

Introducing CPU Time as a Scarce Resource in P2P Systems to Achieve Fair Use in a Distributed DNS

Thomas Bocek¹, David Hausheer¹, Reinhard Riedl², Burkhard Stiller^{1,3}

¹Communication Systems Group, Department of Informatics IFI, University of Zurich, Switzerland

²Information Systems Group, Department of Informatics IFI, University of Zurich, Switzerland

³Computer Engineering and Networks Laboratory TIK, ETH Zürich, Switzerland

E-Mail: [bocek|hausheer|riedl|stiller]@ifi.unizh.ch

Abstract - Peer-to-peer (P2P) systems are flexible, robust, and self-organizing resource sharing infrastructures which are typically designed in a fully decentralized manner. However, a key problem of such systems are peers overusing a resource. This paper presents a fully decentralized scheme to achieve fair use in P2P systems, which does not require a priori information about a peer. The approach developed is based on a scarce resource trading scheme (SRTCPU), which utilizes CPU time as a form of payment. SRTCPU provides an incentive to offer CPU time in return of consuming a scarce resource. A distributed DNS has been implemented as an example application that uses SRTCPU.

Decentralized resource trading allows nodes to be independent from other nodes. Resource trading denotes the concept that a node receives a resource in return for providing a resource [7]. Resource trading is fully decentralized, as a node can trade with its own resources and can decide on its own whether or not to provide resources. Thus, the node is independent of other nodes.

In this paper, a fully decentralized resource trading scheme is developed. The scheme prevents overuse of resources in a P2P system by introducing a payment for services, which is based on calculations using CPU time. It maintains a self-regulated, self-organized, and sustainable resource exchange targeted at a fair resource allocation. This approach is termed SRTCPU (scarce resource trading with CPU time) [5]. The scheme proposed is evaluated using a name service as an application example. Thus, a Distributed Domain Name System (DDNS) prototype has been built that overcomes the resource overuse problem by applying SRTCPU. An analysis of this DDNS has been performed demonstrating that SRTCPU can be used in an integrated fashion with an application. The implementation and evaluation focuses on DDNS, where SRTCPU is a key component of it.

A first approach in developing a name service using P2P networks was developed in [8]. The key problem described there is the problem of *insertion denial of service*, which is considered in the context of this paper as a resource overuse. This problem deals with the possibility of a massive insertion of name value pairs. A node reserving every possible name combination in an endless loop could induce a stop in the network operation as new names can no longer be inserted. In contrast, in the traditional non-distributed DNS it is not possible to reserve such a great number of name combinations unless a huge amount of money is spent, since every name has a price to be paid.

Therefore, SRTCPU is proposed as a payment for name insertions into a DDNS. Unlike crypto puzzles, which determine a specific type of calculation in order to keep the CPU busy, SRTCPU can use the CPU for any type of calculation, *e.g.*, a useful calculation. This enables to contribute CPU time to a grid or other tasks which are in the need of computational power, *e.g.*, for solving large-scale computation problems, such as in SETI@home [21].

SRTCPU achieves a fair use, because a node can only use as much resources as an average node does, which means that

I. INTRODUCTION

A P2P system is fault-tolerant, robust and usually does not require any special administrative arrangements [3]. Currently, several P2P infrastructures exist, which can be used to implement large-scale applications, *e.g.*, CAN [19], Chord [23], Pastry [20], or Tapestry [25]. Applications based on these P2P systems may not only suffer from malicious nodes that can stop an application to operate by deploying a sibyl attack [11] or by pseudospoofing [10], but may also suffer from overuse of resources they do offer [8].

A node in a P2P system has typically a limited amount of resources that it may share with other nodes, *e.g.*, bandwidth, storage space, or central processing unit (CPU) power. However, if these resources are overused by other nodes, an unbalanced load is created such that nodes, which actively contribute to the network, are punished [14]. To reach a fair allocation of resources every node should only be allowed to use the amount of resources it provides. Three different accounting concepts can be applied to achieve such a fair allocation:

Centralized accounting can account for resource usage of every node in a P2P system. However, a central element is not a good option as the specific flexibility and robustness properties introduced by a decentralized P2P system would be weakened.

Decentralized collection of information about a node's behavior is an accounting concept that requires cooperation. Nodes collect information about resource usage and behavior of other nodes and provide other nodes with this information. Thus, decentralized collection of information is cooperative because the information is provided partly by third party nodes.

a node cannot overuse a given resource. This does not completely compare to freeriding, since SRTCPU does not directly enforce that a node using resources has to provide resources as well. However, SRTCPU prevents freeriding indirectly. If a node a has been identified as malicious (*e.g.*, a freerider) by node b , then the provided resources from node a in order to use resources on node b are lost. The following example illustrates the difference between freeriding and fair use achieved by SRTCPU: *E.g.*, pure freeriding can be described as a user traveling by train without paying a fee. In contrast, with SRTCPU users potentially pay 5 cents for a single train ticket. If the price is too low, everyone may be traveling by train, thus, the train will become overcrowded and the price will rise to 5 dollar.

The remainder of this paper is organized as follows: While Section II . compares related work, Section III . defines the design of SRTCPU. Section IV . discusses the implementation of the distributed DNS prototype and validates requirements. Finally, Section V . summarizes and discusses future work.

II. RELATED WORK

A review of related work has revealed a number of different approaches tackling the problem of resource overuse. Key characteristics taken into account cover the level of decentralization and the dependency on other nodes. These characteristics clearly distinguish the three concepts. Consequently, related work is discussed and compared according to the following dimensions:

- **Existence of a central element.** This indicates whether a central element is present or not.
- **Identification of fair resource usage based on local data.** Trust is good, control is better. A decision taken based on local data judging if a node is overusing a resource is always better than to trust other, potentially malicious nodes.
- **Efficient resource usage.** Counting back a hash is an exhaustive calculation. CPU time cannot be contributed to calculate anything else, *e.g.*, SETI@home [21].
- **Resource symmetry.** Resource symmetry denotes the trading with the same kind of resources, for example bandwidth in exchange for bandwidth.

For further information on related work, refer to [5].

A. Comparative Dimensions

Distributed systems applying decentralized or centralized schemes do show different behaviors. Thus, a comparative study has been performed. The notion of cooperation has been introduced to denote a high degree of dependency. In cooperative systems, *e.g.*, transitive trust management ([1] [9] [12] [15]), a node is dependent on more nodes than in independent systems. For example in tit-for-tat (TFT), a node can make a decision about sharing a resource based on the interaction between itself and another node. Examples for deploying a

central element are: DNS, SETI@home [21], PPay [24], or A4C [22].

Resource trading can be divided into symmetric and asymmetric resource trading. Symmetric resource trading happens when the same kind of resource is traded between two nodes, *e.g.*, if one node requests x data sets from a second node, the first node has to provide the second node with x data sets, too. Bittorrent [6] is an example of symmetric resource trading with bandwidth taking place. However, symmetric resource trading only works, if the first node is interested in the resource of the second node

If symmetric resource trading is not possible another resource available for trading has to be found. Asymmetric resource trading happens when one type of resource is traded for a different type of resource, *e.g.*, storage space in return for CPU power. HashCash [2] and SRTCPU deploy an asymmetric resource trading scheme. Asymmetric Resource Trading implies dealing with exchange rates for trading different resources with different values. However, in a decentralized system, no supervisor or resource allocator is present. Therefore, decentralized resource trading has to happen in a completely self-regulated and self-organized way.

B. Comparison

With respect to SRTCPU, TFT and hashcash, no central elements are necessary and they are not dependent on data from other nodes for verifying fair resource usage. In addition, with SRTCPU CPU time can be used for arbitrary calculations. In contrast, hashcash keeps the CPU busy with a very specific computational task without any freedom of choosing an arbitrary calculation. This can be considered as a waste of CPU time. Finally, in contrast to TFT, no resource symmetry is required for SRTCPU. Thus, SRTCPU defines an optimal approach to prevent the overuse of a resource in a fully decentralized system. The full comparison of approaches is shown in Table , where “+” indicates the presence of the characteristic, while “-” indicates its absence.

TABLE I
COMPARISON OF DIFFERENT APPROACHES FOR RESOURCE TRADING

	No central element	Identification of fair resource usage based on local data	Efficient resource usage	No resource symmetry required
Tit-for-tat	+	+	+	-
Trust Management	- ^a	-	+	+
Hashcash [2]	+	+	-	+
Centralized Accounting	-	-	+	+
PeerMint [13]	+	-	+	+
SRTCPU	+	+	+	+

a. Trust Management has to limit the number of possible identities of a client. [11] show that without resource trading, a limitation is only possible with a central element.

III. EXAMPLE-DRIVEN DESIGN

Based on the investigation of related work, the key requirements for SRTCPU have been collected. The detailed design of SRTCPU includes the set of mechanisms in charge of sending computational tasks. Those need CPU time to be solved, basically in exchange for another resource to become as scarce as the CPU time. The basic design of such a use of CPU time includes a process to choose randomly a third party acting as a task provider.

A. Technical Prerequisites and Requirements

A node ID identifies a node, and SRTCPU relies on the fact that a node ID cannot be freely chosen. A prevention of arbitrary node ID selection can, *e.g.*, be achieved by calculating the node ID based on the IP address [23]. A second possibility is to use a central element, as proposed by [11] to assign node IDs.

The following 7 requirements have been defined to enable the design of a fully decentralized asymmetric resource trading scheme which will meet the key goals of a distributed system: efficiency and scalability with respect to the number of resources being dealt by and the number of users utilizing the scheme developed:

- Fair. Provide a fair allocation of resources, i.e. prevent resource overuse.
- Fully decentralized. No central elements should be present.
- Load balanced. A node should not become a bottleneck.
- Fault tolerant. A failure of nodes should not affect the service.
- Self-organized. The scheme is able to adapt to changes in the network.
- Efficient. The scheme should consume as little resources as possible.
- Trustworthy. In order to reduce the risk of attacks, minimal trust relations should be deployed.

B. Main Challenge

The main challenge of the architectural design is to ensure that a node has calculated a task by itself. Without this assurance the node could re-label the task and send it as a task provider to another node. To prevent such behavior a checksum of what has been calculated is being made. The same task from the same task provider results in the same checksum. The checksum is calculated using the instructions that have been performed and the node ID of the node that has commissioned the task. A malicious node that relabels a task and sends it back as a new task will receive a different checksum from that relabeled task. The node which received the relabeled task will use the node ID from the malicious node to calculate the checksum. As stated in the technical prerequisites, a node ID cannot be freely chosen and other nodes can verify that. As a consequence, a malicious node cannot fake its node ID without being detected.

C. SRTCPU Design

The basic design of SRTCPU is depicted in Figure 1. Three participants can be identified: A task provider that has a calculation to be outsourced, a requesting node (node 2) and a node that provides a resource (node 1). The number of requesting nodes is denoted r , the percentage of malicious nodes is m .

The communication starts with node 2 requesting a resource (1). In order to get that resource node 2 must be willing to provide CPU time for that resource. Node 1 sends this request to the task provider (2). Node 2 receives and calculates the task (3) which was requested from node 1. During task calculation, a checksum is generated which includes the node ID of node 1 and the instructions that have been executed during the task. The result and the checksum are sent back to the task provider (4). Node 1 receives the checksum from node 2 and the task provider (5), (6). This allows node 1 to determine if the task provider has received the result. If the checksum is not received from the task provider, an error has occurred either at node 2 or the task provider. Node 1 will mark both nodes in a local list as potentially malicious. If a certain threshold is reached as a node continues to behave incorrectly, that node will not be considered anymore as a trading partner in future.

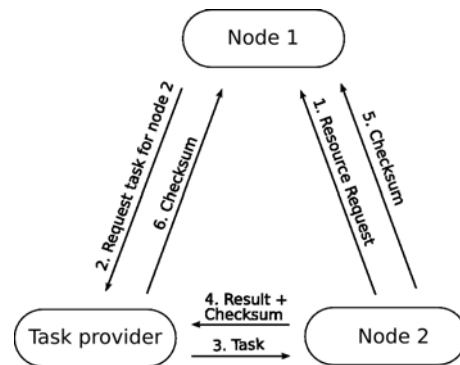


Figure 1. Basic design of SRTCPU with three participants ($r=1$, $m=0$).

A task provider can announce itself to any node. The task provider that announces itself to a node does also have to compute a task to get listed. A listed task provider is then randomly selected to provide a computational task.

A design with three participants and $r=1$ is not feasible if a malicious node may send back arbitrary data and checksums. Therefore, more nodes ($r>1$) have to be introduced. Figure 2 shows the extended design with $r=2$. Node 4 has been introduced and is calculating the same task as node 3. It is obvious that a task has to be deterministic. Node 1 receives r checksums that have to be equal. If they are not, one node has made a mistake and both nodes are marked as potentially malicious. When the number of mistakes reaches a certain threshold, a malicious node is detected. Note, that this scheme only works when $m<0.5$.

Node 1 is also able to send a computational task, if no task provider is available or $r=1$ with $m>0$. The task looks similar to hashcash, calculating back a hash, i.e. node 1 sends node 2 a hash value of an initial value. Node 2 has to calculate back a possible initial value for that hash value.

It is also possible to have $r > 2$. With more checksums sent back, node 1 can determine faster if a node is behaving maliciously. However, with more nodes, more overhead is created.

The number of malicious nodes may vary from none to any number. Through parameter r and the type of task (crypto puzzle or arbitrary calculation), the impact of malicious nodes can be reduced or even eliminated. When using crypto puzzles the scheme can tolerate any number of malicious nodes independent of r , because the result of the task can be verified by the node that has sent the task. When using arbitrary calculations, with $r > 1$ a certain amount of malicious nodes can be present, because the checksum can be compared and, thus, malicious nodes can be detected and ignored. By varying the parameter r and the type of task, SRTCPU can be adapted to a certain number of malicious nodes trying to overuse a resource.

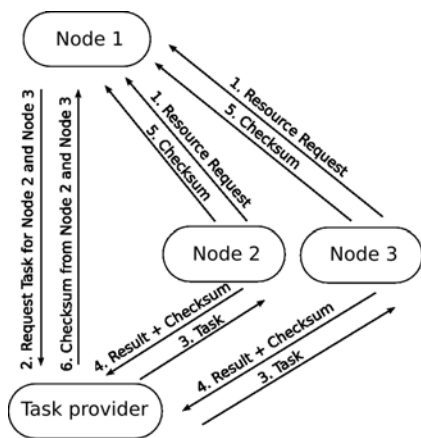


Figure 2. Extended design of SRTCPU with four participants. 2 nodes (node 2,node 3) have received the same task ($r=2$)

D. Self-Regulating Task Difficulty

With SRTCPU a resource is paid by calculating a task. The amount of payment is defined by the difficulty of the task which is an important factor in making a resource scarce. If the difficulty is set too low, overuse will be the result, if it is set too high, underuse will happen. Along with every checksum sent back, the number of processed instructions is sent to indicate if more computation is needed. Node 1 knows how much node 2 and node 3 have calculated by indicating the number of processed instructions that has been returned to node 1. Node 1 now needs to determine the average CPU time per inserted name value pair to keep the scheme self-regulated. This way, the scheme keeps track of an increase in CPU power over time [18].

Node 1 can determine the average CPU time with a benchmark on itself, under the assumption of having an average CPU. A tolerance factor will be applied to allow for a wider range of different CPU power. The tolerance factor determines how homogeneous in terms of CPU processing speed the system can be. This is an important factor because not every node has the same CPU power. Small and embedded devices have usually lower CPU power. SRTCPU rather focuses on homogeneous systems where a similar amount of resources are available on every node.

To calculate the average CPU time per inserted name, node 1 needs to know the average amount of data stored in the P2P system. Again, the node assumes that its stored data is the average. With the average CPU time and the average stored data, node 1 can decide if node 2 and node 3 have calculated enough to be allowed to store a name on node 1. Hence, the system is self-regulated.

IV. IMPLEMENTATION AND EVALUATION

DDNS has been implemented to validate the feasibility of a decentralized name service using SRTCPU. SRTCPU comes in place when a name is stored. With every store command, SRTCPU provides a task for the node that wants to store a data record.

The calculations of the task are executed in a virtual machine which performs only mathematical tasks. For the sake of simplicity, a prototypical MathVM [4] has been implemented in Java. The MathVM can update the checksum on every processed instruction. It uses its own language and has an assembler that translates the source code into instructions (opcodes). A task description in the MathVM has a fixed size of several Kilobyte. Only one task description representing a crypto puzzle has been implemented for SRTCPU. The prototype does not take into account any malicious nodes. DDNS is based on a DHT with an XOR metric [16] with an adaptation of the DDNS message format. DHTs provide a good performance for lookup, i.e. in $O(\log n)$, where n is the number of nodes. The prototype has been tested with 1000 nodes to validate that the search scales gracefully [4].

A. DDNS Efficiency Analysis

The two most important commands of DDNS are *get* and *store*. A *get* is a data lookup that takes $O(\log n)$ in a structured overlay network [16]. A *store* performs first a lookup in $O(\log n)$, in order to find the node where to store the data. For the detailed communication protocol, please refer to [4].

Let n be the number of participating nodes. It is assumed that the number of queries scales with $O(1)$, which means that the number of queries a user performs are constant over time. The analysis of the *store* and *get* commands shows in the following that the message complexity remains $O(\log n)$ as in the Kademlia network [16].

The *get* command has been modified for SRTCPU to obtain always more than one result, but this does not affect $O(\log n)$. Based on the fact that the number of queries $p(n)$ is $O(1)$, the overall number of messages which equals to $p(n) * O(\log n)$ is also $O(\log n)$. The *store* command has been modified with the SRTCPU mechanism. First, every *store* performs a *get* to check if the name exists. For every *store* the message complexity is $p(n) + (p(n) * O(\log n))$. Again $p(n)$ is $O(1)$ and the efficiency is $O(\log n)$. Then the tasks and the results are being exchanged without any further lookup.

Even if $p(n)$ would be $O(\log n)$, the complexity of the store command would still be $O(\log n)^2$. However, with $p(n) = O(n)$ the complexity would be $O(n * \log n)$, which is too high. On

the other hand, with the same situation that $p(n) = O(n)$, the complexity of the DNS would be $O(n) * O(1) = O(n)$, thus the scheme would still not scale.

As a result of this analysis, all modifications do not exceed $O(\log n)$ and the lookup remains scalable. Nevertheless, some optimizations could boost the performance. One optimization is clustering as done in SHARK [17]: The namespace can be divided into parts. A part is built by grouping similar nodes. Similar in this context means that nodes searching the same names are similar. Let $p(n)$ be the amount of nodes that searches within its parts and $q(n)$ be the amount of nodes that searches in other parts. Having constant part sizes and growing numbers of parts, a *get* command in such a part is $O(1)$. The *get* command for searching in- and outside of a part is $p * O(1) + q * O(\log n)$. The complexity remains $O(\log n)$, but if a lot of searches are within a part, the *get* command becomes faster. The second optimization is caching, where the same situation arises. When caching a lot of entries the *get* command becomes faster, but there will also be uncached names, consequently the system remains $O(\log n)$.

B. DDNS Experiments

The prototype implemented has been tested and several simulation experiments with up to 100 nodes per machine have been performed. 10 machines have been used to simulate a network.

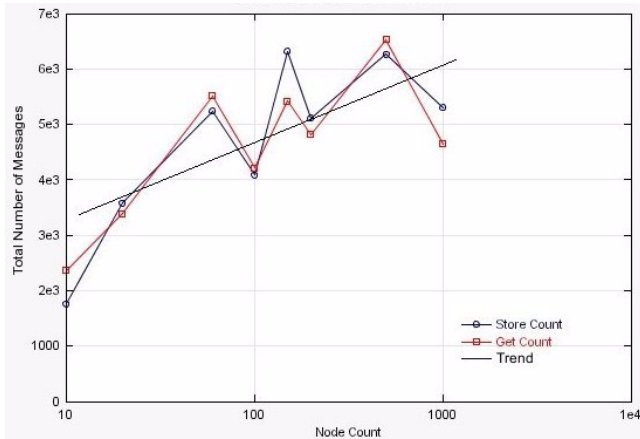


Figure 3. Simulation experiments of the prototype with SRTCPU as a part of DDNS.

These experiments have confirmed the results of the efficiency analysis. The simulation has been performed using DDNS as an example application that uses SRTCPU. To enable SRTCPU in DDNS the *store* command has been modified. With each *store* command, a request for a task is sent. A reply to a store command includes the task and after calculating the task, the result is being sent back. The result for the number of messages in total for *get* and *store* are depicted in Figure 3. This figure indicates that the communication cost grows with $O(\log n)$. In the experiment 15 names were stored and 100 *get* requests were made. The *get* requests were made once and the *store* commands were repeatedly called. After all names had been stored and 100 requests had been made, the run was aborted. The numbers of nodes tested were 10, 20, 60,

100, 150, 200, 500, 1000. In each step, 2 runs were made and the mean was calculated. A node could fail with a probability of 10%. Malicious nodes were not considered ($m=0$), no task providers were present, and $r=1$.

Due to the implementation of the prototype with just one thread per communication channel, only a small number of nodes (app. 150 nodes) could be tested per machine.

C. Validation and Discussion

Key requirements discussed above have been met.

- Fair. Fairness is achieved since everyone has to pay about the same amount of CPU time for a resource. The amount of CPU time to be paid is self-organizing. Resource over-use is limited through the introduction of SRTCPU offering the possibility of trading different types of resources. In DDNS, a name value pair is traded for CPU time.
- Fully decentralized. SRTCPU is fully decentralized. No central element is necessary to account for resource usage.
- Load balanced and fault tolerant. Due to redundant storage a failure does not affect the scheme and the load is balanced. Additionally, a caching mechanism in DDNS could optimize the balance especially with popular names.
- Self-organized. With the use of CPU time to prevent over-use of a resource, every node can decide how much to charge for its resource. A central element is not necessary and a self-regulating task difficulty establishes a balanced charge of CPU time with respect to increasing processing power (Moors' law [18]) over time. Therefore, even when the resources change over time the system itself remains self-organized.
- Efficient. Although CPU time has to be spent, SRTCPU adds a benefit over a crypto puzzle by making the CPU time available for other calculations as well.
- Trustworthy. Trust can be measured based on local data. Thus, minimal trust relations have been achieved.

In SRTCPU, malicious nodes can either be detected by introducing redundancy and comparing the results of the calculations from other nodes, or by sending crypto puzzles as computational task. Malicious task providers can also be detected based on redundancy, i.e. nodes can randomly pick a task provider. Information about unexpected behavior of task providers and nodes will be collected and evaluated. If a certain threshold of unexpected behavior is reached, the node or task provider is considered as malicious. The number of task providers can also be limited by using SRTCPU when a new task provider joins the system, i.e. to get listed on a node, a task provider would have to pay with CPU time. The type of calculation that a task provider wants to have calculated is not important. To protect against misuse, an endless loop in a calculation will result in a time-out.

A problem arises when the network is very heterogeneous, i.e. with a lot of different CPU capabilities. [11] states that *large-scale distributed systems are inevitably heterogeneous*. Especially embedded and small devices have usually fewer

resources to allocate. However, if a participation in a system requires certain resources, the problem of heterogeneity is less relevant. If a node wants to join the network it has to make sure that it has enough resources. If there are not sufficient resources, the node cannot join and has to be upgraded, *e.g.*, with more storage space or a faster processor. The result is that every SRTCPU participant will have a similar amount of resources available.

Efficiency is limited by the possible presence of malicious nodes and the absence of nodes for the verification of a computational task. Overhead is created as SRTCPU has to send a task to multiple nodes to verify the result. Therefore, the calculation is not as efficient as in a supervised system. A fallback strategy similar to hashcash is used when just one node requests a resource. In this case, the CPU time cannot be contributed to calculate anything else.

The bandwidth capacity of a node may become a bottleneck, if the task description is large. In the DDNS prototype, the task description is limited to a fixed size. Without this limitation, the required bandwidth would be considered as part of the payment.

It is not possible to guarantee uniqueness of names in a fully decentralized system. However, the ambiguities in DDNS may exist only for a limited time. If two names are inserted at the same time but from different peers, the name propagated faster will prevail [4].

V. SUMMARY AND FUTURE WORK

Introducing CPU time as a scarce resource in a P2P system can make other resources as scarce as the CPU time by binding these two resources together. As an example application of SRTCPU, DDNS has been implemented. DDNS is an architectural concept of a decentralized name service based on a P2P network. It implements SRTCPU to protect the critical resource *storage space for name value pairs*. The names can be retrieved in $O(\log n)$ due to the underlying DHT. With the introduction of task providers, one can use the CPU time, resulting from name insertions, for any task. It can, *e.g.*, be used in a grid, to solve large-scale computation problems similar to SETI@home. Contrary to the altruistic donation of CPU time to large-scale computation communities, DDNS creates an incentive to provide CPU time to a grid. Thus, the feasibility of DDNS has been validated with a prototypical implementation.

Future approaches may be based on different resources, other than CPU time. For example, a decentralized web page system with a name service could have storage space and bandwidth as the scarce resource. In order to store the name of a web page, storage space would have to be provided to other nodes.

To achieve a more detailed view of the impact of malicious nodes and to retrieve additional data from the simulation, simulation experiments are prepared to be made on a group of machines, mainly by varying the parameter r and changing the number of malicious nodes.

VI. REFERENCES

- [1] K. Aberer, Z. Despotovic. *Managing trust in a peer-2-peer information system*. H. Paques, L. Liu, and D. Grossman (Edts): 10th International Conference on Information and Knowledge Management (CIKM'01), pp 310 — 317, New York, U.S.A., November 2001.
- [2] A. Back. *Hash Cash — A denial of service counter-measure*, URL: <http://www.hashcash.org/papers/hashcash.pdf>, August 2002.
- [3] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica. *Looking up data in P2P systems*. Communications of the ACM, 46(2):43 — 48, February 2003.
- [4] T. Bocek. *Feasibility, Pros and Cons for Distributed DNS*, Master Thesis, IFI, University of Zurich, May 2004.
- [5] T. Bocek, D. Hausheer, R. Riedl, B. Stiller. *Introducing CPU Time as a Scarce Resource in P2P Systems*, University of Zurich, IFI, Technical Report No. ifi-2006.01, January 2006.
- [6] B. Cohen. *Incentives Build Robustness in BitTorrent*. 1st Workshop on Economics of Peer-to-Peer Systems, Berkeley, California, U.S.A., June 2003.
- [7] B. F. Cooper, H. Garcia-Molina. *Peer-to-Peer Resource Trading in a Reliable Distributed System*, Lecture Notes in Computer Science, Volume 2429, Springer, Berlin, pp 319 — 327, January 2002.
- [8] R. Cox, A. Muthitacharoen, R. Morris. *Serving DNS using a peer-to-peer lookup service*, 1st International Workshop on Peer-to-Peer Systems, Cambridge, Massachusetts, U.S.A., March, 2002.
- [9] E. Damiani, De Capitani di Vimercati, S. Paraboschi, P. Samarati, F. Violante. *A reputation-based approach for choosing reliable resources in peer-to-peer networks*. 9th ACM Conference on Computer and Communications Security, Washington DC, U.S.A., pp 207 — 216, November 2002.
- [10] R. Dingleline, M. Freedman and D. Molnar. "Accountability", *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, Chapter 16, pp 271 — 340, O'REILLY Press, 2001.
- [11] J. R. Douceur. *The sybil attack*. 1st International Workshop on Peer-to-Peer Systems (IPTPS02), Cambridge, Massachusetts, U.S.A., March 2002.
- [12] M. Feldman and K. Lai and I. Stoica and J. Chuang. *Robust Incentive Techniques for Peer-to-Peer Networks*, ACM Electronic Commerce, New York, U.S.A., pp. 102 — 111, May 2004.
- [13] D. Hausheer, B. Stiller. *PeerMint: Decentralized and Secure Accounting for Peer-to-Peer Applications*, IFIP Networking 2005, Ontario, Canada, May 2005.
- [14] S. Kamvar, M. Schlosser, and H. Garcia-Molina. *Incentives for Combating Freeriding on P2P Networks*. Euro-Par, Klagenfurt, Austria, June, 2003.
- [15] S. D. Kamvar, M. T. Schlosser, H. Garcia-Molina. *The eigentrust algorithm for reputation management in p2p networks*. 12th International World Wide Web Conference (WWW), Budapest, Hungary, May 2003.
- [16] P. Maymounkov, D. Mazieres. *Kademlia: A peer-to-peer information system based on the xor metric*. 1st International Workshop on Peer to Peer Systems (IPTPS'02), Cambridge, Massachusetts, U.S.A., March 2002.
- [17] J. Mischke, B. Stiller. *Rich and Scalable Peer-to-Peer Search with SHARK*, Autonomic Computing Workshop — 5th Annual International Workshop on Active Middleware Services (AMS'03), Seattle, Washington, U.S.A., 2003.
- [18] G. Moore. *Cramming more components onto integrated circuits*; Electronics, Vol. 38, No. 8, April 1965.
- [19] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker. *A scalable content addressable network*. ACM SIGCOMM'01, San Diego, California, U.S.A., pp 161 — 172, August 2001.
- [20] A. Rowstron, P. Druschel. *Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems*. 18th IFIP/ACM Conference on Distributed Systems Platforms, Heidelberg, Germany, November 2001.
- [21] SETI@home, URL: <http://setiathome.ssl.berkeley.edu/>, August 2005.
- [22] B. Stiller, J. Fernandez, Hasan, P. Kurtansky, W. Lu, D.-J. Plas, B. Weyl, H. Ziemek, B. Bhushan. *Design of an Advanced A4C Framework*, Whitepaper, EU IST Project Daidalos, <http://www.ist-daidalos.org/>, August 2005.
- [23] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan. *Chord: A scalable peer-to-peer lookup service for internet applications*. ACM SIGCOMM'01, San Diego, California, U.S.A., pp 149 — 160, March 2001.
- [24] B. Yang, H. Garcia-Molina. *PPay: Micropayments for Peer-to-Peer Systems*. Technical Report, Stanford University, California, U.S.A., 2003.
- [25] B. Y. Zhao, J. D. Kubiatowicz, A. D. Joseph. *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing*, Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, California, U.S.A., April 2001.