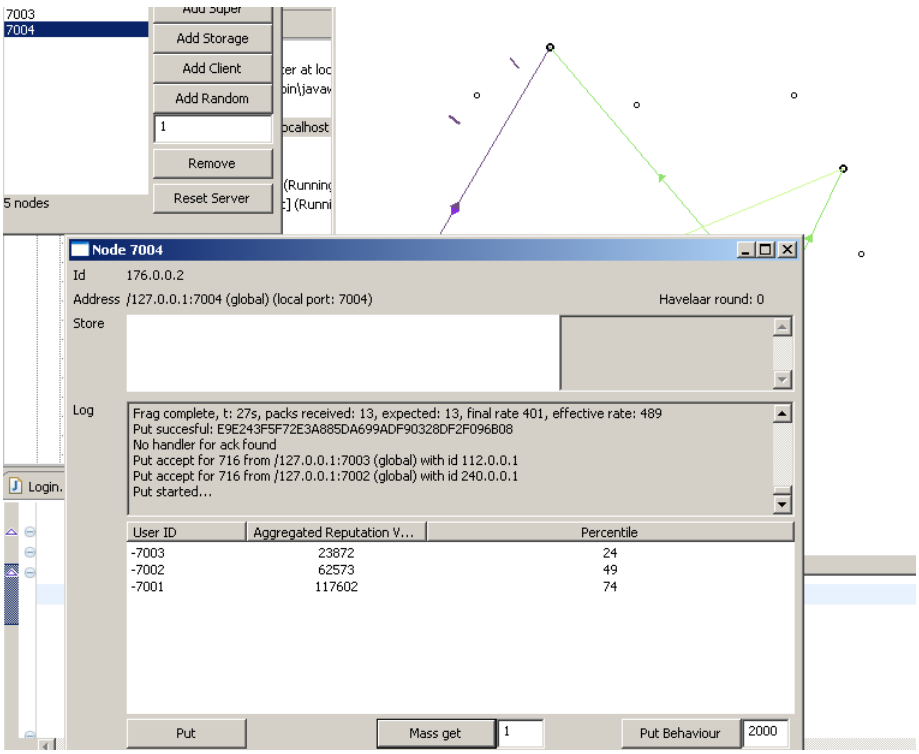

Figure 3: Node 7004 is the client node. It uploads fragments into the DHT, which should statistically be equally distributed to the three storage nodes 7001, 7002 and 7003. When node 7004 downloads those fragments back, the different storage nodes serve their fragments with their respective maximal upload speed, which is reflected in their users' reputation values at the client node. If we let the upload rate of a storage node n_i be $r(n_i)$, and its user's reputation value $c(n_i)$, then it holds that $r(n_1) < r(n_2) < r(n_3) \Rightarrow c(n_1) < c(n_2) < c(n_3)$, if the storage nodes serve similar amounts of fragments.

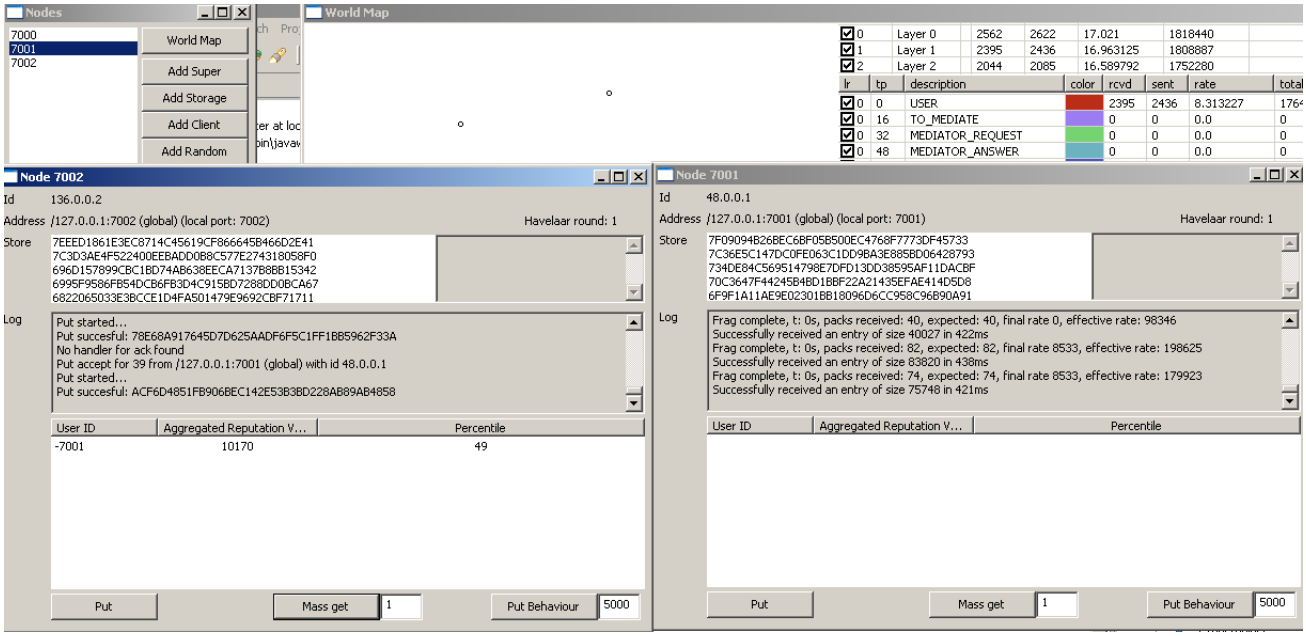

Figure 4: Node 7002 has made some observations about (the user of) the storage node 7001 by getting some fragments. Node 7001 has not made any observations yet.

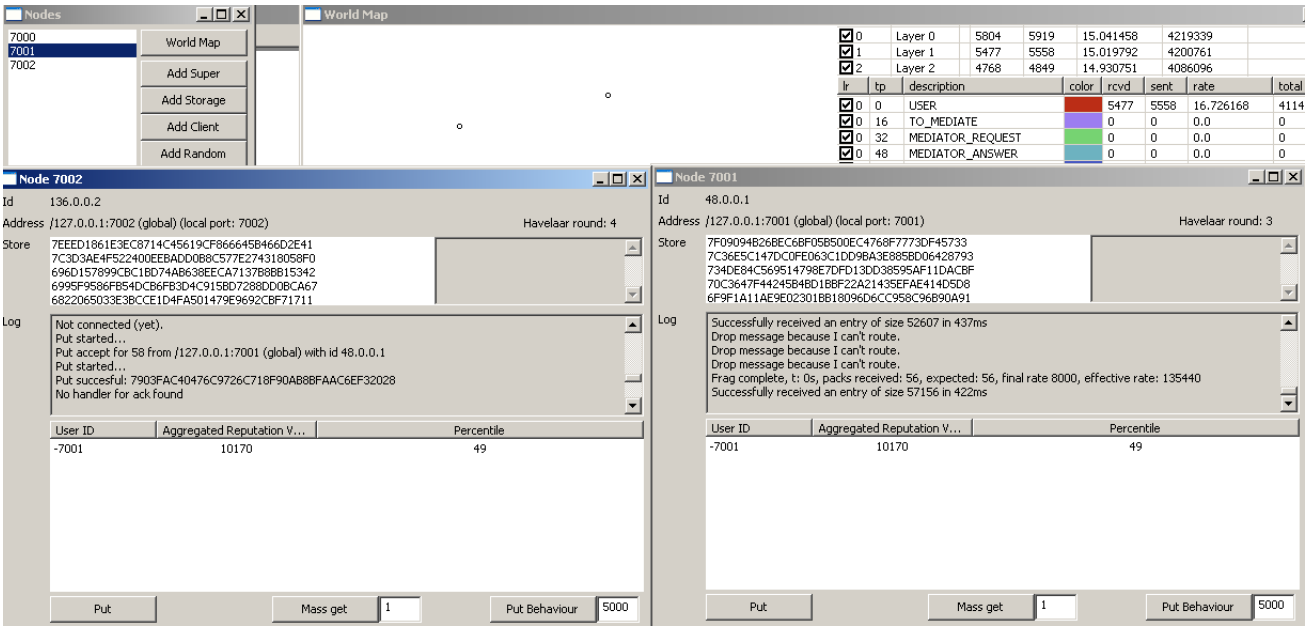

Figure 5: Node 7002 has sent its reputation matrix to node 7001, whose aggregated vector has been updated to include those observations.

The actual distribution of the upstream bandwidth (covered in 4.6) was evaluated with a separate test unit and found to be correct and adequately fast.

6 Conclusions

6.1 Havalaar

The first evaluations, while being small scale, show that the *Havelaar* algorithm is sound and is a viable way of distributing the reputation values into the network. If the duration of a *Havelaar* round is chosen large enough, the substantial size of a reputation matrix should not put an insensible burden on an average user's network connection, especially with broadband internet access still on the rise. The impact on performance is negligible, as the most CPU-intensive tasks (serializing the reputation matrix, checking and aggregating received matrices) only need to be performed once a round and are not time-critical. There is still work to do, but we believe that with the *Havelaar* system, there exists for the first time a reliable, practical and efficient system for collecting reputation about users in a peer-to-peer network and using that information to provide incentives for contributing to the network. It can be hoped that other networks will follow suit, maybe learning from the experiences of *Havelaar*, so that the "freerider" becomes a problem of the past, and that peer-to-peer networks continue to thrive.

6.2 Personal Conclusions

I would like to give some personal thoughts about this thesis in this section. Never having worked with Eclipse before, I felt a bit intimidated by the prospect of working on a project this big (*Kangoo* consists now of over 50'000 source lines of code, divided into numerous modules). Also, my experiences with Java, while coming from different subjects during my studies, were not that great. However, it can be said that the Eclipse IDE in combination with the Java programming language make for a quick adaption phase and the comprehensive Java API documentation proved extremely valuable. The clean structure of *Kangoo* also helped (using Interface types instead of actual classes where appropriate, division of the code in various packages with well-defined and minimal interaction, adequate but not excessive documentation, to name a few aspects). Communication with my advisors was excellent. My e-mails with questions to specific parts of *Kangoo* were answered promptly and always helpfully.

Problems mainly arose due to the fact that a small scale simulation, helpful as it is, makes evaluation and testing not that meaningful, especially as *Havelaar* is explicitly designed for large networks, because it relies on large amounts of observations to achieve approximate results. Also, the fact that *Kangoo* itself is still in development led to some difficulties with hard-to-reproduce bugs that were difficult to trace back to their origins. Especially the Visualizer was in constant change and thus not always very stable (a fact that the author greatly contributed to...)

All in all, it can be said that the work on *Havelaar* was a very interesting and challenging task, especially with the prospect that the code written for this

thesis will perhaps later be run by millions of users around the globe. I finally wish to thank my principal advisors Dominik Grolimund and Luzius Meisser for the opportunity to work on *Havelaar* and *Kangoo*, and for all the feedback, support and enthusiasm they provided.

References

[Bit03] Bram Cohen: *Incentives Build Robustness in BitTorrent*. In Proceedings of the First Workshop on Economics of Peer-to-Peer Systems, 2003.

[Coop03] Kevin Lai, Michal Feldman, Ion Stoica and John Chuang: *Incentives for Cooperation in Peer-to-Peer Networks*. In Proceedings of the Workshop on Economics of Peer-to-Peer Systems (P2PEcon), 2003.

[Cred05] Kevin Walsh and Emin Gün Sirer: *Thwarting P2P Pollution Using Object Reputation*. Technical Report TR2005-1980, Cornell University, Computer Science Department, 2005.

[Curr05] Flavio D. Garcia and Jaap-Henk Hoepman: *Off-line Karma: A Decentralized Currency for Peer-to-peer and Grid Applications*. In Proceedings of the Third International Conference on Applied Cryptography and Network Security (ACNS), 2005.

[Free00] Eytan Adar and Bernardo Huberman: *Free Riding on Gnutella*. First Monday, 5(10), 2000. Available at http://www.hpl.hp.com/research/idl/papers/gnutella/gnutella.pdf.

[Mech05] Lukas Füllemann, Stefan Schmid and Roger Wattenhofer: *P2P Mechanism Design*. Semester Thesis. Available at http://dcg.ethz.ch/theses/ss05/p2pmd_report.pdf. ETH Zürich, Switzerland, 2005.

[Game04] Richard T. B. Ma, Sam C. M. Lee, John C. S. Lui and David K. Y. Yau: *A Game Theoretic Approach to Provide Incentive and Service Differentiation in P2P Networks*. In SIGMETRICS, pp. 189–198, 2004.

[Have06] Dominik Grolimund, Luzius Meisser, Stefan Schmid and Roger Wattenhofer: *Havelaar: A Robust and Efficient Reputation System for Active Peer-to-Peer Systems*. Extended Abstract for the First Workshop on the Economics of Networked Systems, ACM Conference on Electronic Commerce, Ann Arbor, Michigan, 2006.

[Have06b] Dominik Grolimund, Luzius Meisser, Stefan Schmid, and Roger Wattenhofer: *Havelaar: A Robust and Efficient Reputation System for Active Peer-to-Peer Systems*. Technical report, TIKReport 246, available at http://www.tik.ee.ethz.ch/. ETH Zürich, Switzerland, 2006.

[Kang05] Dominik Grolimund and Luzius Meisser: *Implementation of the Kangoo Distributed Hash Table*. Semester Thesis at the Distributed Computing Group, ETH Zurich, 2005. Report available upon request.

[Knuth98] Donald E. Knuth: *The Art of Computer Programming, Volume II: Seminumerical Algorithms*, Third Edition, Addison-Wesley Publishing Company, 1998.

[Rep04] Thanasis G. Papaioannou and George D. Stamoulis: *Effective Use of Reputation in Peer-to-Peer Environments*. In Proceedings of the IEEE/ACM CCGRID 2004 (Workshop on Global P2P Computing), pp. 259–268, 2004.