Tag-Based Cooperation in Peer-to-Peer Networks with Newscast

Andrea Marcozzi **David Hales** **Gian Paolo Jesi**
Stefano Arteconi **Özalp Babaoğlu**

Technical Report UBLCS-2005-15

June 2005

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)

The University of Bologna Department of Computer Science Research Technical Reports are available in PDF and gzipped PostScript formats via anonymous FTP from the area ftp.cs.unibo.it:/pub/TR/UBLCS or via WWW at URL http://www.cs.unibo.it/en/research/reports/. Plain-text abstracts organized by year are available in the directory ABSTRACTS.

Recent Titles from the UBLCS Technical Report Series

2004-14 *Intelligent Web Servers as Agents*, Gaspari, M., Dragoni, N. Guidi, D., July 2004.

2004-15 *SIR: a Model of Social Reputation*, Mezzetti, N., October 2004.

2004-16 *Algorithms for Large Directed CARP Instances: Urban Solid Waste Collection Operational Support*, Maniezzo, V., October 2004.

2004-17 *Supporting e-Commerce Systems Formalization with Choreography Languages*, Bravetti, M., Guidi, C., Lucchi, R., Zavattaro, G., November 2004.

2004-18 *Decentralized Ranking in Large-Scale Overlay Networks*, Montresor, A., Jelasity, M., Babaoglu, O., December 2004.

2004-19 *Advanced Collective Communication in WDM Optical Rings*, Margara, L., Simon, J., Vassura, V., December 2004.

2005-1 *ARTIS: Design and Implementation of an Adaptive Middleware for Parallel and Distributed Simulation (Ph.D. Thesis)*, D'Angelo, G., March 2005.

2005-2 *Analysis and Prototype of a Metamodeling Environment for Engineering Grid Services (Ph.D. Thesis)*, Moretti, R., March 2005.

2005-3 *On some combinatorial optimization problems arising from computer networks (Ph.D. Thesis)*, Vassura, M., March 2005.

2005-4 *Experiences with Synthetic Network Emulation for Complex IP based Networks (Ph.D. Thesis)*, Cacciaguerra, S., March 2005.

2005-5 *Interactivity Maintenance for Event Synchronization in Massive Multiplayer Online Games (Ph.D. Thesis)*, Ferretti, S., March 2005.

2005-6 *Reasoning with preferences over temporal, uncertain, and conditional statements (Ph.D. Thesis)*, Venable, K. B., March 2005.

2005-7 *Whole Platform (Ph.D. Thesis)*, Solmi, R., March 2005.

2005-8 *Loss Functions and Structured Domains for Support Vector Machines (Ph.D. Thesis)*, Portera, F., March 2005.

2005-9 *A Reasoning Infrastructure to Support Cooperation of Intelligent Agents on the Semantic Grid*, Dragoni, N., Gaspari, M., Guidi, D., April 2005.

2005-10 *Fault Tolerant Knowledge Level Communication in Open Asynchronous Multi-Agent Systems*, Dragoni, N., Gaspari, M., April 2005.

2005-11 *The AEDSS Application Ontology: Enhanced Automatic Assessment of EDSS in Multiple Sclerosis*, Gaspari, M., Saletti, N., Scandellari, C., Stecchi, S., April 2005.

2005-12 *How to cheat BitTorrent and why nobody does*, Hales, D., Patarin, S., May 2005.

2005-13 *Choose Your Tribe! - Evolution at the Next Level in a Peer-to-Peer network*, Hales, D., May 2005.

2005-14 *Knowledge-Based Jobs and the Boundaries of Firms: Agent-based simulation of Firms Learning and Workforce Skill Set Dynamics*, Mollona, E., Hales, D., June 2005.

Tag-Based Cooperation in Peer-to-Peer Networks with Newscast[1]

Andrea Marcozzi[2] **David Hales**[2] **Gian Paolo Jesi**[2] **Stefano Arteconi**[2]
Özalp Babaoğlu[2]

Technical Report UBLCS-2005-15

June 2005

Abstract

Recent work has proposed how socially inspired mechanisms, based on "tags" and developed within social science simulations, might be applied in peer-to-peer overlay networks to maintain high cooperation between peers even when they act selfishly. The proposed mechanism involves a dynamic re-wiring algorithm called the "SLAC" algorithm. So far the algorithm has been demonstrated by simulation in abstracted task domain on non-publicly available testbeds. Also, the algorithm assumes a random sampling service over the entire population of nodes but does not implement this itself.

In this paper we re-implement SLAC on an open source peer-to-peer simulation testbed called "PEERSIM". In order to provide the random sampling service we utilise an existing protocol called "NEWSCAST" which has already been implemented on the PEERSIM platform. We present the results of some experiments we performed in which peers play the Prisoner's Dilemma game with their neighbours.

Our results demonstrate that SLAC does indeed produce high levels of cooperation running in the PEERSIM / NEWSCAST environment. This increases our confidence that previous results from SLAC are generally applicable and valid and also that SLAC could have applications in real implemented systems.

Finally we discuss the open issues that need to be addressed for SLAC to progress to a valuable deployable protocol.

1. Based on the Laurea thesis of Andrea Marcozzi, March 2005. Partially supported by the EU within the 6th Framework Programme under contract 001907 "Dynamically Evolving, Large Scale Information Systems" (DELIS).
2. Dept. of Computer Science, The University of Bologna, Mura Anteo Zamboni 7, 40127 Bologna, Italy.

1 Introduction

In recent works a novel socially inspired algorithm based on the ”tag” idea [6] has be applied to the problem of sustaining cooperation in peer-to-peer networks composed f nodes behaving selfishly [3].

In the earlier tag models, individual agents interact randomly (in the form of mean-field interaction) under the constraint that they are more likely to interact with agents haring an identical or similar tag.

In the context of the social scientific interpretation tags represent arbitrary surface markings ttached to agents that can be observed and copied by other agents. They have no irect behavioural significance [6]. In human society tags can be viewed as fashions: styles of dress, colours of hat, brand logos or cosmetic makeup. The key property of tags s that although they are distinctive and immediately observable they can be quickly changed and copied.

A number of simulation models have demonstrated that over a broad range of parameter values high levels of cooperation and altruism emerge when agents bias their interaction towards others sharing the same tag [4, 10]. These previous models follow an evolutionary approach in which agents reproduce and mutate their behaviours and tags. In such models those gaining higher utility are more likely to reproduce their tags and behaviours with the addition of small amounts of mutation - noise or random variation. The assumptions underlying this approach is that such an evolutionary mechanism can capture the essential elements of cultural learning within a society - that agents are bounded optimisers, copying the tags and behaviours of those who are gaining higher utility than themselves.

Although such models have stimulated debate concerning altruism in human and animal social systems [10, 12, 11], the evolutionary approach needs to be adapted in order to apply the technique to practical applications in distributed computer systems.

We have previously described, in detail, how we transformed (via a series of simulation models) the evolutionary models into an algorithm more applicable to a target application of peer-to-peer file sharing [3]. However, in that work the final algorithm (the SLAC algorithm) was tested on a non-publicly available and highly abstracted simulation testbed. Also, SLAC relies on a random sampling service over the entire P2P network and this was assumed rather than implemented.

In this paper we describe a re-implementation of, and experiments with, SLAC on the open source PEERSIM system [14]. PEERSIM offers a more realistic P2P simulation environment - protocols previously tested on it have been successfully implemented [18]. In addition the missing random sampling service was provided by the NEWSCAST protocol which is already implemented on PEERSIM. Both SLAC and NEWSCAST are highly scalable (up-to millions of nodes), robust (recovering from noise and the removal of nodes) and completely decentralised (requiring no centralised services).

In the following sections we describe the original tag-based evolutionary algorithm and the derived SLAC P2P algorithm. We then discuss briefly the application task we tested our system with - the Prisoner’s Dilemma game followed by an overview of the PEERSIM P2P simulation environment and NEWSCAST protocol. We then discuss the SLAC implementation details and present the results of experiments performed. Finally we conclude with a discussion of the results and review some of the open issues that need to be addressed in order to progress towards a valuable deployable implementation.

2 Previous Tags Models

The basic algorithm has been adapted from previous (quite different) simulation work using “tags”. This work demonstrates a novel method of maintaining high levels of cooperation in environments composed of selfish, adaptive agents. The emphasis of the previous work has been towards understanding biological and social systems [4]. Tags are markings or social cues that are attached to individuals (agents) and are observable by others, often represented in models by a single number, they evolve like any other trait in a given evolutionary model. The key

point is that the tags have no direct behavioral implication for the agents that carry them. But through indirect effects, such as the restriction of interaction to those with the same tag value, they can evolve from initially random values into complex ever changing patterns that serve to structure interactions. The simulated environments in which tags have been applied have generally been very simple with interactions based on pair-wise games with immediate payoffs. Nevertheless, we have attempted to adapt the salient features of such tag systems for application in P2P networks. These features are that agents:

- restrict interact to those with whom they share a group defined by tag value
- selfishly and greedily optimize by preferentially copying the behavior and tag of others with higher utility
- periodically mutate their tags and behaviors

By copying and mutating tags, agents effectively move between interaction groups. By restricting interaction within groups free riders tend to kill (reduce the membership) of their own group over time because exploited agents will tend to move elsewhere to get better payoffs, while cooperative groups tend to spread via mutation of the tag. Previous tag models have demonstrated high levels of cooperation in "commons tragedy" [5] scenarios (e.g. in the Prisoners Dilemma – see below). We will not cover the results of the previous tag models in detail here, since the emphasis is not relevant and space precludes detailed treatment, rather we will present our newly derived algorithm (based on the salient features outlined above) and the results we obtained when applying it to two different simulated P2P scenarios.

3 Cooperation and the Prisoner's Dilemma

Distributed P2P applications often require that nodes behave cooperatively or altruistically to help others in the network. For example, in a file-sharing system, nodes are required to host and upload files on demand to other nodes that require them. Also they need to reply to queries concerning what files they host. But why should nodes do this? In an open system there is an incentive for nodes the behave selfishly - saving their own storage and bandwidth but using other nodes. This problem is not limited to file-sharing because any application that requires other peers to perform actions on their behalf, in some sense, relies on a degree of cooperation from those others. Obviously cooperative behaviours can be built into the peer client software but in an open system how can we ensure that such software will not be changed?

The fundamental issue, then, is: *how can one maintain cooperative (socia lly beneficial) interactions within an open system under the assumption of high individual (peer) autonomy*. An archetype of this kind of social dilemma has been developed in the form of a minimal game called the Prisoner's Dilemma (PD) game. Economic and social scientists have often deployed this minimal game as a canonical form of the contradiction that can arise between individual and collective interests.

In the PD game two players each selected a move from two alternatives and then the game ends and each player receives a score (or pay-off). Figure 1 shows a so-called 'pay-off matrix' for the game. If both choose the 'cooperate' move then both get a 'reward' — the score R. If both select the 'defect' move they are 'punished' — they get the score P. If one player defects and the other cooperates then the defector gets T (the 'temptation' score), the other getting S (the 'sucker' score). When these pay-offs, which are numbers representing some kind of desirable utility (for example, money), obey the following constraints: $T > R > P > S$ and $2R > T + S$ then we say the game represents a Prisoner's Dilemma (PD). When both players cooperate this represents maximising of the collective good but when one player defects and another cooperates this represents a form of free-riding. The defector gains a higher score (the temptation) at the expense of the co-operator (who then becomes the 'sucker').

A game theoretic analysis drawing on the Nash equilibrium solution concept (as defined by the now famous John Nash [9]) captures the intuition that a utility maximising player would

	Cooperate	Defect
Cooperate	R, R	S, T
Defect	T, S	P, P

Figure 1. A payoff matrix for the two-player single round Prisoner's Dilemma (PD) game. Given $T > R > P > S \wedge 2R > T + S$ the Nash equilibrium is for both players to select Defect but both selecting Cooperate would produce higher social and individual returns. However, if either player selects Cooperate they are exposed to Defection by their opponent — hence the dilemma

always defect in such games because whatever the other player does a higher score is never attained by choosing to cooperate.

In the context of a P2P system how do we solve this problem without going back to centralised control or closed systems? In the following section we describe the "tag" inspired SLAC algorithm.

4 The SLAC algorithm

The SLAC algorithm [3] assumes that peer nodes have the freedom to change behavior (i.e. the way they handle and dispatch requests to and from other nodes) and drop and make links to nodes they know about. In addition, it is assumed nodes have the ability to discover other nodes randomly from the network, compare their performance against other nodes and copy the links and (some of) the behaviors of other nodes.

As discussed previously we assume that nodes will tend to use their abilities to selfishly increase their own utility in a greedy and adaptive way, that is if changing some behavior or link increases utility then nodes will tend to select it. The algorithm relies on Selfish Link and behavior Adaptation to produce Cooperation (SLAC) - a task domain independent outline is given below.

Over time nodes engage in some activity and generate some measure of utility U. This might be number of files downloaded or jobs processed etc, depending on the domain. Periodically, each node (i) compares its performance and compares this against another node (j), randomly selected from the population. If $Ui < Uj$ node i drops all current links and copies all node j links and adds a link to j itself - see figure 2.

Also, periodically, and with low probability, each node adapts its behavior and links in some randomized way using a kind of mutation operation. Mutation of the links involves removing all existing links and replacing them with a single link to a node randomly drawn from the network. Mutation of the behavior involves some form of randomized change - the specifics being dictated by the application domain. Previous tag models, on which SLAC is based have indicated that the rate of mutation applied to the links needs to be significantly higher than that applied to the behavior - by about one order of magnitude [2].

When applied in a suitably large population, over time, the algorithm follows a kind of evolutionary process in which nodes with high utility tend to replace nodes with low utility (with nodes periodically changing behavior and moving in the network). However, as will be seen, this does not lead to the dominance of selfish behavior – as might be intuitively expected – since a form of incentive mechanism emerges via a kind of ostracism in the network.

5 The Peersim system

Evaluating the performance of P2P protocols is a complex task. One of the main problems for their evaluation, is the extremely large scale that they may reach. P2P networks involving hundred of thousands of peers (or more) are not uncommon (e.g., about 5 millions machines are reported to be connected to the Kazaa/Fasttrack [16] network). In addition P2P systems are highly dynamic environments; they are in a continuous state of flux, with new nodes joining and leaving (or crashing).

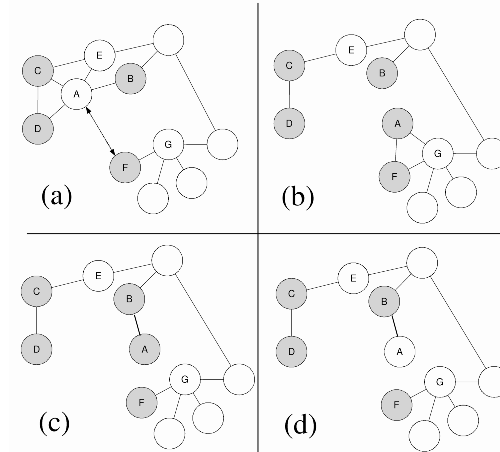

Figure 2. An illustration of 'replication' and 'mutation' as applied in the Selfish Link-based Adaptation for Cooperation (SLAC) algorithm from [3]. Shading of nodes represents strategy. In (a) the arrowed link represents a comparison of utility between A and F. Assuming F has higher utility then (b) shows the state of the network after A copies Fs links and strategy and links to F. A possible result of applying mutation to As links is shown in (c) and the strategy is mutated in (d).

These properties are very challenging to deal with. Evaluating a new protocol in a real environment, especially in its early stages of development, is not feasible. Distributed planetary-scale open platforms (e.g., Planet-Lab [17]) to develop and deploy network services are available, but these solutions do not include more than about 500 (at the time of writing) nodes. Thus, for large-scale systems, a scalable simulation testbed is mandatory.

The Peersim P2P simulator [14] has been developed with the aim to deal with the previously stated issues. Its first goals are: extreme scalability and support for dynamicity. It is a GPL open-source Java based software project; it is an efficient solution and easy configurable. Peersim has proved to be a valuable tool and it is used as the default experimentation platform in the BISON project [15]. In the following, we provide a brief description of its characteristics.

5.1 Peersim Design Goals

The Peersim simulator is inspired by mainly two objectives:

- **High scalability**: P2P networks may be composed by millions of nodes. This result can be achieved only with a careful design of the data structures involved, trying to avoid (when possible) any overhead. But the memory footprint is not the only problem: the simulator engine must be also efficient.

- **Support for dynamicity**: the simulator must manage nodes joining and leaving the network at any time; this feature has tightly relations with the engine memory management sub-system.

Another important requirement is the *modular* or *component* inspired architecture. Every entity in the simulation (such as protocols and the environment related objects) must be easily replaceable with similar type entities. It is important to note that in our vision, the term "components" has no relation with high level component (distributed) architectures such as CORBA or DOM+.

The Peersim extreme performances can be reached only accepting some relaxing assumptions about the simulation details. For example, the overhead introduced by the low level communication protocol stack (e.g., TCP or UDP) in not taken into account because of the huge additional memory and CPU time requirements needed to accomplish this task.

while(TRUE) do	**while(TRUE) do**

wait(Δt);
neighbor = SELECTPEER();
SENDSTATE(neighbor);
n.state = RECEIVESTATE();
my.state.UPDATE(n.state);
	n.state = RECEIVESTATE();
	SENDSTATE(n.state.sender);
	my.state.UPDATE(n.state);

(a) Active Thread	**(b) Passive Thread**

Figure 3. The gossip paradigm.

5.2 Peersim Architecure

As previously stated, Peersim is inspired by a modular and very configurable paradigm, trying to limit any unnecessary overhead. The simulator main component is the *Configurator* entity targeted to read configuration files. A configuration file is a plain ASCII text file, basically composed by key-value pairs; it can also contain variables and basic math expression evaluated at the time of reading. The Configurator is the only not interchangeable simulation component. All the other entities can be easily customized.

In a Peersim simulation, the following three distinct kind of elements can be present: protocols, dynamics and observers. Each of them is implemented by a Java class specified in the configuration file. The network in the simulation is represented by a collection of nodes and each node can hold one or more protocols. The communication between node protocols is based on method calls. To provide a specific kind of service, each component must implement a specific *interface*. For example a protocol has to implement at least the `Protocol` or `CDProtocol` interface to run on Peersim.

The dynamism is introduced by the `Dynamics` interface implementing components. They are scheduled by the engine to run periodically (fine tunable by the configuration file). Dynamics have the global knowledge of the system and can change whatever protocol aspect (such as changing internal parameters or shutting-down nodes).

The `Obsever` implementing interface components are targeted to monitor and analyze the network. The observers are executed periodically like dynamics.

Peersim has also an utility class package to perform statistic computations or to provide some starting topology configuration based on well know models (such as: random-graph, lattice, BA-Graph,...).

The *Simulator* engine is the component that performs the computation; it has to run the component execution according to the configuration file instructions. At the time of writing, Peersim can perform simulation according to the following execution models:

- **Cycle based**: at each step, all nodes are selected in a random fashion and each node protocol is invoked in turn;
- **Event based**: a support for concurrency is provided. A set of events (messages) are scheduled in time and node protocols are run according to the time message delivery order.

This paper work is based on the first simulation model.

6 The Newscast protocol

Newscast [8] is a gossip-based protocol that builds and maintains a continuously changing random graph (or *overlay*). The generated topology is very stable and provides robust connectivity. This protocol has been used successfully to implement several P2P protocols, including broadcast [8] and aggregation [7].

The Newscast state is represented by a partial, fixed c size view of node *descriptors* composed by a node address and a logical *time-stamp* (descriptor creation time).

Referring to the usual gossip scheme (see Figure 3), the protocol behaviour perform the following actions: selects first a neighbour from the local view, exchanges the state with the neighbour, then both participants update their actual state according to the received state. Even though the system is not synchronous, we find it convenient to describe the gossip-scheme execution as a sequence of consecutive real time intervals of length Δ, called *cycles* or *rounds*.

In Newscast, the neighbour selection process is performed in a random fashion by the SELECTPEER() method. The UPDATE() method merges a received view (sent by a node using SENDSTATE()) with the current peer state view. Then, Newscast trims the merged view to obtain the previous c size. The node descriptors discarded are chosen from the most "old" ones (according to the descriptor time-stamp). This approach changes continuously the node descriptors hold in each node view; this implies a continuous rewiring of the graph defined by the set of all node views. This behaviour is shown in Figure 4.

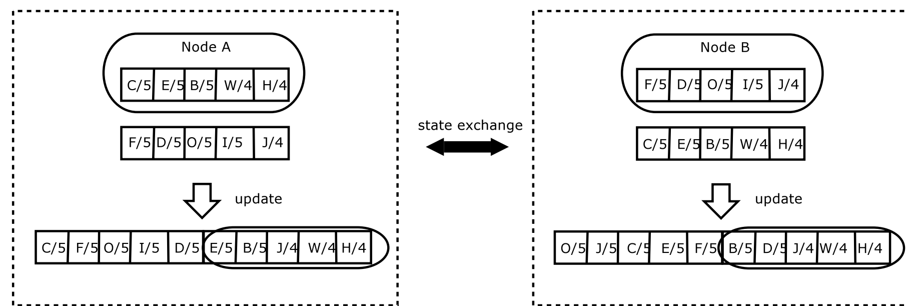

Figure 4. A Newscast exchange between node A (active) and B. Each node has its own 5 descriptor elements view depicted inside the ellipses. A descriptor is a *node-ID*, *timestamp* pair. After the state exchange node A has received the node B view and viceversa; then each partecipant merges the received view with its own. The result is depicted under the empty arrow: each node has selected the "freshest" descriptors at random and has discarted the others (those inside the ellipse) to obtain new 5 element view. Note taht in this basic example, each node sends its entire view; however, the view can be purged by "old" descriptors before sending.

The protocol always tends to insert new informations in the system allowing an automatic elimination of old node descriptors. This feature is particularly desirable to remove crashed node descriptors and thus to repair the overlay with minor efforts. In addition, the protocol does not require any clock synchronization, but only that the timestamps of node descriptors in each view are mutually consistent. To achieve this goal, when a node a sends its view to node b, it sends also its current time-stamp t_a. Then, node b can normalize its time-stamp entry descriptors with the value: $t_b - t_a$. Note that this mechanism is not critical.

The topology generated by Newscast has a low diameter and it is close to a random graph having out-degree c. Experimental results proved that a small 20 elements partial view is already sufficient for a very stable and robust connectivity.

Newscast is also cheap in terms of network communication. The traffic generated by the protocol is estimated in [8]. Summarizing very briefly, the number of exchanges per cycle can be modeled by the random variable 1+ϕ, where ϕ has a Poisson distribution with parameter 1. Thus, on average, we expect two exchanges per cycle (one active and one passive). This involves the exchange a few hundred bytes per cycle for each peer.

A protocol such as Newscast implements a service to pick random nodes from the whole network and we can call it *Randomizer Service*. This provided feature make easier the development of higher level P2P protocol built on top of Newscast providing a valuable fresh node source.

In this vision, the cost effectiveness of Newscast is very useful, because the *Randomizer Service* can be considered a basic and always-on service run by all peers in the network.

Such a randomizer service can also be a key component during the initialisation phase (*bootstrap*) for any higher level protocol and we use this service in the Peersim implementation of the SLAC algorithm. We describe this implementation in the next section.

7 Implementing SLAC in Peersim

Now we purpose to implement and test the solutions proposed by the SLAC algorithm into the Peersim platform, on top of Newcast. We have already seen that Newscast has lots of important properties such as its scalability, its robustness and its ability to maintain a random topology; it's also already implemented in Peersim. This is why it would be interesting to make some test using such infrastructure and that's why we have based our experiments on such protocol.

7.1 The PdProtocol (an overview)

As we have already seen, Peersim is highly modular. So it wasn't so hard to implement a new protocol based on the NetWorld model specifications compatible with the Peersim structure. The name of this protocol is **PdProtocol**.

The simulation, as stated in Peersim documentation [14], is performed through a series of cycles and in each of these some operations are performed. During the simulation our protocol is involved in three phases:

- **phase 0**: initialization of an auxiliary array for neighbors list (nodes are provided by Newscast);
- **phase 1**: the PD game is played and the appropriate payoffs are distributed;
- **phase 2**: two nodes are randomly chosen from the network (these are provided by Newscast) and their payoffs are compared (this is the reproduction phase).

These cycles are more realistic than the "reproduction cycle" algorithms from [3].

In our experiments, our protocol runs on top of Newscast. The key point of this implementation is the idea of copying the neighbors list, generated during the initialization task already implemented in Peersim, to an auxiliary array in order to make the movement of the nodes as easy as possible. This operation (phase 0) is performed at the first cycle, before the system starts evolving.

In phase 1 the PD game is played between nodes. Each node is initialized with a random value chosen from a set of two (True = Cooperate, False = Defect) and at each cycle each node plays a round of the PD game with one of its neighbors randomly chosen. After this game interaction, payoffs are calculated and distributed.

With phase 2 is performed the reproduction task. Every I cycles, two nodes are randomly chosen from the network and their average payoffs are compared. After this comparison, the losing node copies the winner's strategy and it is also moved to its neighborhood.

We implemented three classes in Peersim, whose names are: **PdDistributionInitializer**, **PdObserver** and of course **PdProtocol**. With the first one we just want to initialize the nodes of the network with a strategy (cooperate or defect), while the second one is used to calculate and print results from the simulations. **PdProtocol** instead, is the core of our simulator and hence deserves a more complete discussion.

7.2 PdProtocol details

Class PdProtocol of course implements *CDProtocol* because it defines the operation to be performed at each cycle. This operations are stated in the function *nextCycle* and, as told before, they walk through three phases.

Function *nextCycles* has as arguments a *node* and the relating *protocolID*. In each of the three phases a node is taken into account and this node is the one the function holds as first argument.

At first cycle, after the initialization step has been performed, the neighbor lists of the nodes are copied from Newscast to PdProtocol (see code below):

```
1Linkable linkable = (Linkable) node.getProtocol(lpid);
2   for (int i=0; i<linkable.degree(); ++i)
3     {
4         this.nList.add(linkable.getNeighbor(i));
5     }
```

On line 1 the *"lpid"* protocol identifier is taken from the running node; *lpid* is referred to the Newscast protocol, while *protocolID* is referred to PdProtocol. With this operation we get the neighbors of the nodes from the Newscast protocol and put them into the auxiliary array that is accessible by PdProtocol.

A node can be seen as a container of protocols which can be accessed through a protocol identifier as *lpid* or *protocolID*. From a node it is possible to get access to all the data of a specified protocol through the function *getProtocol(pid)*. Figure 5 represents what happens with our program.

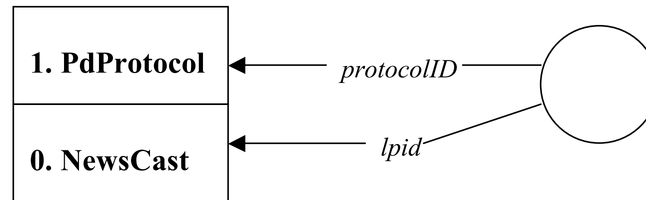

Figure 5. Protocols handling from a node.

From the second cycle the PD game is played. In this phase we only deal with *protocolID*. The node taken into account is the first argument of the function *nextCycle*; from the neighbor list of this node a peer is randomly selected and a round of the PD game is played between them. Their strategies are compared and the appropriate payoffs are distributed. After the game is played, the average payoff of each node is calculated and stored.

Every I cycles (in our experiments we mainly use $I = 4$) phase 2 takes place. With this phase we want to compare the average payoffs of two nodes chosen within the network. The first node we take into account is, as usual, the node that's the first argument of the function *nextCycle*; the second one, as in phase 0, is taken from Newscast: it's a randomly chosen neighbor of the Newscast instance of the first node. Once these nodes (i, j) are selected, reproduction phase can start. Let's consider the case in which node i has a higher average payoff than j, the following operations are performed:

- all j's neighbors delete their links to j itself; [3]
- j's neighbors list is cleared;

3. Actually, in this implementation of SLAC, this first action was not implemented, interestingly, this change did not stop high cooperation from emerging.

- j's neighbors list is filled in with new items: i's neighbors;
- the winner itself (i) is added to j's neighbors list;
- strategy is copied from i to j;
- some variables held by j are cleared (e.g. number of games played);
- j is added to the respective neighbors lists of the winners neighbors;
- j itself is added to i's neighbors list;
- **mutation** is applied with a certain probability.

Neighbors lists have a fixed size F, so it's not permitted to add nodes when such limit is reached. That's why this operation is performed by an appropriate function which makes not exceed such limit: before adding a new node, a check on the actual size of the list is performed; if such value is equal to F, a randomly chosen node is deleted and then the new one is added, else, if the actual size is smaller than F, the new node is simply added. These operations stand for movement between neighborhoods.

This is what happens when i has an average payoff greater than j. Of course if j had an average payoff greater than i, the same algorithm would be performed but with i and j in inverted positions. But what does it happens when the two nodes hold the same value? The winner is randomly chosen between the two nodes and then the same operations listed above are performed. That's for introducing some noise in the model.

8 Simulation Experiments and Results

A series of experiments were done with this new "tag-based" protocol on Newscast. Obtained results relate to experiments done varying several parameters: number of cycles, size of the network, mutation rate, seed for the simulation, payoffs, strategy initialization task, etc..

In the next subsections we'll show some of these results and draw some conclusions.

8.1 Settings

Peersim needs to have its configuration file properly set before a simulation starts. For our experiments we mainly used the configuration file shown in the box below, sometimes changing some parameters.

```
1  # PEERSIM EXAMPLE 1
2  random.seed 1234567890
3  simulation.cycles 300
4  simulation.shuffle
5
6  overlay.size 40000
7  overlay.maxsize 120000
8
9  protocol.0 example.newscast.SimpleNewscast
10 protocol.0.cache 20
11
12 protocol.1 example.aggregation.PdProtocol
13 protocol.1.linkable 0
14
15 init.0 peersim.dynamics.WireRegularRandom
16 init.0.protocol 0
17 init.0.degree 20
```

```
18
19 init.1 example.loadbalance.PdDistributionInitializer
20 init.1.protocol 1
21
22 observer.0 example.aggregation.PdObserver
23 observer.0.protocol 1
```

At line 2 is set the seed used. In the experiments we propose we changed only one parameter per time, in order to emphasize the differences between an experiment and another one.

At line 3 and 6 we set respectively the number of cycles we want to perform and the size of the network. Line 9 indicates that we want our protocol to run on top of Newscast. Line 15 indicates that the topology of our network is always set at random and line 17 indicates that the degree of the network that is going to be created is 20.

1st player	Strategies	2nd player
1	DC	0
0.8	CC	0.8
0.1	DD	0.1
0	CD	1

Table 1. PD payoffs adopted in the model.

Other parameters are then set for each simulation, these are: payoffs (in the greater part of our experiments, we used those shown in table 1), the mutation rate (both for the strategy MR and the tag MRT) and the number of cycles occurring between reproduction phases (indicated with I). We mainly use $MR = MRT = 0.001$ and $I = 4$. In the next section we give two interesting results.

8.2 Cooperation with different network sizes

Results shown in figures 6 and 7, relates experiments done on networks having different sizes.

The diagram in figure 6 represents the number of cycles needed to obtain high levels of cooperation (about 93%) over a series of 1000 cycles. The diagram compares the results obtained performing the experiments on four different network size. For each network were performed experiments with different seeds and the results shown in the figure represents the average number of cycles needed to obtain cooperation and the standard deviation for each network. The seeds were chosen in order to not create correlation between different experiments. The network was started from complete defection, the mutation rate used was the same both for the strategy and the neighbor list ($MR = MRT = 0.001$) and the number of cycles occurring between a reproduction phase and the next one was $I = 4$. Let's note the average cycles number for a network of 4000 nodes: it is much higher than the average for the other network sizes and this is because with seed 1, the 93% of cooperation was obtained just after 772 cycles. The same experiment was also performed with a different seeds table but the results we obtained are nearly identical to those just given.

Figure 7 gives the percentage of cooperating nodes over a series of 4000 cycles. The figure just shows the first 150 cycles since after that point there are no relevant changes in the results. Even here the mutation rate was the same both for the strategy and the tag, experiments were performed over different network size and parameter I was set to 4. On the contrary of the previous experiment, here was used always the same seed, hence the percentages we give are not an average. Nodes were initialized at random: at the beginning of the simulation we had a population composed of about half cooperating nodes and half defecting nodes; after the first few cycle, the percentage of cooperating nodes decreased but soon after the 23th cycle it started increasing toward good levels of cooperation.

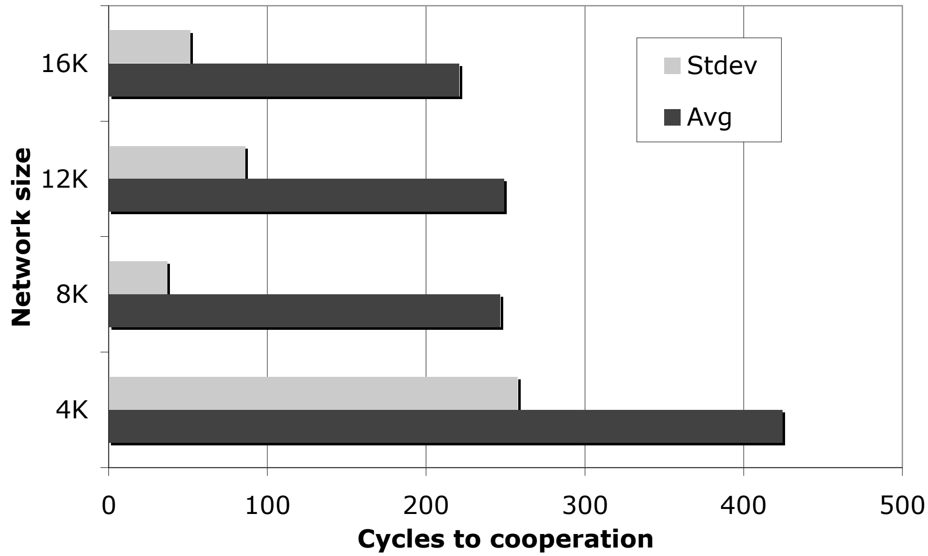

Figure 6. Average number of cycles needed to obtain high levels of cooperation (about 93%) and the realtive standard deviation with four different seeds. MR = MRT = 0.001, I = 4, payoffs from table 1. Results from different network size are compared.

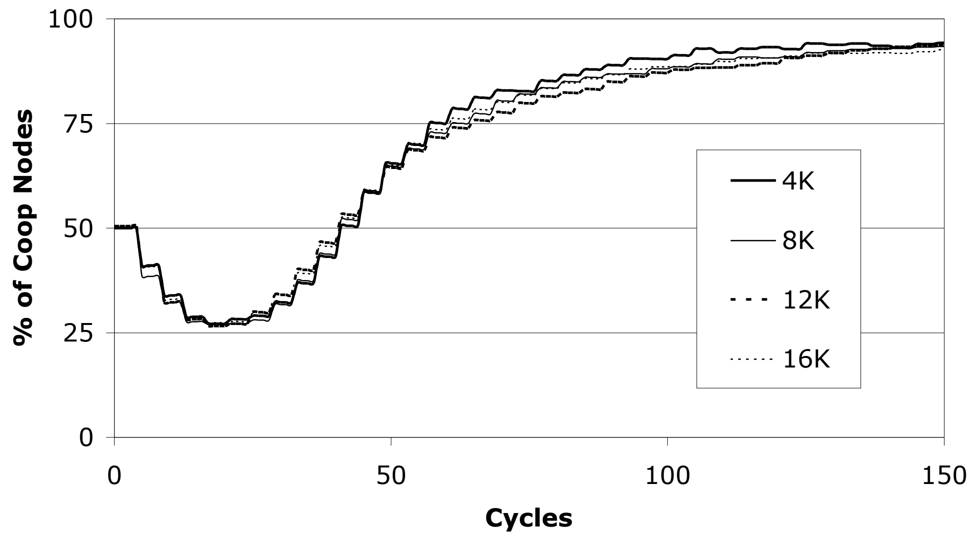

Figure 7. Cooperating nodes over a series of 4000 cycles (single run). MR=MRT=0.001, I=4, payoffs from table 1. Nodes are initialized with a random strategy. Results after cycle 150 do not increase significantly. Note: the "staircasing" effect is an artefact of the synchronous reproduction at every 4 cycles - with asynchronous reproduction the artefact is not visible.

8.2.1 Observations

From both the diagrams it's easy to note the good level of scalability of the model we are testing: results of figure 6 are an average of results obtained with different seeds and we succeeded in obtaining similar results for three different network size. The only difference is found with the network composed of 4000 nodes where we obtained a very high value with just one seed. It would be interesting to make further experiments with more seeds. Figure 7 gives the same important result: here using always the same seed, we obtained the same trend with all the sizes.

8.3 Cooperation with long runs

Some experiments were done with a big number of cycles. On a network composed of 4000 nodes were performed a series of experiments; for each of them we used a different random seed, $MR = MRT = 0.001$, $I = 4$ and performed 10000 cycles.

Results are very close for each seed used, hence in figure 8 we just propose those obtained with one of them. The diagram shows that good levels of cooperation can be obtained from cycle 645 (95.5 %): from this cycle to cycle 10000 the average percentage of cooperating nodes is 95.40 and the standard deviation is 1.03.

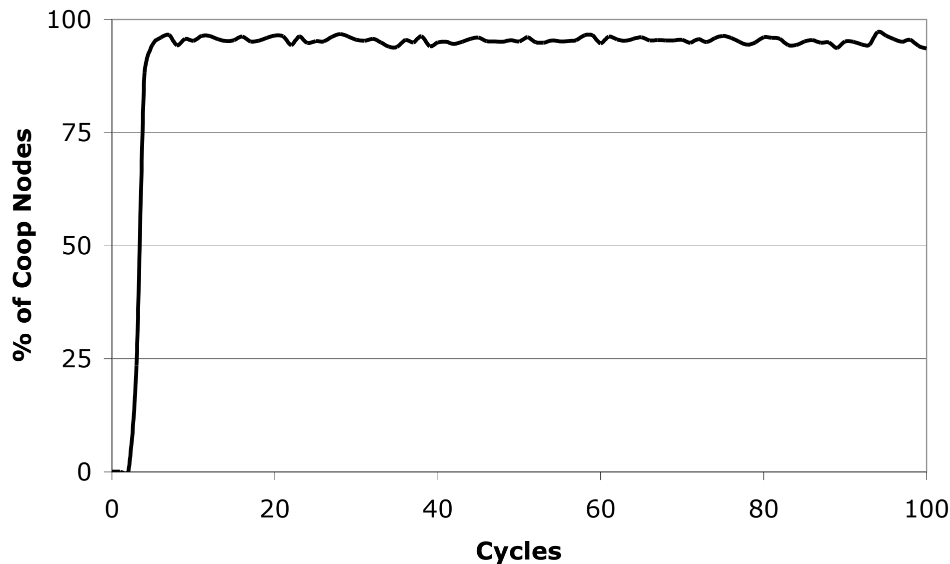

Figure 8. Cooperation with a series of 10000 cycles (single run). Nodes = 4000, MR=MRT=0.001, I=4, payoffs from table 1.

8.3.1 Observations

The experiments just proposed have a great importance since they test the reliability of the model in the time. From all the tests we made (some even with 50000 generations), and also in those proposed in this section, we found that once cooperation has started, it never claps and can be sustained for long times; we also learned that cooperation can be sustained at good levels.

9 Discussion and Conclusion

The results we obtained indicate that high cooperation is produced when nodes follow the SLAC algorithm. Even though the SLAC algorithm implements nodes that behave selfishly in a myopic

and greedy way - that is, they copy other nodes in the network that have higher utility - high levels of cooperation are produced in the single round Prisoner's Dilemma (PD) game.

These new results therefore confirm those results previously found[1, 3] in similar simulation experiments and this adds confidence that SLAC is robust to different simulation implementation details. Here, for example, we interleaved the reproduction phase with the interaction phase whereas in previous simulations reproduction followed at the end of each cycle of interaction. Also in further experiments (not shown here) we found that similar results were obtained when reproduction was interleaved with interaction in a fully asynchronous way - where each node has a probability of reproduction after interaction.

Interestingly, the results shown in figure 6 appear to recover some of the reverse scaling properties demonstrated in an early non-network based tag model [4] which appeared to have been lost in our initial network model [1]. However, more runs and analysis are needed to explore this question. However, we can certainly state that in all the experiments so far performed with the SLAC algorithm larger networks do not take longer to converge and often converge more quickly. This is obviously a valuable property for any candidate algorithm for large scale systems.

A further finding of these new results appears to contradict earlier generalisations [2] that tag-type models needed to have higher mutation rates on the "tag" than the strategy - in the case of SLAC this would mean a higher mutation rate on the neighbour list or view which contains the links to neighbour nodes than the behavioural strategy of the node (either to cooperate or defect). But here high levels of cooperation were produced when the mutation rate was the same. This indicates that further work is needed to circumscribe such a generalisation since it is currently unclear what difference in implementation has allowed this assumption to be relaxed.

These results also demonstrate that the NEWSCAST protocol can be used to provide the random sampling service required by SLAC but not previously explicitly modelled in simulation. This is important since any actual implementation of SLAC must have access to such a service that is both scalable and robust. NEWSCAST provides such a service[8] with the additional benefit that it has actually been tested in the form of a real implementation [*ref newscast imp].

We have argued, and demonstrated previously, that cooperation in the single round PD indicates that cooperation can be produced in other more realistic task domains [3]. We are therefore confident that these results indicate that the PEERSIM implementaion could support cooperation in other task domains (such as file sharing or other kinds of resource sharing).

Finally, we note two major issues that could destroy cooperation within SLAC. Firstly, we currently assume that nodes are able to compare utilities correctly, that is, we assume nodes report their utilities honestly when requested to do so by nodes. But what would happen if nodes lied about their utilities or just failed to report anything? This introduces a kind of "second order" free-rider problem at the informational level because if we assume nodes may behave selfishly and / or maliciously then we need to demonstrate individual incentives for supplying correct utility values. Secondly, we also assume nodes will allow themselves to be copied by supplying their behaviour strategy and their current neighbour list or view (containing their node links) to other nodes. Again, this may not be case with malicious and selfish nodes in certain contexts. Both of these issues we aim to address in future work.

Acknowledgements

This work would not have been possible without perceptive discussions with many people, particularly those in the Bologna group including: Mark Jelasity, Alberto Montresor and Simon Patarin.

References

[1] D. Hales. Self-Organising, Open and Cooperative P2P Societies From Tags to Networks. *Proceedings of the 2nd Workshop on Engineering Self-Organising Applications (ESOA 2004), LNCS 3464*, pp.123-137. Springer, 2005.

[2] D. Hales. Change Your Tags Fast! a necessary condition for cooperation? *Proceedings of the Workshop on Multi-Agents and Multi-Agent-Based Simulation (MABS 2004), LNAI 3415*. Springer, 2005.

[3] D. Hales. From selfish nodes to cooperative networks – emergent link based incentives in peer-to-peer networks. In *Proc. of the 4th IEEE International Conference on Peer-to-Peer Computing (P2P2004)*. IEEE Computer Soc. Press, 2004. Available at: http://www.davidhales.com

[4] D. Hales. Cooperation without Space or Memory: Tags, Groups and the Prisoner's Dilemma. In *Moss and Davidsson (eds.) Multi-Agent-Based Simulation.* LNAI 1979:157-166. Springer. Berlin. 2000

[5] G. Hardin. The tragedy of the commons. *Science*, 162, 1243-1248. 1968

[6] J. Holland. The Effect of Lables (Tags) on Social Interactions. *Santa Fe Institute Working Paper 93-10-064*. Santa Fe, NM 1993

[7] M. Jelasity and A. Montresor, Epidemic-Style Proactive Aggregation in Large Overlay Networks, in *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, March 2004, pp. 102–109, IEEE Computer Society, Available at: http://www.cs.unibo.it/bison/publications/icdcs04.pdf

[8] M. Jelasity and W. Kowalczyk and M. van Steen Newscast Computing *Technical Report IR-CS-006*, Vrije Universiteit Amsterdam, Department of Computer Science, November 2003, Available at: http://www.cs.vu.nl/globe/techreps.html#IR-CS-006.03

[9] J. F. Nash. Equilibrium Points in N-Person Games, *Proc. Natl. Acad. Sci.* USA 36, 48-49, (1950).

[10] R. Riolo, M. D. Cohen, R. Axelrod. Cooperation without Reciprocity. *Nature* 414, 441-443, (2001).

[11] G. Roberts and T. N. Sherratt. Similarity does not breed cooperation. *Nature* 418, 449-500, 2002.

[12] K. Sigmund and M. Nowak: Tides of tolerance. *Nature* 414, 403-405, (2001).

[13] R. Tivers. The evolution of reciprocal altruism. *Q. Rev. Biol.* 46, 35-57. 1971

[14] Peersim Peer-to-Peer Simulator, Available at: http://peersim.sf.net

[15] The BISON Project, http://www.cs.unibo.it/bison

[16] Kazaa Web Site, http://www.kazaa.com

[17] PlanetLab Planetary-Scale Testbed, http://www.planet-lab.org

[18] T. Binci. EpidEm: EPIDemic EMulator, Graduate Thesis in Computer Science, University of Bologna, Department of Computer Science. Available at: http://bincit.web.cs.unibo.it/index.htm