A Modular Paradigm for Building Self-Organizing Peer-to-Peer Applications*

Márk Jelasity**, Alberto Montresor, and Ozalp Babaoglu

Department of Computer Science, University of Bologna, Italy
{jelasity,montresor,babaoglu}@cs.unibo.it

Abstract. Peer-to-peer (P2P) technology has undergone rapid growth, producing new protocols and applications, many of which enjoy considerable commercial success and academic interest. Yet, P2P applications are often based on complex protocols, whose behavior is not completely understood. We believe that in order to enable an even more widespread adoption of P2P systems in commercial and scientific applications, what is needed is a modular paradigm in which well-understood, predictable components with clean interfaces can be combined to implement arbitrarily complex functions. The goal of this paper is to promote this idea by describing our initial experiences in this direction. Our recent work has resulted in a collection of simple and robust components, which include aggregation and membership management. This paper shows how to combine them to obtain a novel load-balancing algorithm that is close to optimal with respect to load transfer. We also describe briefly our simulation environment, explicitly designed to efficiently support our modular approach to P2P protocol design.

1 Introduction

Recent years have witnessed a rapid growth in both the body of scientific knowledge on peer-to-peer (P2P) technology and its commercial applications [10]. There are several features that make P2P systems interesting for scientific research, which include their ability to exploit distributed resources, circumvent censorship and their potential for extreme scalability and robustness. As such, an important candidate consumer for this technology is the computational grid, which is supposed to enable optimal exploitation of large amounts of resources available over the Internet using a P2P approach [5].

The promises of P2P technology have already been partially fulfilled by numerous applications. Unfortunately, the underlying protocols on which they are built are often complex and unpredictable. Their behavior is not fully understood, and often, can be explained only in terms of the theory of complex networks or dynamic systems. Given the lack of traditional hard guarantees regarding expected outputs, users outside the scientific community—especially engineers and application developers—might experience difficulties in exploiting the growing body of available knowledge.

In our opinion, a more significant exploitation of P2P technology requires a *modular paradigm* where well-understood and predictable components with clean interfaces can

* This work was partially supported by the Future & Emerging Technologies unit of the European Commission through Project BISON (IST-2001-38923).

** also with RGAI, MTA SZTE Szeged, Hungary

be combined to implement arbitrarily complex functions. The goal of this paper is to report on our ideas and initial results towards this objective.

The first step in our advocated paradigm is to identify a collection of primitive components, that is, simple P2P protocols that can be used as *building blocks* for constructing more complex protocols and applications. An informal classification of these building blocks in two broad classes is possible:

Overlay protocols maintain connected communication topologies over a set of nodes. We refer to such topologies as *overlays*, as they are built over underlying networks like the Internet. An example is NEWSCAST [6], that maintains a random overlay.

Functional protocols are aimed at implementing basic functions for other components. An example for such a function is *aggregation* [17], a collective name for functions that provide global information about a distributed system. These functions include finding extremal values, computing averages and sums, counting, etc.

A modular approach offers several attractive possibilities. It allows developers to plug different components implementing a desired function into existing or new applications, being certain that the function will be performed in a predictable and dependable manner. An even more interesting possibility is to combine building blocks to form more complex applications that perform relatively sophisticated functions like file sharing or load balancing.

Building blocks must be simple and predictable, as well as extremely scalable and robust. In this way, research can focus on self-organization and other important emergent features, without being burdened by the complexity of the protocols. Our building blocks are typically no more complicated than a cellular automaton or a swarm model which makes them ideal objects for research. As a result, practical applications can also benefit from a potentially more stable foundation and predictability, a key concern in fully distributed systems.

In order to demonstrate the usefulness of our approach, we describe how to implement a fairly complex function, *load balancing*, using the two building blocks introduced above—NEWSCAST and aggregation. It turns out that the resulting protocol is close to optimal with respect to the amount of load it transfers.

The outline of the paper is as follows. In Section 2 we define our system model. Section 3 describes the basic building blocks that will be used by the load balancing example described in Section 4. In Section 5 we give a brief overview of PEERSIM, the high-level network simulator that we have specifically developed to support our modular paradigm.

2 System Model

Figure 1 illustrates our system model. We consider a network comprising a large collection of *nodes* that communicate through the exchange of messages and are assigned unique identifiers. The network is highly dynamic (new nodes may join and old ones can leave at any time) and subject to failures (nodes may fail and messages can be lost).

Each node runs a set of *protocols*. Protocols can be standalone applications, or may provide some service to other protocols. Each protocol instance may communicate with

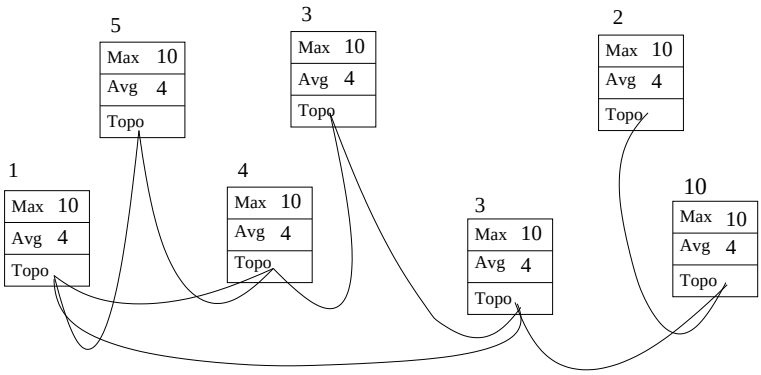


Fig. 1. A simple example network. The environment consists of a set of numeric values, one at each node. Two protocols, MAX and AVG, find the maximum and the average of these values, respectively. They are based on protocol TOPO, whose task is to maintain an overlay topology (represented by the connections between nodes).

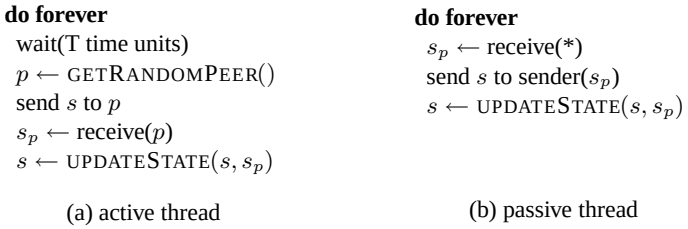
other protocols located at the same node (e.g., to import or export a service) and with other instances of the same protocol type located at remote nodes (e.g., to implement a function).

We assume that nodes are connected by an existing *physical network*. Even though the protocols we suggest can be deployed on arbitrary physical networks, including sensor and ad-hoc networks, in the present work we consider only fully connected networks, such as the Internet, where each node can (potentially) communicate with every other node. In this way, arbitrary overlay topologies may be constructed, and a functional protocol may deploy the most appropriate overlay for implementing its functions.

The physical network provides only the possibility of communication. To actually communicate with its peers, a node must know their identifiers. At each node, the task of an overlay protocol is to collect and maintain up-to-date identifiers in order to form a connected topology with some desired characteristics. Given the large scale and the dynamicity of our envisioned system, these collections are normally limited to a rather small subsets of the entire network.

Apart from communicating with other peers, protocols may also interact with their *environment*. Any input that originates from outside the protocol set falls into this category. The environment may include user interactions, sensor information, and any application-specific data such as load in a load balancing system and free space in a distributed storage system.

In our model, modularity is implemented at the level of protocols. Protocols must provide and implement well-defined interfaces, in order to allow developers to plug different implementations of the same protocol into their applications. For example, as explained in the next section, the aggregation protocols illustrated in Figure 1 make use of an overlay protocol to communicate with peer nodes so as to compute the aggregates. Any implementation of overlay can be used, as long as it provides standard interfaces and a topology with the appropriate characteristics.



(a) active thread (b) passive thread

Fig. 2. The skeleton of a push-pull epidemic-style protocol. Notation: s is the state of this node, s_p is the state of the peer p.

3 Example Building Blocks

In this section we describe two general-purpose protocols, NEWSCAST [6] and *epidemic-style aggregation* [7], which will be used to build our load balancing scheme in Section 4. The descriptions we provide here are necessarily brief and informal; they serve simply to introduce their structure and provide some details of their characteristics. Additional information can be found in the related papers [6, 7].

Both protocols are based on the push-pull epidemic-style scheme illustrated in Figure 2. Each node executes two different threads. The *active* one periodically initiates an *information exchange* with a peer node selected randomly, by sending a message containing the local state and waiting for a response from the selected node. The *passive* one waits for messages sent by an initiator and replies with its local state. The term push-pull refers to the fact that each information exchange is performed in a symmetric manner: both peers send and receive their states.

Method UPDATESTATE builds a new local state based on the previous local state and the state received during the information exchange. The output of UPDATESTATE depends on the specific function implemented by the protocol. The local states at the two peers after an information exchange are not necessarily the same, since UPDATESTATE may be non-deterministic or may produce different outputs depending on which node is the initiator.

The period of wall clock time between two consecutive information exchanges is called the *cycle length*, and is denoted by T. Even though the system is not synchronous, it is often convenient to talk about *cycles* of the protocol, which are simply consecutive wall clock time intervals of length T counted from some convenient starting point.

3.1 Newscast

The NEWSCAST protocol [6] is an example of an overlay protocol. It maintains a random topology, which is extremely robust, and can be used as the basis for several functional protocols, including broadcast [6] and aggregation [7].

In NEWSCAST, the state of a node is given by a *partial view*, which is a fixed-size set of peer descriptors. A *peer descriptor* contains the address of the peer, along with a

timestamp corresponding to the time when the descriptor was created. The (fixed) size of a partial view is denoted by c.

Method GETRANDOMPEER returns an address selected randomly among those in the current partial view. Method UPDATESTATE merges the partial views of the two nodes involved in an exchange and keeps the c freshest descriptors, thereby creating a new partial view. New information enters the system when a node sends its partial view to a peer. In this step, the node always inserts its own, newly created descriptor into the partial view. Old information is gradually and automatically removed from the system and gets replaced by new information (hence the name, NEWSCAST). This feature allows the protocol to "repair" the overlay topology by forgetting dead links, which by definition do not get updated because their owner is no longer active.

3.2 Properties of Newscast

In NEWSCAST, the overlay topology is defined by the content of partial views. Each descriptor in the partial view represents a directed edge in the topology, linking the node holding the descriptor to the node named in the descriptor. The basic assumption in the design of the protocol is that the set of nodes that form the network is highly dynamic, with a continuous flow of nodes joining and leaving the system. This dynamicity is reflected in the overlay topology, that is constantly changing over time, by removing obsolete information and disseminating descriptors of joining nodes.

We have shown in [6] that the resulting topology has a very low diameter and is very close to a random graph with out-degree c. According to our experimental results, choosing $c = 20$ is already sufficient for very stable and robust connectivity.

We have also shown that, within a single cycle, the number of exchanges per node can be modeled by a random variable with the distribution $1 + \text{Poisson}(1)$. In other words, on the average, there are two exchanges per cycle (one is actively initiated and the other one is passively received) and the variance of this estimate is 1. The implication of this property is that no node is more important (or overloaded) than others.

3.3 Epidemic-style Aggregation

In the case of epidemic-style aggregation [7], the state of a node is a numeric value. In a practical setting, this value can be any attribute of the environment: storage capacity, temperature, load, etc. The task of the protocol is to calculate an aggregate value over the set of all numbers held by the nodes. Here, we will focus on the two specific cases of average and maximum. Other aggregate functions, including sum, counting, variance estimation, etc., may be easily computed using a similar scheme.

In order to function, this protocol needs an overlay protocol that provides an implementation of method GETRANDOMPEER. In the present paper, we assume that this service is provided by NEWSCAST, but any other overlay protocol could be used.

In the case of averaging, let method UPDATESTATE(a, b) return $(a + b)/2$. After one state exchange, the sum of the values maintained by the two nodes does not change, since they have just balanced their values. So the operation does not change the global average either; it only decreases the variance over all the estimates in the system. In

the case of maximum, let method UPDATESTATE(a, b) return $\max(a, b)$. In this case, the global maximum value will be effectively broadcast like an epidemic.

3.4 Properties of Epidemic-style Averaging

Maximum and averaging protocols have different mathematical properties. For maximum, existing results about epidemic-style broadcasting [3] are applicable. From now on, we focus on averaging only.

For our purposes, the most important feature will be the *convergence speed* of averaging. As mentioned above, it is guaranteed that the value at each node will converge to the true global average, as long as the underlying communication topology is connected. In [7] it was shown that if this communication topology is not only connected but also sufficiently random, the speed of convergence is exponential.

In a more precise mathematical formulation, let μ_i be the empirical mean and σ_i^2 be the empirical variance in cycle i,

$$\mu_i = \frac{1}{N}\sum_{k=1}^{N} a_{i,k}, \quad \sigma_i^2 = \frac{1}{N-1}\sum_{k=1}^{N}(a_{i,k} - \mu_i)^2 \tag{1}$$

where $a_{i,k}$ is the value maintained at node $k = 1, \ldots N$ during cycle i and N is the number of nodes in the system. It can be shown that we have

$$E(\sigma_{i+1}^2) \approx \frac{E(\sigma_i^2)}{2\sqrt{e}}. \tag{2}$$

Simulations show that this approximation holds with high precision. From this equation, it is clear that convergence can be achieved with very high precision in only a few cycles, irrespective of the network size which confirms extreme scalability.

In addition to being fast, our aggregation protocol is also very robust. Node failures may perturb the final result, as the values stored in crashed nodes are lost; but both analytical and empirical studies have shown that this effect is generally marginal [7]. As long as the overlay network remains connected, link failures do not modify the final value, they only slow down the aggregation process.

4 Load Balancing: an Example Application

Let us define the load balancing problem, which will be our example application for illustrating the modular design paradigm. We assume that each node has a certain amount of load and that the nodes are allowed to transfer all or some portions of their load between themselves. The goal is to reach a state where each node has the same amount of load. To this end, nodes can make decisions for sending or receiving load based only on locally available information.

Without further restrictions, this problem is in fact identical to the averaging problem described in Section 3. In a more realistic setting however, each node will have a limit, or *quota*, on the amount of load it can transfer in a given cycle (where *cycle* is as defined in Section 3). In our present discussion we will denote this quota by Q and assume that it is the same for each node.

Let $a_{i_1}, \ldots, a_{i_N}$ be the decreasing order of load values $a_1, \ldots, a_N$
$j \leftarrow 1$
while ($a_{i_j} > \mu$ and $a_{i_{N+1-j}} < \mu$)
 $a_{i_j} \leftarrow a_{i_j} - Q$
 $a_{i_{N+1-j}} \leftarrow a_{i_{N+1-j}} + Q$
 $j \leftarrow j + 1$

Fig. 3. One cycle of the optimal load balancing algorithm. Notation: μ is the average load in the system, N is the network size, Q is the quota.

4.1 The Optimal Algorithm

For the sake of comparison, to serve as a baseline, we give theoretical bounds on the performance of any load balancing protocol that has access to global information.

Let $a_{i,1}, \ldots a_{i,N}$ represent the individual loads at cycle i, where N is the total number of nodes. Let μ be the average of these individual loads over all nodes. Note that the global average does not change as a result of load transfers as long as work is "conserved" (there are no node failures). Clearly, at cycle i, the minimum number of additional cycles that are necessary to reach a perfectly balanced state is given by

$$\max_j \left\lceil \frac{|a_{i,j} - \mu|}{Q} \right\rceil \tag{3}$$

and the minimum amount of total load that needs to be transferred is given by

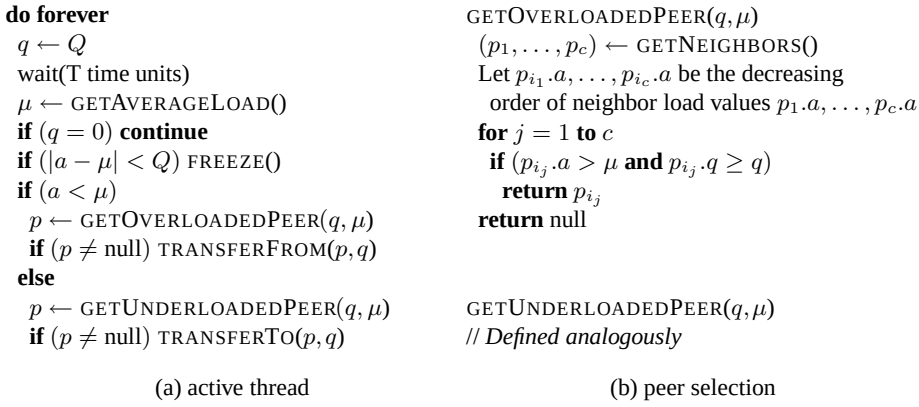
$$\frac{\sum_j |a_{i,j} - \mu|}{2}. \tag{4}$$

Furthermore, if in cycle i all $a_{i,j} - \mu$ $(j = 1, \ldots, N)$ are divisible by Q, then the optimal number of cycles and the optimal total transfer can both be achieved by the protocol given in Figure 3. This algorithm is expressed not as a local protocol that can be run at each node, but as a global algorithm operating directly on the list of individual loads. It relies on global information in two ways. First, it makes a decision based on the overall average load (μ) which is a global property and it relies on globally ordered local load information to select nodes with specific characteristics (such as over- or under-loaded) and for making sure the quota is never exceeded.

It is easy to see that the total load transfered is optimal, since the load at each node either increases monotonically or decreases monotonically, and when the exact global average is reached, all communication stops. In other words, it is impossible to reach the balanced state with any less load transfered.

The algorithm also achieves the lower bound given in (3) for the number of cycles necessary for perfect balance. First, observe that during all transfers exactly Q amount of load is moved. This means that the property that all $a_{i,j} - \mu$ $(j = 1, \ldots, N)$ are divisible by Q holds for all cycles, throughout the execution of the algorithm. Now, we only have to show that if $\max_j |a_{i,j} - \mu| = kQ \geq 0$ then

$$\max_j |a_{i,j} - \mu| - \max_j |a_{i+1,j} - \mu| = Q. \tag{5}$$



(a) active thread (b) peer selection

Fig. 4. A modular load balancing protocol. Notations: a is the current load, Q is the total quota, q is the residual quota and c is the number of peers in the partial view as determined by the overlay protocol.

To see this, define $J = \{j^* | \max_j |a_{i,j} - \mu| = |a_{i,j^*} - \mu|\}$ as the indices which belong to nodes that are maximally distant from the average. We have to show that for all nodes in J, a different node can be assigned that is on the other side of the average. We can assume without the loss of generality that the load at all nodes in J is larger than the average because (i) if it is smaller, the reasoning is identical and (ii) if over- and under-loaded nodes are mixed, we can pair them with each other until only over- or under-loaded nodes remain in J. But then it is impossible that the nodes in J cannot be assigned different pairs because (using the definition of J and the assumption that all nodes in J are overloaded) the number of under-loaded nodes has to be at least as large as the size of J. But then all the maximally distant nodes got their load difference reduced by exactly Q, which proves (5).

Motivated by this result, in the following we assume that (a) the initial load at each node is an integer value, (b) the average is also an integer and (c) we are allowed to transfer at most one unit of load at a time. This setting satisfies the assumptions of the above results and serves only as a tool for simplifying and focusing our discussion.

4.2 A Modular Load Balancing Protocol

Based on the observations about the optimal load balancing algorithm, we propose a protocol that is based purely on local knowledge, but that approximates the optimal protocol extremely well, as we show in Section 4.4.

Figure 4 illustrates the protocol we propose. The basic idea is that each node periodically attempts to find a peer which is on the "other side" of the global average and has sufficient residual quota. If such a peer can be found, load transfer is performed.

The approximation of the global average is obtained using method GETAVERAGELOAD, and the peer information is obtained using method GETNEIGHBORS. These methods can

be implemented by any appropriate component for average calculation and for topology management.

We assume that in each cycle, each node has access to the current load and residual quota of its peers. This latter value is represented by local variable q at each node, which is initialized to Q at the beginning of each cycle and is updated by decrementing it by the actual transfered load. This information can be obtained by simply asking for it directly from the peers. This does not introduce significant overhead as we assume that the load transfer itself is many orders of magnitude more expensive. Furthermore, as we mentioned earlier, the number of peers is typically small ($c = 20$ is typical).

Note that once the local load at a node is equal to the global average, the node can be excluded from future considerations for load balancing since it will never be selected for transfers. By excluding these "balanced" nodes, we can devote more attention to those nodes that can benefit from further transfers. The protocol of Figure 4 implements this optimization through the method FREEZE. When a node executes this method, it starts to play "dead" towards the overlay protocol. As a result, the node will be removed from the communication topology and the remaining nodes (those that have not yet reached the average load) will meet each other with higher probability. In other words, peer selection can be more efficient in the final phases of the execution of the balancing protocol when most nodes already have reached the average load. Although the optimization will result in a communication topology that is partitioned, the problem can easily be solved by adding another overlay component that does not take part in load balancing and is responsible only for maintaining a connected network. Note also that the averaging component uses the same overlay component that is used by the load balancing protocol.

A key feature of the averaging and overlay protocols is that they are potentially significantly faster than any load balancing protocol. If the quota is significantly smaller than the variance of the initial load distribution, then reaching the final balanced state can take arbitrarily long (see Equation (3)). On the other hand, averaging converges exponentially fast as defined by Equation (2). This fact makes it possible for load balancing to use the approximation of the global average as if it were supplied by an oracle with access to global information. This scenario where two (or more) protocols operate at significantly different time scales to solve a given problem is encountered also in nature and may characterize an interesting general technique that is applicable to a larger class of problems.

4.3 A Basic Load Balancing Protocol

In order to illustrate the effectiveness of using the averaging component, we suggest a protocol which does not rely on the average approximation. The protocol is shown in Figure 5.

This protocol attempts to replace the average approximation by heuristics. In particular, instead of choosing a peer from the other side of the average, each node picks the peer which has a maximally different load (larger or smaller) from the local load. The step which cannot be replaced however is the FREEZE operation. Performing that operation depends crucially on knowing the global average load in the system.

```
do forever
  q ← Q
  wait(T time units)
  if (q = 0) continue
  p ← GETPEER(q, a)
  if (p.a < a) TRANSFERTO(p, q)
  else TRANSFERFROM(p, q)
```

(a) active thread

```
GETPEER(q, a)
  (p_1, ..., p_c) ← getNeighbors()
  Let p_{i_1}.a, ..., p_{i_c}.a be the decreasing
    order of neighbor load values p_1.a, ..., p_c.a
    according to the ordering defined by
    |a − p_1.a|, ..., |a − p_c.a|
  for j = 1 to c
    if (p_{i_j}.q ≥ q) return p_{i_j}
  return null
```

(b) peer selection

Fig. 5. The basic load balancing protocol. Notations: a is the current load, Q is the total quota, q is the residual quota and c is the number of peers in the partial view as determined by the overlay protocol.

4.4 Empirical Results

Empirical studies have been performed using the simulator described in Section 5. We implemented the three protocols described above: the optimal algorithm, the modular protocol that is based on the averaging protocol and NEWSCAST and the basic protocol that has no access to global average load. As components, the methods of Figure 4 were instantiated with the aggregation protocol of Section 3 for averaging and NEWSCAST for the overlay.

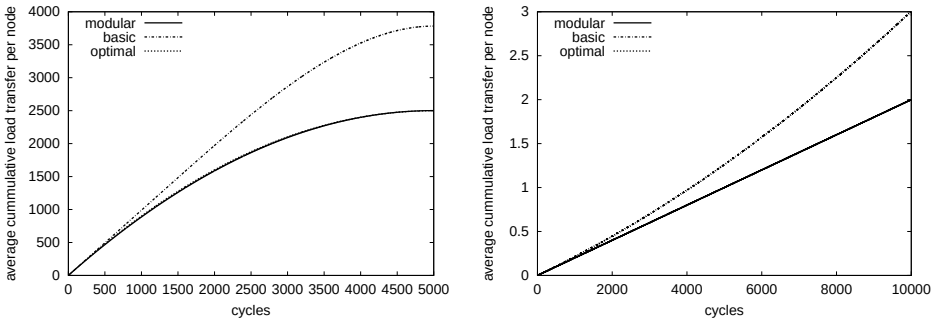
In all our experiments, the network size was fixed at $N = 10^4$ and the partial view size was $c = 40$. We examined two different initial load distributions: linear and peak. In the case of linear distribution, the initial load of node i ($i = 0, \ldots, N$) was set to exactly $i - 1$ units. In the case of peak distribution, the load of exactly one node was set to 10^4 units while the rest of the nodes had no initial load. The total quota for load transfer in each cycle was set to one load unit ($Q = 1$).

During the experiments the variance of the local load over the entire network was recorded along with the amount of load that was transfered during each cycle. We do not show the data on variance—which would give information about the speed of reaching the balanced state—because all three protocols have identical (i.e., optimal) convergence performance for both initial distributions.

Figure 6 presents results for total load transfered during the execution of the three solutions. Each curve corresponds to a single execution of a protocol, as the variance of the results over independent runs is diminishing. As can be seen from the figures, the load transfered by the modular protocol is indistinguishable from the amount that is optimal for perfect balancing in the system.

5 A Dedicated Simulator: PeerSim

Evaluating the performance of P2P protocols is a complex task. One of the main reasons for their success, i.e. the extremely large scale that they may reach, is also one



(a) linear load distribution

(b) peak load distribution

Fig. 6. Cumulative average load transferred by a node until a given cycle in a network of size 10^4. The curves corresponding to the optimal algorithm and the modular protocol overlap completely and appear as a single (lower) curve. The final point in both graphs (5000 and 10000 cycles, respectively) correspond to a state of perfect balance reached by all three protocols.

of the major obstacles for their evaluation. P2P networks that incorporate hundreds of thousands of nodes are not uncommon; at the time of writing, more than 5 million hosts are connected to the Kazaa/Fasttrack network. Another source of complexity is the high dynamicity of such systems: P2P networks are in a continuous state of flux, with new nodes joining and existing nodes leaving or crashing.

Evaluating a protocol in real settings, especially in the first phases of its design, is clearly not feasible. Even the larger distributed testbeds for deploying planetary-scale network services [12] do not include more than 220 nodes, a tiny figure compared to the size of modern P2P applications. Furthermore, nodes in such testbeds are not characterized by the same degree of dynamicity that is typical of P2P nodes.

For some protocols, an analytical evaluation is possible. For example, simple epidemic-style protocols (such our average aggregation mechanism) may analyzed mathematically due to their simplicity and their inherent probabilistic nature. Nevertheless, simulation is still an invaluable tool for understanding the behavior of complex protocols or validating theoretical results.

The results illustrated in the previous section have been obtained using our simulator called PEERSIM, developed by our group and specialized for supporting the simulation of P2P protocols based on the modular paradigm pursued in this paper. PEERSIM complements our model by enabling developers to experiment with protocols and their composition.

5.1 Design Objectives for PEERSIM

A simulator for P2P systems may have very different objectives from general-purpose networks simulators [11]:

– *Extreme scalability*. Simulated networks may be composed of millions of nodes. This may be obtained only if a careful design of the memory layout of the simulator is performed. Being able to store data for a large number of nodes, however, is not the only requirement for large-scale simulations; the simulation engine must be optimized as well, trying to reduce, whenever possible, any form of overhead.
– *Support for dynamicity*. The simulator must be capable to deal with nodes that join and leave the network, either definitively or temporarily. This has some implications on memory management in the simulator, requiring mechanisms for removing nodes that are not useful any more.

In addition to these requirements, the modular approach we are pursuing in this paper must be reflected in the architecture of the simulation environment as well. The idea is to provide a composition mechanism that enables the construction of simulations as collections of components. Every component of the simulation (for example, protocols or the environment) must be easily replaceable through simple configuration files. The flexibility offered by this mechanism should enable developers to re-implement, when needed, every component of the system, with the freedom of re-using existing components for fast prototyping.

Some of these goals may appear contradictory. For example, a modular approach may introduce overhead that limits overall performance of the simulator, or smart but large data structures may improve speed, but they may also limit the scalability of the simulator. A careful design is needed trying to obtain the best equilibrium.

5.2 Simplifying Assumptions

The strong scalability requirements outlined in the previous section force us to introduce several simplifying assumptions. For example, low-level details, such as the overhead associated to the communication stack (e.g. TCP or UDP) cannot be taken into consideration because simulating the underlying protocols requires a lot of additional memory and time, a price that cannot be easily paid when nodes are in the range of millions.

However, in many cases the properties of certain P2P protocols enable us to apply not only these assumptions, but also additional ones without sacrificing much of the realism of the simulations. For example, let us consider the membership and aggregation protocols presented in Section 3. In each cycle, every node sends and receives two messages on average. In both protocols, messages are small: a few hundred bytes for NEWSCAST, and a few bytes for average aggregation. Given the fast convergence of these protocols, in a real implementation the cycle length can be chosen large enough to guarantee that messages will arrive before the start of the next cycle. (For example, choosing a cycle length of five seconds and performing 20 cycles (sufficient to obtain a variance reduction of 10^{-9}), less than two minutes are needed to obtain the average, independently of the size of the network.)

As a result of these properties even latency and bandwidth may be dropped from our models, without rendering the simulations unrealistic. For these reasons, the simulation model that is adopted by PEERSIM ignores concurrency and in fact it is very similar to a cellular automaton model. The model is based on the concept of cycle. In each cycle all nodes have a chance to perform a basic operation based on their current state

and possible communication with their current neighboring nodes, where neighborhood relation is defined by an overlay protocol or a fixed communication topology.

5.3 The PEERSIM Architecture

The architecture of PEERSIM is illustrated in Figure 5.3. As described above, PEERSIM has been designed to be highly modular and configurable, without incurring in excessive overhead both in terms of memory and time.

The *configuration manager* is the main component. Its task is to read configuration files and command-line parameters, and compose a simulation starting from the components listed in the configuration. The configuration manager is the only fixed module of a simulation; every other component is interchangeable, to allow developers to write their customized and optimized version when needed. The configuration mechanism is currently based on Java property files, that are collections of pairs associating a property name to a property value. Each configuration file is substantially a list of associations between component identifiers and the name of the Java class implementing the particular protocol. After the instantiation, each component is responsible for reading the additional parameters needed by its implementation. For example, a membership component may read the configured maximum size of its partial view.

Following the definitions provided in Section 2, the simulated network is composed of a collection of *nodes*, each of them may host one or more *protocols*. Communication between protocol instances belonging to the same node are based on method invocations: in order to provide a service, each protocol must implement a well-defined interface. For example, protocols that maintain an overlay topology must implement the Linkable interface, that enables higher-level services to obtain information about the neighbors known to that node. Protocols implementing aggregation must provide an interface for setting the local value and obtaining the aggregated one.

The interaction between the environment and the protocols is represented by *Dynamics*, that are executed periodically by the simulation engine, and may interact with the simulated systems at different levels; for example, they may modify the network composition, either by adding new nodes or by destroying existing ones; or, they may act at the level of protocols, for example modifying an overlay topology or changing the aggregated value.

Observers play the role of global observation points from which it is possible to analyze the network, the nodes composing it and the state of the protocols included on them, in order to collect statistics about the behavior of the system as a whole. Observers, like dynamics, are executed periodically. Observers may be highly customized for a particular protocol (for example, to report the variance reduction rate in an aggregation protocol), or may be more general (for example, to analyze graph-theoretical properties of maintained topologies, like diameter, clustering, etc.).

Protocols, dynamics and observers give designers of P2P protocols complete freedom to simulate whatever system they want, at the desired level of accuracy; yet, the presence of a growing library of pre-built components enable the construction of fast prototypes.

The *simulation engine* is the module that will actually perform the simulation; its task is to orchestrate the execution of the different components loaded in the system. As

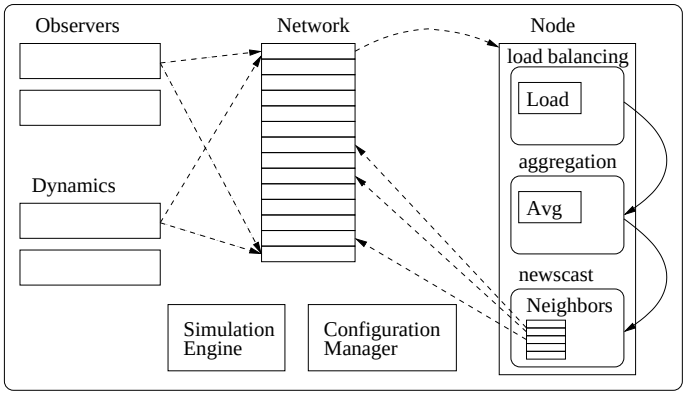


Fig. 7. Architecture of the PEERSIM simulator. A simulation is composed of a set of nodes, observers and dynamics objects. Nodes are composed of a set of protocols. The composition of a single node is shown, with the protocols used to implement load balancing.

described in the previous section, the engine adopts a time-stepped simulation model instead of more complex and expensive event-based architecture. At each time step, all nodes in the system are selected in a random order, and a callback method is invoked on each of the protocols included in that node. In this way, all protocol instances get a chance to execute at each cycle. Dynamics and observers are executed periodically; each of them is associated with a *scheduler* object, that based on the information stored in the configuration file decides when and how frequently the particular dynamics or observer must be executed.

Communication between nodes is left to the designer of protocols; in the algorithms developed so far, protocol instances invoke methods on each other, in order to reduce the overhead associated to the creation, the enqueuing and the garbage collection of messages. This is compatible with the kind of protocols simulated so far. Using this approach, nodes are responsible to check whether a particular node cannot be reached, either due to a node crash or a communication failure; if so, the method invocation should be discarded.

6 Related Work

Another well-known functional building block is a *distributed hash table* (DHT), which is an abstraction of a distributed data structure that maintains associations between keys and nodes in the network. There have been proposals for applying DHTs as abstract building blocks to construct more complex applications [2] including event notification systems [14] and distributed storage systems [4]. Yet, DHTs themselves are often complex and in our conceptual framework, we are looking for possibilities for decomposing them into smaller components [18].

The load balancing problem, which we have used as an example to demonstrate the usefulness of our approach, is one of the oldest problem in distributed systems.

Past research has proposed several different load balancing strategies and has evaluated their performance on both distributed systems and multiprocessors. In these studies, the topologies considered are either fixed, structured graphs (such as trees, rings, stars, multi-dimensional hypercubes, etc.) [9], or complete graphs, where each node knows the identifier of every other node [1]. As such, these results are not easily applicable to P2P networks. Due to the high dynamicity and large-scale of these networks, constructing and maintaining topologies as overlay networks is a complex task, and for the same reasons, complete overlays are not possible. Furthermore, and more importantly, these structured topologies do not exploit the possibility of constructing close-to-optimal overlay networks, as we did in this paper.

More recently, Rao et al. have presented a set of algorithms for solving the load balancing problem in DHT networks [13]. One of these algorithms, called one-to-many, is similar to our approach: they assume the knowledge of a known target for the average load , and each over-loaded node selects the most under-loaded nodes among a subset. Their algorithm, however, does not explain exactly how the average load may be obtained, and does not exploit the possibility of progressively reducing the overlay network to those nodes that need to be balanced.

Surveying existing literature on P2P simulation, the picture that is obtained is highly fragmented; most current P2P projects base their results on home-grown simulators that have been tuned for the particular protocols employed by the target systems. Very few papers [19] make use of general-purpose, detailed network simulators such as NS2 [11]. This choice is due to the high costs associated with packet-level simulations that pose very severe limits to scalability.

Despite this fragmentation, several interesting simulation environments have appeared recently [15, 16, 8]. Most of them are limited to file-sharing applications, as they incorporate notions of documents, keywords, etc. The most promising one is NeuroGrid [8], that has been designed with extensibility in mind. Yet, none of them approach the scalability that can be obtained by PEERSIM.

7 Conclusions and Future Work

In this paper we have presented our initial results towards a modular paradigm for designing complex, self-organizing protocols. A load balancing protocol was presented to confirm the validity of the idea that simple components can be combined to implement more complex functions. We demonstrated the utility of applying an averaging component along with an overlay component to obtain a performance with respect to the amount of transfered load that is indistinguishable from the optimal case.

Naturally, our present and future work focuses on developing this paradigm further by extending it with more components and applications. It is interesting to consider the similarities of this approach to object-oriented design and programming. In a sense, at least in the design phase, the designer can treat these building blocks as objects that maintain a state and that have an interface for modifying or monitoring that state or for performing functions. Some of the usual relations such as dependence or inheritance can also be applied. Our future work includes developing this analogy further.

References

1. A. Barak and A. Shiloh. A Distributed Load Balancing Policy for a Multicomputer. *Software Practice and Experience*, 15(9):901–913, Sept. 1985.
2. F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, CA, USA, Feb. 2003.
3. A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Management. In *Proc. of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 1–12, Vancouver, Aug. 1987. ACM.
4. P. Druschel and A. Rowstron. PAST: A Persistent Anonymous Store. In *HotOS VIII*, May 2001.
5. I. Foster and A. Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, CA, USA, Feb. 2003.
6. M. Jelasity, W. Kowalczyk, and M. van Steen. Newscast Computing. submitted for publication.
7. M. Jelasity and A. Montresor. Epidemic-Style Proactive Aggregation in Large Overlay Networks, 2004. To appear.
8. S. Joseph. An Extendible Open-Source Simulator. *P2P Journal*, Nov. 2003.
9. P. Kok, K. Loh, W. J. Hsu, C. Wentong, and N. Sriskanthan. How Network Topology Affects Dynamic Load Balancing. *IEEE Parallel & Distributed Technology*, 4(3), Sept. 1996.
10. D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-Peer Computing. Technical Report HPL-2002-57, HP Labs, Palo Alto, 2002.
11. Network Simulator 2. `http://www.isi.edu/nsnam/ns/`.
12. L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. of the First ACM Workshop on Hot Topics in Networks (HotNets-I)*, Princeton, NJ, Oct. 2002.
13. A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load Balancing in Structured P2P Systems. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, CA, USA, Feb. 2003.
14. A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The Design of a Large-scale Event Notification Infrastructure. In J. Crowcroft and M. Hofmann, editors, *Networked Group Communication, Third International Workshop (NGC'2001)*, volume 2233 of *Lecture Notes in Computer Science*, pages 30–43, Nov. 2001.
15. M. Schlosser and S. Kamvar. Simulating a file-sharing p2p network. In *Proc. of the First Workshop on Semantics in P2P and Grid Computing*, 2002.
16. N. S. Ting and R. Deters. A P2P Network Simulator. In *Proc. of the 3rd IEEE International Conference on Peer-to-Peer Computing (P2P'03)*, Linkoping, Sweden, Sept. 2003.
17. R. van Renesse. The Importance of Aggregation. In A. Schiper, A. A. Shvartsman, H. Weatherspoon, and B. Y. Zhao, editors, *Future Directions in Distributed Computing*, number 2584 in Lecture Notes in Computer Science, pages 87–92. Springer, 2003.
18. S. Voulgaris and M. van Steen. An epidemic protocol for managing routing tables in very large peer-to-peer networks. In *Proceedings of the 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, (DSOM 2003)*, number 2867 in Lecture Notes in Computer Science. Springer, 2003.
19. B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *To appear in IEEE Journal on Selected Areas in Communications*, 2003.