Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki
Vrije Universiteit Amsterdam
Faculty of Sciences

Tomasz Pylak

Nr albumu: 181288

Gossip-based computing in the presence of node failures

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem

Janiny Mincer-Daszkiewicz
Instytut Informatyki, Uniwersytet Warszawski
oraz
Maartena van Steena i Wojciecha Kowalczyka
Instytut Informatyki, Vrije Universiteit, Amsterdam

Sierpień 2004

Streszczenie

Gossip-based protocols provide scalable, reliable and robust way of spreading information in large and widely distributed networks. The Author has build a flexible, efficient and configurable simulator to explore properties of this class of protocols. Using the simulator we demonstrate big potential of gossip-based protocols, also in scenarios where the set of network members is very dynamic because of node failures. We analyze these protocols' capabilities to compute aggregative functions of distributed values.

Słowa kluczowe

peer-to-peer, epidemic protocol, membership management, dissemination-oriented communication, large-scale distributed system, unstructured overlay, random graph, failure, recovery

Klasyfikacja tematyczna

Category and subject descriptor according to ACM CCS:
C.2.4 [Computer-Communication Networks]:
Distributed Systems— gossip-based protocols, P2P, an overlay network, distributed applications

Acknowledgments

I would like to thank all the people who helped me writing this thesis.

I would especially like to thank my supervisors, Maarten van Steen and Wojciech Kowalczyk, for introducing me to the subject of epidemic protocols, helpful suggestions and constant interest in my work.

I also want to thank my supervisor in Poland, Janina Mincer-Daszkiewicz, for her help in improving the quality of the paper and interesting remarks.

I would also like to thank Spyros Vouglaris, a PhD student at the VU, for explaining me the details of his research during a number of informal meetings.

Last, but not least, I would like to express my gratitude to all my friends in the guesthouse 'Hospitium' for their support and friendship during the whole year in Amsterdam.

Contents

Chapter 1

Introduction

1.1. Background

Because of the rapid growth of the Internet, peer-to-peer systems have become more and more common. A characteristic feature of such systems is that they maintain a highly dynamic network based on the logical relationships between peers. That network is often called an *overlay network*, as it is built on top of the connections which make up the Internet.

A proper overlay network management is a quite big challenge. Lately it have become popular to solve this problem by usage of the class of protocols called *epidemic* (or *gossip-based*). Epidemic protocols provide scalable, reliable and robust way of spreading information in large and widely distributed networks. What makes them interesting is that their guarantee of the effectiveness comes from probabilistic calculations.

Recently, interesting research in the domain of gossip-based peer-to-peer systems has been done. Simple and efficient algorithms for information dissemination in large overlay networks have been introduced ([3, 7, 2]).

Overlay networks can be viewed in an abstract sense as constituting a distributed computing engine — peers cooperate exchanging data to compute a result which is a function of their collective knowledge. So far it has been shown that this computing engine is capable of performing simple yet interesting tasks such as aggregation of distributed values, load balancing, broadcasting, etc.

The underlying assumption has always been that failing nodes leave the network forever. That is, failures have been treated as voluntary departures from the network.

1.2. Problem statement

In this thesis we would like to examine the behavior of a class of gossip-based protocols — *Newscast*, *Shuffling* and *Cyclon*. We will focus on scenarios in which failing nodes may recover and rejoin the network, as well as the computation that they were previously engaged in. However, to gain insight into system behavior and provide a good comparison, stable networks were considered as well.

Issues that we examine include basic network properties, such as average path length, clustering and connectivity. In addition, we wish to examine the effect on the convergence speed of an aggregation algorithm computing the average. In this way, we check the behavior of our protocols when used in an application area.

To carry out experiments the need for a flexible, efficient and configurable simulator emerged. Since we expect enhancements of protocols managing the overlay network, as well as the emergence of new applications running on them, it is clear that the source code of such a simulator must be extendable and also easy to maintain. We believe that the simulator, built by the author of this thesis, fulfills all these requirements. The simulator's general design issues and abilities are discussed here. The well-documented simulator's source code for Linux, Solaris and Windows, together with the user manual and configuration examples, are publicly available[1].

1.3. Related work

1.3.1. Epidemic protocols

The inspiration for epidemic protocols, also called gossip-based protocols, came from observations of the way in which infectious diseases spread among population. The theory [12] shows that even if at the beginning only one side is infected, the whole population will be eventually infected by an epidemic in expected time proportional to the logarithm of the population size. The goal is to build systems disseminating information as fast as an epidemic spreads. A good introduction to this field can be found in [9].

Epidemic algorithms do not give us reliability guarantees which are so strong as those offered by costly deterministic algorithms. Instead of that they allow to create highly scalable solutions. Epidemic protocols became more popular since the publication of [1], where they were used to maintain consistency of distributed database replicas. Since that time they have been successfully applied in various new areas, including failure detection [13], resource monitoring [10], aggregation [3, 6, 14] and broadcasting [11, 15].

1.3.2. Membership management

The basic idea of epidemic-style protocols is that every node repeatedly contacts a peer which is randomly selected among all nodes in the network and exchanges information with it. Such an approach [15, 14, 16] requires every node to know all other nodes in the system. In a distributed system, where the network tends to be very dynamic (nodes join and leave frequently), maintaining such a list of nodes causes scalability problems. For this reason, solutions in which every node has only a *partial view* of the network have been proposed [11, 3]. In these cases every node knows only a continuously changing sample of all nodes in the system. Gossip-based protocols are used to update partial views, providing a membership management mechanism which copes very well with dynamic environments.

The paper [17] presents a generic framework of the *peer sampling service*, which maintains partial views of nodes, to provide each node with a peer when a node asks about it. A general gossip-based protocol scheme which can be used by the framework is described there. There is also an experimental evaluation of different protocols. It turns out that an overlay network topology maintained by considered protocols does not resemble traditional random graphs, which was often a basic assumption of gossip-based protocols analysis. Instead, an

[1]see http://www.cs.vu.nl/globesoul/sim.tgz

overlay network has common features with so called ‘small-world’ graphs, characterized by small diameter and large clustering.

1.3.3. Distributed aggregation

The topic of computing aggregates like extreme values, mean or variance in large distributed systems is quite new, though very useful and interesting [18]. As was shown in [3], aggregation can be used for monitoring network size, distributing alarm signals or measuring the total amount of resources.

The problem can be solved using a hierarchical architecture like Astrolabe [10]. Astrolabe sees the system as a hierarchy of zones, which comprises smaller non-overlapping zones or a host. The structure of Astrolabe’s zones can be viewed as a tree, the leaves of this tree represent the hosts. A user can initiate an aggregate query concerning hosts in the chosen part of the system and results of such a query will mimic the state of a group of hosts. Although this approach reduces the cost of computing aggregates, it has one main drawback — it requires a non-negligible overhead to maintain the hierarchical topology in a dynamic distributed system. In contrast, approaches to find aggregates discussed in this thesis are much more simple and lightweight. Moreover, the result is known to all nodes without extra efforts, which is a desired feature when aggregation is used to build self-organizing systems that are able to monitor their state in a decentralized way.

Theoretical analysis of the convergence speed of various aggregation algorithms working on an unstructured overlay network and using gossip-based protocols can be found in [14, 5, 6]. The latter two papers validate the scalability of discussed solutions through simulations, also. The presented algorithms proved to be efficient and very robust.

The behavior of the averaging algorithm when the Newscast protocol is used is examined in a variety of scenarios in [4]. The paper presents theoretical and empirical evidence showing the robustness of the averaging algorithm under scenarios when node and communication failures occur. However, in contrast to this paper, it was not considered that a node can recover and resume its work.

1.3.4. GlobeSoul project

This thesis is a part of the GlobeSoul project. The goal of GlobeSoul is to explore the P2P technology and find out how can it help in building large-scale distributed systems. The current research activities focus on scalable epidemic protocols, superpeers and decentralized clustering. The important part of GlobeSoul is Globule — an open-source Apache Web server module which provides functionality of a user-centric content delivery network. The home page of GlobeSoul with more detail description of that project can be found at www.cs.vu.nl/globesoul.

1.4. Overview

This paper is organized as follows. First, in Chapter 2 we briefly explain the model of the gossip-based computing engine which will be used. In Chapter 3 protocols maintaining an overlay network which were tested in this thesis are described. Chapter 4 presents the way in which general properties of the network depend on the protocol which is used. Among other things the effects of the network size, cache size and failures scenarios are examined. In Chapter 5 we test the performance of some applications computing aggregate functions. The relationship between application performance and the network properties examined in the previous chapter is discussed. Finally, in Chapter 6 we describe the simulator built by us and used in all reported experiments. Its design, functionality and usefulness in further research are presented. We conclude in Chapter 7, showing possible directions of future work.

Chapter 2

System model

First, we briefly describe the model of our gossip-based computing engine. The engine works in agent-based, large-scale distributed systems. It uses an uncomplicated, peer-to-peer data exchange protocol to maintain an overlay network and to keep it connected. The protocol is based on an epidemic algorithm. The overlay network consists of nodes; each node has a small amount of memory to store information about addresses of other nodes. This set of addresses is continuously changing.

Our gossip-based computing engine, presented in Figure 2.1, consists of two layers. The first one can be thought of as a *news agency*. A news agency is responsible for membership management — it takes care of joining and leaving nodes. It also determines the way in which information is disseminated in the network.

The second part of the computing engine is an application layer which comprises autonomous *agents* running specific computations.

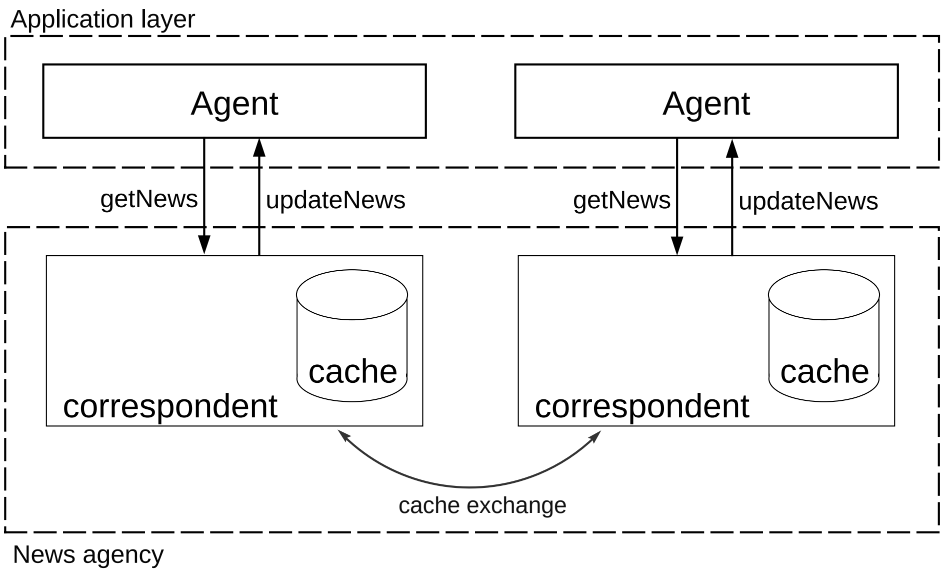


Figure 2.1: The conceptual organization of a computing engine. Arrows describe data flow.

2.1. News agency

The news agency cooperates with agents by collecting their news and passing it on to other agents. This task is performed via an agent's callback functions: `getNews()` and `updateNews(news)`, which will be described later.

The news agency is implemented by a set of *correspondents*. Each correspondent is associated with an agent running on the same machine. A correspondent is equipped with a small amount of memory, which is used as a *cache*. If a correspondent knows the address of the other correspondent, it can establish a connection and initiate a cache exchange. The method of choosing a correspondent to contact and the way in which cache entries are then exchanged and used are determined by a *news agency protocol*. In this paper, properties of the Newscast and the related Shuffling and Cyclon protocols will be examined in detail.

As we can see in Figure 2.2, a cache entry consists of 3 fields. The first one is mandatory and stores the address of a correspondent who created that entry. The two others are optional: one for protocol-specific data and one for some application-specific data concerning the correspondent's agent. The number of entries in the cache is limited by a global parameter $c > 0$. Usually c is between 20 and 50.

Address	Protocol-specific data optional	Application-specific data optional

Figure 2.2: The structure of the cache entry.

2.2. Application layer

The correspondent acquires news from its agent by calling the `getNews()` function. News can be anything, e.g., the measurement done by the agent's sensors or a result of a computation in which the agent takes part. Note that the news may have changed since the last invocation of `getNews()` — it can, for example, be affected by the current time or by new information the agent has acquired.

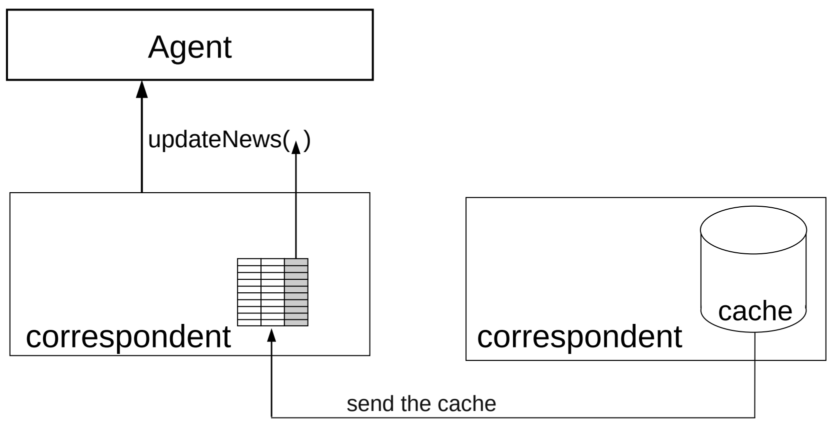
The exact task of the agents does not concern the computing engine. From our point of view the most important thing about the application layer is its integration with the news agency. This can be done in two ways.

The first one, depicted in Figure 2.3(a), can be characterized as a strong integration — cache entries carry agent news as application-specific data. The correspondent informs its agent about data collected by the peer from other agents by means of the `updateNews(news)` function. This function is called by the correspondent who has initialized a cache exchange, as well as by the correspondent who has been contacted. Although this kind of integration can be useful (see [6]), we will not consider it.

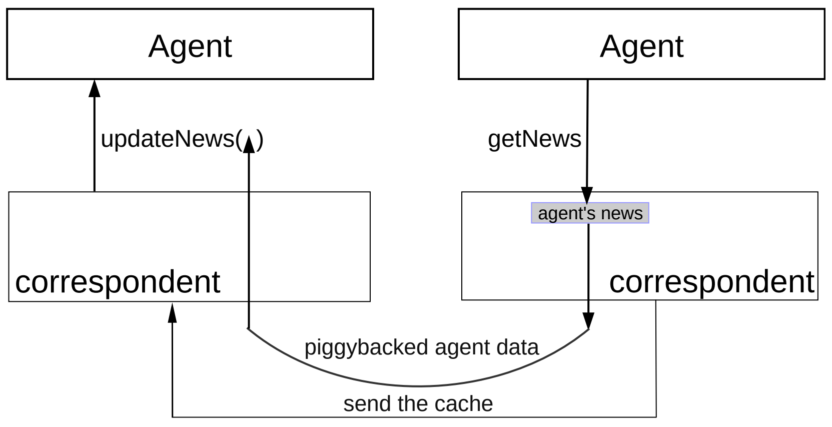
Figure 2.3(b) presents a scenario in which the integration between the application layer and the news agency is weaker than in the previous case. It uses the `updateNews(peerAgentNews)` function, instead of the `updateNews(news)` function. The difference is that the correspondent's agent updates its news only on the basis of its peer agent's news. The required data from the peer agent must be, for example, piggybacked with the caches when they are exchanged.

Since no application-specific data is stored in the cache, the information dissemination service of the protocol is not used directly. Nevertheless, the protocol's influence on the way in which information spreads is noticeable — the protocol determines which agents will exchange news. As in the previous case both the initiator of a cache exchange and the receiver of the request call the function updating news. In this paper we will focus only on the weak model of integration.

Note that in the models of integration described above, application data has no influence on the way in which the news agency protocol works. However, sometimes such models could also be useful.



(a) strong integration



(b) weak integration

Figure 2.3: Two kinds of integration between the application layer and the news agency. Grey fields represent application-specific data created by agents. Note that the data flow is symmetrical for both agents, although the flow in only one direction is depicted.

Chapter 3

Protocols

The basic idea of epidemic-style protocols is that every node repeatedly contacts a peer which is randomly selected among all nodes in the network and exchanges information with it. But this approach is not scalable when the number of nodes becomes large and the network is dynamic — it is no longer possible that every node keeps track of all others available in the network.

Protocols like Cyclon, Shuffling or Newscast tackle this problem by providing each node with a small, continuously changing random sample of nodes from the whole network. The size of this sample, c, is a global parameter of the protocol. As we will show later, this approach successfully solves scalability problems giving us a robust membership management tool. At the same time information can be efficiently disseminated.

In the following sections the Newscast, Shuffling and Cyclon protocols are described in detail.

3.1. Newscast

The basic idea of the Newscast protocol is that each correspondent periodically chooses one entry from its cache and contacts the correspondent referred to in that entry. Correspondents then send their cache content to each other and construct new caches from the exchanged entries.

The Newscast protocol uses the protocol-specific part of cache entries for storing the time when the entry was created.

More formally, each correspondent performs the following steps every ΔT time units:

1. randomly select a cache entry to find the address of a peer correspondent,
2. send the cache content to that peer, along with an additional entry filled with your own address and timestamped with the local current time,
3. receive the peer's cache,
4. merge the received cache with your own cache, discarding the oldest entries, until there are at most c entries left. Also, no two entries may refer to the same correspondent.

The selected correspondent performs the same steps, except the first one.

Note that it is not required that the clocks of the correspondents are synchronized. It is sufficient that the timestamps of entries in each particular cache are consistent. To assure this, when correspondent A receives the cache from B, it can normalize the timestamp of each received entry by adding to it the difference between the local time of A and B. The local time of B can be sent to A together with its cache.

3.2. Shuffling

The Shuffling protocol was introduced in [7] (below we describe a slightly modified version of it). In contrast to Newscast no protocol-specific data is stored in a cache entry, so each entry is equivalent to a reference to an agent. The main issue is that each correspondent periodically selects a subset of cache entries and exchanges them with a peer whose address was randomly selected among the cache entries.

More formally, the following steps, called a *shuffle operation*, are performed by each correspondent every ΔT time units:

1. randomly select a cache entry to find the address of a peer correspondent,
2. randomly select $l-1$ cache entries (different than the one selected in the previous step), where l is a global parameter of the protocol and is called the *shuffle length*, $1 \leq l \leq c$,
3. send these entries to the peer together with an additional one with a reference to the local agent,
4. receive l entries from the peer,
5. update the cache with the received entries, according to the following rules:
    - discard a new entry referring to an agent if a reference to this agent already exists in the cache,
    - if there are empty slots in the cache, fill them first, otherwise:
    - replace the entry referring to the peer and the entries sent to him.

The peer correspondent sends to the initiator l randomly selected entries and performs steps 4 and 5. If one correspondent has less than l entries in its cache, then the number of sent or received entries can be smaller.

As it was shown in [2], an overlay network's properties are insensitive to the shuffle length l. In all our experiments l was equal to 5.

The important property of the shuffling operation is that it cannot divide the overlay network into unconnected parts if it constituted one connected part before. A formal proof of this quite intuitive fact can be found in [2].

3.3. Cyclon

The Cyclon protocol is an extension of Shuffling introduced in [2]. Thanks to a small, yet smart, enhancement it improves the process of information dissemination, increases overlay network connectivity and balances load on nodes.

Cyclon uses the protocol-specific part of a cache entry to store its age counting from the moment when it was created. The main, and almost the only, difference with Shuffling is the first step of the shuffling operation — the selection of a peer. To select a peer, Cyclon does not use a randomly selected cache entry. Instead, the entry which has appeared in the network the longest time ago is used.

When a new entry is created, it gets the age 0. Before the shuffle operation, the initiator increases the age of every cache entry by 1.

3.4. Common properties

The nodes that constitute the network must have access to local clocks. These clocks do not have to be synchronized, but should be able to measure time with reasonable accuracy — in a short period a clock's drift cannot be too big.

Although the whole system is not synchronized, we find it convenient to describe a protocol execution in terms of consecutive wall clock time intervals of length ΔT. These intervals will be called *cycles* of the protocol and will be counted from some convenient starting point.

For every protocol, we assume that the communication between 2 correspondents takes less than ΔT time units. ΔT should be big enough to fulfill this assumption, otherwise the protocols will not be able to work correctly — note that, according to each of them, every correspondent has to complete at least one data exchange during that time.

3.5. Membership management

Each protocol described above solves the problem of joining and leaving nodes in a similar, simple way. Cache entries store correspondents' addresses and these addresses are disseminated during the cache exchange — as we will show in a moment, the whole membership management functionality is based on it.

When a correspondent wants to subscribe to a network, all it must do is to initialize its cache with at least one entry for another correspondent and start executing the protocol. In [2, 3, 7] it was shown how the cache can be initialized keeping good overlay network properties.

If a correspondent wants to unsubscribe, it simply stops communicating — voluntary departures from the network are treated in the same way as failures. When another correspondent chooses a cache entry to find a peer, it can happen that the address stored in that entry is a *dead pointer* — it is not valid any more because the peer has left the network[1]. Every dead pointer is removed from the cache as soon as it is encountered. In this way the system forgets about the nodes which have left.

[1]When a peer selected by the correspondent turns out to be unavailable, the correspondent waits until the next cycle to try again.

If the system is capable of repairing an overlay after a serious disaster in a few cycles, we can say it can heal itself. As was shown in [4] and [2], even if a big fraction of nodes suddenly crash, the overlay network stays connected and nodes which survived quickly replace dead pointers with links to valid nodes.

The Newscast protocol has an additional weapon against dead pointers — it favors the freshest entries, so if a correspondent stops injecting its own entries, all the references to it will be quickly removed from the system to the newer entries' advantage.

The Cyclon protocol also fights against system contamination by dead pointers. The peer selection procedure bounds the time an entry can stay in the system unreferenced, assuring that entries with links to unavailable nodes will not stay in the system too long.

Note that sometimes forgetting too fast about nodes which have left the system can have undesirable consequences. Just imagine that because of some errors the network becomes temporarily partitioned into two parts. All pointers in the overlay network from one part to the other become dead pointers. If all these pointers are removed before connections between two parts of the network can be restored, it will cause the permanent partition of the overlay network.

Chapter 4

Overlay properties

Before examining specific applications which need the distributed overlay network to run, we want to gain a good insight into the general properties of that network. It will allow us to understand better the overlay network's capability of spreading information. At the beginning we will introduce some basic notions which allow us to characterize a given overlay network. Then we will compare properties of overlay networks managed by the Newscast, Shuffling and Cyclon protocol.

To examine behavior of the gossip-based protocols in very large networks, we build an efficient simulator which allows us to control the state of the overlay network and the application layer. The simulator is described in detail in Chapter 6. In this chapter we present results of simulations for different protocol's configurations and for different network sizes. Two kinds of scenarios are considered — the one in which the set of nodes taking part in the simulation is fixed and the one in which that set changes dynamically due to the fact that nodes join and leave the network.

4.1. Basic characteristics

In the systems which we consider, every machine stores (in a cache) a few addresses of other machines. Such a network can be perceived as a graph in which every machine constitutes a node. If a machine A keeps the address of another machine B, then there is an edge A $\rightarrow$ B in the graph. When we forget about edge directions, we get an undirected graph which reflects the ability of the nodes to communicate with one another. This graph is continually changing in time, but its statistical properties have a tendency to stabilize. Analysis of such a communication graph can give us some interesting insights into the capabilities of our system. We will take a look at several graph properties.

- Let V_A be a set of neighbors of a node A (nodes connected with A by one edge). Let E_A be the set of edges in the graph induced by V_A. Now for a given node A we can define its **clustering coefficient** as:

  $$|E_A| / \binom{|V_A|}{2}$$

  In other words the clustering coefficient of a node A is the proportion of the number of edges in the graph induced by V_A to the number of edges in a full graph with $|V_A|$ vertices.

The clustering coefficient of a graph is the average of the clustering coefficients of all its nodes. For example, in a full graph the clustering coefficient is equal to 1. In a random graph with n nodes where each node has c neighbors, the clustering coefficient is approximately equal to c/n. A cluster can be perceived here as a group of nodes which are densely connected among themselves but have weaker connections with other nodes. Thus if the clustering coefficient is high, it is more probable that some clusters will get disconnected from the rest of the network.

The bigger the partition of the nodes into clusters, the worse the ability of quick information dissemination throughout the whole graph — a lot of redundant messages will be delivered to the cluster.

- The length of the shortest path between two arbitrary nodes gives us some information about the speed with which these nodes can exchange a message. To characterize this property for the whole graph, we can measure the length of the shortest path for all pairs of nodes and look at the average length of these paths — **the average shortest path length**.

- When some nodes leave the network, a situation of **dead pointers** leading to unreachable nodes can arise. The average percentage of addresses in the cache referring to live nodes (i.e. those that are still part of the network) gives us some idea of the network's ability of effective membership management. The number of addresses in a node's cache leading to live nodes is called the **effective cache size** — in this chapter we will look at the average effective cache size of all nodes.

- The node's degree in an undirected graph is defined as the number of its neighbors. In a directed graph we can distinguish between the number of nodes known by a given node — its *out-degree*, and the number of nodes which know about a given node — its *in-degree*. In our case the out-degree of every node is bounded by the cache size. The protocols which we consider very eagerly fill free cache slots during the cache exchange operation, so if we take into account also dead pointers, the out-degree of an arbitrary node will be in general close to the cache size.

  That's why we focus on the in-degree and examine the **in-degree distribution**. It is from our point of view a very important feature of a graph. It determines how the load will be balanced among nodes — the nodes with higher in-degree will be contacted more often. The in-degree distribution influences also the way in which information is disseminated among the nodes. So the more evenly the nodes' in-degree is distributed, the better the properties of our overlay network.

- As we explained in Chapter 3, protocols examined by us do not provide each node with a complete list of the other members of the network, but only with a random sample of them. But how can we check that this sample is uniformly random? One way is to compare properties of our communication graph to the properties of a **random graph**, where the out-degree of every node is equal to the cache size and neighbors of every node have been picked randomly among all the nodes. From now on, if we talk about random graphs, we mean graphs constructed in this way.

4.2. Stable network

First we present the results of simulations conducted on a stable network, where the set of participating nodes is fixed (nodes neither fail nor join or leave the network). Although the most interesting case for us is the one in which the network is dynamic, first we have to analyze the simpler scenario in which the network is stable to apprehend the properties of the whole system and to have a good frame of reference.

We examine the following important properties of the communication graph: the distribution of nodes in-degree, the average shortest path length and the clustering coefficient.

4.2.1. In-degree distribution

Figure 4.1 presents the in-degree distribution of nodes. Measurements were done in a stable, converged network of 10000 nodes, the cache size was 20.

In case of the Cyclon and Shuffling protocols, like in a random graph, the majority of nodes have in-degree equal to the cache size and the in-degree distribution is symmetrical around the cache size. However the variance of in-degree is not the same in these three cases. Cyclon exhibits an in-degree that is evenly distributed among all nodes, so the number of nodes with different in-degree drops quickly. A node's in-degree does not differ from the cache size by more than 20%. The Shuffling protocol is not so good in spreading the in-degree evenly, although the distribution has better properties when compared to a random graph.

In this light, the in-degree distribution when the Newscast protocol is used does not look too impressive. About 40% of all nodes have an in-degree that differs from the cache size by at least 50%. We can also see that there is a relatively large number of nodes with a high in-degree.

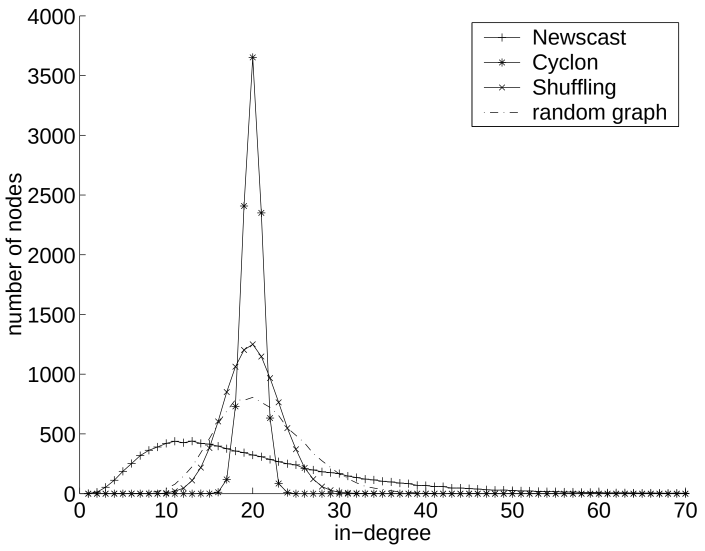


Figure 4.1: The in-degree distribution when different protocols are used in a stable network of 10000 nodes. The cache size was 20. Results are averaged over 20 runs, the standard deviation was in all cases smaller than 0.5% of the network size.

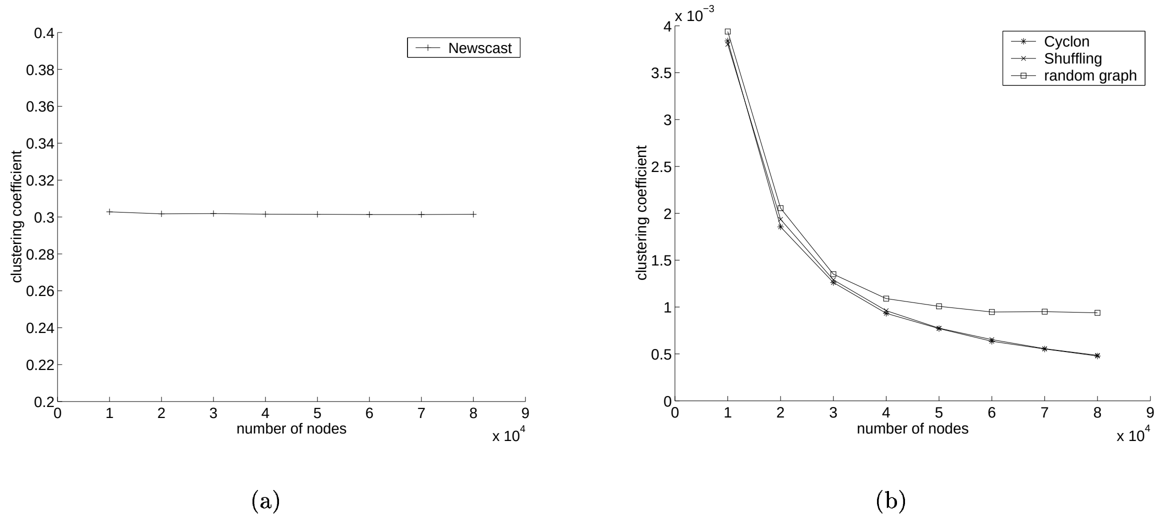


Figure 4.2: The clustering coefficient as a function of the network size when different protocols are used in a stable network. The cache size was 20. Results are averaged over 20 runs, the standard deviation was negligible. Note that the scale is different on (a) and (b).

4.2.2. Clustering coefficient

Figures 4.2 and 4.3 present measurements of the clustering coefficient in the simulations we conducted. Networks with different number of nodes (from 10000 to 80000) and configurations with different cache size (from 10 to 80) are considered.

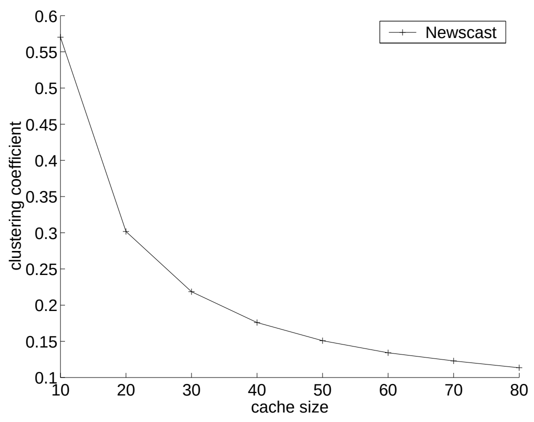
One general observation which can be made is that the Shuffling and Cyclon protocols maintain a communication graph with a low clustering coefficient, that is comparable and even slightly lower than in random graphs. The similarity is visible both in networks of different sizes (Figure 4.2(b)) and in networks where a different cache size is used (Figure 4.3(b)).

We can also notice the large, two orders of magnitude difference between Newscast and other protocols. It shows that an overlay network maintained by the Newscast protocol is not a random graph — the cache-exchange operation causes the nodes to prefer to group into clusters. It can adversely affect the information dissemination abilities of the Newscast protocol.

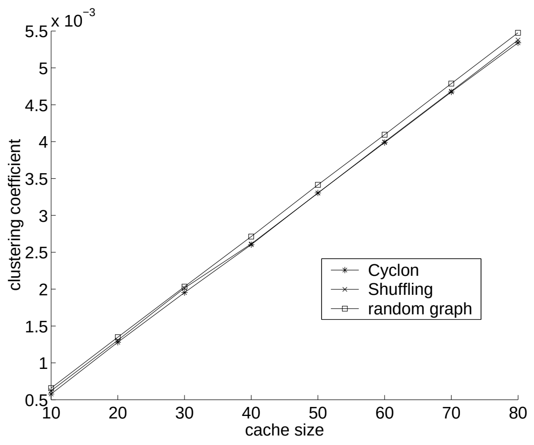
Going into details we see in Figure 4.2(a) that the clustering coefficient in an overlay network maintained by Newscast weakly depends on the network size — it stays on a similar level when the number of nodes is between 10 thousand and 80 thousand. Sensitivity to the cache size is much higher — as the cache size grows, the clustering coefficient is decreasing quickly as a power function, with exponent $\alpha < 1$. However the clustering stays much bigger than in a random graph even if the cache is large.

4.2.3. The shortest path length

Let's have a look at Figure 4.4, where the average shortest path length is plotted. As we can see in Figure 4.4(a), whichever of the three protocols we use, the average shortest path length is small and it grows very slowly with the number of nodes in the network. Although the difference between the Newscast and other protocols is noticeable, it does not exceed 1 edge.



(a)



(b)

Figure 4.3: The clustering coefficient as a function of the cache size when different protocols are used in a stable network. The network consisted of 30000 nodes. Results are averaged over 20 runs, the standard deviation was negligible. Note that the scale is different for (a) and (b).

There is almost no difference in the length of the average shortest path when we compare communication graph maintained by Cyclon or Shuffling with the random one.

Figure 4.4(b) presents how the average shortest path length depends on the cache size. The Newscast protocol turns out to be sensitive to this parameter, especially if the cache size is smaller than 20. However when the cache size was bigger than 40, the difference almost completely disappeared.

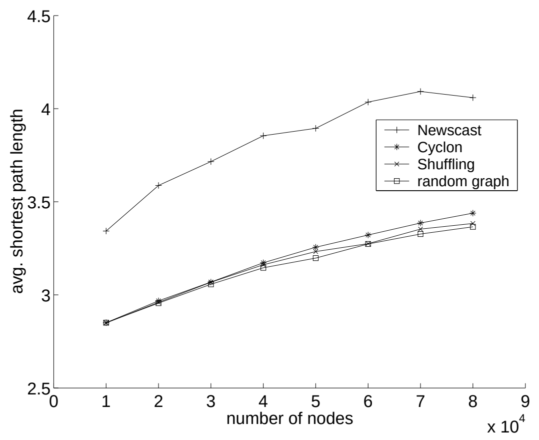
In general we can conclude that both Shuffling, Cyclon and Newscast maintain a communication graph with the low average shortest path length, similar to the one in a random graph. However in the case of the Newscast protocol it is crucial to configure the cache size appropriately.

4.3. Dynamic network

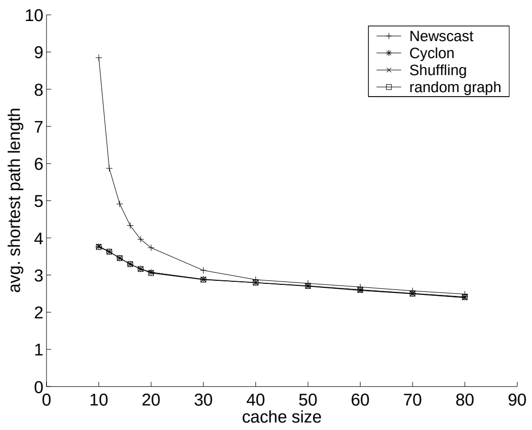
4.3.1. Failures model

We want to examine the behavior of an overlay network's properties in the presence of nodes failures. We assumed the following scenario: each node is working for a number of cycles, then failure occurs and a node is idle during a period called the *recovery time*. After recovery the node tries to return to the network with the cache content it had before the crash (it requires saving the cache content to a log in persistent memory). In our model both faultless work time, also called *time between failures*, and recovery time of a random node were defined by a random variable with exponential distribution. Note that in such a network the fraction of nodes which are working will converge to:

$$MTBF/(MTBF + meanRecoveryTime)$$



(a) The results for different network sizes. The cache size was 20.



(b) The results for different cache sizes. The network consisted of 30000 nodes.

Figure 4.4: The average shortest path length as a function of the number of nodes (left) and cache size (right), when different protocols are used in a stable network. The shortest path length was measured from a random sample of 5 nodes to all others and then the average was taken. Results are averaged over 20 runs, the standard deviation was negligible. On (a) the cache size was set to 20. On (b) the network consisted of 30000 nodes.

where MTBF stands for mean time between failures.

The mean time between failures was set to 20 cycles. We conducted several experiments changing the ratio of mean recovery time to MTBF. Figures 4.5, 4.6 and 4.7 show the results, which are averaged over 20 runs. If the standard deviation is not shown it means that it was several orders of magnitude lower than the average. All experiments were conducted on the network of 50000 nodes, the cache size was in all cases set to 20.

To provide comparison, experiments with the same parameters but on a stable network were also done — the ratio of mean recovery time to MTBF was in this case equal to 0, because the MTBF was then infinite.

4.3.2. Effective cache size

In Figure 4.5 we can see how fast information about dead nodes is removed from the system in the presence of continuous node failures.

At first glance, we can see that the Newscast protocol quickly forgets about non-available nodes. Even if every node recovers from crashes two times longer than it is present in the system, the cache is contaminated on average by only 2 addresses of dead nodes. This is caused by the fact that Newscast keeps only the freshest addresses in the cache — information about nodes which were up and running recently is stored. The probability that such nodes are still working after a few cycles is much higher compared to the case of a random node.

The Shuffling and Cyclon protocols do not have this property. They try (with some success) to keep the cache content very close to a random sample of all nodes in the network. It implies that if on average X% of all nodes is not available (because of recovering), then X%

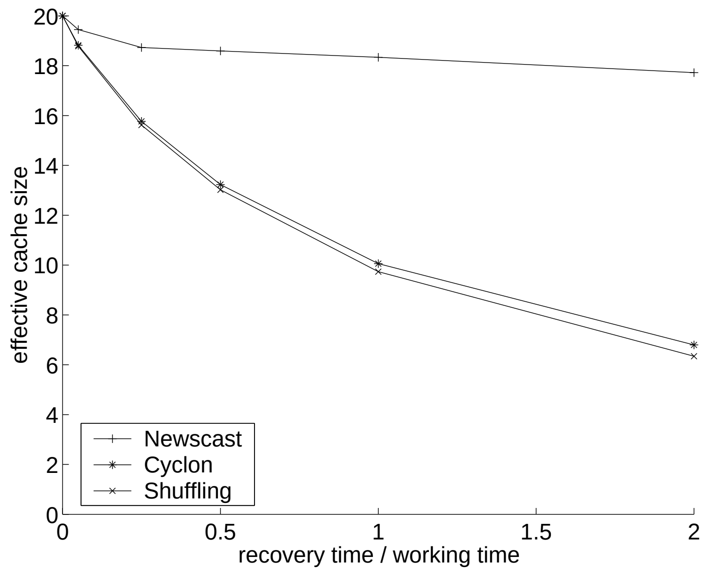


Figure 4.5: A comparison of the effective cache size when different protocols are used in a dynamic network. The network size was 50000, the cache size was 20. Results are averaged over 20 runs, the standard deviation was negligible.

of addresses in the cache will lead to unavailable nodes — and this is exactly what we see in Figure 4.5. The Cyclon protocol tries to contact peers with the oldest timestamp in the cache. This fact is manifested by slightly faster removal of dead pointers comparing to the Shuffling protocol. Thus the effective cache size is also a little bit bigger.

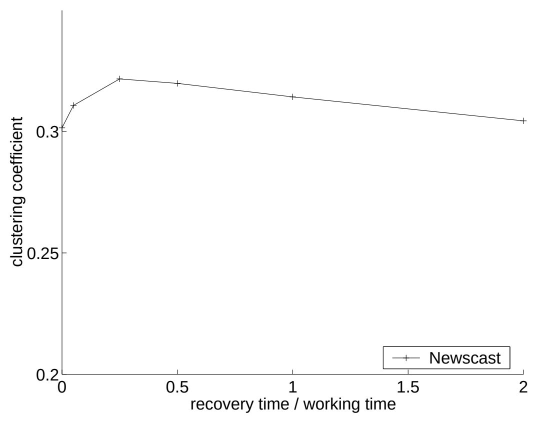
4.3.3. Clustering coefficient

Figure 4.6 demonstrates sensitivity of the clustering coefficient to the recovery speed. We can observe that the clustering coefficient of a communication graph when the Newscast protocol is used stays at a high level and is relatively stable. To understand the latter phenomenon we have to realize that when Newscast is used:

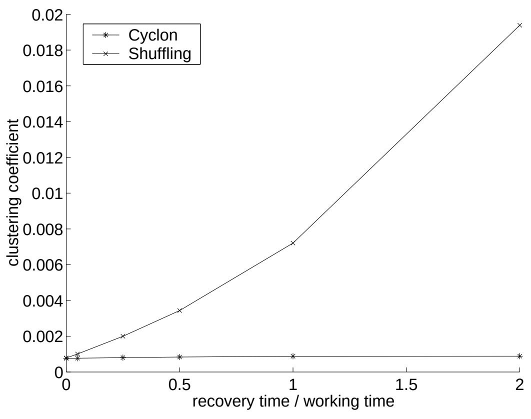
- the effective cache size is not very sensitive to the recovery time — dead links are thrown away very quickly (see Figure 4.5).
- the clustering coefficient is not very sensitive to the number of nodes in a network (see Figure 4.2(a))

So, as the effective cache size is stable the clustering coefficient is also stable, even if the number of nodes decreases.

The most interesting thing is that the clustering coefficient of a communication graph managed by the Cyclon protocol stays on the same low level even if the mean recovery time is bigger than MTBF. It implies that a graph induced by active nodes still possesses properties of a random graph — addresses of live nodes in a cache are a uniform random sample of all live nodes. This is not the case when Shuffling is used. The clustering coefficient increases by about one order of magnitude when the mean recovery time grows. This fact implies that



(a)



(b)

Figure 4.6: A comparison of the clustering coefficient when different protocols are used in a dynamic network. The network size was 50000, the cache size was 20. Note that the scale is different on (a) and (b). Results are averaged over 20 runs, the standard deviation was negligible.

a network has stopped to appear as a random graph and live nodes have started to group in clusters. But still the clustering is one order of magnitude smaller than in the case of Newscast.

4.3.4. The shortest path length

Figure 4.7 shows us how the average shortest path length changes when the recovery time increases. We can see that the length is low even if nodes recover very slowly.

When the mean recovery time increases, the path length decreases in the case of Newscast and increases in other cases. To understand why is it so, we have to look again at Figures 4.4 and 4.5. Looking at Figure 4.4 we could conclude that the average shortest path length is more sensitive to the cache size than to the number of nodes in the network. From Figure 4.5 we know that the long recovery time causes only a small decrease of the effective cache size when Newscast is used, and a quite serious decrease in other cases. So when the recovery time is long, in the case of Newscast, the network will be smaller and the effective cache size will not change too much, so the average shortest path length will be smaller. In the case of Cyclon and Shuffling both the network size and the effective cache size will be smaller, but because the cache size plays a more important role here, the average shortest path length will increase.

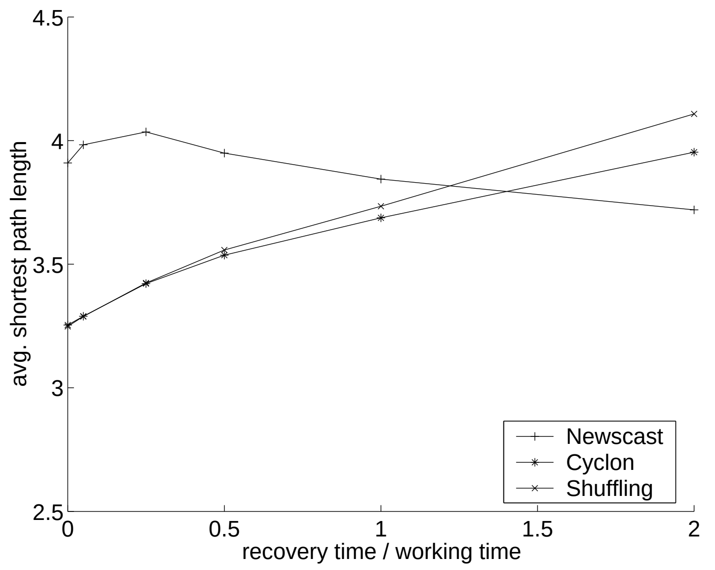


Figure 4.7: A comparison of the average shortest path length when different protocols are used in a dynamic network. The network consisted of 50000 nodes, the cache size was 20. Results are averaged over 20 runs, the standard deviation was in all cases smaller than 0.08.

Chapter 5

System applications

We are already equipped with knowledge about the properties of the overlay network managed by various protocols, so it is time to see, how this system can be used and how efficient it can be in practice. Among a broad range of our system's applications we will focus on aggregation algorithms.

Let's consider a system with N agents, where agent A_i stores one number a_i. The goal is to program these agents in such a way that in a small number of protocol cycles the system would be able to compute some aggregate function of $a_1 ... a_N$, such as maximum, average or variance. We are also interested in the behavior of the algorithm in a network where nodes could crash and after some time recover joining the computation again.

In the following sections we describe a particular aggregation algorithm, which computes average of distributed values. Such an algorithm can be used e.g. to monitor the average load of machines in the network or the average amount of their free memory. As was shown in [6], the averaging algorithm can be also used directly to compute a sum of distributed values. The ability of computing the average of some attributes in a distributed environment gives us a powerful tool to monitor such an environment.

The aggregation algorithm discussed here provides the result of a computation to all nodes. It is also not difficult to make this algorithm adaptive [4]. Adaptivity means that if a network is dynamic or values which are aggregated can change, the output of the aggregation protocol should follow these changes quickly. These important properties can be used e.g. by protocols responsible for self-organization in large-scale systems.

As we have mentioned earlier, protocols examined by us do not provide each node with a complete list of the other members of the network, but only with a random sample of them. However it is much easier to analyze the behavior of some applications considering the 'ideal case', where a correspondent can select its peer among all nodes in the network. To validate theoretical analysis of aggregation algorithms presented in [5], we performed tests using the *Random-peer* protocol which simulates the 'ideal case'. This simple protocol does not use caches of correspondents. When a correspondent wants to find a peer, the peer is randomly selected from the set of all nodes in a network.

5.1. Averaging algorithm

The averaging algorithm, presented in [5], allows every agent to get to know the average of all agents' values in a small number of protocol cycles with very good precision. The agent

A_i stores a current estimation of the average in a variable x_i. The algorithm requires that all agents start executing it synchronously, but later no synchronization is necessary. At the beginning every agent initializes x_i to a_i. The `getNews()` function simply returns x_i. The implementation of the `updateNews(peerAgentValue)` function updates the current x_i value with the average of x_i and the peer's agent value:

$$x_i := (x_i + peerAgentValue)/2$$

When the correspondent's agent and the peer's agent update their estimates, we say that an *averaging step* has been completed. As was shown in [5], the approximation of the average known by each agent converges to the true value exponentially fast when this algorithm is used.

In our simulations the value stored at the beginning by the agent A_i was set to i. We looked at the number of protocol cycles needed until all nodes know the average with certain precision. In the figures presented in the following sections precision X means that all available nodes knew the average with an error less than $X * 100\%$.

5.2. Stable network

Figure 5.1 shows us how the performance of the algorithm depends on the protocol which is used. The measurements have been taken in a stable network.

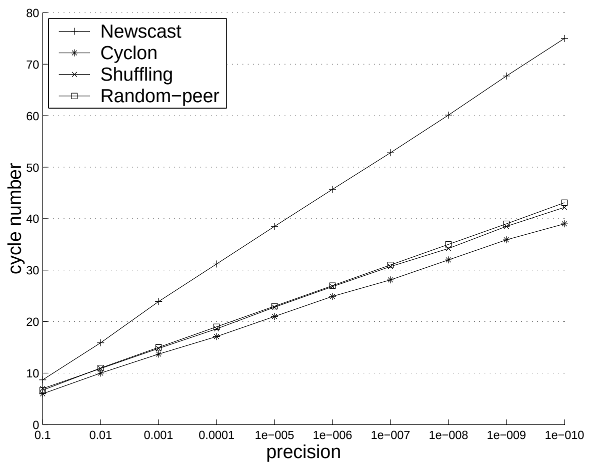


Figure 5.1: Comparison of the convergence speed of the averaging algorithm, when different protocols are used in a stable network. The network consisted of 50.000 nodes, the cache size was 20. Results are averages of 20 runs, the standard deviation of all samples was always smaller than 1.

For every protocol it takes a similar number of cycles to reach a small precision between 1% and 10%. But differences grow if we want to reach better accuracy. As we can see, if the Shuffling or Cyclon protocol is used, it takes from 30 to 40 cycles until all nodes know the average with very good precision. If the Newscast protocol is used, we need 70% more time to reach that point.

Note that the performance of Shuffling and Cyclon is almost identical to the ‘ideal case’, when peers are randomly picked from all nodes in the network. It proves that the small set of peers known to each node constitute a truly random view of the whole network.

Another interesting observation which can be made is the fact that the performance of Cyclon is a few cycles better than performance of the ‘ideal case’. How can it be possible? The reason is that Cyclon spreads the number of pointers to an arbitrary node (its in-degree) very evenly among all nodes. If all nodes have similar in-degree, then they perform a similar number of averaging steps. In contrast, when a peer is randomly selected among all nodes, then some peers are selected more often than others, because the distribution of the number of times when an arbitrary peer is selected is similar to the in-degree distribution of a random graph shown in Figure 4.1. Thus the number of averaging steps is not evenly distributed among nodes, what makes the convergence to the true average slower.

5.3. Dynamic network

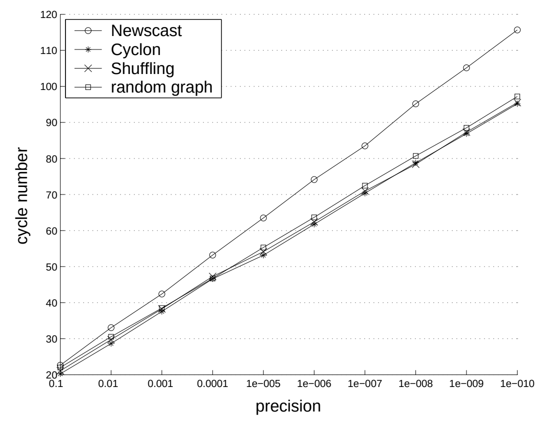
We have tested three scenarios using the failures model described in Section 4.3.1. Figure 5.2 shows the results when the mean recovery time is 10%, 20% and 40% of the mean time between failures.

In the first case in Figure 5.2(a), we can see that although the time needed to reach a certain accuracy increased about 2.5 times, relations between the protocols stayed similar to the case when the network was stable. When Newscast is used, convergence takes noticeably more time, albeit that the difference slightly diminishes. The fastest convergence is still observed when the Cyclon protocol is used.

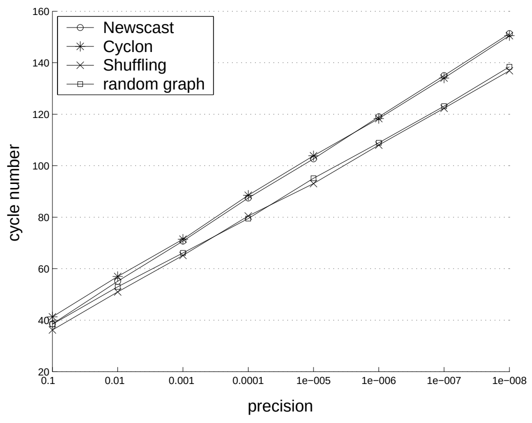
As we can see in Figure 5.2(b) and 5.2(c), the situation changes when extreme scenarios with relatively long recovery time are considered. The fact that Newscast maintains an overlay network with a high clustering is compensated by the low cache contamination with dead pointers (as we saw in Figure 4.5). That’s why the convergence speed of the averaging algorithm when the Newscast protocol is used is comparable to an ‘ideal case’, when a peer can be randomly selected among all nodes.

In all scenarios the convergence speed when the Shuffling protocol is used is similar to the ‘ideal case’. The low effective cache size is compensated by the clustering coefficient much lower than in the case of Newscast.

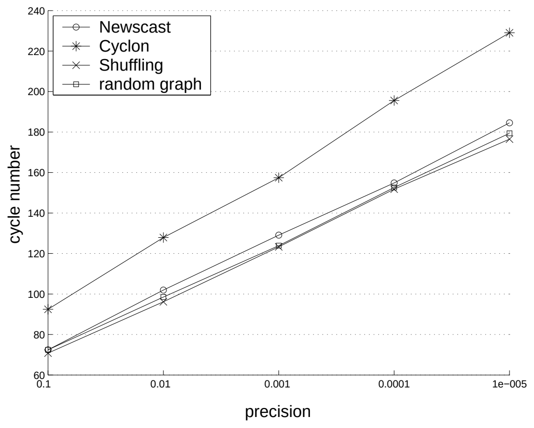
At the same time Cyclon performance decreases. As we remember from Section 3.3, the Cyclon protocol is the only one that tries to contact peers with the oldest timestamp in the cache, instead of choosing them randomly. When the cache contamination is high, the oldest chosen cache entry often contains a reference to an unavailable peer. In this way the number of averaging steps is lower and this is the reason of the slower convergence. However, we have to keep in mind that the examined scenarios were rather extreme and the performance degradation was not so drastic.



(a) Mean recovery time — 2 cycles.



(b) Mean recovery time — 4 cycles.



(c) Mean recovery time — 8 cycles.

Figure 5.2: A comparison of the convergence speed of the averaging algorithm for different protocols in a dynamic network. The network consisted of 50000 nodes, the cache size was 20. The mean time between failures was 20 cycles. Results are averaged over 20 runs, the standard deviation was always smaller than 1.

Chapter 6

Simulation engine

To carry out the experiments described in this thesis the need for a flexible, efficient and configurable simulator emerged. Since we expect enhancements of protocols managing an overlay network, as well as the emergence of new applications running on it, it is clear that the source code of such a simulator must be extendable and also easy to maintain. We believe that the simulator built by the author of this thesis fulfills all these requirements. The simulator was written in C++ and was carefully optimized, which makes it fast. We took advantage of the object-oriented approach, taking care of the clarity and flexibility of the design. The well-documented simulator's source code for Linux, Solaris and Windows, together with the user manual and configuration examples are publicly available[1].

The simulator lets us test both new protocols managing the network overlay and applications working on top of them. All aspects can be tested in a stable network as well as in the presence of crashes and recoveries of nodes.

We have to be aware that every simulator models reality introducing some simplifications. In our simulator the whole distributed network is simulated on one machine. During one cycle a node can serve all data exchange requests which come from other nodes during that cycle. In practice we can imagine that the number of requests is too big to fulfill this requirement. Fortunately errors caused by this simplification are small. The in-degree of a node bounds the number of requests which can be sent to that node. And as we saw in Figure 4.1, the majority of nodes has the low in-degree. Moreover the probability that there are 'hubs' with huge in-degree in the system is really small.

We verified correctness of our simulator by comparing part of our results obtained for the Newscast protocol with simulations results described in [3] and [6]. Our results were also coherent with results obtained in [8], where the Newscast protocol was emulated on a real network of agents distributed across a 320-node wide-area cluster of workstations.

The simulator consists of 3 types of modules:

- a protocol independent framework which allows us to specify the agents we want to use and aspects which we want to observe and measure in the simulation,
- modules that contain implementations of specific protocols, agents and simulations,
- facilities that help in preparing statistics, histograms, etc.

In the following sections each module is described.

[1]see http://www.cs.vu.nl/globesoul/sim.tgz

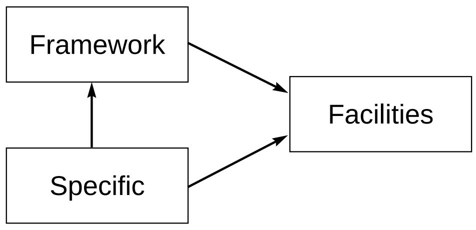


Figure 6.1: The simulator's modules dependencies

6.1. Framework

6.1.1. Design

Our simulator perceives a network as the set of *nodes*. Each node has the following properties:

- its own unique **address**,
- a correspondent with some **cache**,
- an **agent** connected to a correspondent.

During every protocol cycle every correspondent initiates exactly one connection with a peer and proceeds according to the rules given by the protocol:

- the initiator and its peer update their caches on the basis of the other correspondent's cache contents,
- the initiator's agent and the peer's agent update their state on the basis of the other agent's state.

The user can specify what type of agents should be used, choosing in this way the application which will run on an overlay network. The framework specifies an interface, which every agent must implement. Thus it is easy to create new types of agents — there is a small set of methods with a precisely defined semantic which have to be implemented.

Moreover, a **protocol**, which should be used by all correspondents during a simulation, can be specified as well. Again, the simulator's framework is based on the generic interface defining the protocol's responsibilities and no specific assumptions about a protocol are made.

The progress of the simulation can be observed by a **simulation monitor**. Such a monitor is informed about the beginning and the end of every cycle of the protocol. When a monitor receives a signal about such an event, it can perform some analysis of the agents' state or check properties of a communication graph. In this way we can explore the dynamics of particular parts of the system, gaining more insight into it.

The fact that the main components of the simulator: a protocol, an agent and a simulation monitor, are generic makes the simulator very flexible. Testing new kinds of agents, introducing improved protocols or monitoring either the network or the application layer can be done by adding simple and well–defined pieces of code.

6.1.2. Simulation properties

We can configure the simulator program to use a specific protocol and agents in a simulation. The simulation monitor can be specified as well. Such a construction lets us conduct different simulations and compare their results without the need for burdensome code recompilation.

Parameters such as the cache size or the number of protocol cycles after which a simulation should end can be chosen by the experimenter. Sometimes we would like to conduct several experiments for some range of parameters. The simulator allows to specify the range of cache sizes and the range of node numbers for which simulations should be performed. There is also a possibility of performing simulations with certain parameters a given number of times, which allows the simulation monitor to compute averaged results.

Sometimes, as in the case of a communication graph analysis, the simulator monitor has a lot of work to do. For this reason we can specify the protocol cycle in which monitoring should start. We can also perform measurements once for several cycles.

If for some reason we need to make the simulator more configurable, we can always do it by implementing our own component as a subclass of one of the framework's classes.

6.1.3. Management of nodes presence

During a simulation some nodes can suddenly crash and lose contact with the rest of the network for some time. At a certain moment, a node can recover and continue its work. The special component called **presence manager** is responsible for deciding if a node should crash and how long the recovery process should take.

The framework provides a general interface of a presence manager, which can be implemented to fulfill specific experimentation needs.

Starting a simulation without any nodes and waiting while they are joining one by one, until the network will be large enough, would last quite long. The simulator gives us an alternative — we can specify the number of nodes which take part in a simulation from the start.

It is not obvious how to initialize caches of such nodes. However for the protocols we have considered, we've found that we can take a group of nodes working from the beginning and initialize each node's cache with the addresses of nodes randomly selected from this group. It turns out, that after a few cycles (20–30) the caches' properties are the same as in a converged network. In a particular simulation, we can choose the number of protocol cycles needed for full convergence of the network. The application layer starts working after that time, which makes the simulation's results reliable.

6.2. Specific functionality

To conduct experiments needed for this thesis several interfaces provided by the Framework have been implemented.

Specific protocols, described in Chapter 3 and 5, which realize the *Protocol* interface contract were created: Newscast, Shuffling, Cyclon, Random-Peer. We have made efforts to make important protocols' features configurable. For instance, it is possible to specify in the configuration file how many cache items should be exchanged when the Shuffling or Cyclon protocol is used. We can also decide what to do if a peer randomly selected by a node does not respond

— the node can wait in such a case until the next cycle or repeat selection a given number of times.

We also created specific types of agents which compute aggregative functions of values stored by all agents, like the average, the sum or the maximum.

Different types of monitors were created. With their help we can measure:

- the communication graph properties,
- the number of cycles needed to calculate average or sum by all agents with certain accuracy,
- the number of nodes (in each cycle) which know the average or sum with certain accuracy,
- the convergence speed of the algorithm computing maximum.

All the details of using specific components can be found in the user manual.

6.3. Facilities

The main goal of all simulations is to gain an insight into system's characteristics. The *Facilities* module provides several utilities which can support the simulation monitor in e.g. collecting statistics or analyzing network properties.

6.3.1. Collecting statistics

The *Histogram Creator* is a utility which allows us to create histograms in a simple way. It can help if we want to see what's the distribution of a sequence of values in a given range. We can specify the number of 'buckets' in this range and the histogram will count the number of values in each bucket. We can also get to know how many values lay outside the range.

Usually we want to conduct one simulation several times to produce averaged results. The *Statistics* utility can help us in this situation. It makes it really simple if we want to compute an average and the standard deviation of one or many sequences of values. There is also a possibility of saving the results to the chosen file.

6.3.2. Network analysis

As we wrote in Section 4.1, a network of nodes can be perceived as a communication graph. To analyze important properties of this graph a special component was created. Using it we can analyze, for example:

- the clustering coefficient of a graph,
- the average shortest path length in a graph,
- graph connectivity — we can quickly find an answer to questions like:
    - is the graph connected?

  - how many connected components are in the graph?
  - how many randomly selected nodes should be removed to partition the graph? (in average)
  - what's the number of nodes which have no neighbors?
- the average effective in-degree of nodes (cache contamination).

6.3.3. Random numbers

During simulations we often need a random number generator. To model some phenomena like the mean time between failures it is useful to have random numbers which have exponential or Poisson distribution. A special component of the *Facilities* module lets us generate random numbers with a given range and with a chosen distribution — uniform (real numbers and integers), exponential or Poisson.

Chapter 7

Conclusions

In this thesis we examined the properties of the distributed computing engine, as well as a class of gossip-based protocols used by it — *Newscast*, *Shuffling* and *Cyclon*. We explored scenarios in networks where the set of participating nodes was stable and scenarios where this set was very dynamic because of node failures and recoveries. We considered situations where nodes after recovery from a failure could rejoin the network and resume the computation that they were previously engaged in.

We provided a detailed analysis of the considered protocols, examining the overlay network which they maintain and the behavior of applications running on this network. We presented their strong and weak points.

We demonstrated via simulations that both in stable and dynamic networks the recently introduced Cyclon protocol can maintain an overlay network with properties similar and even better than a random graph has — short paths between nodes, a low clustering coefficient and a good in-degree distribution.

We examined the effect of failing nodes on the convergence of the aggregation algorithm computing the average. For the first time we showed that we can obtain a really good convergence speed of this algorithm when the Cyclon protocol is used.

To obtain all the results we developed the flexible, efficient, configurable and multiplatform simulator of the gossip-based computing engine. Thanks to this fact new aspects of the system can be examined during future research. Using the simulator it is extremely easy to test enhancements of protocols managing an overlay network or new applications running on it.

We think that future work should be focused on extending Cyclon with features which will allow the construction of an overlay network more aware of the connections bandwidth and computational power of nodes. Then it would be possible, for example, to avoid network bottlenecks. Additionally taking proximity of nodes into account would help to decrease traffic in a network.

Appendix A

Example simulator configuration

All we need to configure the simulation is a text file which has a simple format of name–value pairs collection. Below we present examples of the configuration files used by the simulator. The simulator program will conduct a simulation for every configuration file given as a parameter. Details of the configuration are described in the simulator's user manual.

A.1. Stable network

Below we present an example configuration file for several simulations with different network sizes: 100000, 300000 and 500000 nodes. In each simulation the convergence speed of the averaging algorithm is measured. Agents start working after 30 cycles, when the network is converged. After 100 cycles the simulation stops. The network during the whole simulation is stable.

```
#   Kind of observer, which will monitor the state of
#   the simulation and measure something
#   (here: convergence speed of the averaging algorithm)
monitor     = cyclesForAcc

#   Class of agents used in the simulation. Here: agents computing the average
agent       = AgentAvg

#   Protocol which will be used: Cyclon
protocol    = cyclon

#   the number of nodes which take part in a simulation from the start.
initNumNodes = 100000

#   when one simulation ends, the initNumNodes parameter is updated:
#       initNumNodes = mulNumNodes * initNumNodes + incNumNodes
#   If initNumNodes does not exceed the limit set by the maxNumNodes parameter,
#   the next simulation with different network size is performed.

maxNumNodes = 500000
```

```
incNumNodes = 200000
mulNumNodes = 1

#   the cache size of each node
cacheSize   = 20

#   after 100 protocol cycles the simulation will stop
cyclesInSimulation=100

#   After 30 cycles agents start working.
#   Aim of this parameter is to allow agents to start their computation
#   on the converged network.
agentMeetingsStart=30
```

A.2. Dynamic network

This is an example configuration file for the simulation in which communication graph properties are analyzed every 20th cycle, starting from the cycle 40. After 100 cycles the simulation stops. This time the network is dynamic. Every node works on average 20 cycles, then crashes and recovers on average during 2 cycles. Both the time between failures and the recovery time of a random node are defined by a random variable with exponential distribution.

```
# every node has the same crash/recovery time distribution
presenceManager = uniform
    # the mean time between failures (20 cycles)
    meanFaultlessWorkTime    = 20
    # ... and its distribution (exponential)
    distrFaultlessWorkTime   = exp
    # the mean recovery time (2 cycles)
    meanRecoveryTime     = 2
    # ... and its distribution (exponential)
    distrRecoveryTime    = exp

    # no joining or leaving nodes.
    distrLifeTime        = const
    meanLifeTime         = -1
    distrJoiningRate     = const
    meanJoiningRate      = -1

#   Network analysis will start after 40 cycles
measurementStart = 40

# the measurement will be performed every 20th cycle.
measureCycle = 20

#   Kind of observer, which will monitor the state of
#   the simulation and analyze the communication graph.
monitor = graphAnalysis
```

```
#   We do not need agents during network analysis
agent = None

#   Protocol which will be used: Newscast
protocol        = newscast

#   the number of nodes which take part in the simulation from the start.
initNumNodes    = 10000

#   the cache size of each node
cacheSize   = 20

#   after 100 protocol cycles the simulation will stop
cyclesInSimulation=100
```

Bibliography

[1] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson. *Epidemic algorithms for replicated database maintenance*, In Proceedings of the 6th Annual ACM Symp. Principles of Distributed Computing (PODC '87), pages 1–12, 1987.

[2] S. Vouglaris, D. G. Simonetti, M. van Steen. *Inexpensive membership management for unstructured P2P overlays*, submitted for publication, 2004.

[3] M. Jelasity, M. van Steen. *Large-scale newscast computing on the Internet*, Technical Report IR-503, Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands, Oct. 2002.

[4] A. Montresor, M. Jelasity, O. Babaoglu. *Robust aggregation protocols for large-scale overlay networks*, Technical Report UBLCS-2003-16, University of Bologna, Department of Computer Science, Dec. 2003.

[5] M. Jelasity, A. Montresor. *Epidemic-style proactive aggregation in large overlay networks*, to appear in the proceedings of The 24th International Conference on Distributed Computing Systems (ICDCS 2004).

[6] M. Jelasity, W. Kowalczyk, M. van Steen. *An approach to massively distributed aggregate computing on peer-to-peer networks*, In Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'04), pages 200–207, A Coruna, Spain, 2004. IEEE Computer Society.

[7] A. Stavrou, D. Rubenstein, S. Sahu. *A lightweight, robust P2P system to handle flash crowds*, IEEE J. Selected Areas Commun., 22(1):6-17, Jan. 2004.

[8] S. Voulgaris, M. Jelasity, M. van Steen. *A robust and scalable P2P gossiping protocol*, to appear.

[9] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, L. Massoulié. *Epidemic information dissemination in distributed systems*, IEEE Computer, vol. 37(5): 60-67. May 2004.

[10] R. Van Renesse, K. P. Birman, W. Vogels. *Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining*, ACM Transactions on Computer Systems, 21(2), May 2003.

[11] P. T. Eugster, R. Guerraoui, S. B. Handurukande, A.-M. Kermarrec, P. Kouznetsov. *Lightweight probabilistic broadcast*, ACM Transactions on Computer Systems, 21(4):341–374, 2003.

[12] N. Bailey. *The mathematical theory of infectious diseases and its applications*, Hafner Press, 1975.

[13] R. van Renesse, Y. Minsky, M. Hayden. *A gossip-style failure detection service*, in Middleware '98, Nigel Davies, Kerry Raymond, and Jochen Seitz, pp. 55–70, Springer, Eds. 1998.

[14] D. Kempe, A. Dobra, J. Gehrke. *Gossip-based computation of aggregate information*, In Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS'03), pages 482–491. IEEE Computer Society, 2003.

[15] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, Y. Minsky. *Bimodal multicast*, ACM Transactions on Computer Systems, 17(2):41–88, May 1999.

[16] Q. Sun, D. Sturman. *A gossip-based reliable multicast for large-scale high-throughput applications*, In Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN 2000).

[17] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, M. van Steen *The peer sampling service: experimental evaluation of unstructured gossip-based implementations*, to appear.

[18] R. van Renesse. *The importance of aggregation*, In A. Schiper, A. A. Shvartsman, H. Weatherspoon, B. Y. Zhao, editors, Future Directions in Distributed Computing, number 2584 in Lecture Notes in Computer Science, pages 87-92. Springer, 2003.