Warsaw University
Faculty of Mathematics, Informatics and Mechanics
Vrije Universiteit Amsterdam
Faculty of Sciences

Konrad Iwanicki
Student id. no.: 189391 (WU), 1493817 (VU)

Gossip-Based Dissemination of Time

Master's Thesis
in COMPUTER SCIENCE
in the field of DISTRIBUTED SYSTEMS

Supervisors:

Maarten van Steen and Spyros Voulgaris
Dept. of Computer Science,
Vrije Universiteit Amsterdam

and

Janina Mincer-Daszkiewicz
Institute of Informatics,
Warsaw University

May 2005

Abstract

Time synchronization between computers within a very large, highly dynamic network is a challenging task. Current solutions operate mostly in a hierarchical client-server mode based on a static configuration of logical connections, which tends to lack the scalability and robustness to failures.

In this thesis, the author presents the Gossiping Time Protocol (GTP) — an approach to time synchronization employing the theory of epidemics. GTP is a completely decentralized solution in which all the hosts form a peer-to-peer network. They gossip with each other in order to propagate accurate time. The algorithms constituting GTP have desired properties of scalability and robustness, while offering fast and quite accurate synchronization. Experimental results obtained with a prototype implementation on an emulated network of more than 64,000 hosts scattered across the machines of a wide-area cluster computer confirm the above claim.

Keywords

epidemic protocols, P2P networks, time synchronization, Gossiping Time Protocol, GTP, unstructured overlay, large-scale distributed system, large-scale experiments

Thesis domain (Socrates-Erasmus program codes)

11.3 Informatics

Subject classification

Category and subject descriptor according to ACM Computing Classification System:
C.2.4 [**Computer-Communication Networks**]:
Distributed Systems – *Distributed applications*
C.2.2 [**Computer-Communication Networks**]:
Network Protocols – *Protocol architecture*

To all my friends from VU and UvA that were with me in Amsterdam

– KI

Contents

Chapter 1

Introduction

Considering the expansion of the Internet in the last few years as well as the variety of applications being used, one may observe a gradual shift from the traditional client-server model towards peer-to-peer (P2P) systems. The key feature of these systems is symmetrical distribution of data and control among peers. The peers collaborate in order to carry out large-scale tasks in a simple manner. Moreover, the distribution is performed in such a way that processes are highly autonomous — they can join or leave at any time without severely disrupting the whole system. These factors clearly indicate scalability and robustness of P2P technology and make it suitable for building large-scale, easy-to-manage applications.

Except for scalability and fault tolerance, synchronization between processes is also one of the most crucial aspects of distributed systems. Despite the fact that many of them utilize algorithms based on some variants of logical clocks [31, Chap. 5.2], there are a lot of situations (especially in business) in which an accurate physical clock is required. Moreover, the ability to synchronize time on a wide scale over the Internet encourages developing novel techniques for solving well-known problems. Examples described in [16] include enforcing at-most-once message delivery semantics in the face of failures, maintaining cache consistency, using time-out tickets for authentication and achieving commitment in atomic transactions.

1.1. Contribution of the Thesis

Nowadays, a majority of time synchronization protocols (see Section 1.2.1) operate in a (hierarchical) client-server mode. Such approaches tend to have drawbacks concerning fault tolerance and scalability. Even the algorithms that are claimed to be adaptive to changes in the environment — in particular, crashes of the time server or network connection — are usually based on a static configuration of a logical network topology.

The author's research was focused on exploring the potential of gossiping (see Section 1.2.2) in the field of time synchronization. His main tasks included: designing, implementing and testing a time protocol for P2P networks. The thesis presents a set of developed algorithms based on different assumptions together with their properties. The implementations of the protocols were subsequently used to conduct a series of experiments emulating large-scale operation in a real network.

The results described in the thesis constitute a part of the GlobeSoul project [7]. The project aims at exploring possible applications of P2P technology for building large-scale distributed systems. Current research activities include designing and experimenting with scalable epidemic networks, super-peers, unstructured overlays and decentralized clustering.

1.2. Related Work

1.2.1. Time Protocols

There are currently many algorithms for time synchronization in the network. They differ in the techniques used, but the general principles usually remain similar. The time server is responsible for maintaining accurate time which is propagated across the network by either clients fetching the time (*pull-based* approach), or servers broadcasting the time (*push-based* approach). Alternatively, a combination of both methods can be used — *push-pull-based* approach.

The most popular time synchronization software is the Network Time Protocol ver. 3 (NTP) [20]. It can operate in several different modes — the most common one involves a passive server and active clients. There is also a simplified version — the Simple Network Time Protocol ver. 4 (SNTP) [22] — designed for end users. NTP distinguishes three types of nodes: primary time servers (with the most accurate time), secondary time servers (synchronize[1] with primary time servers or other secondary time servers) and clients (can synchronize with both types of servers, but do not provide their time information for other nodes). NTP is described in detail in Section 2.2.

Another example of a time protocol is the Digital Time Service (DTS). According to [20, 19], DTS has similar functional objectives as NTP, but emphasizes configuration management and correctness principles when used in a managed LAN or on a cluster, while NTP focuses on the accuracy when operating in the Internet. In the terminology of DTS, the network consists of time providers, couriers, servers and clerks. The time providers (primary time servers in NTP nomenclature) maintain the most accurate time which is imported by couriers (secondary time servers which are being synchronized with primary time servers in NTP) for local redistribution. The task of the servers (secondary time servers in NTP nomenclature) is to provide time for possibly many clerks (clients in NTP).

In the UNIX 4.3BSD, the time server (actually, a time daemon, called *timed*, of a master host) [8] periodically polls every slave host to obtain its time. Based on the answers, it computes a new time and tells all other machines to adjust their clocks. In this model, the master host is determined by an election algorithm. The election process requires broadcasting which is unavailable in WAN. Although the basic protocol was later adapted to work in the Internet in a hierarchical way, it requires manual configuration of the host hierarchy.

Other time synchronization protocols exist, e.g. Time Protocol [25] — the ancestor of NTP — or the protocol incorporated in the Fuzzball [18] routing algorithm, but they are not described in this paper. Further information and examples can be found in [15], [20] and [31, Chap. 5.1.2].

1.2.2. P2P Systems

Modern P2P systems usually consist of two or more layers. The lowest one is responsible for managing the topology of the logical network and for routing messages, whereas higher layers implement the functionality required by a specific application or group of applications. Depending on the design principles of the lowest layer, P2P systems can be divided into three categories.

The most popular one is dedicated to content-based searching (e.g. KaZaa [13]). The

[1] For the purpose of this chapter — *synchronizing A with B* denotes an activity after which A will correct its time to be equal (with given tolerance) to the time of B. The term *to synchronize* will be formally defined later.

majority of such systems operate with indexing peers that are dynamically constructed usually in the form of super-peers [36].

A second group consisting of systems known as distributed hash tables aims at efficient id-based routing of messages through a collection of intermediate peers. Examples include CAN [26], Chord [30], Pastry [27] and Tapestry [37].

Finally, the epidemic (gossip-based) protocols [31, Chap. 6.4.3] try to exploit randomness in order to disseminate some information across a large set of peers. Their origin comes from the theory of spreading diseases [2], which states that with even only one initially infected specimen, the whole population will be infected in an expected time proportional to the logarithm of the number of population members.

The general concept of gossiping is that every peer repeatedly contacts another 'random' peer to exchange information. A simple and scalable solution of choosing a peer to communicate with is allowing each peer to maintain a partial view of the network, which can be changed from time to time.

Some examples of application of epidemic protocols cover aggregation computation [14, 11], broadcasting [4], dealing with hot spots [29], load balancing [10], network management [33] and resource monitoring [32].

1.3. Overview

The rest of this thesis is organized as follows. The author starts with explaining the basic theory of time synchronization in distributed systems and gives an overview of NTP in Chapter 2. It is necessary to understand Chapter 3 describing the model of the system, Chapter 4 presenting developed time protocols and Chapter 5, which discusses the algorithms employed for maintaining the logical network for the purpose of time dissemination. Chapter 6 contains the analysis of the experimental results, in particular obtained by the large-scale emulation of the environment, while the utilized implementation is described in Chapter 7. Chapter 8 lists examples of possible applications of the designed algorithms and discusses the work for the nearest future. The thesis is concluded in Chapter 9.

Chapter 2

Time Synchronization

Throughout this thesis a standard terminology is adopted. Following [21], the *epoch* of an event is an abstraction which determines the ordering of events in some given frame of reference or *time-scale*. An *oscillator* is a generator capable of precise frequency (relative to the given time-scale) within a specified *tolerance*. A *clock* contains an oscillator and a counter which records the number of cycles since being initialized with a given value at a given epoch. The value of the counter at epoch t defines the *time* of that epoch $T(t)$.[1]

Let $T(t)$ be the time displayed by a clock at epoch t relative to the standard time-scale:

$$T(t) = T(t_0) + R(t_0)\frac{(t-t_0)}{1!} + D(t_0)\frac{(t-t_0)^2}{2!} + \rho(t),$$

where $T(t_0)$ is the time at some previous epoch t_0, $R(t_0)$ is the frequency and $D(t_0)$ is the *drift* (first derivative of frequency) per unit time. T and sometimes R are estimated in the synchronization process, while D and ρ (characterizing the random nature of the real clock) are assumed to be zero.

The *stability* of the clock indicates how well it can maintain a constant frequency, the *accuracy* is how well its time compares with the reference time and the *precision* is the granularity of the clock. The *time offset* (or simply *offset*) of clock A relative to clock B is the time difference between them $T^{AB}(t) = T^A(t) - T^B(t)$ at a particular epoch t, while the *frequency offset* (or just *skew*) is the frequency difference between them $R^{AB}(t) = R^A(t) - R^B(t)$. For all t the following equations are true: $T^{AB}(t) = -T^{BA}(t)$, $R^{AB}(t) = -R^{BA}(t)$, $T^{AA}(t) = 0$ and $R^{AA}(t) = 0$.

As the reference time, the Universal Coordinated Time (abbreviated as UTC) [31, Chap. 5.1] is assumed. It is later denoted as $T^{REF}(t)$.

2.1. Synchronization Theory

Synchronization of the clocks in the network requires a way to compare them — directly or indirectly — in time and, possibly, in frequency. There are many ways of doing this comparison. The one presented here was widely adopted (e.g. NTP) and is similar to Cristian's algorithm [31, Chap. 5.2.1]. Again following a standard nomenclature, to *synchronize the time* of the clocks means to set them to agree at a particular epoch with respect to UTC, to *synchronize their frequency* means to adjust the clocks to run with the same frequency. To *synchronize clocks* means to synchronize them in both time and frequency.

[1] In general, time is not continuous and depends on the precision of the counter.

The *time server* is a host with authoritative time. Such time can be maintained with special, commercially available hardware and by means of short-wave radio broadcasts or satellite services for UTC. Alternatively, depending on the required accuracy, standard crystal-based oscillator, which requires manual adjustments from time to time, can be used. Parameters of the most popular technologies are presented in [20, Appendix E].

Other (*slave*) hosts in the network own their *local clocks*.

2.1.1. Basic Algorithm

A time server (B) is passive. Each slave host (A), on the other hand, periodically sends a synchronization request message to the time server and also records a timestamp T_1^A (see Figure 2.1) according to its local clock.

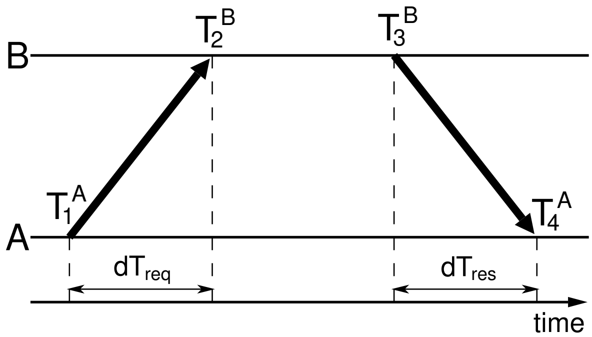


Figure 2.1: Measuring round-trip delay and offset.

Immediately after reception of a synchronization message the time server records a time T_2^B according to its clock and starts to prepare a response message containing the recorded time. When the message is ready, the time server records a time T_3^B, stores it within the response (together with T_2^B) and sends the message back to the slave host.

As soon as the synchronization response is delivered to the host, the latter records a time T_4^A according to its clock. At that point, slave host A has the following times: T_1^A, T_2^B, T_3^B and T_4^A. These values allow A to compute the offset of B and a *round-trip delay*.

Let:

$$T_1^A = T^A(t_1), \qquad T_2^B = T^B(t_2), \qquad T_3^B = T^B(t_3) \qquad \textit{and} \qquad T_4^A = T^A(t_4),$$

where $t_1 \leq t_2 \leq t_3 \leq t_4$ indicate epochs of appropriate events. Without loss of generality, assume $T^A(t_1) \leq T^A(t_4)$ and $T^B(t_2) \leq T^B(t_3)$ (time is not running backwards). Also, for the moment assume that both clocks are stable and run at the same rate (which implies $T^A(t_4) - T^A(t_1) \geq T^B(t_3) - T^B(t_2)$). In addition, as shown in Figure 2.1:

$$dT_{req} = T^{REF}(t_2) - T^{REF}(t_1) \qquad \textit{and} \qquad dT_{res} = T^{REF}(t_4) - T^{REF}(t_3).$$

If the difference between propagation delays from A to B and from B to A, called *differential delay*, is small ($dT_{req} \simeq dT_{res}$), the round-trip delay δ and clock offset θ of B relative to A at time T_4^A (epoch t_4) are close to:

$$\delta = T_4^A - T_1^A - (T_3^B - T_2^B) = T_4^A - T_1^A + T_2^B - T_3^B \tag{2.1}$$

$$\theta = T_3^B + \frac{\delta}{2} - T_4^A = \frac{T_2^B - T_1^A + T_3^B - T_4^A}{2} \tag{2.2}$$

Based on the θ value, host A can adjust its time in order to synchronize with host B by either changing the frequency of its clock for a certain amount of time, or resetting the clock immediately to the appropriate time. The value of δ provides the capability to send a message to arrive at B at a specified time. It is sometimes also used for estimating a *dispersion* (denoted as ε), which represents the expected maximal error of the local clock relative to the reference clock.

2.1.2. Synchronization Errors

Exhaustive analysis of errors for the presented algorithm is part of the work of D.L. Mills. Because of its length, it is not repeated here. The details and further references can be found, for example in [20, Appendix H] and [21]. However, important issues are the types and the reasons of errors that are listed below.

1. Errors in reading clocks of both hosts, which depend on the precision of the clocks and the method of adjustment.
2. Errors due to the frequency tolerance of the clocks since their time was last set.
3. Errors contributed by delay variations in the network and in the operating system on the path to the time server.
4. Errors contributed by multiple time servers used for synchronizing the local clock, which depend upon the differences between members of the time servers set.

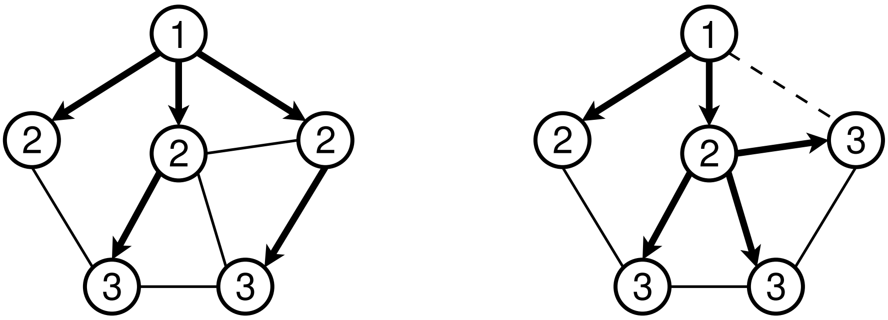
According to [21], in practice, errors due to network delays dominate all others. Moreover, if a hierarchical synchronization network is assumed — host A is a time server for host B, host B is a time server for host C, and so on — the errors accumulate — the time of B will be influenced by errors in synchronization with A, the time of C will be influenced by the previous errors of B and errors in synchronization with B, and so on.

2.2. Application Example: NTP

The Network Time Protocol [20] evolved from the Time Protocol [25]. It operates on top of UDP and can run in several modes, e.g. unicast and broadcast, private workstations, public servers, etc. The accuracy of time provided by NTP varies from 1 to 10 milliseconds in local area networks to tens or even hundreds of milliseconds in the Internet [21].

The model of NTP introduces a *synchronization subnet* — a network consisting of primary and secondary time servers, clients and transmission paths. A primary time server is directly synchronized to a reference time source — usually a UTC radio clock. A secondary time server synchronizes, possibly via other secondary servers, with one or more primary servers over network transmission paths, possibly shared with other services. Clients can synchronize with both types of servers. The synchronization subnet assumes a hierarchical master-slave configuration with primary servers at the root and secondary servers of decreasing accuracy at successive levels towards the leaves.

An example of a synchronization subnet is shown in Figure 2.2a, in which the nodes represent servers and the edges transmission paths. The accuracy of the server is defined by a *stratum*, which indicates the hop count from the primary (stratum 1) server. Bold lines indicate active transmission paths with time information flow marked with arrows. Normal lines represent backup synchronization paths.



(a) Before crash (b) After crash

Figure 2.2: Sample synchronization subnet.

Figure 2.2b shows the same subnet after a crash of a communication path (dashed line). NTP reconfigures itself to use backup paths, which results in one of the servers having increased its stratum from 2 to 3.

In order to improve the quality of time provided by a secondary server to its users, the server usually synchronizes with many other primary or secondary servers. Moreover, the synchronization samples estimated by the algorithm described in Section 2.1.1 are examined for possible errors by estimating their dispersion. This value allows NTP to reject samples that might have experienced excessive network delay and should not be used for synchronization. After filtering the data (per transmission path) the intersection and clustering algorithms are used to select from among all servers a suitable subset capable of providing the most accurate and trustworthy time. That process tries to divide the servers into two disjoint sets of *truechimers*, which own correct clocks, and *falsetickers*, which may not. Finally, the samples from selected servers are combined and the clock adjustment can be started.

All these steps correspond to building a directed acyclic graph (DAG) from primary time servers to a given secondary server or client, with edge weights minimizing the estimated synchronization error. More information about these algorithms, which are too elaborate to be repeated in the thesis, can be found in [21] and [20].

For the same reason, the simplified version of NTP called the Simple Network Time Protocol [22] was designed. It can cooperate with NTP or be utilized on its own. However, it is strongly recommended to use SNTP only for the end nodes (the highest stratum) of the synchronization subnet.

Chapter 3

System Model

The model of the system for the developed time protocols assumes a logical network formed by *nodes*. In this network there is at least one node maintaining accurate time, called a *time source*. Other nodes are equipped with local clocks capable of time-keeping with a reasonable accuracy and stability, e.g. default crystal-based oscillator timers.

According to the system architecture invented by the author, the protocol-related software of a node consists of layers (see Figure 3.1), described in detail later in this chapter.

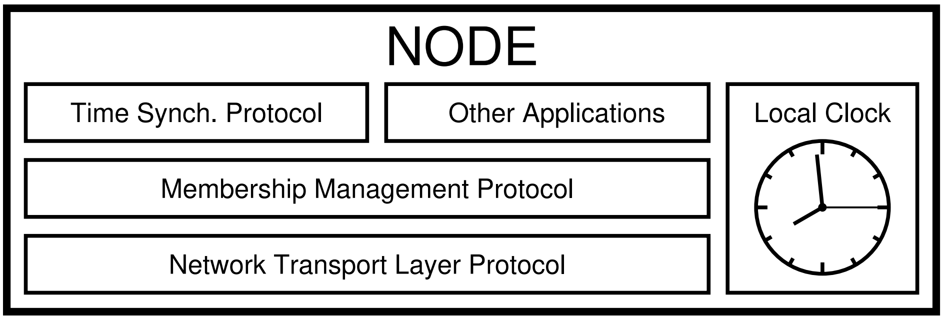


Figure 3.1: Layers of a single node.

The lowest layer is formed by an operating-system-level protocol designed for message passing. The middle layer is responsible for membership management of the logical network. Finally, the time synchronization protocol resides in the highest (application) layer. Such design is very common in distributed systems and allows sharing or replacing some parts without affecting the others [31, Chap. 1].

3.1. Local Clock Model

The local clock model assumed for the system is a standard one — described in Chapter 2. The clock consists of an oscillator and a counter. However, two variants of adjustment methods, called *immediate* and *gradual*, are distinguished. They differ in the way the time correction is applied to the clock. Although both methods can be utilized interchangeably, the thesis introduces two distinct clock models which are named after the methods used for time correction.

3.1.1. Immediate Adjustment

The programming interface of the immediate model consists of two operations[1] which are specified in Listing 3.1.

```
interface Clock;
begin
    function  getTime():integer;
    procedure correctTimeIm(offset:integer);
end;
```

Listing 3.1: Interface for the immediate clock model.

Function `getTime()` returns the current time, while procedure `correctTimeIm(offset)` increases the value of the current time by the value of the parameter `offset`[2]. Such functionality can be easily implemented with basic arithmetic operations on the counter of the clock.

3.1.2. Gradual Adjustment

The gradual model was designed to meet the requirement that time cannot be set backwards. Such an assumption is crucial for some applications (e.g. the `make` program). The adjustment complying with the above specification can be performed in a variety of ways, therefore this section presents only a solution implemented by the author.

Instead of moving the clock directly, the method simulates either slowing the clock down or speeding it up for a period of time necessary for the time changes to be completed. More specifically, the *clock adjustment period* indicates the number of time units during which one time unit will be either added (simulating a fast clock) or subtracted (simulating a slow clock) from the time. This operation is performed during the last time unit of the clock adjustment period. For the user of the clock it seems like that time unit was lasting for half of a normal time unit (fast clock) or two normal time units (slow clock). If time correction requires to move a clock by N time units, N clock adjustment periods are required.

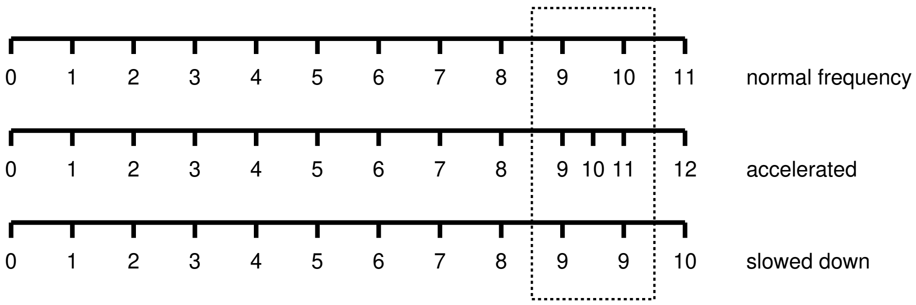


Figure 3.2: Gradual clock adjustment (only one clock adjustment period of the length of 10 time units is depicted). Because of limited precision of the clock, value 10 of the accelerated clock will not appear.

[1] In all specifications and algorithms it is assumed that the time is represented by an **integer** type of a size capable of storing all required values. It can for example denote the number of milliseconds since the beginning of the millennium.

[2] Which can be negative.

Figure 3.2 shows an example of gradual time adjustment and the interface of a gradually-adjustable clock is presented in Listing 3.2.

```
interface Clock;
begin
    function  getTime():integer;
    procedure correctTimeGr(offset:integer);
    function  getCorrection():integer;
end;
```

Listing 3.2: Interface for the gradual clock model.

Function `getTime`() returns the current time, like in the immediate model, procedure `correctTimeGr`(`offset`) starts the adjustment process of `offset` time units and function `getCorrection`() returns the number of time units by which the time has to be further corrected (0 indicates that no adjustment process is active). If operation `correctTimeGr`(`offset`) is called when the clock is being adjusted then the number of time units remaining to adjust is overwritten with the `offset` value.

That adjustment mode can be implemented with a supplementary counter storing the number of time units to be either added or removed from the normal counter.

3.2. Membership Management

One of the major assumptions concerning the system is that the logical network connecting the nodes can be very large and dynamic. Therefore information about all nodes should be distributed across the system — solutions using centralized components for membership management or forcing every node to store information about all other members are unacceptable. Moreover, crashes of nodes (including time sources) or communication paths are considered normal and should not disrupt the operation of the system.

The problem is easily solvable if the P2P protocols, described in Section 1.2.2, are employed for handling membership. In particular, a node is assumed to know some number of other nodes called its *neighbors*. It can directly exchange information (gossip) only with these nodes in a P2P fashion. The neighborhood relation is represented by a directed graph called the *overlay graph* (or, simply, the *overlay*). Because of the fact that the time synchronization protocols designed by the author operate in an epidemic manner, the overlay graph can be either structured (e.g. a multidimensional torus in CAN [26], or a lattice in Chord [30]), or unstructured (e.g. CYCLON [35], SCAMP [6]) — the important issue being that it has good data dissemination properties (see Chapter 5).

The neighborhood information has to be supplied for the time synchronization protocol (and possibly other applications). To accomplish that task the author extended the concept of a *peer sampling service* [12]. The peer sampling service acts as a connection between the membership/overlay management layer and the application layer. Its original interface is presented in Listing 3.3.

```
interface PeerSamplingService;
begin
    function getNeighbor():NeighborData;
end;
```

Listing 3.3: Basic interface for the peer sampling service.

The sole function `getNeighbor`() is to be called by the application whenever it is willing to exchange data with another node. The result of this function (`NeighborData`) contains information necessary to contact that node (e.g. an IP address and a port number of the application). The task of the membership management layer is to choose one of the node's neighbors and pass its data to the requesting application. The operation of the peer sampling service depends only on the implementation of the overlay layer. It can, for example, choose a random neighbor or the first neighbor from the list (with the ordering enforced by the overlay management protocol).

The extended version (see Figure 3.3) proposed by the author allows to store within the neighborhood information not only data necessary to contact a given node, but also some (small) amount of application-specific data.

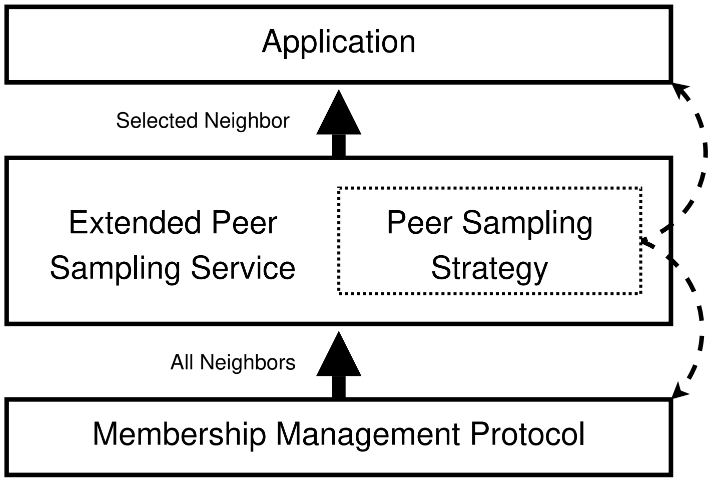


Figure 3.3: Overview of the extended peer sampling service. The bold arrows show information flow, while the dashed arrows show dependencies.

This approach allows to implement different *peer sampling strategies*. The peer sampling strategy, based on the application data, may choose the best suited neighbor to be contacted (by the application) from the set of neighbors currently maintained by the membership management protocol. The new interface of the peer sampling service is presented in Listing 3.4.

```
interface PeerSamplingServiceEx;
begin
    function  getNeighbor():NeighborDataEx;
    procedure setSamplingStrategy(
              strategy:PeerSamplingStrategy);
    callback  getApplicationData():Object;
end;
```

Listing 3.4: Interface for the extended peer sampling service.

Function `getNeighbor`() using the current peer sampling strategy chooses a neighbor to be contacted and returns its contact information and application-specific data to the caller. Procedure `setSamplingStrategy(strategy)` changes the current sampling strategy. The `strategy` object can be both overlay management layer and application specific. Finally, callback `getApplicationData`() is implemented by a given application. Its task is to provide the membership management protocol with current application data of the same node. That callback is usually called when information about the node is to be sent to one of its neighbors as a part of the overlay protocol message.

In further chapters of the thesis, the extended peer sampling service is assumed. The peer sampling strategy and application-specific data utilized by described time synchronization protocols are always stated explicitly.

3.3. Time Synchronization Protocol

The goal of the time synchronization protocol is to keep time synchronized with the time source. A node does not, in general, have information allowing it to contact the time source directly. It may only establish its time by gossiping with, the possibly changing, set of its neighbors accessible via the peer sampling service.

It should be noted that clock synchronization requires by nature long periods and many comparisons in order to maintain accurate time-keeping. While only a few measurements are usually adequate to reliably determine local time to a reasonable accuracy, periods of many hours and tens of measurements (involving exactly the same nodes) are required to resolve oscillator skew [20]. This is the main reason that protocols designed by the author do not aim at synchronizing frequency. Another reason is the fact that services provided by operating systems usually do not contain functions allowing to change the frequency of the system clock.[3]

From now on the term to *synchronize clocks* denotes synchronizing clocks in time, but not in frequency.

[3] Of course such operations can be simulated by a user-space software layer that is placed on top of kernel services, but applications using kernel functions directly would not benefit from it.

Chapter 4

Gossiping Time Protocol

The time synchronization algorithm designed by the author has been called the Gossiping Time Protocol (GTP). There are three major variants of GTP, presented later in this chapter. For the purpose of clarity, the thesis does not describe precisely every aspect of the protocol (e.g. order of fields in the messages, unified format of time for heterogeneous environments, ways of dealing with different time zones or handling errors). It aims at explaining the algorithms developed for synchronization along with their properties.

Gossiping in GTP follows a timestamp exchange pattern similar to the algorithm described in Section 2.1.1. However, simple application of that algorithm would result in the pull-based only information dissemination strategy. To achieve the push-pull-based approach, which has much better properties [12], additional actions are required. More specifically, node A initializes a timestamp exchange (gossiping) by sending a *request* message to node B. Upon reception of a request by node B, it performs all necessary operations and sends a *response* back to node A. Node A, based on the sample estimated from the response, may either synchronize its clock to the clock of B, send a *feedback* message to B, or ignore the sample. If node B receives the feedback message, it may use it to synchronize its clock to the clock of node A. Such a solution, among other things, allows a time source to be pro-active, which increases the speed of time dissemination — the time source may initiate gossiping with other nodes and then always send feedback messages.

4.1. Common Elements

All algorithms involve two separate paths of execution[1] called *active* and *passive*. For the purpose of the description of GTP, the following symbols are used:

- `CLOCK` — a module providing access to the local clock with one of the interfaces specified in Section 3.1 (exact clock version is stated for each algorithm separately);
- `PSS` — a module implementing extended peer sampling service conforming to the interface described in Section 3.2 (peer sampling strategy and application-specific data utilized are a part of a particular GTP variant);
- `GOSSIPING_DELAY` — a variable storing the current delay (in time units) between consecutive GTP gossiping attempts;

[1] The author avoids using the word *thread*, because the algorithms can be implemented in both single-threaded and multi-threaded manner. Moreover, an efficient implementation would preferably utilize a pool of threads for handling synchronization requests from other nodes on multi-processor machines.

- `TIME_SOURCE` — a boolean value[2] which is true if, and only if, the given node is a time source.

Additionally the following operations are available:

- **procedure** `sleep(n:`**integer**`);`
  — suspends execution of the current path for `n` time units;

- **procedure** `send(m:Message;a:NetworkAddress);`
  — sends message `m` to the network address specified by `a`;

- **function** `receive(`**out** `m:Message;` **out** `a:NetworkAddress);`
  — waits for a message and, when it arrives, stores it in `m` (variable `a` receives address of the sender).

The algorithms do not require neither reliable, nor ordered message delivery. It is assumed that the type `Message` contains a flag that can be equal to one of the following values: `MSG_REQUEST`, `MSG_RESPONSE` or `MSG_FEEDBACK`. This flag is referred to by a field `msgType`. Depending on the value of the flag, the message may contain up to four timestamps denoted by fields: `t1`, `t2`, `t3` and `t4`, corresponding to the symbols used in the synchronization algorithm described in Section 2.1.1.

The active execution path is responsible for initiating gossiping with another node. Its pseudo-code is presented in Listing 4.1.

```
1  var
2      target    : NeighborData;
3      request   : Message;
4
5  while true do
6  begin
7      sleep(GOSSIPING_DELAY);
8      target := PSS.getNeighbor();
9      if target <> null then
10     begin
11         request.msgType := MSG_REQUEST;
12         PrepareRequest(request);
13         request.t1 := CLOCK.getTime();
14         send(request, target.gtpAddress);
15     end;
16 end;
```

Listing 4.1: GTP active execution path.

In the infinite loop, first, GTP waits for `GOSSIPING_DELAY` time units (7) before starting synchronization. After waking up, it retrieves a neighbor to gossip with from the peer sampling service (8). If the node is connected to the network (9), the message with GTP request is prepared (11-12), timestamped (13) and sent (14) to the address of the GTP layer of the chosen neighbor.

Procedure `PrepareRequest(m:Message)` is responsible for setting supplementary message fields that depend on a particular variant of GTP.

The pseudo-code of the passive execution path, whose task is handling requests from other nodes and responses to the node's own requests, is presented in Listing 4.2.

[2] Although presented versions of GTP do not modify this value, it can be used as a variable if the fault tolerance is considered. More specifically, if the time source crashes, another node can take over its role.

```
1  var
2      sender    : NetworkAddress;
3      message   : Message;
4      timestamp : integer;
5
6  while true do
7  begin
8      receive(message, sender);
9      timestamp := CLOCK.getTime();
10     case message.msgType of
11         MSG_REQUEST:    (* t1 is set *)
12             begin
13                 message.t2 := timestamp;
14                 message.msgType := MSG_RESPONSE;
15                 PrepareResponse(message);
16                 message.t3 := CLOCK.getTime();
17                 send(message, sender);
18             end;
19         MSG_RESPONSE:   (* t1, t2 and t3 are set *)
20             begin
21                 message.t4 := timestamp;
22                 if ExamineSample(message) then
23                 begin
24                     message.msgType := MSG_FEEDBACK;
25                     PrepareFeedback(message);
26                     send(message, sender);
27                 end;
28             end;
29         MSG_FEEDBACK:   (* t1, t2, t3 and t4 are set *)
30             begin
31                 ExamineSample(message);
32             end;
33     end;
34 end;
```

Listing 4.2: GTP passive execution path.

Inside the infinite loop, GTP waits for a message from an arbitrary node (8). Immediately after reception, it records the current time (9). If the message contains a synchronization request sent by an active path of another node (11), then the response is prepared (13-15), timestamped (16) and returned to the sender (17). On the other hand, if the message is a response to the request of the current node (19), it is examined for possible application for clock adjustment (22), based on criteria described in detail for each version of the algorithm separately. If the information from the message can be utilized by the sender to synchronize its clock[3], a feedback message is prepared (24-25) and returned to the sender (26). Finally, if the received message is a feedback sent by another node (29), it is examined, in a similar way, for possible synchronization usage (31).

Procedures `PrepareResponse(m:Message)` and `PrepareFeedback(m:Message)` are responsible for setting fields specific to a GTP variant. The task of the function `ExamineSample(m:Message):`**boolean** is to estimate whether a given message can be used for

[3] Such situation implies that the current node does not use the received message for synchronization of its clock.

synchronization purposes, and, if so, to apply the computed offset to the local clock. If the information cannot be used by the current node, but there is a possibility that the sender can use it, the function returns true. Otherwise, false is returned.

To sum up, all versions of GTP follow the framework presented above. To fully describe a given algorithm, the following issues have to be specified:

1. Chosen clock adjustment model.
2. Application-specific data for the overlay layer and a peer sampling strategy.
3. Additional variables and objects.
4. Additional fields in the `Message` type.
5. Procedures `PrepareRequest`, `PrepareResponse` and `PrepareFeedback`.
6. Function `ExamineSample`.

The following sections discuss GTP in detail, starting from the simplest algorithm to the most elaborate one.

4.2. Simple Algorithm

The initial approach (denoted as *basic* version of GTP) is based on the immediate clock adjustment model. It does not require any application-specific data for the overlay layer. The neighbor to gossip with is always chosen as a random one from the neighbor set provided by the membership management protocol, which corresponds to the classical definition of gossiping.

In order to have a mechanism allowing for estimating quality of time provided by nodes, an additional **integer** variable `TS_DISTANCE` is employed. It records the distance (hop count) of a given node from the time source, which is similar to a stratum number in NTP. In particular, the value of `TS_DISTANCE` is always equal to zero for a node that is a time source. For other nodes it is initially set to infinity. Whenever node A adjusts its clock after gossiping with node B, its `TS_DISTANCE` value is set to the value of this variable for node B incremented by one. This solution requires an additional field in the message (denoted as `tsDistance`), containing the `TS_DISTANCE` value of the sender.

Moreover, the time displayed by the clock of a node may be adjusted during the exchange of synchronization messages between two nodes, if one of them is contacted by some other node. Such situations cause timestamps carried by a message to be useless and the algorithm must be able to detect it. This is one of the reasons for introducing the variable `LAST_UPDATE` (of type **integer**) storing the time of the last update of the clock according to the new local time of a node. This variable can also be used for different purposes, as described later in this section. In addition, the message is extended by two **integer** fields denoted as `luSender` and `luReceiver` storing the values of `LAST_UPDATE` for appropriate nodes.

The implementations of procedures `PrepareRequest`, `PrepareResponse` and `PrepareFeedback`, presented in Listing 4.3, are straightforward. The only operation they perform is copying the values of `TS_DISTANCE` and `LAST_UPDATE` of the node to the appropriate message fields.

The most important part of the algorithm is the function `ExamineSample`, whose goal is to decide whether to synchronize the clock and send a feedback message. Its pseudo-code is presented in Listing 4.4.

```
procedure PrepareRequest(m:Message);   (* executed by A *)
begin
    m.luSender := LAST_UPDATE;
end;

procedure PrepareResponse(m:Message); (* executed by B *)
begin
    m.luReceiver := LAST_UPDATE;
    m.tsDistance := TS_DISTANCE;
end;

procedure PrepareFeedback(m:Message); (* executed by A *)
begin
    m.tsDistance := TS_DISTANCE;
end;
```

Listing 4.3: Message preparation procedures for *basic* GTP.

In the beginning (6-8), the algorithm determines if the samples have any synchronization value, i.e. whether the clock of the node was adjusted during the message exchange. If this is the case, the function returns indicating that received information cannot be used as a feedback message.

Otherwise, the node checks whether it is a time source (10) and, if so, it leaves the function informing that the feedback message can be sent to the other node.

The goal of the next statements is to determine whether the node is going to synchronize its clock. First (12), it is checked whether the other node was synchronized before. After that the node checks whether its hop count is smaller than or equal to the hop count of the other node (13), effectively, evaluating the quality of the time provided by that node. Alternatively, it may turn out that the node has not been synchronized for a long time and it is willing to accept the sample, even despite the fact that it may degrade its time quality (14-15). The reason for the last heuristic is the ability to deal with skewed clocks that require periodical corrections.

If the node decides to accept the sample, it synchronizes its clock (17-19), updates other variables (20-21) and returns indicating that a feedback message is not needed.

Otherwise, it checks if its time was synchronized before and, based on the result, it indicates whether a feedback message can be sent (25).

Table 4.1 summarizes the contents of the particular messages exchanged during a single gossiping process. As stated before, the feedback message is optional.

msgType	MSG_REQUEST	MSG_RESPONSE	MSG_FEEDBACK
t1	T_1^A	T_1^A	T_1^A
t2	-	T_2^B	T_2^B
t3	-	T_3^B	T_3^B
t4	-	-	T_4^A
tsDistance	-	TS_DISTANCE^B	TS_DISTANCE^A
luSender	LAST_UPDATE^A	LAST_UPDATE^A	-
luReceiver	-	LAST_UPDATE^B	LAST_UPDATE^B

Table 4.1: Contents of different messages of *basic* GTP.

```
function ExamineSample(m:Message):boolean;
var
    msglu  : integer;
    offset : integer;
begin
    msglu := m.msgType = MSG_RESPONSE ?
        m.luSender : m.luReceiver;
    if msglu <> LAST_UPDATE then return false;

    if TIME_SOURCE then return true;

    if (m.tsDistance < ∞ and
            (TS_DISTANCE > m.tsDistance or
                CLOCK.getTime() − LAST_UPDATE >=
                    _STANDALONE_PERIOD_)) then
    begin
        offset := (m.t2 − m.t1 + m.t3 − m.t4) / 2;
        if m.msgType = MSG_FEEDBACK then offset := −offset;
        CLOCK.correctTimeIm(offset);
        TS_DISTANCE := m.tsDistance + 1;
        LAST_UPDATE := CLOCK.getTime();
        return false;
    end;

    return TS_DISTANCE < ∞;
end;
```

Listing 4.4: Basic clock synchronization procedure for *basic* GTP.

4.2.1. Observations

For the simple theoretical analysis assume that all nodes gossip with the same frequency and let t_i (where $i = 0, 1, 2, ...$) denote some epoch before the $i + 1$-st gossiping period, but after the i-th gossiping period (if such exists). Moreover, let $S^A(t_i)$ represent the state of node A at epoch t_i. Function

$$S^A(t_i).ERROR = |S^A(t_i).CLOCK.getTime() - T^{REF}(t_i)| \tag{4.1}$$

indicates an absolute error in time displayed by a clock of node A at epoch t_i. If the node A is a time source, the following condition is met:

$$S^A(t_0).ERROR = 0 \qquad and \qquad S^A(t_0).TS\_DISTANCE = 0, \tag{4.2}$$

otherwise:

$$S^A(t_0).TS\_DISTANCE = \infty. \tag{4.3}$$

Additionally, for all A:

$$S^A(t_0).LAST\_UPDATE = -\infty. \tag{4.4}$$

Without loss of generality, assume that nodes A and B gossip with each other during the $i + 1$-st gossiping period. If the message delivery is reliable, the algorithm ensures that:

$$S^A(t_i).TS\_DISTANCE < \infty \quad or \quad S^B(t_i).TS\_DISTANCE < \infty \Longrightarrow \tag{4.5}$$

$$S^A(t_{i+1}).TS\_DISTANCE < \infty \quad and \quad S^B(t_{i+1}).TS\_DISTANCE < \infty.$$

If the network differential delay is equal to zero, the offsets calculated using the timestamps from exchanged synchronization messages are equal to the real offsets between the clocks. If, additionally, the clock frequencies of all nodes are equal to the perfect frequency, then for all $i \leq j$:

$$S^A(t_i).ERROR = 0 \Longrightarrow S^A(t_j).ERROR = 0 \tag{4.6}$$

and also:

$$S^A(t_i).TS\_DISTANCE < \infty \Longrightarrow S^A(t_i).ERROR = 0. \tag{4.7}$$

It should be noted, that the formulas above allow to estimate the speed of time dissemination in a perfect environment. By combining them together, one may notice that if during a gossiping cycle two nodes, one of which has already been synchronized, exchange timestamps, then both of them will be synchronized afterwards. Such behavior is similar to the model of spreading infectious diseases, mentioned earlier. On the assumption that the peer sampling service provides GTP with an overlay graph of appropriate properties, the expected time of network synchronization should depend logarithmically on the size of the network.

Experiments conducted by the author (see Chapter 6) indicate that, indeed, the latter statement is true. However, the assumption of small network differential delays is not very realistic and special measures have to be taken to alleviate the influence of these errors on the accuracy of a time synchronization algorithm.

4.2.2. Improvements

The approach to dealing with network differential delays, introduced in *basic* GTP utilizes *sample filtering*. To be specific, each node owns a cyclic buffer (called *filter*) of size N holding round-trip delays of the last N samples that could have been potentially used for synchronization. Initially, the buffer is filled with infinite values.

For each sample that passes lines 6-10 of the `ExamineSample` function (see Listing 4.4), the round-trip delay is estimated and stored within the filter, before checking the conditions in lines 12-15. Later, if the sample satisfies these conditions, a value of some statistical function (e.g. a median or an average) on the round-trip delays stored within the buffer is estimated. That value is then compared to the round-trip delay of the current sample (possibly with some weights). If this (weighted) round-trip delay is smaller than or equal to the (weighted) value of the statistical function, then the sample can be used for synchronization purposes (lines 17-21). Otherwise, function `ExamineSample` exits without synchronizing the clock, but returning true — although the sample was not appropriate for the current node, it may be utilized by the other node, so it can be sent as a feedback message.

Despite its simplicity, the sample filtering solution addresses two important problems. Firstly, it is capable of rejecting the samples with round-trip delays higher than usual. This does not have to be equivalent to rejecting samples with high differential delays. However, without well-synchronized clocks it is not possible to measure differential delays during timestamp exchange. On the other hand, high differential delays may be caused by a packet being slowed down on its way (in one direction), which leads to a higher round-trip delay. Moreover, the filter is adaptive. When round-trip delays increase because of high network load, the buffer is gradually filled with these increasing values and after some time (depending on the size of the buffer and function used) a sample will pass filtering conditions.

4.3. Refined Algorithm

The next algorithm (referred to as *gradual* version of GTP) aims at improving the accuracy of time synchronization in the presence of network differential delays. Like the previous version, it does not require any application-specific data for the overlay management layer and employs random selection as the peer sampling strategy. The meaning of variables `LAST_UPDATE` and `TS_DISTANCE` remains unchanged.

The first major modification to *basic* GTP is using the gradual clock adjustment model, fulfilling the requirement of constant time flow.

Another problem the algorithm addresses is improving the granularity of the time quality information maintained by nodes. In the previous version of GTP, the quality of time of a node is determined only by the value of the `TS_DISTANCE` variable — the nodes with higher values are considered to be less useful for synchronization. However, this is not always the case, because if during synchronization between nodes A and B (with a small hop count) an undetected (by the filter) error caused by the network differential delay occurs, the time of node A may be much worse than the time of some other node with higher hop count. Moreover, there is a high probability that many nodes will become unsynchronized due to such an error and errors accumulating on the synchronization paths from node A to these nodes.

The solution requires every node to store an additional **integer** variable, called `DISPERSION`, which contains the quality of the last sample used by the node for synchronization. Additionally, GTP messages are extended by a field `dispersion` of the same type.

It is also assumed that the frequency and the tolerance of the clock are known[4]. In particular, during `_TD_` time units the error of the clock may change by at most `_TN_` time units. Message preparation procedures in Listing 4.5 present the usage of the new elements (procedure `PrepareRequest` is empty).

```
procedure PrepareResponse(m:Message); (* executed by B *)
begin
    m.tsDistance := TS_DISTANCE;
    m.dispersion := CalculateDispersion();
end;

procedure PrepareFeedback(m:Message); (* executed by A *)
begin
    m.tsDistance := TS_DISTANCE;
    m.dispersion := CalculateDispersion();
end;

function CalculateDispersion():integer;
begin
    return DISPERSION + abs(CLOCK.getCorrection()) +
        ceil((_TN_ * (CLOCK.getTime() - LAST_UPDATE)) / _TD_);
end;
```

Listing 4.5: Message preparation procedures for *gradual* GTP.

Function `CalculateDispersion` tries to estimate an expected error that the time of given node may exhibit under the nominal operating conditions.[5] That error incorporates the

[4] Such information is usually a part of a clock specification.

[5] The formulas used for estimating errors have been inspired by the research concerning NTP.

estimated error of the last sample used by the node for synchronization, the time correction, that has still to be applied to the clock in the gradual adjustment process, and a possible error due to the clock skew. For simplicity of the algorithm description, it is assumed that infinity is an ordinary value. Comparisons of ∞ with other values have intuitive results. Moreover, $x \pm \infty = \pm\infty$ for all x and $x \cdot \pm\infty = sign(x) \pm \infty$ for all x except 0.

Listing 4.6 shows the clock adjustment function `ExamineSample`. In the beginning of the synchronization process, the value of `DISPERSION` is zero for the time source and infinity for other nodes.

```
1  function ExamineSample(m:Message):boolean;
2  var
3      offset, roundtripDelay        : integer;
4      ourDispersion, msgDispersion  : integer;
5  begin
6      if TIME_SOURCE then return true;
7
8      if m.dispersion = ∞ then
9          return DISPERSION < ∞;
10
11     roundtripDelay := m.t4 − m.t1 + m.t2 − m.t3;
12     ourDispersion := CalculateDispersion();
13     msgDispersion := m.dispersion + roundtripDelay / 2;
14
15     if ((msgDispersion − ourDispersion) * _TD_ <=
16             (CLOCK.getTime() − LAST_UPDATE) * _TN_ and
17             (TS_DISTANCE > m.tsDistance or
18                 CLOCK.getCorrection() = 0)) then
19     begin
20         offset := (m.t2 − m.t1 + m.t3 − m.t4) / 2;
21         if m.msgType = MSG_FEEDBACK then offset := −offset;
22         CLOCK.correctTimeGr(offset);
23         TS_DISTANCE := m.tsDistance + 1;
24         LAST_UPDATE := CLOCK.getTime();
25         DISPERSION := msgDispersion;
26         return false;
27     end;
28
29     return DISPERSION < ∞;
30 end;
```

Listing 4.6: Basic clock synchronization procedure for *gradual* GTP.

Initially, the node checks whether it is a time source (6) and, if so, the function finishes without synchronizing the clock, but informing that a feedback message should be sent. The next statement (8-9) is responsible for abandoning the clock adjustment if the dispersion of the sample received is infinite — the other node has not been synchronized yet.

After that the dispersions of the node's time and the sample are estimated (11-13). For calculating the dispersion of the sample its round-trip delay is utilized as an additional source of error, together with a dispersion of the other node.

Lines 15-18 determine whether the sample should be used for synchronization purposes. The first condition checks whether a possible error of the node's time after using the given sample for synchronization would be smaller than or equal to the error without correcting the clock at that moment. The second heuristic is responsible for preventing multiple clock correction processes to be held at the same time, unless the second sample comes from a node with a smaller hop count.

If the sample is accepted, the clock synchronization begins (20-22) and the variables are updated (23-25). The value returned by the function is similar to that from the previous version of GTP.

Table 4.2 summarizes the contents of particular messages exchanged during a single gossiping process.

msgType	MSG_REQUEST	MSG_RESPONSE	MSG_FEEDBACK
t1	T_1^A	T_1^A	T_1^A
t2	-	T_2^B	T_2^B
t3	-	T_3^B	T_3^B
t4	-	-	T_4^A
dispersion	-	ε^B	ε^A
tsDistance	-	TS_DISTANCE^B	TS_DISTANCE^A

Table 4.2: Contents of different messages of *gradual* GTP.

4.3.1. Observations

The first issue that can be easily noticed is the difference in the time the network needs in order to synchronize the clocks, caused by using the gradual clock adjustment method. The speed no longer depends only on the number of nodes, but also on the length of the clock adjustment period (see Section 3.1.2) and the initial offsets between the time of different nodes.

The algorithm constantly tries to decrease the error of the node's clock, which, on the other hand, is increasing due to the clock skew. The dispersion mechanism provides the capability to estimate the accuracy of time maintained by a given node at a given moment. Such information allows other nodes to make a decision about adjusting their clocks based on more precise data concerning the quality of samples. This solution has much better accuracy than the one based solely on hop count, which is confirmed by the results of experiments (see Chapter 6).

One may argue that *gradual* GTP is similar to *basic* GTP with the filter of an appropriate length and the minimum function for deciding whether to accept a sample or not. That statement, however, is not true — such a filter would choose only the samples with a minimal round-trip delay during the last gossiping, whereas in *gradual* GTP the dispersion represents the cumulated error of a sample on a path from the time source to a given node. As an example consider a *basic* GTP node (called A) contacting another node (B). Assume that B has its hop count much smaller than A and its filter contains only samples with high round-trip delays. This probably indicates that the time error of B is high. Furthermore, assume that the synchronization sample between A and B has very small round-trip delay, so that A decides to use it for clock adjustment — the sample is not rejected by the filter of A and the hop count of A is much higher than the one of B. Such behavior is different from the behavior of *gradual* GTP, which would reject the sample, effectively avoiding error propagation.

So far all the algorithms assumed constant frequency of gossiping. Although the time-stamp exchange may be implemented to be inexpensive, it nevertheless, consumes resources. One may notice that after the network is synchronized the frequency may be lower than in the beginning of the synchronization process. It should, however, change when a sudden event, like reseting the time in the time source or joining of a large number of unsynchronized nodes, occurs in the network. After returning to the normal network synchronization state, the gossiping frequency may be decreased again.

4.3.2. Improvements

The improvements to *gradual* GTP target mainly at a dynamic control of gossiping frequency. More specifically, in the proposed solution two constants: `_MAX_GOSSIPING_DELAY_` and `_MIN_GOSSIPING_DELAY_`, store the maximal and minimal, respectively, cycle lengths in which the active execution path of GTP initiates gossiping. Furthermore, each node is equipped with a cyclic buffer storing absolute values of offsets calculated for the last N samples and a variable `SAMPLES_COUNTER` counting the number of samples used to synchronize the clock. Initially, for each node, the buffer is filled with zeros, the value of `SAMPLES_COUNTER` is zero and the value of the variable `GOSSIPING_DELAY` (see Section 4.1) is equal to `_MIN_GOSSIPING_DELAY_`.

For each sample that failed the tests of conditions in lines 6-9 of the `EstimateSample` function (see Listing 4.6), the absolute value of the clock offset is calculated and inserted into the buffer. When a given sample is used to adjust the clock (19-24), the counter is incremented. If it reaches the value m ($m \leq N$), it is set back to zero and the following actions are performed.

The minimal (denoted as O_{min}) and maximal (O_{max}) values from the buffer are selected. Additionally, the value O_f of some function (e.g. a weighted average or a median) over the last M ($m \leq M \leq N$) values recorded in the buffer is computed. If $O_{min} < O_{max}$ then the value of variable `GOSSIPING_DELAY` is modified to:

$$(\texttt{\_MAX\_GOSSIPING\_DELAY\_} - \texttt{\_MIN\_GOSSIPING\_DELAY\_}) \cdot x + \texttt{\_MIN\_GOSSIPING\_DELAY\_},$$

where x is the fractional number:

$$x = \frac{O_{max} - O_f}{O_{max} - O_{min}}.$$

On the other hand, if $O_{min} = O_{max}$, then `GOSSIPING_DELAY` is set to `_MAX_GOSSIPING_DELAY_`. The purpose of all these actions is obtaining a linear mapping of the O_f value, which represents some recent 'average' absolute offset, from the interval $[O_{min}...O_{max}]$ to an appropriate gossiping delay within the specified bounds [`_MAX_GOSSIPING_DELAY_` ... `_MIN_GOSSIPING_DELAY_`].[6]

The described solution is not the only one possible. Its advantages include simplicity, adaptability to changes in the environment and many available configurations of values m, M and N influencing the behavior. One may consider a simple modification that delays inserting a sample to the buffer untill the moment the sample is used for clock adjustment. The other way to deal with the gossiping frequency problem is defining values of 'small' and 'large' offsets, which can be subsequently used for adjusting the current gossiping delay.

4.4. Further Enhancements

Both previous versions of GTP conform to the traditional gossiping scheme — a node to gossip with is chosen in a 'random' way from all nodes forming the network. The next

[6] For clarification see the mapping of some values of O_f: O_{min} is mapped to `_MAX_GOSSIPING_DELAY_`, O_{max} is mapped to `_MIN_GOSSIPING_DELAY_`, $\frac{1}{2} \cdot (O_{min} + O_{max})$ is mapped to $\frac{1}{2} \cdot$ (`_MAX_GOSSIPING_DELAY_` + `_MIN_GOSSIPING_DELAY_`).

version of the protocol (denoted as *selective* version of GTP) has been designed to explore the possibilities of changing the gossiping scheme in order to improve the properties of time dissemination. In particular, it aims at further decreasing the dispersion of samples used for synchronization.

The base of the algorithm is *gradual* GTP with one modification — introducing application-specific data for the overlay layer. The data consists of the current dispersion and the hop count value of a node. The callback `getApplicationData` for the peer sampling service is straightforward (see Listing 4.7).

```
class SelectiveGTPData extends Object;
    var
        dispersion : integer;
        tsDistance : integer;
end;

callback  getApplicationData():Object;
var
    result : SelectiveGTPData;
begin
    result := new SelectiveGTPData;
    result.dispersion := CalculateDispersion();
    result.tsDistance := TS_DISTANCE;
    return result;
end;
```

Listing 4.7: Callback for the peer sampling service for *selective* GTP.

Such information is utilized by the peer sampling strategy (the `getNeighbor` function), which now works as follows:

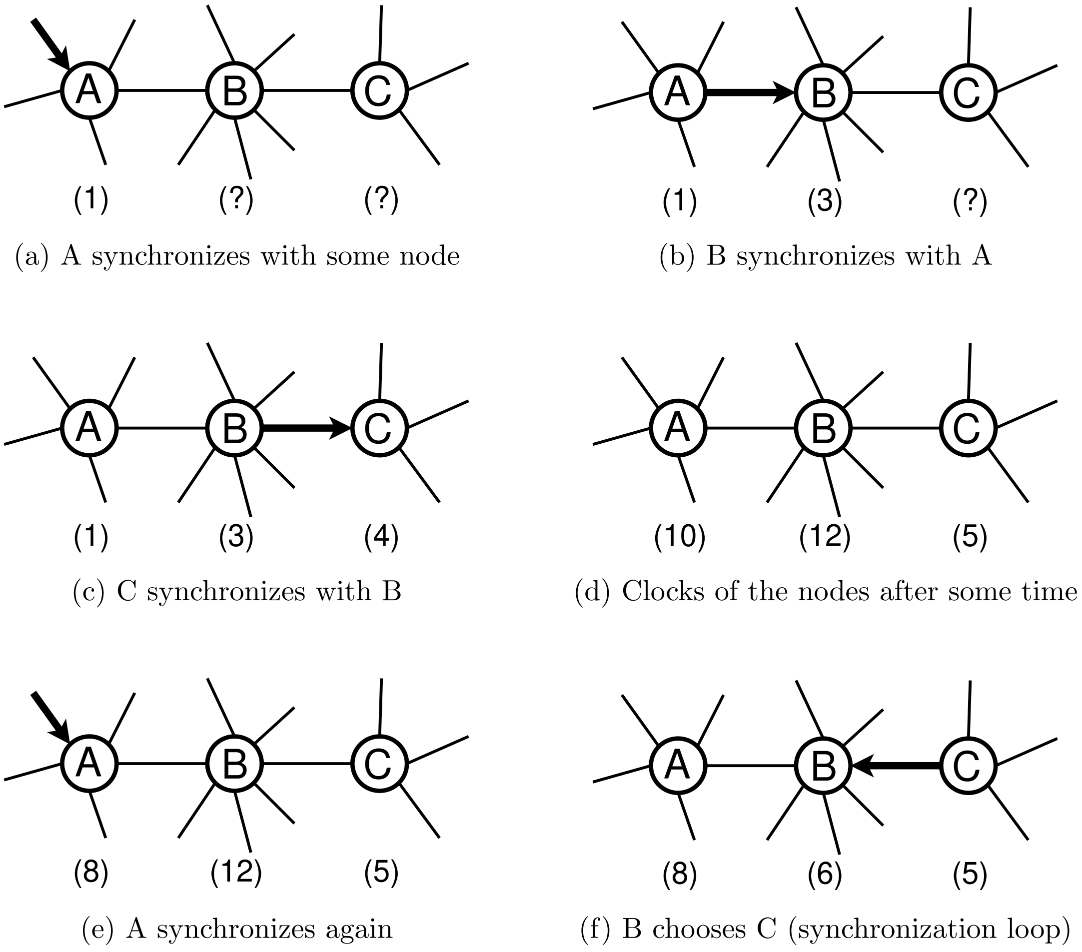
1. From the neighborhood set choose nodes with the minimal dispersion.

2. From the nodes selected in the previous point, choose a set of nodes with minimal hop count.

3. If the set contains at least one element, then pick and return a random node from it.

All other aspects of the protocol remain unchanged compared to *gradual* GTP.

It should be noted that for the two previous versions of GTP all nodes are stateless, according to the definition proposed in [31, Chap. 3]. However, in *selective* GTP a node maintains some state of other nodes that is used by the protocol. The management of that state is the responsibility of the overlay layer. If the latter is able to keep the state of nodes' neighbors up-to-date, GTP always selects the nodes with the smallest dispersion and hop count. Such an algorithm is similar to NTP, described in Section 2.2, but loops during the synchronization process are not forbidden, as long as they lead to smaller errors. An example of such a situation involves three nodes A, B and C (see Figure 4.1a). B synchronizes its clock with A, which has the most accurate time at a certain epoch (Figure 4.1b), and C, which has the most accurate and stable local clock, synchronizes with B (Figure 4.1c). Assume that due to the clock skew A has had to synchronize in order to correct its time (Figure 4.1d), but because of the high round-trip delay of a new sample, its new dispersion is high (Figure 4.1e). If B is to synchronize due to the skew of its clock, it may select C, since its clock is more accurate and the dispersion may be still lower than the new dispersion of A (Figure 4.1f).



(a) A synchronizes with some node

(b) B synchronizes with A

(c) C synchronizes with B

(d) Clocks of the nodes after some time

(e) A synchronizes again

(f) B chooses C (synchronization loop)

Figure 4.1: An example of a synchronization loop in GTP. The values in parentheses indicate dispersion of nodes. The clock of the node C is the most stable and has the smallest skew.

There is, however, a possible drawback of *selective* GTP. Traditional gossiping ensures even load balancing of work among the nodes. In *selective* GTP the nodes with smaller dispersion and hop count are more likely to be contacted than the nodes with greater values of these attributes. In other words, they have more work to do, which may result in decrease of the synchronization accuracy. Possible improvements are discussed in Section 8.2.

4.5. Remarks

The outline of the earlier sections presenting consecutive versions of GTP has been designed in order to explain the reasoning that influenced the development of the protocol. Although the naming convention is meant to be descriptive, it may be confusing due to the variety of modifications and improvements introduced for the algorithms. For the purpose of clarification, whenever the author refers to a certain version of the protocol later in this thesis, all proposed improvements are assumed, unless otherwise specified. As a reminder:

- *basic* version of GTP denotes the algorithm presented in Section 4.2 (using immediate clock adjustment and based solely on the hop count value) with the sample filtering improvement described in Section 4.2.2;

- *gradual* version of GTP is the algorithm from Section 4.3, employing gradual clock adjustment and the dispersion mechanism, with the gossiping frequency control enhancement described in Section 4.3.2;
- *selective* version of GTP denotes the algorithm utilizing application-specific data for the selection of neighbors (see Section 4.4).

Chapter 5

Overlay Management

In order to propagate the time among the nodes, GTP requires logical connections allowing direct communication between pairs of nodes. As stated in Section 3.2, such a network is maintained by means of a P2P protocol for membership management. The topology of the network is modeled as an overlay graph. More specifically, let $G = \langle V, E \rangle$ denote the overlay graph, where V is the set of all nodes forming the network and E represents the set of edges. The condition $\langle u, v \rangle \in E$ (where $u, v \in V$) is true if, and only if, the neighbor set of u contains v. Both sets V and E are continuously changing in time, either due to arrivals and departures of nodes, changes in the physical network or operation of the overlay management protocol.

Although it may not be obvious, certain properties (presented in detail in the next section) of both the shape of G and the protocol used to maintain this shape influence the way the data is disseminated across the network.

5.1. Overlay Properties

For simplicity of analyzing the overlay the undirected graph $G^* = \langle V, E^* \rangle$, where, for all $u, v \in V$:

$$\{u, v\} \in E^* \iff (\langle u, v \rangle \in E \quad or \quad \langle v, u \rangle \in E) \quad and \quad v \neq u$$

is considered (unless otherwise specified). The reason for this choice is that even though the links described by the E relation are one-way, the actual information flow from the point of view of the applications is potentially two-way, since during gossiping the passive party learns about the active party as well.

The *shortest path length* between u and v ($u, v \in V$) is the minimal number of edges that are necessary to be traversed in order to reach v from u. The average shortest path length is the average of the shortest path lengths over all pairs of elements of V.

The motivation of examining this property is that, in any information dissemination application, the shortest path length defines a lower bound on the communication cost and the time necessary to reach nodes from a given information source.

Let:

$$V_u = \{v \in V | \{u, v\} \in E^*\}$$

denote a set of neighbors[1] of u and:

$$E_u^* = \{\{v, w\} \in E^* | v, w \in V_u\}$$

[1] In the undirected graph.

denote a set of edges between the neighbors of u. The *clustering coefficient* of u is defined as follows:

$$\frac{|E_u^*|}{\binom{|V_u|}{2}},$$

which indicates the proportion of the number of edges between the neighbors of u to the number of the possible edges between them. The clustering coefficient of G^* is the average clustering coefficient of all $u \in V$. For a complete graph, it is 1, for a tree it is 0 and for an undirected graph that was created by dropping the direction of edges of a random directed graph in which all nodes have c neighbors, it is equal to $\frac{2 \cdot c}{|V|}$ (in case $|V| \gg c$).

The purpose of analyzing this property is that a high clustering coefficient has a negative effect on information dissemination (by increasing the number of redundant messages) and on the robustness (see later in this section for an explanation) by weakening the connection of a cluster to the rest of the graph and therefore increasing the probability of partitioning.

The degree of u in G^* is defined as $|V_u|$. In G one may distinguish between the number of nodes known by the given node u (the *out-degree* of u):

$$|\{v \in V | \langle u, v\rangle \in E\}|$$

and the number of nodes that know about u (the *in-degree* of u):

$$|\{v \in V | \langle v, u\rangle \in E\}|.$$

The out-degree of a node provides information about the amount of data that is maintained by the node for the purpose of the overlay management protocol. It is usually bound by the specification of the protocol to a constant value or a value that depends logarithmically on the size of V.

The in-degree (and especially its distribution) allows to determine how well the work load is balanced among the nodes and the way the information is disseminated. The nodes with higher in-degree (the hot spots) are usually contacted more often, so even distribution of this property is a very important task of the overlay management protocol.

Another important issue is whether the protocol ensures *connectivity* of the overlay graph in a fail-free environment. That is, whether the overlay graph can be partitioned due to the actions taken by the protocol, assuming the lack of failures. An associated term — the *robustness* to a (potentially massive) node failure — allows to estimate the number of random nodes that must be removed simultaneously in order to partition the network, while the *self-healing capacity* shows how the protocol can repair the overlay graph after a (severe) damage.

Finally, the *convergence (speed)* describes whether (and how fast) the overlay management protocol is able to achieve the desired shape of the overlay graph after statically configuring the network to a given initial topology or joining of a node.

5.2. Protocols

Due to the epidemic nature of GTP, the overlay graph does not have to be structured. Moreover, the research presented in the thesis was based on protocols for managing unstructured overlays. The following sections describe the overlay protocols used by the author for examining the properties of GTP.

They solve the membership management problem by providing each node in a network with a small, continuously changing neighbor set, called the *overlay cache*. The size of the cache may be fixed or adjustable depending on the protocol.

5.2.1. Newscast

Newscast [9, 34] has been designed as a protocol for communication in a large-scale agent-based system. It defines a system model consisting of two layers: the application layer that runs the agents and the news agency that runs the, so called, correspondents responsible for communication. The agent-specific information is disseminated together with membership management data, but, for the purpose of compatibility with the model presented in Chapter 3, the author of this thesis decided to separate these two issues by extracting the overlay management protocol from the system.

Each node in the Newscast system maintains the fixed-size overlay cache. A single cache entry consists of contact information of a neighbor node (e.g. an IP address and a port number[2]) and a timestamp.

The nodes regularly exchange caches as follows. Assume that the size of the cache is c. Every node executes the following steps once every ΔT time units (ΔT is called the gossiping delay for the overlay layer[3]).

1. Randomly select a node to gossip with from the cache.
2. Create a message containing the whole cache.
3. Substitute (in the message) the cache entry of the node selected in point 1, by your own contact data with the current timestamp.[4]
4. Send the message to the selected node and, in turn, receive a message with all its cache entries.
5. Merge the received cache entries into the local cache discarding the oldest ones. Only at most c cache entries of other different nodes may remain in the cache.

Upon reception of the cache exchange message, the selected node executes the steps below:

1. Send back all your cache entries.
2. Merge the received cache entries into the local cache discarding the oldest ones. At most c cache entries may remain in the cache.

After the exchange both nodes have the same cache, except for a pointer to each other. However, as soon as any of them executes the protocol again selecting a different node, their respective caches will most likely be different again.

The protocol does not require the clocks of the nodes to be synchronized, but only that the timestamps of cache entries in a single cache are mutually consistent[5]. It is assumed that communication time between two nodes is negligible in comparison to ΔT. The messages exchanged between the nodes, apart from their caches, contain the current local time. When a node A receives the cache entries and a local timestamp T^B from a node B, it subsequently adjusts the timestamp of each received entry with a value $T^A - T^B$, effectively normalizing the time of each new entry to those already cached.

[2] Actually, because the model described in Chapter 3 assumes the independence of the application layer on the overlay management layer, multiple ports may be required, based on the implementation.

[3] Which is, in general, different than the gossiping delay for GTP.

[4] In the system model designed by the author (see Chapter 3), this is also the moment when the overlay management layer calls the `getApplicationData` callback to obtain application-specific data that will be sent to the selected node.

[5] This is an example of one more situation where gradual clock adjustment is required.

5.2.2. Lpbcast

The Lightweight Probabilistic Broadcast (abbr. as Lpbcast) [4] is an approach for solving the problem of broadcasting in a large-scale distributed system. Instead of a common solution based on multicasting, Lpbcast employs a probabilistic gossip-based algorithm. Gossip messages are used not only to disseminate event notifications and to propagate information about received notifications, but also to propagate membership data. The protocol also introduces a heuristic for purging out-of-date event notifications and membership management information.

Again, for the purpose of the research the author separated the overlay management algorithm from the broadcasting service. Moreover, since the system model from Chapter 3 assumes that intentional disconnection of a node from a network cannot be distinguished from a failure of that node, the unsubscription mechanisms of Lpbcast (see [4] for more details) have been ignored. Additionally, in the protocol description below the frequency-based optimizations for membership purging (see [4] for explanation) are assumed.

The node executing the protocol has two fixed-size sets: a cache of neighbors (as in Newscast) and a set of subscriptions. The single element of both sets consists of contact information of a node and a number denoting a frequency value.

Assume that the size of the cache is c and the size of the subscription set is equal to s. Additionally let f denote the fanout indicating the number of nodes to gossip with during one gossiping attempt. Each node executes the following steps once every ΔT time units.

1. Randomly select f neighbors from the cache.
2. Prepare a message containing the whole subscription set.
3. Insert your contact data to the subscription set of the message with frequency equal to zero.[6]
4. Send the message to the neighbors selected in point 1.

Upon reception of the gossiping message, the node executes the following algorithm.

1. For all received subscriptions not pointing at you:
   (a) If the node represented by the subscription is in your cache, then increase the frequency of that node in the cache. Otherwise, add the node to the cache after increasing its frequency.
   (b) If the node represented by the subscription is in your subscription set, then increase the frequency of the node within the set. Otherwise, add the node to the subscription set after increasing its frequency.
2. While the cache size is greater than c:
   (a) Select the cache entry using a function described below.
   (b) Remove the selected element from the cache and add it to your subscription set if it is not there.
3. While the size of the subscription set is greater than s:

[6] Similarly to Newscast, at this moment the overlay management layer calls the `getApplicationData` callback to obtain application-specific data of the node.

(a) Select the subscription set element using a function described below.
(b) Remove the selected element from the subscription set.

The function (mentioned in points 2a and 3a) used for selecting elements to be purged from the cache or the subscription set operates as follows.

1. Compute the average frequency of all elements.
2. Select a random element.
3. If the frequency of the selected element is greater than k multiplied by the average frequency of all elements ($0 < k \leq 1$), then return this element. Otherwise, increment its frequency and go to point 2.

The frequency optimization applied is particularly useful for dynamic networks, since it lowers the propagation delay of membership information.

In contrast to Newscast, in which the nodes exchange their caches, Lpbcast is a push-based protocol — the membership information during one gossiping process is propagated only in one direction.

5.2.3. Shuffling

The Shuffling protocol is a part of the PROOFS system [29]. PROOFS (the P2P Randomized Overlays to Obviate Flashcrowd Symptoms) has been designed in order to deal with, so called, Internet flash crowds (or hot spots) — a phenomenon that results from a sudden, unpredicted increase in an on-line object's popularity. Clients running the system form a P2P overlay network that allows those clients that have received copies of popular content to forward the content to the clients that desire it but have not yet received it.

PROOFS consists of two protocols. The first one is responsible for managing the overlay network, while the second one participates in searches of objects in this network. The overlay management protocol runs continuously, in contrast to the location protocol, which runs only when flash crowd phenomena exist within the network. For the purpose of GTP, only the overlay management protocol is interesting.

Each node has a fixed-size cache containing contact information of its neighbors. The nodes periodically perform what is called a shuffle operation. The shuffle is an exchange of a subset of neighbors between a pair of nodes that can be initiated by any node. More specifically, assume that the cache size is c and the size of the subset used during the shuffle operation (called the shuffle length) is l. Every ΔT time units the node performs the following activities.

1. Select a random subset of l neighbors ($1 \leq l \leq c$) from the cache and a random node within this subset.
2. Create a message containing the selected neighbors.
3. Replace the contact information of the node selected in point 1 within the message with your contact information.[7]
4. Send the message to the selected node.

[7] Again, at this moment the overlay management layer calls the `getApplicationData` callback in the system model designed by the author.

5. Receive a subset of no more than l neighbors from the selected node.

6. Update your cache by including the received neighbors. The replacement is done according to the four rules:

   (a) No neighbor appears more than once within the cache.

   (b) A node is never its own neighbor.

   (c) If the size of the cache lies below the bound c, new entries are added without overwriting previous entries (until the number of entries in the cache reaches c).

   (d) Entries in the cache can only be overwritten (i.e. removed) if they were sent to the selected node during the shuffle.

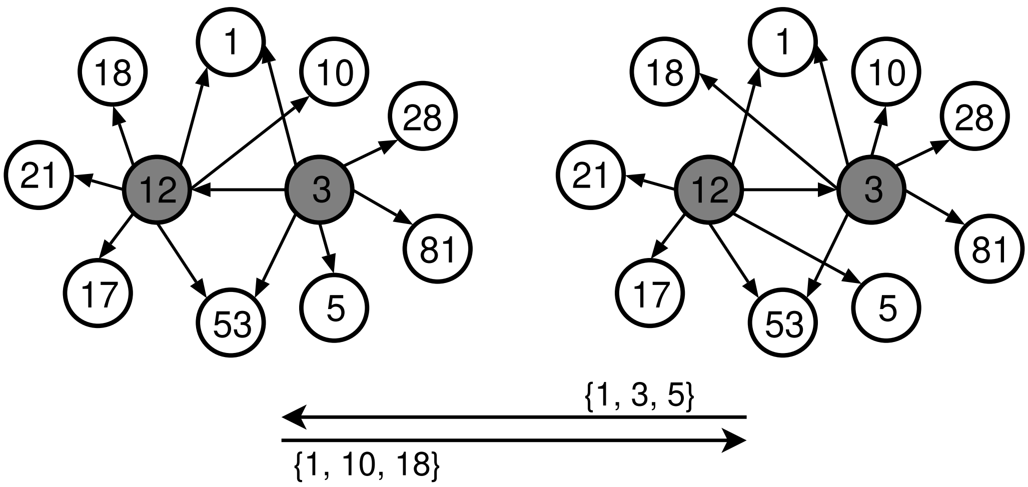
Upon reception of a shuffle request, the node executes the following steps.

1. Randomly select a subset of l neighbors from the cache.

2. Send the selected subset to the node that initiated the shuffle.

3. Update your cache including the received entries, according to the rules listed earlier.

A sample shuffle operation is shown in Figure 5.1.



(a) Before the shuffle (b) After the shuffle

Figure 5.1: An example of a shuffle operation.

The nodes are represented by numbered circles. Directed edges indicate the neighborhood relation, where an arrow pointing from A to B means that B is a neighbor of A. Neighbors are depicted only for the grey nodes numbered 3 and 12. These nodes start with the cache shown in Figure 5.1a, and, after the shuffle operation initiated by node 3, they end with the cache shown in Figure 5.1b.

Two important points need to be noted. First, if a node A is B's neighbor and B initiates a shuffle with A, then after the shuffle, B is A's neighbor (i.e. the edge reverses direction). Second, in a fail-free environment no client becomes disconnected after the shuffle operation — it simply moves from being the neighbor of one node to being the neighbor of another.

It may happen that two nodes start the shuffle operation simultaneously forming a cycle. Since there are two execution paths that can modify the cache, the conflicts may appear when $l > \lfloor \frac{c}{2} \rfloor$.[8] The original solution to the problem (called later the *optimistic* one) states that a node may reject the shuffle request if it initiated one on its own and has not received a response. Upon reception of a rejection message, a node chooses the next time to initiate the shuffle from a uniform distribution.

The experiments conducted for the purpose of the thesis indicate that this situation is not uncommon. Therefore, the author designed and applied a different solution called the *pessimistic* one. Because the research concerning Shuffling [35] shows that the overlay formed by the protocol is independent of the shuffle length l, the pessimistic solution utilizes locking. More specifically, let N denote the number of instances of execution paths that may modify the cache concurrently. The instance may lock up to l cache entries ($l \leq \lfloor \frac{c}{N} \rfloor$) that can be later replaced with entries received from another node. Such an approach limits the shuffle length to at least $\lfloor \frac{c}{2} \rfloor$, in case each execution path has one instance, but also eliminates the need for the shuffle rejection messages.

5.2.4. CYCLON

CYCLON [35] is a novel protocol designed specifically for inexpensive unstructured overlay management. It is also based on the shuffle operation, but due to some changes described later, the overlay graph has different properties. Additionally, it introduces a mechanism of random walks that allows joining nodes to have their caches fully filled without initiating shuffle operations.

The main difference between Shuffling and CYCLON is that in case of the latter one, nodes do not randomly choose which neighbor to shuffle caches with. Instead, they select the node whose information was the earliest one to have been injected in the network. The first goal of this enhancement is to limit the time the entry can be passed around until it is chosen by some node for a cache exchange. Another motivation is to impose a predictable lifetime on each pointer, in order to control the number of existing pointers to a given node at any time.

These objectives are accomplished by extending a single cache entry of a node with a number indicating the age of the entry. Assume that the cache size is c and the size of the subset used during the shuffle length is l. Every ΔT time units the node executes the following steps.

1. Increase by one the age of all neighbors in the cache.
2. Select the neighbor with the highest age among all cache entries.
3. Select $l - 1$ other entries randomly and store them in the shuffle request message.
4. Add your own contact information with age equal to zero to the message.[9]
5. Send the message to the node selected in point 2.
6. Receive a subset of no more than l neighbors from the selected node.
7. Update the cache by including the received neighbors using the same rules as for Shuffling.

[8] For the implementation purposes the execution paths may have many (N) instances (threads) executing them. In this case the conflicts may appear when $l > \lfloor \frac{c}{N} \rfloor$.

[9] Like for the other protocols, at this moment the `getApplicationData` callback is called.

The activities performed during the reception of the shuffle request are exactly the same as in the Shuffling protocol. Also the pessimistic solution is used for the resolution of the shuffle conflicts.

5.2.5. Other Solutions

The author does not claim that the choice of the overlay management protocols used for the purpose of the experiments with GTP is exhaustive. This was not a goal of the conducted research. However, the protocols presented earlier have been designed for many different purposes: agent-based computations, broadcasting, dealing with hot spots or specifically for membership management, and thereby constitute an interesting test group. Other solutions that were considered include, among the others, SCAMP [6] and Saxons [28].

SCAMP, similarly to Lpbcast, has been developed for membership management in large-scale systems that require efficient broadcasting. Its properties considering capability of information dissemination are comparable to the properties of other protocols. The distinguishing feature of SCAMP is that it is reactive, in the sense that cache exchanges take place only when nodes join, leave, or a failure is detected. Such an approach, although reasonable for information dissemination, without any special enhancements seems inappropriate for *selective* GTP, where the caches of nodes should be kept up-to-date.

Saxons is another example of a general-purpose overlay management protocol. It aims at constructing an overlay graph that has low path latency, low shortest path length and high path bandwidth. Its model consists of six modules. The bootstrap process determines how new nodes join the network. The structure quality maintenance component maintains an overlay mesh of desired properties while the (optional) connectivity support component aims at detecting and repairing overlay partitions. They run periodically to accommodate dynamic changes in the system. The above Saxons components are all supported by the membership management module whose goal is to track a random subset of nodes. The structure quality maintenance is further supported by two other components responsible for acquiring performance measurement data for network links and finding nearby nodes. The shape of the overlay meeting the design objectives of Saxons might be an interesting solution for GTP.

5.3. Comparison

Due to the lack of the space required in the thesis in order to fully describe all aspects of the presented overlay management protocols, this section focuses only on the most important issues. More extensive analysis can be found in the papers cited earlier. The results presented in this section come from these papers as well as from the experiments that have been conducted by the author in order to verify the correctness of the system implementation introduced in Chapter 7.

One of the most important requirements for the overlay protocol is the ability to keep the network connected during normal operation. Both Shuffling and CYCLON guarantee connectivity in a fail-free environment, independent of the values of protocols' parameters. The experiments show that Lpbcast also keeps the network connected with very high probability. Unfortunately, during the tests involving 1500 nodes it turned out that Newscast is unstable for cache sizes less than 20, which leads to overlay partitions independent of the initial (or bootstrapping) topology of the network. Such an observation was confirmed by the people who developed Newscast[10] and was a main reason for increasing the size of the cache for this protocol during further experiments. After that step the connectivity has always been preserved.

The speed of convergence for Shuffling, CYCLON and Newscast is high, e.g. less than 100 gossiping cycles is required for a network of 1500 nodes with a bidirectional ring as a bootstrapping topology (assuming the cache size of 10, or 20 in case of Newscast, and the shuffle length of 5). On the other hand, Lpbcast needs almost 200 gossiping cycles, in spite of the fanout greater than one (which means that during one cycle a node is allowed to gossip with more than one node), in order to fully converge. This behavior is caused by the fact that the latter protocol is push-based only while the first three are push-pull-based solutions.

After having converged, the network is stable. In case of Shuffling and CYCLON, the average clustering coefficient and the average shortest path length is almost equal to the appropriate values for a random graph. For Newscast and Lpbcast the average shortest path length is slightly higher, but the clustering coefficient is significantly bigger than for a random graph. Examples of these values for a network of 1500 nodes with the cache size equal to 10 (20 for Newscast) are shown in Table 5.1.

Protocol	**Avg. Clust. Coeff.**	**Avg. Shortest Path Len.**	**Avg. In-Deg.**
Newscast	0.295921990	2.718564409	20.0
Lpbcast	0.118435210	2.951563876	10.0
Shuffling	0.012477643	2.769968958	10.0
CYCLON	0.012381255	2.757673446	10.0
random graph	0.012530221	2.793862575	10.0

Table 5.1: The properties of a sample overlay graph for various protocols.

Another important issue is the change of these values when the size of the network or of the cache changes. For all presented protocols the average shortest path length depends logarithmically on the network size. The size of the cache influences the base of the logarithm in this dependence. The clustering coefficient drops rapidly with the increase of the network size for CYCLON and Shuffling. Newscast and Lpbcast show rather small changes.

The distribution of the in-degree in a network of 1500 nodes and the cache size set to 10 (or 20 in case of Newscast) is depicted in Figure 5.2.

As can be easily observed, the most even distribution is provided by CYCLON, where almost one of three nodes has an in-degree equal to the size of the cache. On the other hand, the distribution achieved by Newscast introduces many hubs (nodes with high in-degrees), which has negative effect on load balancing. Shuffling and Lpbcast are characterized by distributions closer to that of a random graph. The shape of the distribution charts look similar when the size of the network or of the cache changes.

The protocols, in general, are very robust. With the cache size equal to 20 above 90% of nodes from a 100,000-node network have to be removed at once in order to partition the overlay managed by CYCLON. For Newscast this value is about 75%. Moreover, even if partitioning occurs, most nodes form a single large connected cluster.

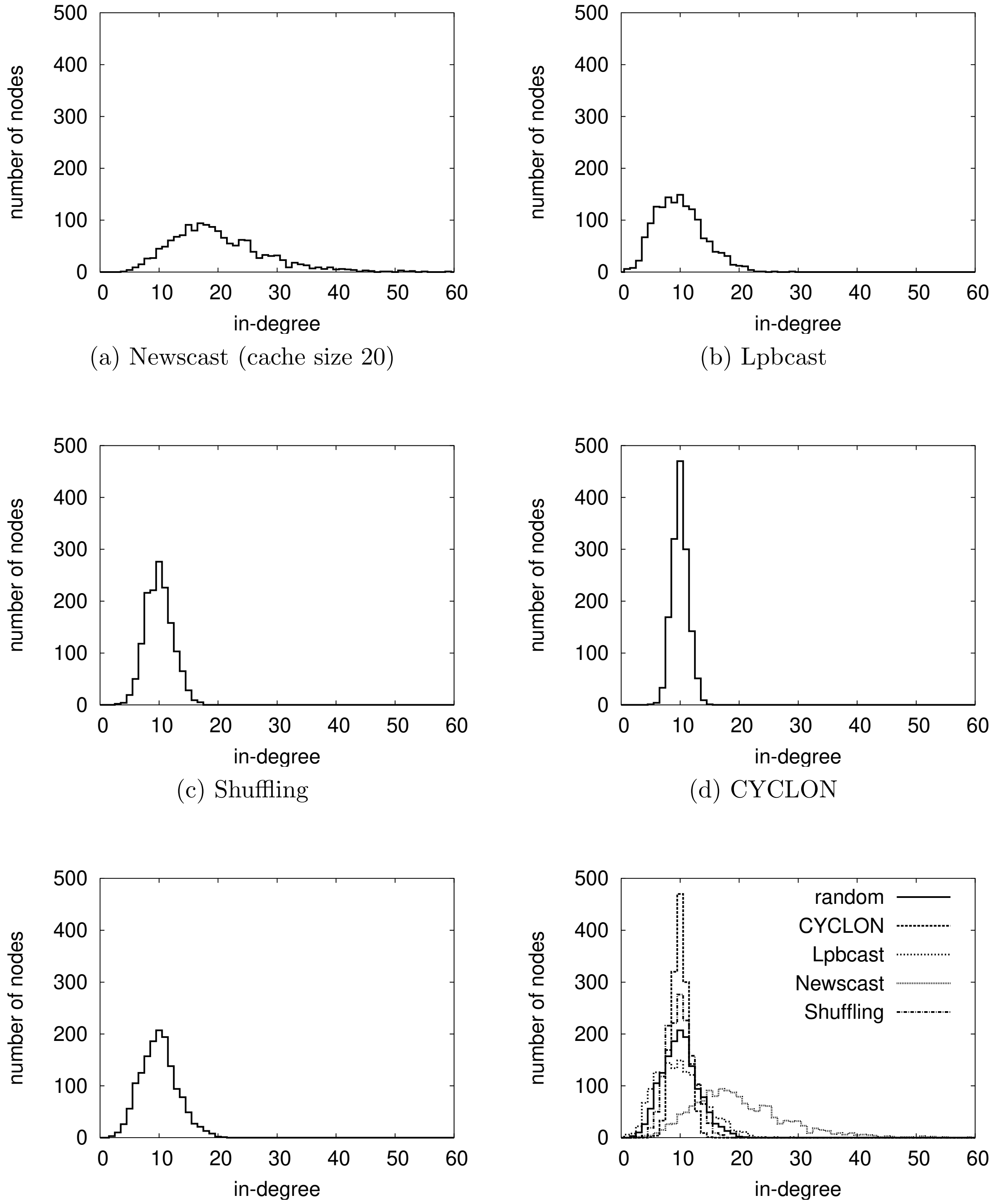
The next advantage of the protocols is the self-healing capacity. After massive failures of nodes, they are able to gradually repair the overlay graph by removing the links pointing to the dead nodes. The speed of healing depends on a particular protocol. For example after a sudden crash of 50% of nodes from a 100,000-node network Newscast and Shuffling may

[10] Private communication between the author and Maarten van Steen.

require about 500 and 300 gossiping cycles respectively in order to clean all dead links, while, due to the design goals, CYCLON can heal even up to four times faster than Shuffling.

To sum up, Shuffling and CYCLON have the average clustering coefficient and the average shortest path length very similar to a random graph, while offering better in-degree distribution (especially in case of CYCLON) and robustness to massive nodes failures. The topology of the overlay formed by Newscast or Lpbcast resembles complex networks observable in nature, biology, sociology and computer science called the *small-world* topologies [1, Chap. VI]. They are characterized by the average shortest path length almost as small as for random graphs, but at the same time, a significantly higher clustering coefficient.

The most important observation is that all described protocols provide scalable, fully-decentralized, robust and inexpensive ways to manage a (highly) dynamic overlay network.



(a) Newscast (cache size 20)
(b) Lpbcast
(c) Shuffling
(d) CYCLON
(e) random graph
(f) comparison

Figure 5.2: In-degree distribution for the protocols.

Chapter 6

Experimental Results

The theoretical reasoning presented in Chapter 4 concerning GTP reveals certain properties of particular versions of the protocol. Nevertheless, it should be noted that in the world of large-scale systems, theory and practice often diverge. In order to examine how GTP (and the whole system model) behave in practice and to verify the correctness of theoretical assertions, the author performed various emulation experiments with GTP. Emulation, in contrast to simulation, involves implementing the protocols and conducting the experiments on a real network of computers. In the case of GTP, they have been performed with a set of up to 64,500 nodes distributed across a 200-machine wide-area cluster of workstations.

The implementation utilized for the purpose of the experiments is described in Chapter 7.

It should be emphasized, that not only for GTP, but also for some of the overlay management protocols presented in Chapter 5, the experiments conducted by the author have been the first emulations, since the results presented in the papers concerning the overlay protocols are usually based on simulations.

The rest of this chapter is organized as follows. The next section describes the test environment. After that the properties of GTP revealed by the experiments on networks up to 1500 nodes are discussed. The following section describes the results obtained by the large-scale experiments. Finally, the summary of the results is given.

6.1. Test Environment

The experiments with small networks (up to 1500 nodes) have been performed on a 2.4-GHz Pentium 4 workstation with 0.5GB of RAM running Windows XP Professional (SP1) and using the standard Sun JVM 1.4.2. For the communication purposes the local object passing implementation (see Chapter 7) has been used.

The environment utilized for large-scale tests constitutes a part of the DAS-2 (the Distributed ASCI Supercomputer [3]) — a wide-area system consisting of five clusters of PCs located at different sites across the Netherlands.

For the purpose of the experiments only the cluster belonging to the Vrije Universiteit has been used. It consists of 72 machines. Each machine contains two 1-GHz Pentium III processors and at least 1GB of RAM. Machines within the cluster are connected by a Fast Ethernet (100Mbps) network, which, in addition, is used for the file transport. The system runs Red Hat Linux (version 7.2). Machines' clocks are synchronized with NTP (version 3). All tests have been performed using Sun JVM 1.4.2 with UDP as the transport-level protocol.

6.2. Properties of GTP

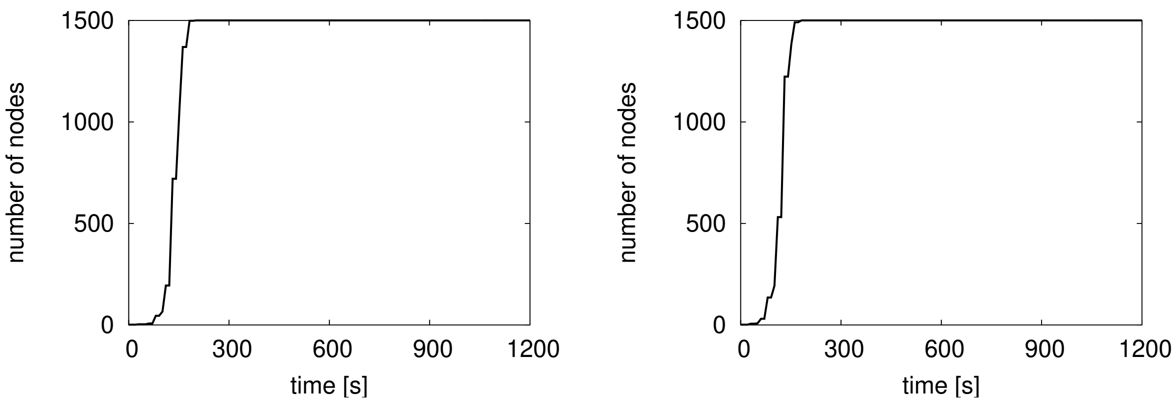
The small-network experiments aimed at exploring GTPs' capability of propagating and maintaining accurate time. In order to examine these issues many properties have been analyzed including: error in the time displayed by the node's clock (average absolute error, maximal positive error, maximal negative error, error distribution, changes in error values), hop count of a node from a time source (average and maximal value, hop count distribution), changes in gossiping frequency.

All results presented in this section come from tests on the networks of 1500 nodes. The GTP layer was always activated after the overlay layer had fully converged. The gossiping delay of the overlay management protocol was set to 250 seconds (unless otherwise specified), the cache sizes were equal to 10 (20 for Newscast), the shuffle length for Shuffling and CYCLON was 5, the subscription set size and the fanout for Lpbcast were equal to 10 and 2 respectively. Other parameters were changed on a per-test basis.

6.2.1. *Basic* GTP

The first interesting question is how fast the time information can be disseminated across the network. More specifically, assume that there is only one time source. Initially, all other nodes do not know which one is the time source and whether there is any (the `TS_DISTANCE` variable of a node is set to infinity). A node becomes *aware* of the time source existence when its `TS_DISTANCE` variable changes to a finite value. Furthermore, it should be examined whether the overlay management protocol utilized influences awareness speed.

To answer the question the experiments involving all the membership management protocols have been performed. The gossiping delay for GTP (`GOSSIPING_DELAY` variable) and `_STANDALONE_PERIOD_` were both set to 25 seconds and no sample filtering was used. The results of the tests for CYCLON and Newscast as the overlay management protocols are depicted in Figure 6.1.



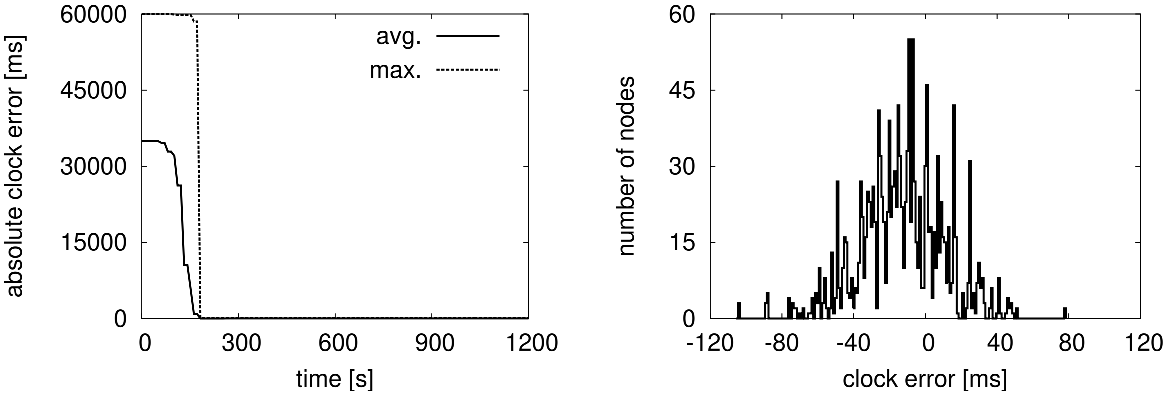
(a) CYCLON (b) Newscast

Figure 6.1: Speed of discovering the time source existence (no sample filtering).

The charts look almost identical, which is also the case for other protocols. Such an observation allows to claim that the overlay management protocol (from the set described in Section 5.2) does not influence the information dissemination speed. In every test scenario GTP usually required 8 gossiping cycles (200 seconds) to propagate information about the presence of the time source and the number of aware nodes grew exponentially.

However, the opposite relation is observable when Newscast is employed with some nodes having their initial time greater than the time of the time source — during synchronization their time is set backwards which causes inconsistencies with the timestamps of entries within the overlay cache. Due to this factor, the average clustering coefficient of the network grows rapidly to almost 0.95 (cf. Table 5.1), since the caches of nodes become similar — containing entries with future timestamps. Although during the experiments conducted by the author the network remained unpartitioned and the clustering coefficient returned to its normal value rather fast (depending on the initial time offset distribution), this behavior confirms the claim that Newscast should not be used in practice with *basic* GTP.

The next interesting issue is the speed the network needs to reach the stable state of the clocks and the time accuracy in such a state. Figure 6.2 shows the results derived from the tests described above. The initial time error of all nodes except for the time source was chosen randomly with a uniform distribution from the sum of intervals: 10 to 60 seconds and -10 to -60 seconds. All clocks were ticking with the same frequency.



(a) Synchr. speed (Lpbcast) (b) Synchr. quality (Shuffling)

Figure 6.2: Synchronization speed and quality of unfiltered *basic* GTP.

The chart from Figure 6.2a indicates that the speed of reaching the stable state[1] by the system is exactly the same as the speed of discovering the presence of the time source. This fact is consistent with the theoretical analysis presented in Section 4.2.1, more specifically, with the formula (4.7).

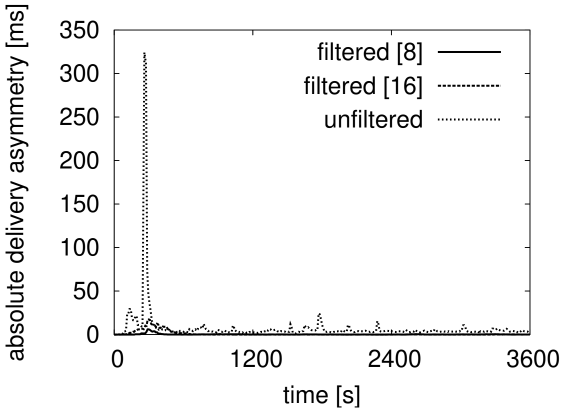
Unfortunately, in the stable state the time errors are very high. The distribution depicted in Figure 6.2b contains arbitrary data after about three hours of operation. After discovering the presence of the time source by all nodes the errors range from -277 ms to 152 ms. Not even 50 of the nodes maintain accurate time. Such an observation challenges the legitimacy of using the term 'stable' for describing the state of the network.

The reason for this behavior is the influence of network differential delays appearing during the gossiping. The filtering improvement introduced in Section 4.2.2 allows to decrease the number of such errors. Figure 6.3 illustrates the relationship between sample filtering and the number of asymmetric deliveries[2] (the values in square brackets denote the length of the filter used). Because the tests were run on one machine, measuring the differential delays was easy due to access to a global clock. The charts show only differential delays of the samples

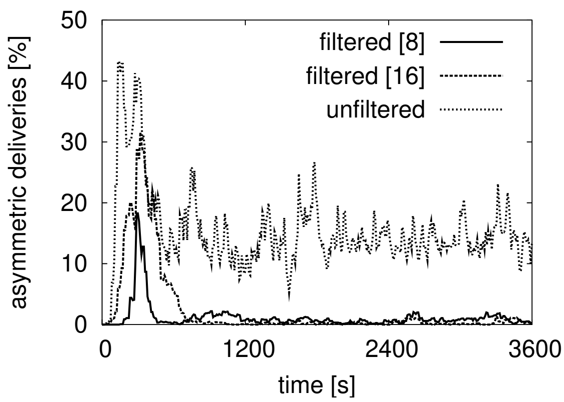
[1] For a moment assume that the stable state definition is intuitive and denotes the state corresponding to the almost flat absolute clock error line in Figure 6.2a.

[2] Asymmetric delivery denotes a situation in which the time of delivering the GTP request from A to B is different from the time of delivering the GTP response from B to A.

that passed the filter (if it was used) and were tested for possible synchronization usage.



(a) Avg. abs. delivery asymmetry



(b) Perc. of asymmetric deliveries

Figure 6.3: Influence of sample filtering on the asymmetric deliveries.

In Figure 6.3a the average absolute difference between the time required to deliver a GTP request and the time required to deliver a GTP response is depicted. During the initial synchronization phase the differences are high (reaching almost 350 ms), because all the nodes are activated and start gossiping at the same time, which leads to network congestion. Later the situation is more stable — the average and maximal differential delays for the unfiltered GTP range between 1.14-24.31 ms and 32-876 ms respectively. For the filtered version the maximal values of these numbers are equal to 0.51 ms and 32 ms for the filter length 8, and 0.35 ms and 32 ms for the filter length 16.

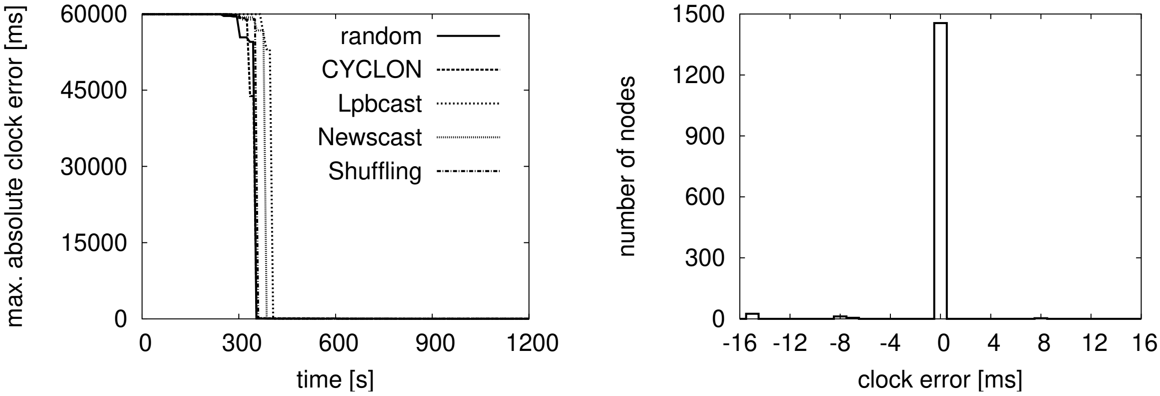
Additionally, Figure 6.3b shows the proportion between the asymmetric and all deliveries. In the case of the unfiltered protocol version, this value oscillates between 5 and 30 percent, while it significantly decreases with the increase of the filter length. Note that in the beginning longer filters present worse properties, since they are not completely filled with historical data. Such a problem however, may be dealt with easily if the filtering initially uses only the values collected so far.

It should be obvious that sample filtering improves the quality of the time synchronization algorithm. In order to confirm this claim a series of experiments have been conducted. The results are depicted in Figure 6.4. During the tests the length of the filter was set to 8 and the other parameters remained unchanged.

Due to filtering the speed of the network synchronization decreases slightly from 8 to 14-20 gossiping cycles (350-500 seconds). Moreover, very small differences in the speed of convergence of GTP between different overlay management protocols are observable. In particular, the overlay with a random graph topology provides better properties than the overlay formed by Lpbcast.

On the other hand, the quality of synchronization improves significantly. The maximal absolute errors are on average smaller than 16 ms and most of the nodes are perfectly synchronized with the time source. However, despite the filtering, if long lasting network load fluctuations occur, the systems tends to lose its accuracy.

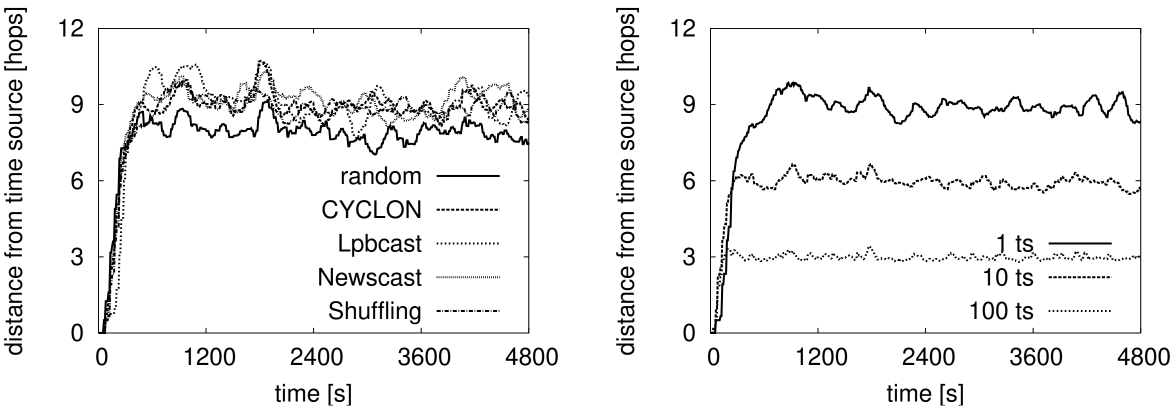
Another important issue allowing to partially explain the reason for errors is the hop count from a node to the time source. In *basic* GTP the neighbor to gossip with is chosen randomly, so the average value of the `TS_DISTANCE` variable for a node is usually higher than the shortest path length to the time source. Moreover, the interesting question concerns the



(a) Synchr. speed (b) Synchr. quality (Shuffling)

Figure 6.4: Synchronization speed and quality of *basic* (filtered) GTP.

way these values change when the number of time sources increases.



(a) Hop count for different overlay protocols (b) Multiple time sources (Shuffling)

Figure 6.5: Convergence of the `TS_DISTANCE` variable.

Figures 6.5a and 6.5b show the change of the time source distance in time. The overlay protocol does not significantly influence this property, although the difference between the random graph and other protocols is visible. Moreover, changes in the time source distance depend logarithmically on the number of time sources in the network. With only one time source the average time source distance is equal to 9. Adding 9 more time sources reduces this value to 6, while with 100 time sources the average distance is equal to 3. Such a phenomenon is directly related to the properties of the overlay graph.

The distribution of the time source distance is depicted in Figure 6.6. For all overlay management protocols, most of the nodes achieve a hop count in the range from 9 to 11, which is quite a big value comparing to the average shortest path length in the overlay graph equal to ∼2.8 (see Table 5.1). This behavior, which may have destructive effects in large-scale systems, is caused by two facts: the random choice of the neighbor for gossiping and the synchronization loops that occur in the network.

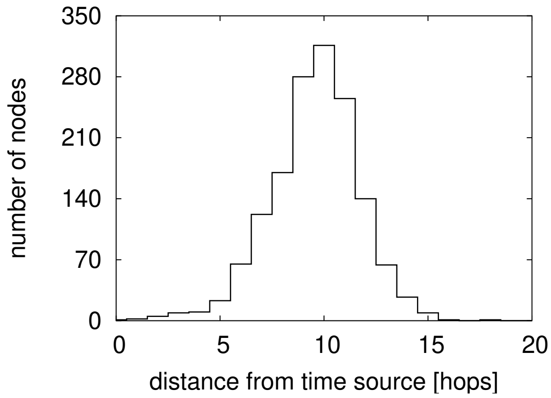
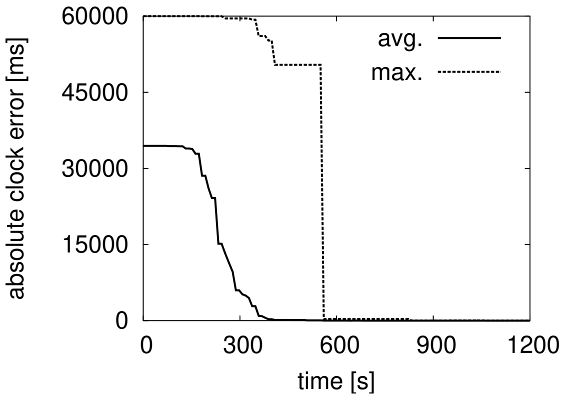
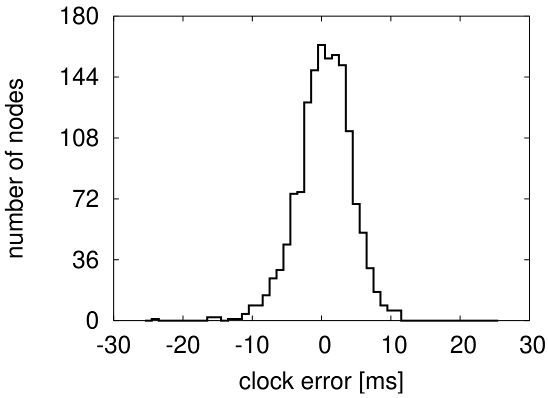


Figure 6.6: Distribution of the `TS_DISTANCE` variable (Newscast).

The final series of experiments concerns the ability of *basic* GTP to maintain accurate time in a system whose nodes' clocks are skewed. More specifically, the clock skew of a node was chosen randomly with a uniform distribution from the set $\{-\frac{4}{10^5}, -\frac{3}{10^5}, ..., \frac{3}{10^5}, \frac{4}{10^5}\}$, the other parameters remained the same. Note that these values are exaggerated, since in practice default computer clocks have much better oscillator tolerance. It also means that in case of a node having a clock with the skew equal to $\pm\frac{4}{10^5}$, the error of one millisecond appears between consecutive gossiping cycles, assuming that the sample from every cycle is used for clock adjustment, i.e. the filter does not reject any of them. Moreover, as stated in Section 4.3, if the time of a node with a small hop count has an error, the error is propagated among many other nodes. In face of the aforementioned facts, the interesting question is whether GTP is capable of maintaining the time within some bounds. The results of experiments are presented in Figure 6.7.



(a) Synchronization speed



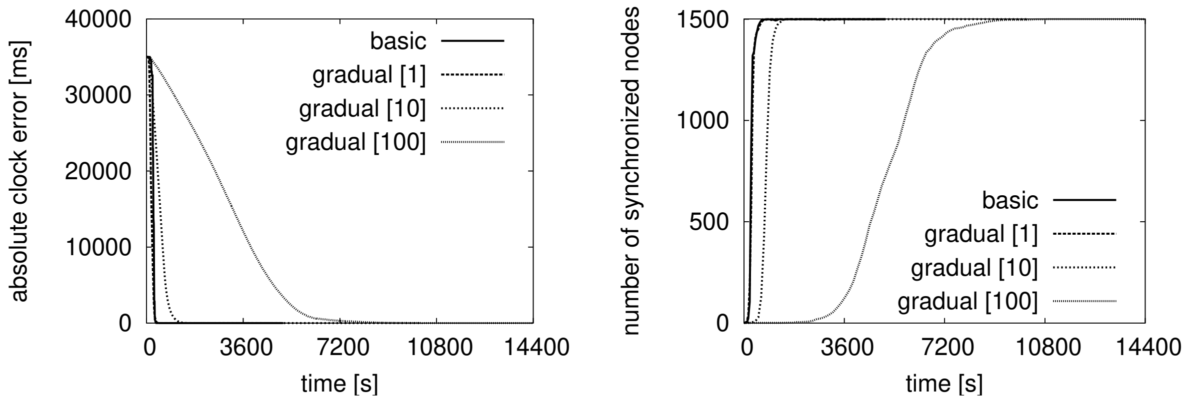
(b) Error distribution

Figure 6.7: The synchronization speed and accuracy in the system with skewed clocks.

They indicate that the period of time needed for the network to synchronize increases to 33 gossiping cycles (825 seconds). The most important observation however, is that the offsets of the nodes are limited. In the stable state the maximal absolute error of a node is equal to 28 ms and for the majority of nodes it does not exceed 5 ms. Such results allow to claim that the protocol is able to limit the influence of the clock skew on the accuracy of time.

6.2.2. *Gradual* GTP

Gradual GTP utilizes gradual clock adjustment, which, in general, requires longer periods of time to complete applied time changes. Therefore, the first issue to be examined is the speed of network synchronization. The parameters of the experiments conducted by the author were similar to the parameters for tests of *basic* GTP. All clocks had the same frequency and the initial time error of all nodes (except for the time source) was chosen randomly with a uniform distribution from the sum of intervals: 10 to 60 seconds and -10 to -60 seconds. In order to obtain better knowledge on how the clock adjustment period (see Section 3.1.2) influences the network synchronization speed, many tests with different values of this constant have been performed. Figure 6.8 depicts the change of the average absolute error and the number of synchronized nodes. The values in square brackets indicate the length of the clock adjustment period, *basic* GTP is included for comparison.



(a) Avg. absolute error (Shuffling) (b) Number of synch. nodes (Shuffling)

Figure 6.8: Synchronization speed for *gradual* GTP with different clock adjustment periods.

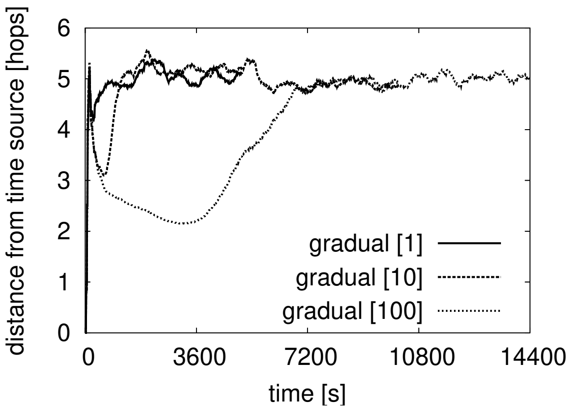
Firstly, the experiments show that the speed of *gradual* GTP, similar to *basic* GTP, is independent from the overlay protocol used and therefore these charts are not presented here.

Figure 6.8 clearly indicates that network synchronization is still fast. If the clock adjustment period is equal to 1 the network requires 910 seconds for convergence (cf. 560 seconds in case of *basic* GTP). Increasing this period to 10 and 100 changes the above value to less than 0.5 hour and about 3 hours respectively.

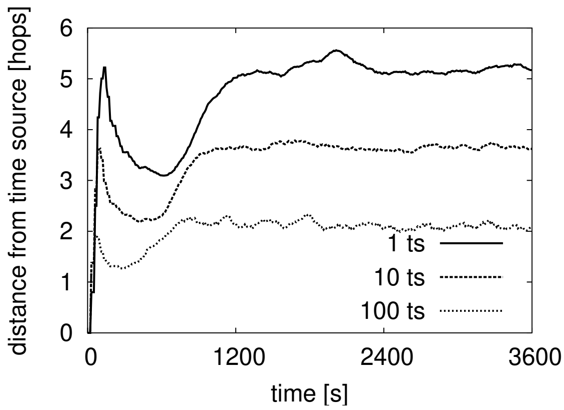
Another important observation is the quality of synchronization. All the nodes synchronize perfectly with the time source — the errors are equal to zero — and such a state is maintained during the whole time. These results show that the dispersion mechanism introduced in *gradual* GTP, significantly improves the accuracy of the time synchronization protocol.

The next issue to examine is the change in the distance of nodes from the time source and the distribution of this value. Although it may not be obvious, the author has expected the hop count to be smaller than in *basic* GTP. This idea is based on the observation that the dispersion of a node depends on its distance from the time source, since the errors constituting the node's dispersion accumulate on the synchronization path. The results of experiments targeting at exploring the properties of the hop count are shown in Figure 6.9.

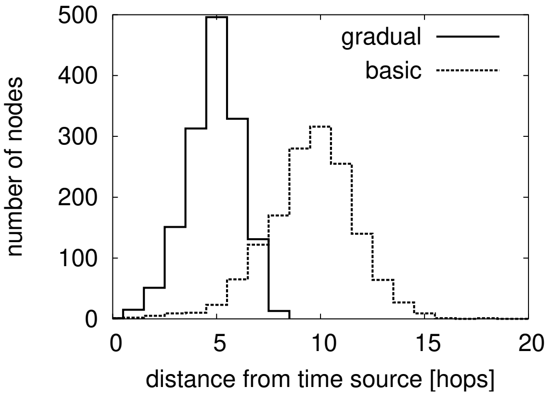
The strange behavior of the time source distance is very easy to explain. In the beginning of the synchronization process, when the nodes are discovering the presence of the time source,



(a) Different clock adjustment periods



(b) Multiple time sources



(c) Distribution of the `TS_DISTANCE` value

Figure 6.9: Properties of the `TS_DISTANCE` value. Note different time-scale for (a) and (b).

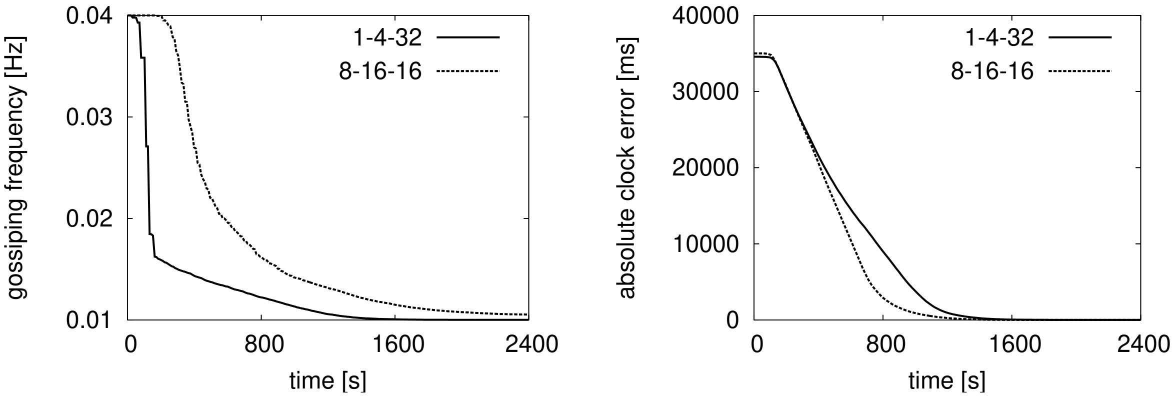
the hop count increases fast.

After initial communication the nodes try to correct their first time approximation using only the samples that come from the nodes with the small dispersion. Such nodes have a small hop count, since in the beginning both the offset and the dispersion are high. Moreover, as stated multiple times here, the dispersion is cumulating on the synchronization path. Therefore, a node which has a small dispersion must have estimated it during the direct gossiping with the time source or one of the nodes 'near' the time source. Due to this fact after rapid initial growth of the time source distance, the value decreases gradually (depending on the clock adjustment period) as nodes correct their time.

When the whole network is synchronized (cf. Figure 6.8), all the nodes have their dispersion on a comparable level, so they can synchronize with each other, which explains the later slow growth of the hop count to a stable value.

As can be easily observed the time source distance in *gradual* GTP is almost two times smaller than in the previous protocol version. Adding time sources to the network decreases the value in the same manner as in *basic* GTP, however, the constants are smaller in this case. Furthermore, the distribution of hop count is more even — nearly one out of three nodes has its value equal to the average.

Another question concerns the operation of the gossiping frequency adjustment improvements. In the experiments, whose results are shown in Figure 6.10, the values of `_MIN_GOSSIPING_DELAY_` and `_MAX_GOSSIPING_DELAY_` were equal to 25 and 100 seconds respectively. The clock adjustment period was set to 10 ms. Other parameters remained unchanged. The symbol m-M-N in the legend of the chart denotes values of appropriate parameters as described in Section 4.3.2. As a reminder, N denotes the length of the buffer storing absolute offset values of the last N samples, M is the number of the last samples utilized for calculating gossiping delay, while m specifies how often (considering the number of samples) the gossiping delay can be changed.



(a) Average gossiping frequency

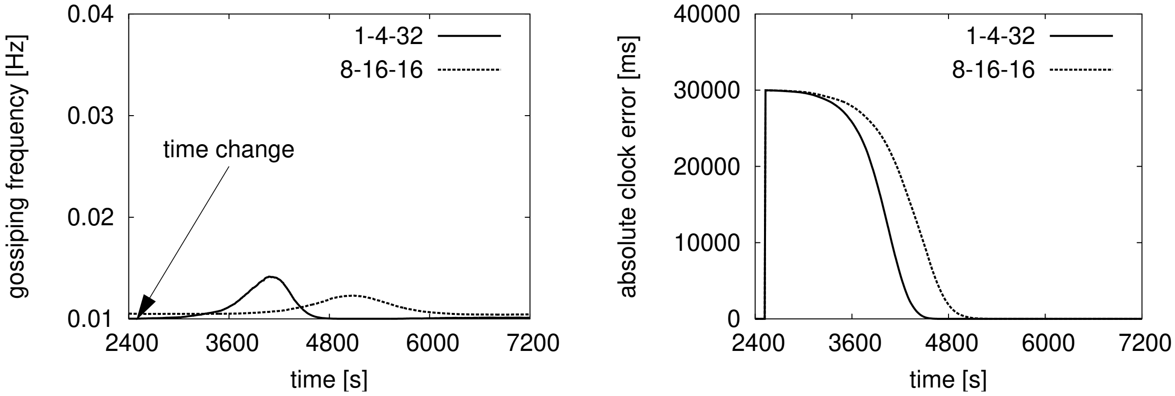
(b) Average absolute offset

Figure 6.10: Convergence of the gossiping frequency.

The graphs clearly indicate that the synchronization of the network is accompanied by the gradual decrease of the gossiping frequency. The setting denoted as 1-4-32 corresponds to the scenario in which the gossiping frequency is modified after accepting each sample for synchronization. To compute the new frequency 4 last samples are used, while the buffer stores 32 last samples for bounds mapping. In the 8-16-16 the buffer holds the 16 last samples which all are used for computation of the new frequency. The frequency is changed every 8 samples accepted for synchronization. The shape of the gossiping frequency curve depends on the attributes of the algorithm. It is smoother when more historical data is used for computation (8-16-16). Also the influence of the gossiping frequency on the synchronization speed is visible — decreasing the frequency sooner when using 1-4-32 setting increases the period necessary for the network to synchronize.

The capability of the network to adapt this value when some change in the system occurs is directly associated with the gossiping frequency. The next experiments show the reaction of the network to the change of time in the time source. More specifically, after the network synchronizes, the time in the time source is changed by 30 seconds. The results of conducted tests are depicted in Figure 6.11.

One may observe that the setting 1-4-32 has better adaptation properties. The system reacts almost immediately which makes the time required for synchronization shorter. Using 8-16-16 configuration slows down the reaction and therefore increases the time required for synchronization, which is quite obvious. It should, however, be noted that the average gossiping frequency of the whole network does not reach the value of the maximal gossiping frequency, although if analyzing the behavior of a single node, such a situation is possi-

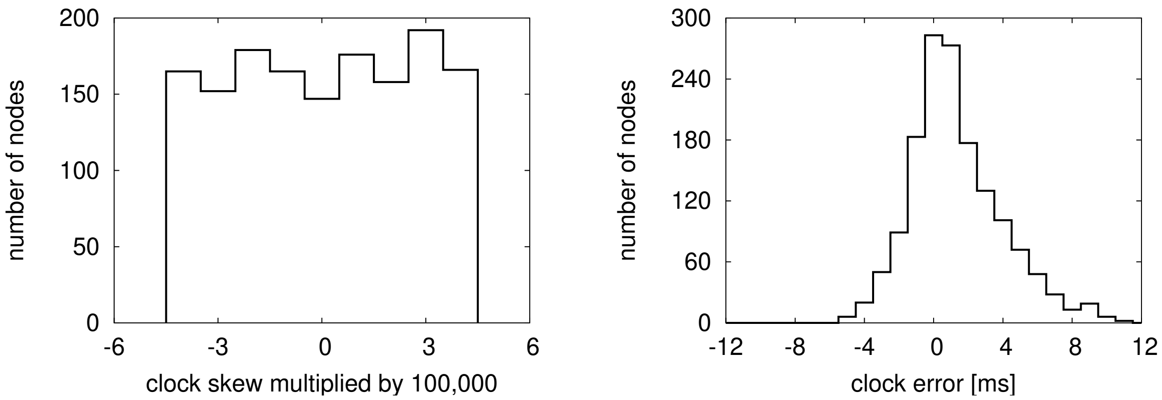


(a) Average gossiping frequency (b) Average absolute offset

Figure 6.11: A change of the gossiping frequency as a reaction to a change in the system.

ble. This is clearly a positive property, since it prevents network congestion while letting some nodes, that require it at a given moment, the opportunity to gossip with the maximal frequency.

The final experiments concern the operation of *gradual* GTP in a system with skewed clocks. The clock settings were similar to the settings for the corresponding tests of the previous version of the protocol — exaggerating the clock frequency errors. Figure 6.12 presents the results of these tests.



(a) Skew distribution (b) Error distribution

Figure 6.12: The synchronization accuracy in the system with skewed clocks.

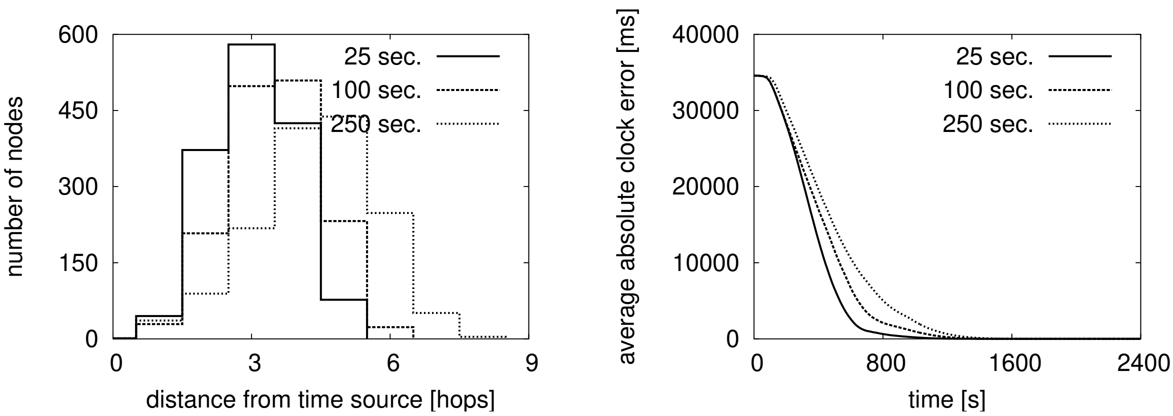
The experiments revealed that the clock skew does not influence the speed of network synchronization. In the stable state the absolute errors are limited to 18 ms. The charts indicate that slightly more errors lean towards positive values, but such a behavior is caused by the fact that there were 32 more clocks with positive skew during the experiments.

The most important conclusion is that *gradual* GTP is able to maintain the time within reasonable bounds even in the presence of very poor clocks. Furthermore, the dispersion mechanism provides the ability to estimate the accuracy of time provided by a given node. Such a value may be used by applications to examine to what extent the local time is consistent with the reference one.

6.2.3. *Selective* GTP

The difference between *selective* GTP and the previous version of the protocol is the usage of application-specific data in the overlay management protocol. The latter one should keep the information about neighbors in the cache as fresh as possible in order to provide reliable data for the peer sampling service. For all protocols presented in Section 5.2, the update of neighborhood information is a result of gossiping of the overlay management protocol. Therefore, the first interesting question is the relationship between the gossiping frequency of the overlay, the gossiping frequency of GTP and the freshness of the overlay cache entries. The answer can be found by analyzing the distribution of values of the `TS_DISTANCE` variable, because the peer sampling strategy for *selective* GTP chooses the neighbor with the smallest dispersion and hop count. It is expected that increasing the gossiping frequency of the overlay should decrease the age of the cache entries and thereby improve the freshness of data provided for the peer sampling strategy.

During the tests the parameters' values were exactly the same as for the previous version of the protocol. The only thing modified was the gossiping frequency of the overlay management protocol (more specifically, ΔT value). The results are presented in Figure 6.13.



(a) Hop count distribution (CYCLON)　　(b) Convergence speed (CYCLON)

Figure 6.13: The influence of the ΔT value on the freshness of the application-specific data for the peer sampling service.

It can be easily observed that if the gossiping frequency of the overlay management protocol is equal to the maximal possible gossiping frequency of GTP (ΔT equal to 25 seconds) then the data is really fresh. For the majority of nodes the hop count value is smaller than or equal to the shortest path length property of the overlay graph (cf. Table 5.1), which is consistent with the theoretical reasoning presented in Section 4.4. On the other hand, if the overlay gossiping frequency used is the same as for *gradual* GTP tests (ΔT equal to 250 seconds which is 2.5 times higher than minimal gossiping frequency bound of GTP) the distribution is almost unaffected (cf. Figure 6.9c).

Furthermore, the decrease of the `TS_DISTANCE` value increases the speed of the network synchronization, as depicted in Figure 6.13b. Such behavior can be explained by that the nodes pick the neighbors with the smallest dispersion (which usually means the most accurate time) to gossip with. The slow-down in the synchronization speed at the end of the synchronization process is caused by the fact that although the nodes contact the neighbors with the optimal dispersion value getting the most accurate time, they still have to adjust their clocks in the gradual adjustment process. Note that correcting the error of 60 seconds, using the parameters selected for the test, requires 60,000 clock adjustment periods which accounts for 600 seconds.

For the rest of the tests described in this section the value of ΔT parameter is equal to 25 seconds. In other words, the overlay management protocol gossips with the maximal allowed gossiping frequency for GTP.

The next issue to examine is the influence of the overlay management protocol on the freshness of application-specific data. Some of the presented protocols try to limit the time the entry remains in the cache by using timestamps or other similar methods and the question is how their heuristics work in practice. During the experiments the methodology remained the same — the distribution of hop count was analyzed. The results are shown in Figure 6.14.

It turns out that the timestamp mechanism utilized in Newscast ensures the best properties. The implemented frequency-based optimization of Lpbcast also provides good freshness of data. CYCLON, although presenting relatively high number of nodes with hop count equal to 1 (the neighbors of the time source), has worse properties than the latter two. Finally, as expected, Shuffling, which does not use any algorithms to control the time the entry remains in the cache, performs the worst.

It should be noted that the good distribution of the time source distance is not the only factor that influences the quality of time provided by GTP. It may even cause some problems concerning load balancing (as stated in Section 4.4). However, during the experiments with the small networks, the author has not observed any decrement in the quality of time provided by *selective* GTP when operating with any of the overlay management protocols, comparing to *gradual* GTP.

6.3. Large-Scale Experiments

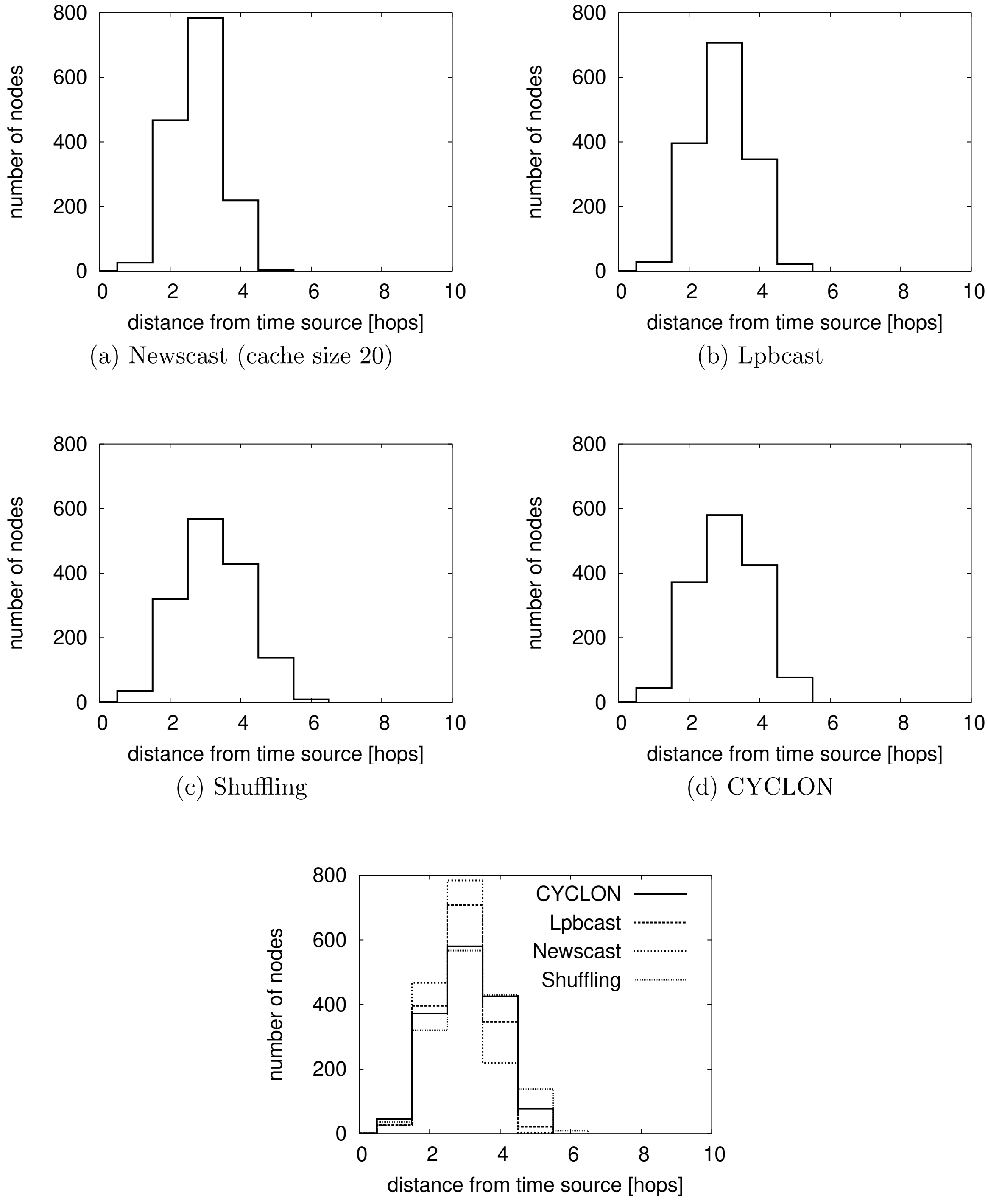
The large-scale experiments have been carried out for systems of 10,500 nodes (denoted later as the *medium network/system*) and 64,500 nodes (denoted as the *large network/system*) distributed among machines of DAS-2 (see Section 6.1). The author focused on analyzing the properties of different GTP versions in large networks as well as examining the influence of the underlying physical network's heterogeneity on the behavior of the protocols. Due to the latter issue, the deployment of nodes across machines of the cluster is described further.

6.3.1. Experimental Setting

During the tests each DAS-2 machine was running two instances of JVM (one instance per processor). One JVM was acting as the RMI registry for the system utilities (see Section 7.2 for explanation). All other instances were hosting 500 nodes each (the medium and large systems were using 11 and 65 machines respectively). The observed errors between the time displayed by the physical clocks of different machines were smaller than or equal to ~1 ms.

The above mapping of nodes to the machines provides physical network heterogeneity. More specifically, the following communication types depending on the relative location of gossiping nodes were involved:

- intraprocess communication — between the threads of the nodes running within the same instance of JVM,
- interprocess communication — for nodes hosted by different JVMs, but on the same DAS-2 machine,
- network communication — between nodes residing on different machines.



(a) Newscast (cache size 20)

(b) Lpbcast

(c) Shuffling

(d) CYCLON

(e) comparison

Figure 6.14: Hop count distribution for the protocols for *selective* GTP.

It should be noted that even though 1000 nodes were running on a single DAS-2 machine, more than 98%, in case of the large system, and more than 90%, in case of the medium system, of the communication between nodes was carried through the network links. For any given node, at most 999 other nodes were running on the same machine and at least 63,500, in case of the large system, were running on different machines, which account for 1.55% and 98.45% respectively. As observed during the tests and in the other similar large-scale experiments concerning overlay management [34], the entries in the overlay cache of a node are distributed over all nodes constituting the system, irrespective of their physical locations. Therefore, the expected communication between the nodes on the same machine for the large system was equal only to 1.55%.

As the overlay management protocol CYCLON was chosen. The reason for such a decision was that Newscast should not be used with *basic* GTP and although Lpbcast has better properties maintaining the cache entries fresh than CYCLON, it has much worse in-degree distribution which might have lead to overflowing some nodes with requests. The cache size was set to 30 and the shuffle length to 10. As the initial topology the random graph was selected to shorten the period required for the convergence of the overlay. The gossiping delay for CYCLON was set on a per-test basis: 25 seconds for the medium systems and 250 seconds for the large systems.

In all scenarios there was only one time source. All clocks were ticking with the same frequency and the initial offsets were chosen randomly with a uniform distribution from the sum of intervals -10 to -60 seconds and 10 to 60 seconds. For *basic* GTP the values of `GOSSIPING_DELAY` variable and `_STANDALONE_PERIOD_` were set to 100 seconds and the sample filter buffer length was equal to 8. For *gradual* and *selective* GTP the clock adjustment period was set to 10 ms and the values of `_MIN_GOSSIPING_DELAY_` and `_MAX_GOSSIPING_DELAY_` were equal to 25 and 100 seconds respectively. Moreover, 8-16-16 was selected as the setting for the buffer controlling the changes of the gossiping frequency.

6.3.2. Results

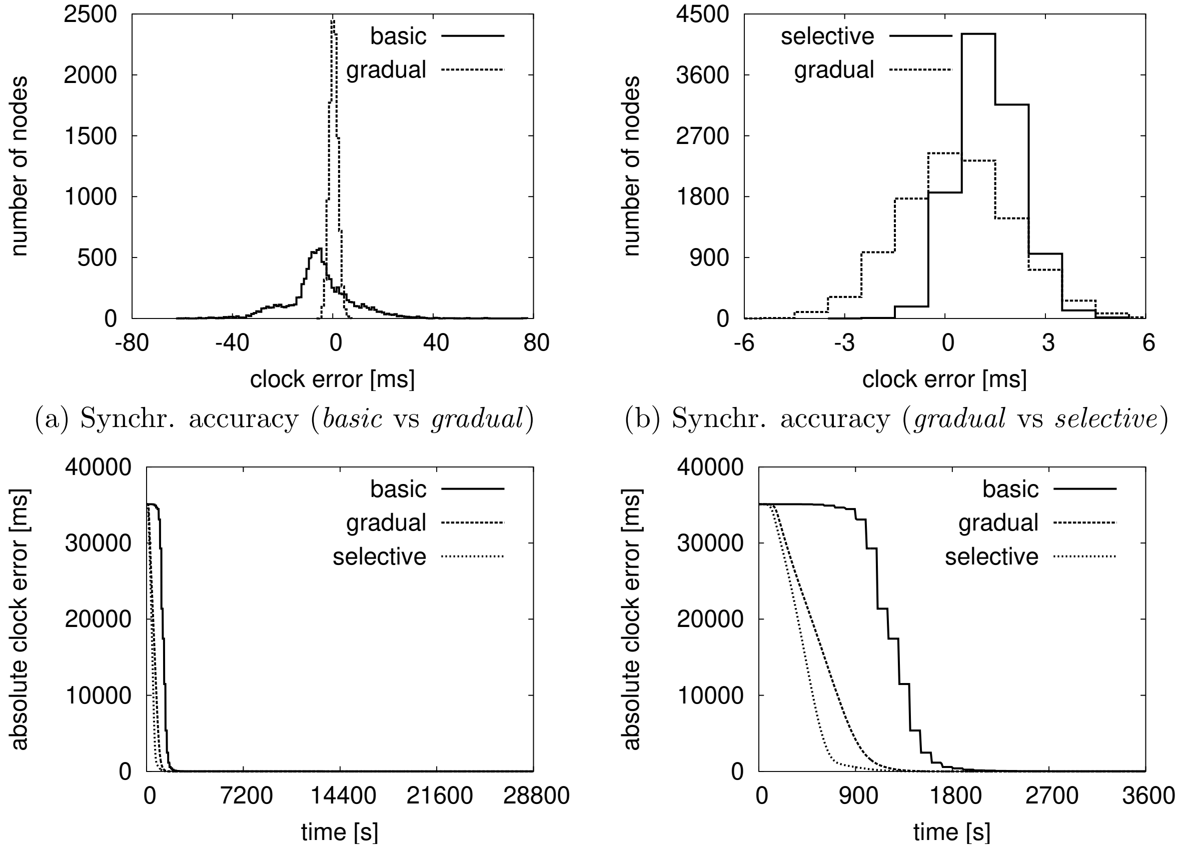
The tests involving medium systems (10,500 nodes) aimed at checking the results of the experiments presented earlier when a heterogeneous environment is used. More specifically, the author was interested mostly in the speed and accuracy of the synchronization as well as in the statistical data concerning the hop count value.

The results concerning the quality of GTP are depicted in Figure 6.15.

They indicate that the speed of convergence remains high despite the network growth: 20-76 gossiping cycles are required for a full convergence of *selective* GTP (the value cannot be estimated perfectly due to the gossiping frequency adjustment mechanism). *Basic* GTP requires only 22 gossiping cycles (note that the real time is longer than for *gradual* or *selective* GTP, because the gossiping delay of *basic* GTP — 100 s — is four times as high as the minimal gossiping delay of *gradual* and *selective* GTP — 25 s).

Unfortunately, the error distribution for *basic* GTP indicates that the algorithm has serious problems with maintaining the accurate time. The errors are unstable and oscillate leaning towards either positive or negative (as in Figure 6.15a) values, despite of sample filtering.

The tests revealed also that *selective* GTP suffers from a load imbalance. During the whole operation most of the nodes have (rather small) positive errors indicating that sending a request lasts shorter than receiving a response. The explanation of such a phenomenon is that the overloaded node processes incoming requests faster than the operating system is able to send responses through the network. Therefore, they are waiting in the outgoing buffer



(a) Synchr. accuracy (*basic* vs *gradual*)

(b) Synchr. accuracy (*gradual* vs *selective*)

(c) Synchronization speed

(d) Synchronization speed (initial hour)

Figure 6.15: The quality of synchronization in the medium network.

for the physical network interface to be ready for transmission. Disabling outgoing message buffering results in an opposite situation. Because the author's implementation does not have any operating system support for timestamping incoming messages, the message sent to an overloaded node, which now has to wait for each processed request to be transmitted by the network interface, resides for a long time in the incoming message buffer before being timestamped. This, however, is a reason for artificial differential delays occurring not due to the physical network behavior, but because of the message path length from the network interface to the processing application. Even the quality of in-degree distribution provided by CYCLON does not solve the problem.

Gradual GTP performs the best. The errors in the stable state do not exceed 12 ms and most of the nodes have the error in the range of the clock precision.

Figure 6.16 presents the distribution of the `TS_DISTANCE` value.

As can be easily observed, the distribution shape is similar to the small networks. Again, *selective* GTP provides good theoretical properties, which, however, in practice lead to a load imbalance.

Due to administrative issues concerning the DAS-2 cluster and the fact that *gradual* and *selective* GTP are similar, but the latter one provides worse synchronization quality, during the experiments with large networks (64,500 nodes) the author has tested only the first two versions of the protocol. These experiments were focused on analyzing the scalability of GTP.

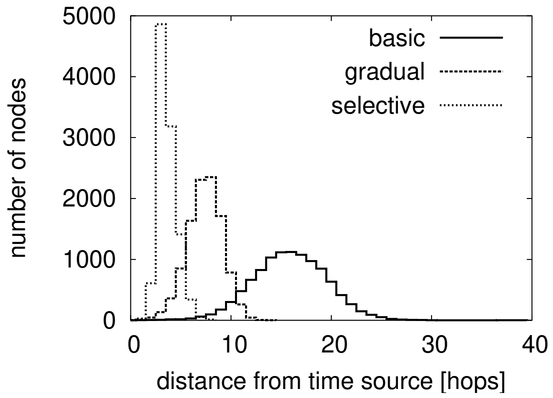


Figure 6.16: Distribution of the hop count value.

Figure 6.17 depicts the interesting properties.

The results of the experiments confirm the claim that, concerning the synchronization speed, both versions of GTP are scalable. *Gradual* GTP requires 26-100 gossiping cycles for the full convergence whereas the previous version of the protocol about 40 cycles. The behavior of the hop count value is consistent with earlier measurements.

The predictions about the accuracy of *basic* GTP based on the previous results turn out to be true. The errors are very unstable and exceed high values ranging even to 322 ms in extreme situations. On the other hand, in the case of the second version of GTP they do not differ from the results of the tests involving medium systems (actually they are better, because the maximal absolute error noticed is 11 ms).

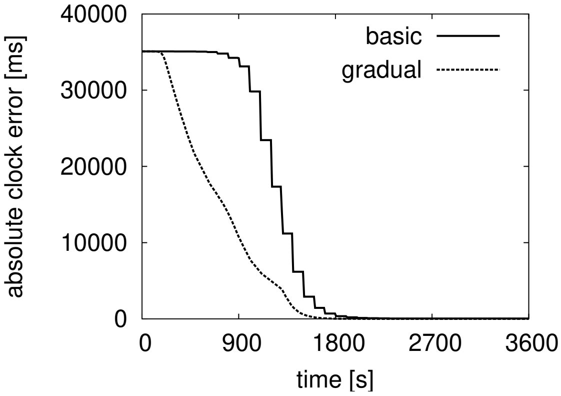
6.4. Summary

From the beginning, the design of *basic* GTP assumed the simplicity of the algorithm. This is the main reason for rather poor accuracy of time maintained by the system, when operating in large networks. It has however, some advantages including high speed of network synchronization. The results constitute good deal of information on what to expect from similar, simple solutions.

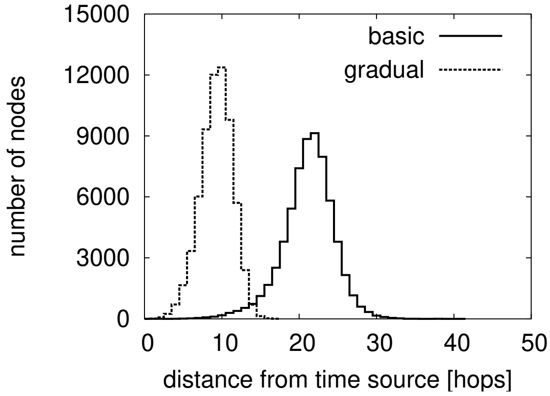
Gradual GTP targets at improving the accuracy of time-keeping and it achieves this objective performing the best of all presented algorithms. It has fine synchronization properties while still being highly scalable. Moreover, when an appropriate overlay management protocol is utilized (e.g. CYCLON), the system is capable of self-organization and is very robust to massive nodes failures.

The experiments conducted with *selective* GTP show that sometimes when trying to improve a certain property of some protocol, one may meet serious problems with some other issues that have not occurred before. Obtaining better theoretical properties of the time propagation paths, results in a load imbalance, which ironically, in case of large systems, leads to decrease in the accuracy of synchronization — the main goal of the time protocol.

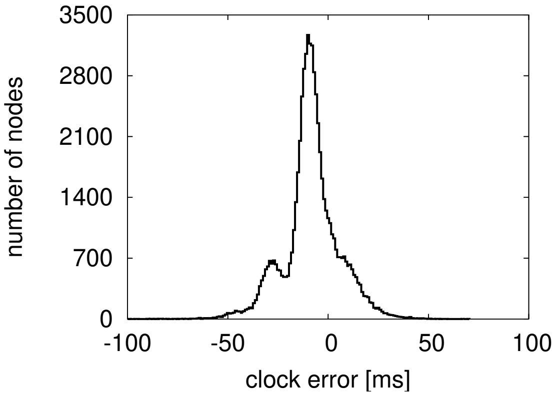
Finally, the results as a whole show one more important issue concerning the procedure of conducting experiments. It should be noted that many properties of the presented algorithms could not be revealed only with simulations, as the number of different aspects that are necessary to be simulated is enormous. Furthermore, emulations conducted only with small networks do not emphasize some intricacies of the presented work that appear only in case



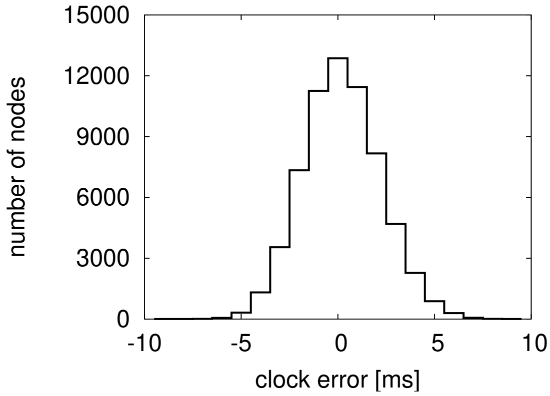
(a) Synchronization speed (initial hour)



(b) Hop count distribution



(c) Error distribution (*basic*)



(d) Error distribution (*gradual*)

Figure 6.17: The properties of synchronization in the large network.

of the large-scale systems (e.g. load imbalance in *selective* GTP or error propagation despite of using sample filtering in *basic* GTP).

The author does not claim that the conducted experiments are exhaustive and thereby Section 8.2 presents possible additional research.

Chapter 7

Implementation

For the purpose of the experiments the author has developed an environment capable of emulating the system on a single machine, a cluster-based computer with a job submissions software (e.g. PBS [24]) or, possibly, in the Internet. The implementation has been created in Java 1.4.2, mainly because of the portability reasons. It is divided into two logical parts: the core implementation and the utilities, described in detail later in this chapter.[1]

The overview of the environment architecture is presented in Figure 7.1.

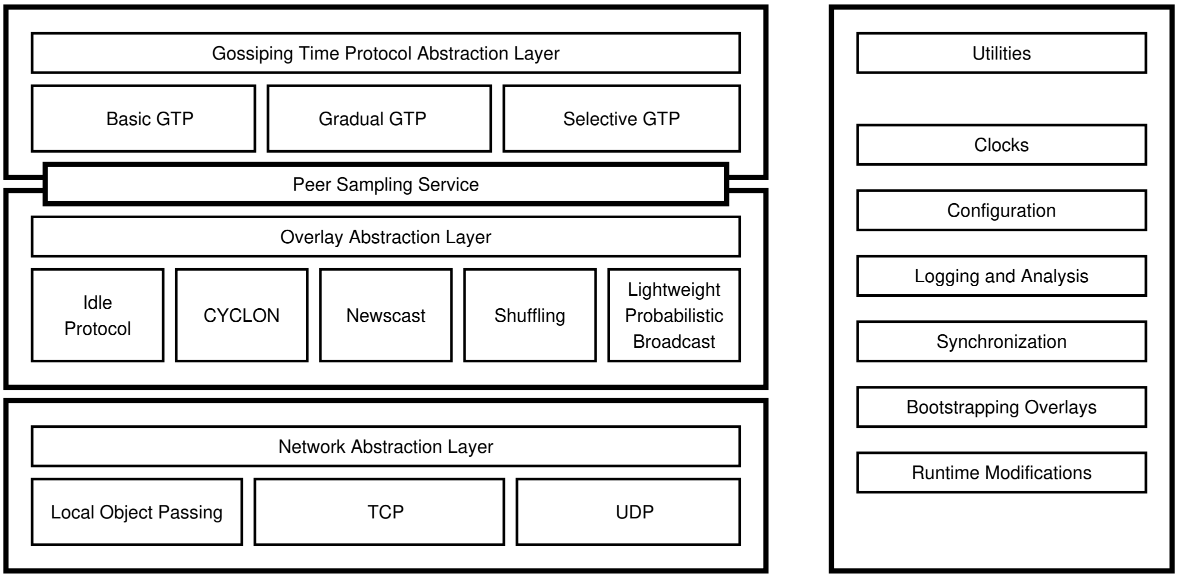


Figure 7.1: The overview of the implementation architecture.

7.1. Core

The core of the environment implements the system model described in Chapter 3. It consists of three layers: networking layer, overlay layer and application (GTP) layer.

The networking layer aims at providing higher layers with uniform communication primitives, that are independent of the underlying protocols. For this reason, a network abstraction

[1] Because of the size of the implementation (about 600KB of source code), the description focuses only on the most important issues.

layer has been designed. Its goal is to specify the following abstract definitions:

- *network address* — used for unique identification of an instance of an application/a protocol in the whole system (e.g. an IP address and a port number in the UDP implementation);
- *network endpoint* — supplies methods for sending and receiving objects from other nodes (e.g. a socket in the UDP implementation);
- *network allocator* — used for transparent allocation of various network-related resources.

The network abstraction layer assumes unreliable (best-effort), connection-less, object-based[2] communication model.

There are three implementations of the networking layer: UDP, TCP and local. UDP and TCP implementations use standard Java classes for network communication. The local implementation utilizes Java synchronization primitives (`wait`, `notify`, `notifyAll` methods and `synchronized` keyword). It can be applied only for communication in the environment run within one instance of JVM and has been developed mainly because of high speed and small resource consumption. All implementations utilize Java serialization mechanism. One-copy semantics for the local implementation is preserved for compatibility reasons with other implementations. Switching between various implementations can be done without any modifications to the source code, by changing a value of one property in the test configuration file.

The overlay layer is responsible for running a membership management protocol. Again, an abstraction layer approach has been used in order to unify access to the peer data by the peer sampling service and classes responsible for starting the environment.

The author has implemented four sample membership management protocols described in Chapter 5: CYCLON, Newscast, Shuffling and Lpbcast. Additionally, an idle protocol, that does not change the overlay shape during the whole test, has been developed — mainly for testing GTP with networks based on random graphs. All protocols (except for idle) utilize one endpoint for communication and two threads: active and passive. The active thread is responsible for periodical initialization of neighborhood information exchanges with other nodes, while the passive thread handles such requests from other nodes and responses to node's own requests. In protocols that employ the shuffle operation, instead of the optimistic solution for handling cyclic exchanges, pessimistic locking has been applied. Additionally, simple mechanisms for recovering after isolation of a node have been incorporated into the implementation.

Switching between various overlay management protocols can be performed by changing the property storing the name of the protocol and setting additional protocol-specific parameters in the test configuration.

The peer sampling service is implemented directly on top of the overlay abstraction layer. The dependence on particular application is exposed only in implementations of various peer sampling strategies.

Finally, the GTP layer has also been built with the design principle involving an abstraction layer for management purposes. All versions of GTP described in Chapter 4 have been implemented. The implementations use one endpoint and two threads — similar to the implementations of the overlay layer. Switching between them is performed by changing the

[2] Basic unit of information that can be sent through the network is a Java object.

protocol version and setting all necessary protocol-specific parameters in the configuration file.

All layers, for measuring time, suspending threads, etc. use time services provided by clock implementations from the utilities components, described in the next section.[3]

7.2. Utilities

The utilities have been intended to automate some parts of experiments and allow almost transparent execution independent of the hardware used (e.g. a single machine or a cluster). They consist of the following packages: clocks, configuration, logging, synchronization, bootstrapping overlays and runtime modifications.

The clocks package includes software implementation of various clock types (perfect clocks, skewed clocks, etc.). Each clock version is based on the functionality provided by Java, in particular, `System.currentTimeMillis` method. This implies the theoretical precision of the clock equal to 1 millisecond.

The configuration package is responsible for loading the configuration of both hardware used for emulation and the test parameters supplied by the user.

For the logging purposes, Log4J library [17] has been incorporated into the system. Each layer on each machine used for emulation owns one file for messages. The logging of system state intended for later analysis is performed by the main thread. Other messages, like warnings or errors, can be logged by an arbitrary thread. In case of emulation on more than one machine, all machines are synchronized before logging a sample. After the experiments, the logs can be processed by classes and scripts designed for analysis of properties of a particular layer.

The synchronization package contains primitives for threads synchronization, e.g. barriers. To optimize the communication during experiments conducted on multiple machines, hierarchical synchronization objects have been developed. Threads within one JVM synchronize locally (without any network communication) and, after that, the JVMs spawned on different machines synchronize by the means of RMI, leading to only one remote method invocation per JVM, instead of one per thread.

Bootstrapping of the initial overlay topology is performed by means of a topology manager. Each JVM, after loading the configuration file and spawning nodes, communicates via RMI with the topology manager which generates neighborhood lists for nodes of that JVM.

Finally, custom runtime modifications, like changing time of the nodes' clocks, adding nodes to the system or other user-defined operations, may be scheduled for an experiment.

[3] This implies that for example sleeping for a certain amount of time lasts an amount of time dependent on the clock skew of a given node.

Chapter 8

Applications and Future Work

Due to the provided functionality, GTP can be applied in many domains of distributed systems, not necessarily associated with P2P networks. The protocol does not require any specific clock services to be supplied by the operating system. In the extreme case, like in the implementation presented before, only one function responsible for returning some form of local time is required. This portability and inherent simplicity of GTP allow to easily incorporate it as a time service into large applications.

8.1. Applications of GTP

As the first trivial use-case one may consider synchronizing clocks of machines forming a large system (e.g. a P2P network) for the purpose of statistical logging or computation of some aggregate function. A similar example involves synchronizing processes in large-scale experiments conducted using resources reserved worldwide, for instance, through PlanetLab [23].

Another application of GTP is associated with systems dealing with distribution by using timestamps for caching or leases [31, Chap. 6.4]. Consider the cache system designed for HTTP/1.1 [5] as an example. It is based on a hierarchy of proxy servers and clients' caches to support documents replication. The documents sent within HTTP messages can be accompanied by some tags, among which the following are of special interest here: *Expires*, *Last-Modified*, *If-Modified-Since* and *If-Unmodified-Since*. Their values allow to make a decision, either by a client's browser or a proxy server, whether a cached copy of a document is still up-to-date. These tags, especially *Expires*, assume that time within the cache hierarchy is synchronized. Instead of depending on this assumption, one can employ GTP to ensure the synchronization. More specifically, the proxies may form an overlay network which can be used by GTP in order to keep accurate time on all machines.

The field of GRID computing, that is recently successfully applying P2P technology, may also benefit from solutions based on GTP. In particular, let the computations be performed by a highly dynamic network of nodes, e.g. idle PCs of many people spread worldwide. The resource reservation system should support allocation of a number of machines for a given period at specific time. In order to provide such functionality, the clocks of the computers (or at least the processes responsible for local reservation of resources) have to be synchronized, which is the task for GTP. Moreover, the error estimation by the means of the dispersion value can be utilized by the parallel application as a part of fault tolerance mechanisms.

The few examples described above should give an idea of possible applications of the developed solutions. One may easily think of many more, including even some misusing GTP (e.g. coordination, in a difficult-to-detect way, of the denial of service attacks by computer viruses). In face of the aforementioned facts, the need of further research concerning the designed algorithms seems obvious.

8.2. Future Work

The nearest research will target at overcoming the load balancing problem of *selective* GTP. The solution being currently worked out by the author introduces some randomness in the peer sampling strategy and gossiping frequency calculation alleviating the results of massive requests aimed at nodes with low hop count.

Moreover, further cooperation between the overlay layer and GTP is considered. Time synchronization protocols are usually not the only processes active, even in the case of time sources. It may happen that a time source may be heavily loaded due to the operation of some other service. The interesting question is whether the system can react to such situations by letting GTP to monitor the incoming samples and changing the peer sampling strategy on demand to choose a different node for gossiping.

Directly associated with the above ideas is the utilization of the overlay management protocols that form the logical network based on some properties of the physical connections between the nodes (e.g. Saxons described in Section 5.2.5). The time service can influence behavior of such a protocol by supplying it with the data collected during the gossiping (e.g. round-trip delay on the path between nodes). Such an approach should further improve the adaptation speed in situations described above.

The protocols should also be extended with mechanisms capable of detecting truechimers and falsetickers (see Section 2.2), which is connected to the problem of reputation in P2P networks. However, in case of the time service the solutions may be different. Some fault tolerance algorithms should also be considered to deal with the crashes of time sources.

Finally, after all proposed improvements, large-scale real-life-operation experiments of GTP have to be conducted. They cause some further problems concerning measurements of the synchronization quality, but surely will give a better picture of the advantages and disadvantages of GTP.

Chapter 9

Conclusions

Many applications require reliable and fairly accurate local time in order to operate properly. Currently utilized time synchronization algorithms are capable of fulfilling this task, however, they are often lacking the properties of robustness and scalability. Therefore, the developers of distributed systems may be forced to seek for alternative solutions.

The Gossiping Time Protocol suite, presented in this thesis has been designed to be robust and scalable, while providing good quality of time synchronization. The algorithms operate in an epidemic style which makes them easy to implement, extend and maintain. Moreover, as shown before, they do not require any operating system support in order to have a reasonable time service accuracy.

The introduced system model extending previous research [10, 12], especially the new peer sampling service designed by the author for the purpose of GTP, forms a flexible environment for building layered large-scale distributed applications. Such systems separate common functionality, like managing group membership and forming a logical network, from application-specific issues allowing for sharing or replacing the (parts of) layers. The examples on how to port existing solutions, which usually incorporate application data in the membership-related messages, are also described here.

Another important contribution of the thesis are the large-scale experiments of the proposed algorithms. Their advantage is the fact that the protocol operation was emulated using a real implementation. Many test configurations have been used to make the results suitable for verifying the claims concerning the behavior of the system in the real-life networks. The research shows that the theoretical reasoning is correct in most cases, while revealing some additional interesting matters.

The consecutive versions of GTP have been being developed in an incremental manner. The author started with the simplest solution and throughout the analysis of the properties of a given version, particular improvements and enhancements dealing with appearing problems were proposed. As noted earlier, some of them turn out to be important and some, in spite of being interesting from the theoretical point of view, are probably useless in practice. Nevertheless, the author believes that the reasoning presented here constitutes a sort of a guide or a starting point for optimizations on the existing time synchronization techniques, or may be even the motivation for designing new ones, with better desired properties.

Bibliography

[1] R. Albert and A.-L. Barabási, *Statistical Mechanics of Complex Networks.* Reviews of Modern Physics, January 2001.

[2] N. T. J. Bailey, *Mathematical Theory of Infectious Diseases and Its Applications.* Charles Griffin & Company Ltd, 1975.

[3] Distributed ASCI Supercomputer (DAS-2) homepage, `http://www.cs.vu.nl/das2/`.

[4] P. Eugster, R. Guerraoui, S. Handurukande, A.-M. Kermarrec and P. Kouznetsov, *Lightweight Probabilistic Broadcast.* ACM Trans. Comp. Syst., December 2003.

[5] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616, June 1999.

[6] A. Ganesh, A.-M. Kermarrec and L. Massoulié, *SCAMP: Peer-to-Peer Lightweight Membership Service for Large-Scale Group Communication.* In Proc. Networked Group Communication Workshop, Springer-Verlag, Berlin, November 2001.

[7] GlobeSoul project homepage, `http://www.cs.vu.nl/globesoul/`.

[8] R. Gusella and S. Zatti, *The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD.* IEEE Trans. Softw. Eng., July 1989.

[9] M. Jelasity and M. van Steen, *Large-Scale Newscast Computing on the Internet.* Internal report IR-503, Vrije Universiteit, Department of Computer Science, October 2002.

[10] M. Jelasity, A. Montresor and O. Babaoglu, *A Modular Paradigm for Building Self-organizing Peer-to-Peer Applications.* In the proc. of The 1st International Workshop on Engineering Self-Organising Applications, Lecture Notes in Computer Science, vol. 2977, Springer-Verlag, Berlin, 2004.

[11] M. Jelasity, W. Kowalczyk and M. van Steen, *An Approach to Massively Distributed Aggregate Computing on Peer-to-Peer Networks.* In Proc. of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, IEEE Computer Society, Coruna, Spain, 2004.

[12] M. Jelasity, R. Guerraoui, A.-M. Kermarrec and M. van Steen, *The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations.* Proc. 5th ACM/IFIP/USENIX International Middleware Conference, Toronto, Canada, October 2004.

[13] KaZaa homepage, `http://www.kazaa.com/`.

[14] D. Kempe, A. Dobra and J. Gehrke, *Gossip-Based Computation of Aggregate Information.* In Proc. of the 44th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, 2003.

[15] J. Levine, M. A. Lombardi and A. N. Novick, *NIST Computer Time Services: Internet Time Service (ITS), Automated Computer Time Service (ACTS), and time.gov Web Sites.* Technical Report, National Institute of Standards and Technology (NIST), May 2002.

[16] B. Liskov, *Practical Uses of Synchronized Clocks in Distributed Systems.* Distributed Computing, vol. 6, 1993.

[17] Log4J homepage, `http://jakarta.apache.org/`.

[18] D. L. Mills, *DCN Local-Network Protocols.* RFC 891, December 1983.

[19] D. L. Mills (review by J. Comuzzi), *A Comparison of the Network Time Protocol and Digital Time Service.* Available in the Internet, February (review in March), 1990.

[20] D. L. Mills, *Network Time Protocol (Version 3) Specification, Implementation and Analysis.* RFC 1305, March 1992.

[21] D. L. Mills, *Improved Algorithms for Synchronizing Computer Network Clocks.* IEEE/ACM Trans. Netw., June 1995.

[22] D. L. Mills, *Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI.* RFC 2030, October 1996.

[23] PlanetLab homepage, `http://www.planet-lab.org/`.

[24] Portable Batch System homepage, `http://www.openpbs.org/`.

[25] J. Postel and K. Harrenstien, *Time Protocol.* RFC 868, May 1983.

[26] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A Scalable Content-Addressable Network.* In Proc. of the ACM SIGCOMM, San Diego, CA, August 2001.

[27] A. Rowstron and P. Druschel, *Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems.* Proc. Middleware 2001, Heidelberg, Germany, November 2001.

[28] K. Shen, *Structure Management for Scalable Overlay Service Construction.* In Proc. of the First USENIX/ACM Sympo. on Networked Systems Design and Implementation, San Francisco, CA, March 2004.

[29] A. Stavrou, D. Rubenstein and S. Sahu, *A Lightweight, Robust P2P System to Handle Flash Crowds.* IEEE J. Selected Areas Commun., January 2004.

[30] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.* In Proc. of the ACM SIGCOMM, San Diego, CA, August 2001.

[31] A.S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms.* Prentice Hall, 2002.

[32] R. van Renesse, K. P. Birman and W. Vogels, *Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining.* ACM Trans. on Computer Systems, May 2003.

[33] S. Voulgaris and M. van Steen, *An Epidemic Protocol for Managing Routing Tables in Very Large Peer-to-Peer Networks.* In Proc. of the 14th IFIP/IEEE Intern. Workshop on Distr. Systems: Operations and Management, number 2867 in Lecture Notes in Computer Science, Springer, 2003.

[34] S. Voulgaris, M. Jelasity and M. van Steen, *A Robust and Scalable Peer-to-Peer Gossiping Protocol.* Proc. 2nd Int'l Workshop on Agents and Peer-to-Peer Computing (AP2PC 2003), Melbourne, Australia, July 2003.

[35] S. Voulgaris, D. Gavidia and M. van Steen, *CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays.* Journal of Network and Systems Management, to appear.

[36] B. Yang and H. Garcia-Molina, *Designing a Super-Peer Network*, In Proc. 19th Int'l Conf. Data Engineering, Los Alamitos, CA, March 2003.

[37] B. Y. Zhao, J. Kubiatowicz and A. D. Joseph, *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing* Technical Report CSD-01-1141, Computer Science Division, University of California, Berkeley, April 2001.

Listings