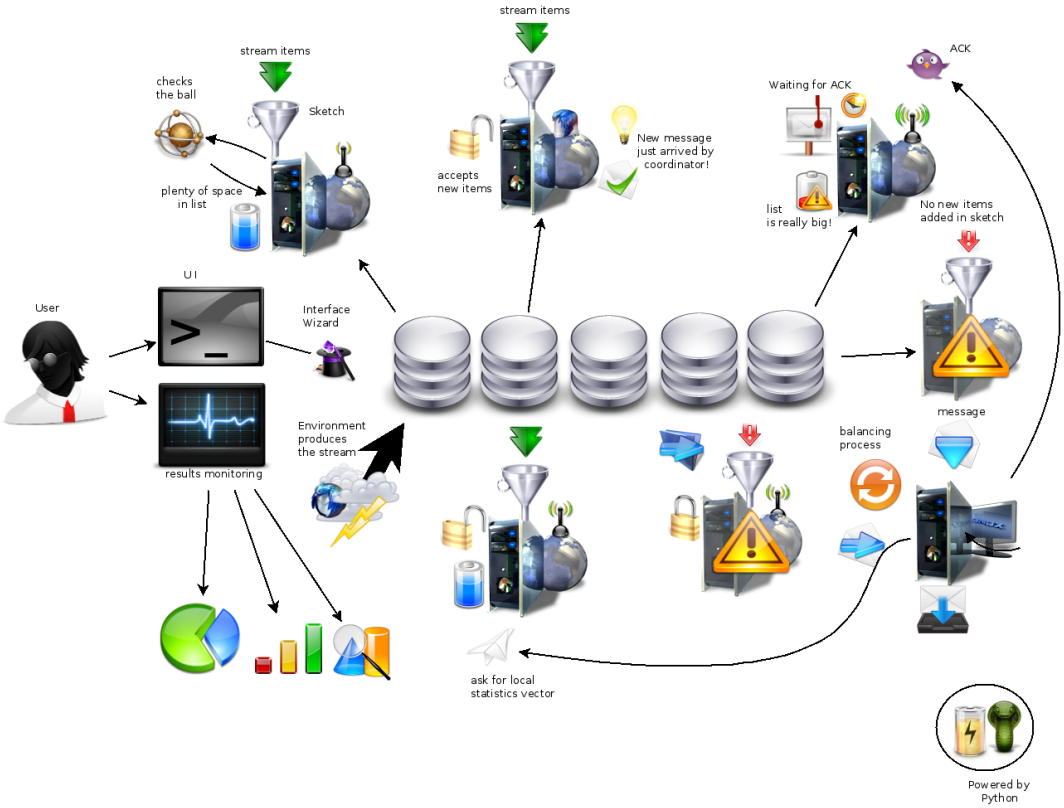
BABIS BABALIS

A SIMULATOR FOR MONITORING DATA STREAMS

A SIMULATOR FOR MONITORING DATA STREAMS

BABIS BABALIS

SUPERVISOR: VASSILIS SAMOLADAS



Development Of Simulation Systems For Monitoring Data Streams

COMMITTEE:

VASSILIS SAMOLADAS

MINOS GAROFALAKIS

ANTONIOS DELIGIANNAKIS

Department of Electronic and Computer Engineering

February 2013 –

Dedicated to the loving memory of my grandpa Dimosthenes Tsaknias.

1921 – 2010

ABSTRACT

A new class of data has come to the foreground and it is used by an increasing number of applications: applications in which the data is modeled as data streams. Examples of such applications are financial applications, network monitoring applications, telecommunication applications, etc. However, the continuous arrival of data in multiple, rapid, possibly time-varying, unpredictable and unbounded streams, defines some new research problems. A lot of work has been done in these fields in recent years and in particular in monitoring data streams. A sub-problem is monitoring data in a distributed system. Most of the proposed publications in this field confine monitoring to simple aggregated streams. On the problem of monitoring data streams, an innovative geometric approach has been suggested where a general monitoring task can be divided into (smaller) local tasks by applying some constraints. The constraints are used locally in order to filter out data that do not affect the monitoring outcome, yet avoiding unnecessary communication.

A problem that we are dealing with, is the lack of good modeling system, so as to simulate and test such approximations. It is very difficult to have a closed-form, analytic solution for a modeling system, due to large number of mathematical constraints and variables that would probably be required. We aim to build a simulator which simulates DES (Discrete Event Simulation) and it will comply to a variety of experiments easily. Moreover, the simulator should be of general purpose, easily customizable and extensible. So, the simulator should be as general as possible, without losing its main purpose which is to simulate particular communication protocols for distributed data streams [15] ...

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [10]

ACKNOWLEDGMENTS

First of all, I would like to thank my parents Dimitris and Evangelia for their support in every decision and move I made all these years (even if it proved totally wrong) and their love. I would also like to thank my sister Vasiliki and my brother Dimosthenes for all their tolerance towards my whims.

Special thanks to my supervisor mr. Vassilis Samoladas for his valuable help and useful conversations due to which I have learnt to think as a software engineer. I would also like to thank Professor Minos Garofalakis and Assistant Professor Antonios Deligiannakis.

I would like to express my appreciation as well as special gratitudes to Kate, who stood (and still stands) by me in my darkest moments and who helped me to understand that difficulties in science and life exist so as to be overcome.

Last but not least, I would like to thank Manos, Dio and Panos, for the great moments we spent together as well as the endless discussion nights and struggle, all of which contributed to what I have become today.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

API	Application Programming Interface
DES	Discrete Event Simulation
SimPy	Simulation in Python
BaSh	Bourne Again Shell

Part I

INTRODUCTION AND BACKGROUND

1 INTRODUCTION

In recent years, interest in data streams and manipulation of data streams is growing more and more. In particular, manipulation and extraction of useful information from a data stream is really important to the scientific computing community.

A data stream is an uninterrupted flow of a long sequence of data. Usually, streams are consisted of large amounts of data and we don't have the opportunity to save a data stream (which often "runs" on-the-fly) for further process. In our times, we can easily observe that data streams are everywhere and we tend to use streams everyday in our lives. Applications like twitter or facebook can be considered as vast reservoirs of data flows. Financial data, sensor networks, or even television-shows or live-games in our computers consist more examples of data streams and of their importance and use.

A big challenge is the real-time monitoring of numerous and large data streams. This can be done with the help of a special class of queries installed onto data, named monitoring queries [1]. An example of a monitoring query is a query which counts the frequency of discrete items that appear in a set of streams. Another one is a query which detects if the frequency of an item into the stream is bigger than a given number. The challenge gets even bigger if we have to apply such a monitored query to a distributed stream. The main problem in such queries that are applied to distributed data streams is the communication overhead that is needed for combining the local results and composing the final answer for the query.

A novel approach [15] has been proposed as a solution for the problem described above. According to this work, two algorithms are proposed for minimizing the communication between the "points" (from now on named nodes) that monitor the stream. The algorithms are based on a geometric analysis of the problem. Even if the algorithms are tested, there is still some concern on their effectiveness and their precise results. The experiments and the simulations that have been done are not quite enough.

1 Monitoring queries are different from the simple queries For more information see chapter 2.

DES *stands for* D*iscrete* E*vent* S*imulation*

The problem we stated is that a set of discrete event simulation (DES) experiments are missing for the given algorithms. In discrete event simulation the whole system is represented as a chronological sequence of events. Each event is happening in a particular moment (of time) and results to a change to the system. Given such a set of experiments (DES experiments), we could extract some precise conclusions about the efficiency of the algorithms proposed and we could encourage and clarify further possible research in the particular scientific area.

1.1 THESIS CONTRIBUTION

As we conclude from above, the lack of a simulator that would be able to simulate a variety of different scenarios and cases, is a drawback for testing algorithms and protocols real-time. This thesis contributes Elita, which is a simulation software application focused on simulating the algorithms given to [15] and also offering a wide range of true, real-time parameters and heavy parameter-tuning for any kind of DES-experiment.

In particular, our software implements a general-purpose simulator for any kind of Discrete Event Simulation (DES) the user may want to run. Furthermore, the simulations the user can run through Elita, may contain a lot of parameters, depending on user's interests and research. Moreover, Elita is written ready to accept any kind of new set of experiments the user wants to run. It also can manipulate any kind of data (text or binary) and it can run many more protocols than the ones given in [15].

Another feature of Elita is the selective use of parts of the system. The system is well defined and implemented in a discrete way. As a result, the user can use only the discrete parts that may need such as the network, and/or the environment that produces data, and/or the protocol implementations, etc.

Finally, in the extreme case that the user wants to change/implement a lot of new features, Elita is able to accept any user's plugin that implements the desired functionality easily (and with great pleasure).

1.2 THESIS OVERVIEW

Chapter 2 describes the necessary background for this thesis. Python, SimPy, sketches, streams, distributed streams and monitoring distributed queries are described as well. Furthermore, Chapter 2 provides basic background information about the

protocols that are tested with the simulator. In Chapter 3 the significance of a good simulator is discussed and we state the requirements of our simulator, while in Chapter 4 we briefly refer to the related work of others. In Chapter 5 we describe the design of our simulator's architecture and describe extensively all the available features and functionalities it provides. In Chapter 6 we present our simulator's implementation from a technical point of view. In Chapter 7 we present the protocol described in Chapter 2 demonstrating the effectiveness and real-time performance of our simulator. Finally, in Chapter 8 we discuss the results of this thesis and we suggest some possible future research enhancements and directions.

2

THEORETICAL BACKGROUND

2.1 STREAMS - MONITORING QUERIES

A data stream is an ordered sequence of instances (data blocks, or other). This special data type can be read only once (or few times) by the applications (we usually name this kind of applications data stream mining applications and the procedure of reading and processing such a data stream, data mining). Such applications include:

1. computer network traffic
2. phone conversations
3. ATM transactions
4. web searches
5. sensor data
6. stock exchange records

etc.

In applications that are called to manipulate data streams, the main goal is to extract useful information and conclusions out of the data stream. This task has significant challenges, due to the large amount of data, their rapid passing and the availability constraints of the stream (data of a stream is often available only once).

To extract knowledge from a stream, a query is installed to the data stream. Depending on data values, the query returns its results.

2.1.1 *Streams*

In telecommunications and computing, a data stream is a sequence of digitally encoded coherent signals (packets of data or data packets) used to transmit or receive information that is in the process of being transmitted.

data stream definition

A formal definition follows: A *data stream* is an ordered pair

(s, Δ) where:

- s: is a sequence of tuples (or bits, elements, etc) and
- Δ: a sequence of positive real time intervals.

Nowadays, data streams are almost everywhere. We are streaming video from the internet. Social media can be seen as stream repositories. More generally, there is so much information that can not be saved in a hard disk for further processing. Volume of data flow is enormous and the stream keeps going really fast. This form of data has the following characteristics:

1. it carries a vast amount of data (hence, it usually can't be saved for further processing).
2. it passes only one time (no repetition in terms of seeing again the same part of a stream).

Hence, this recent data type needs special manipulation techniques.

main stream problems

DIVE INTO STREAMS: Intuitively, a data stream represents input data that comes at a really high rate [13]. This is translated to computer infrastructure and communication stress. Consequently, it often is hard to:

- *transmit* the whole input to the application/node/etc. (see Section 2.1.2 for more information),
- *compute* query results or make data process that require hardware load, at the rate that data is presented, and
- *store* the data stream either short term or long term.

As we easily now understand, the particular data form depends on many parameters that affect the overall system performance and quite often these parameters are the bottleneck in an application.

DATA STREAM MODELS: Muthukrishnan [13] describes three main data stream models. Given a data stream

$$S = [a_1, a_2, \ldots, a_n], n \in \mathbb{N}$$

the relation between i and a_i is described:

- *Time Series Model.* Each i equals a_i and values of a_i (and hence, i) are only being increased by the time.
- *Cash Register Model.* The correlation here resembles to the one of the previous model, but it is more general. Here, $a_j = a_{j-1} + i$. To restate, each stream element could increment an a_j, but not in the linear, smooth way the previous model did.
- *Turnstile Model.* This is the most general model of all. Here, each new element is just an update to stream. It could either increase or decrease the values of the stream.

Presenting the models by ascending order of generality, we have the time series model, the cash register model and the turnstile model. Even if we would love to design elegant algorithms and protocols for the turnstile model, we cannot. Fortunately, all three models are useful, as they successfully model different types of problems. Therefore, an algorithm is designed according to the model that better fits to the problem. Admissibly, the most popular model is the cash register model.

2.1.2 *Distributed Streams*

The problem of data stream manipulation becomes more complex when the stream is distributed. In many instances, streams are generated at multiple distributed nodes. These data streams should be manipulated with different techniques than usual and criteria about what is important are different.[1] In distributed streams communication costs across different nodes or computational and network requirements become more important.

With the rapid development of web and web's infrastructure, distributed streams have been appeared everywhere. Some monitoring tasks and useful applications depend completely on distributed streams. For example, http traffic monitoring, or how many IPs visited a particular node, are simple queries. However, this kind of information is hidden into data streams (and in particular, distributed data streams) and data mining demands special handling.

The main concern is the communication and network overhead. Not only do the algorithms have to deal with data streams' steady nature, but also they have to minimize infrastructure sources' consumption. Even nowadays the scientific community

presents decent activity about distributed data streams and the new challenges that have arisen.

To sum up, algorithms responsible for data mining from streams, should be efficient not only in terms of space and processing time, but also in terms of communication load. [9]

2.1.3 *Monitoring Queries*

Due to the nature of the particular data form (streams), a new category of queries has arisen, the continuous queries. These queries implement the very same operators as the ordinary queries do (select, join, etc) but they are applied continuously to the stream. A special class of queries is the monitoring queries. These queries usually monitor a stream by watching the current (incoming) values. Then an aggregate (usually simple, such as sum) is computed and is being compared to a given threshold. The (monitoring) result depends on whether the threshold has been violated or not. This is the class of queries we are interested in this thesis.

In particular, the monitoring query is defined as follows: Let $X_1, X_2, \ldots, X_d$ be frequency counts for d items over a set of streams. Let $f(X_1, X_2, \ldots . X_d)$ be an arbitrary function over the frequency counts. We are interested in detecting when the value of f rises above or falls below a predetermined threshold value [15].

The problem is more complex than it seems. A big challenge is that the function f is not linear and as a result it needs special treatment.

2.2 SKETCHES

Sketches is an advanced data synopsis. It is an array that it actually saves frequencies of stream items. Naively speaking, sketches are data structures which can be represented as linear transform of the input. They are mostly focused on stream summaries. Each update observed in the stream potentially causes this synopsis to be modified and as a result, the synopsis can be used to answer queries (approximately) over the original data [5].

The basic idea behind sketches is as follows: Let a stream $S = a_1, a_2, \ldots, a_n$. Now, let us define an array A and store to it numbers that correspond to the frequency of each stream element. This is an improvement over the stream and we can

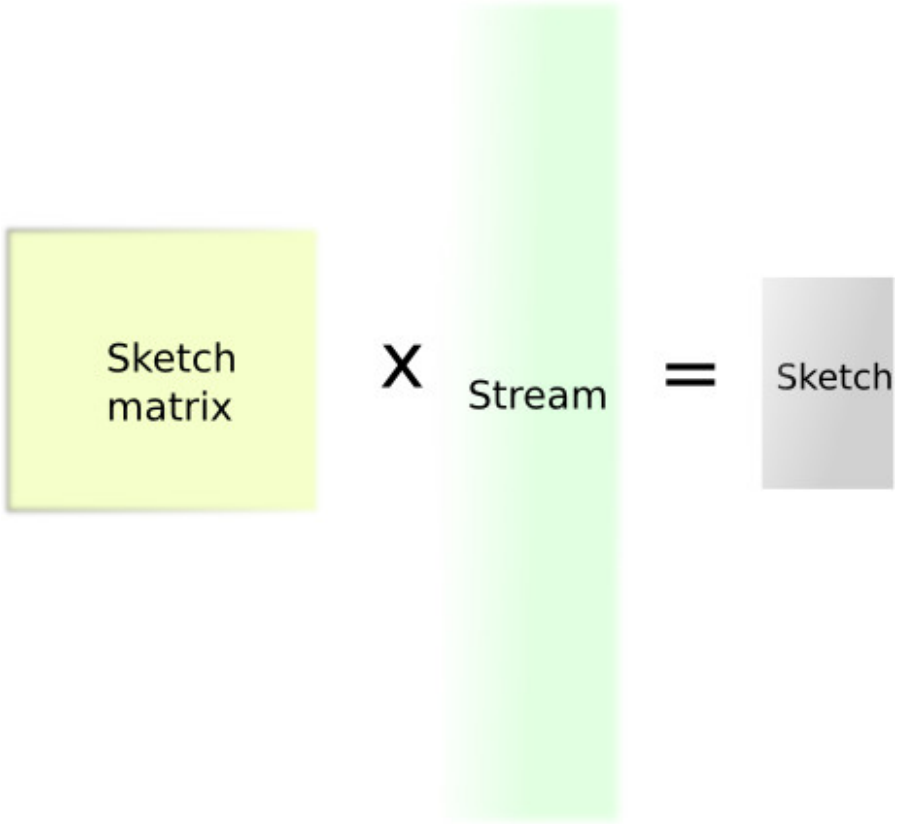


Figure 1: Visualization of sketch creation

have some data conclusions about the stream given by this new-found array.

PROPERTIES OF SKETCHES: We prefer sketches on other data synopses due to the following advantages:

- They support queries
- Their size is log(N), N: size of stream
- Their update speed.
- The time that is needed to answer a query.

There are different kinds of sketches that implement the basic idea. To name some, we have the CM-sketches (count-min sketches), Count sketches, FM-sketches (Flajolet-Martin [8]) etc. In this thesis, AMS sketches [3] are in use.

2.2.1 *AMS Sketch*

Ams sketch is a pioneer variation of basic sketch idea. It was first presented by Alon et al. for confronting a different problem than it is now used, but it turned out to be an optimal data structure for saving and extracting data from a sketch and (as a result) for answering a monitoring query. The main advantages (moreover to other sketches) of the AMS synopsis are that:

- it is very fast
- it guarantees that the answer will be well bounded approximately.

AMS sketch definition

More specifically, this powerful data stream synopsis structure [3] consists of $O(1/\epsilon^2) \times O(\log(1/\delta))$ atomic sketches. An atomic AMS sketch X is a randomized linear projection.

Definition: $X =< \alpha, \xi >= \sum_{i=1}^{n} \alpha[i]\xi(i)$, where ξ : random vector of four-wise independent random variables that map in $[\pm 1]$.

Let us now examine a simple example the AMS sketches work: Let a stream S with elements $S = a_1, a_2, \ldots, a_n$. Let an AMS sketch A. While stream passes, each element is hashed by four-wise independent hash functions, and the number that is produced is the position that each atomic sketch corresponds to the particular element. The idea is that due to many independent hash functions (four-wise independent at least) the number that shows up will be different for each atomic sketch. As a result, when a query is installed into a stream, the query is answered by sketch's saved data. Despite the fact that the sketch is just a summary of the stream, the answer is approximately precise (and the error is bounded, too). For a mathematically strict proof, see [3].

2.3 A GEOMETRIC APPROACH

Monitoring data streams in a distributed system is the focus of much research. A batch of new problems have arisen. The most important set of problems of them refer to the very high communication overhead that is required. The reason why this is happening is the centralized, naive algorithms that are in use. A novel geometric approach has been proposed by Sharfman et al.. Here, an arbitrary monitoring task can be split into a set of constraints applied locally on each of the streams. In this

approach, the constraints are used to locally filter out data increments that do not affect the monitoring outcome and as a result, unnecessary communication is avoided.

In brief, we have a distributed data stream as described above. We also have a monitoring query installed to the distributed stream. The challenge here is to reduce the communication of the nodes to an absolutely necessary level. This piece of work has two main algorithms presented: One fully-distributed (but a bit naive) and another coordinator-based, but truly elegant and effective. We name the algorithms protocols (actually, they *are* communication protocols) and we analyze them.

2.3.1 *Computational Model*

The computational model behind the protocols is based on a geometric approach and description of the problem. The problem in brief is the monitoring of a distributed stream where we check if any violation happens. It is modeled as follows:

Let $S = [s_1, s_2, \ldots, s_n]$ be a set of n data streams, monitored from a set of nodes $P = [p_1, p_2, \ldots, p_n]$ respectively. Each node p_i collects items from the stream corresponding to it. As a result, the node forms a d-dimensional vector $\vec{v_i}(t)$, t : time. This vector is called *local statistics vector* and each node has one. Additionally to the local statistics vector, each node has a positive weight [1] which can be changed over time. Let $\vec{v(t)} = \frac{\sum_{i=1}^{n} w_i \vec{v_i}(t)}{\sum_{i=1}^{n} w_i}$. $\vec{v(t)}$ is called the global statistics vector. Let $f : \mathbb{R}^d \to \mathbb{R}$ be an arbitrary function from d-dimensional vectors to reals. f is the monitored function. Now we want to continuously figure out if $f(\vec{v(t)}) > r$, where r : threshold value.

local statistics vector definition

global statistics vector definition

Except the local and global statistics vectors, more vectors are defined in the computational model. We have the last statistics vector, named $\vec{v_i'}$, consisted of the last m elements of the node p_i. We then have the estimate vector, denoted $\vec{e(t)}$, defined as $\vec{e(t)} = \frac{\sum_{i=1}^{n} w_i \vec{v_i'}}{\sum_{i=1}^{n} w_i}$. Intuitively the estimate vector is a local answer to the function f. One more vector, common in both protocols, is defined, the statistics delta vector. The statistics delta vector

estimate vector definition

1 The weight w_i assigned to the node p_i usually depends on the rate of data items coming from stream.

is denoted by $\vec{\Delta}v_i(t)$. This vector holds the difference between the current local statistics vector and the last statistics vector and it is defined as

$$\vec{\Delta}v_i(t) = v_i\vec{(t)} - \vec{v_i'}$$

.

The vectors we described so far are the same for both of the protocols. Now, we will define two more vectors, necessary to both protocols, but different in the way they are computed.

decentralized protocol drift vector

The first vector is the drift vector, denoted by $u_i\vec{(t)}$. In the decentralized protocol , the drift vector is a displacement of delta statistics vector $\vec{\Delta}v_i(t)$ and it is defined as

$$u_i\vec{(t)} = e\vec{(t)} + \vec{\Delta}v_i(t).$$

The coordinator-based protocol implements a whole mechanism for balancing local statistics vectors of some of the nodes. Hence, another vector is defined, called slack vector and denoted by $\vec{\delta}_i$. In the particular setting, there is one condition that should be satisfied for the protocol to run normally:

$$\boxed{\text{condition: } \sum_{i=1}^{n} \vec{\delta}_i = 0}$$

coord-based protocol drift vector

Consequently, we define the drift vector as

$$u_i\vec{(t)} = e\vec{(t)} + \vec{\Delta}v_i(t) + \frac{\vec{\delta}_i}{w_i} \quad .$$

2.3.2 *Geometric Interpretation*

Having in mind the theory described above, we will try to approach in a more "visual" way the main idea of [15]. Let us think of each node as an individual. As data arrives, each node checks its local constraint on its stream and verifies that it has not been violated. As long as no violation happens, no communication is needed between the nodes.

EXPLANATION: Each node collects data from its local stream (or set of streams). Then, the node forms the local statistics vector. According to the idea, the important node p_i [2] should have some knowledge of others nodes' collected data. This knowledge is extracted by the last statistics vector, that the nodes send to p_i [3]. The last statistics vector though, could be slightly dif-

2 The important node is the coordinator in the coord-based protocol, and every node in the decentralized protocol.

3 in decentralized protocol, each node sends its last statistics vector to every other node, while in coord-based protocol each node sends its last statistics vector to coordinator.

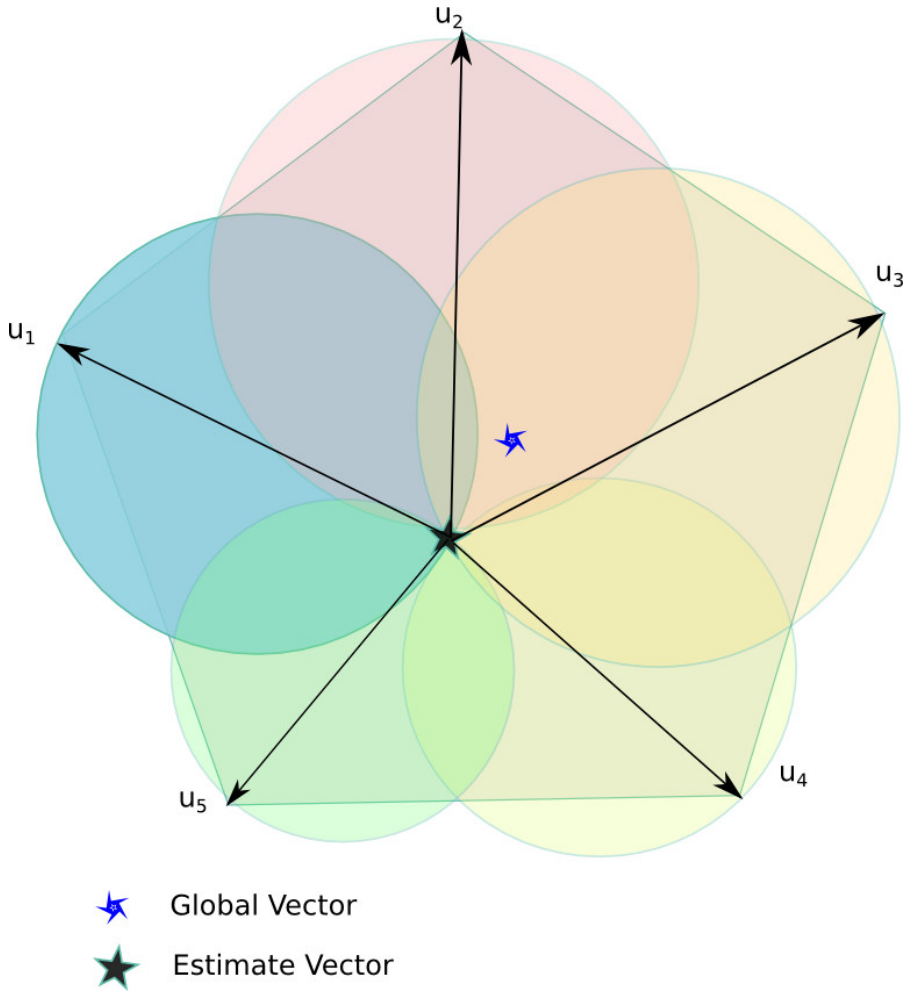


Figure 2: *Illustration of the computational model and the basic idea.*In this case, we have five nodes and each one of them constructs a ball. The grey-highlighted aera is the convex hull of the drift vectors and it is covered by the union of the balls. Even if estimate vector is a bit different than the real global statistics vector, it is covered by the convex hull. **Note:** The model is the same for the d-dimensional space.

ferent than the actual local statistics vector (due to the fact that the local statistics vector is updated every time a new update comes to the stream, while the last statistics vector is updated once in a while [4].

After collecting the last statistics vectors of all nodes, node p_i computes an estimate of all vectors. In our geometric approach, the real-time, real global statistics vector might be found somewhere else than it is estimated, but it is covered by the convex hull. This means that it is safe to assume that the global constraint is satisfied.

An important note here is that regardless of the protocol running, any node has the required information to construct its own ball at any time. For a more detailed description and mathematical proof, see [15].

2.3.3 *Local Constraints*

As this idea is the heart of the protocols and our simulator is focused on simulating these protocols, we will further analyze the local constraints each node applies to its stream(s). Let us imagine that the set of vectors that make the equation $[\vec{x}|f(\vec{x}) > r]$ true, are green, while the set of vectors that make the equation $[\vec{x}|f(\vec{x}) \leqslant r]$ are red.

By mapping the vectors to green and red, now the local constraint each node maintains, is to check whether the ball formed locally by the estimate vector $e(\vec{t})$ and the local statistics vector $u_i(\vec{t})$ is monochromatic or not.

This is a straightforward task. First the ball center and radius are computed, given by the formulas $c = \frac{e(\vec{t})+u_i(\vec{t})}{2}$ and $r = \|\frac{e(\vec{t})+u_i(\vec{t})}{2}\|$ respectively. Hence, each node checks locally if its ball is monochromatic or not. If it is, there is no reason to communicate and overload the network. More generally, each node does the same. If the ball is monochromatic, then the set of vectors which form the union of the balls is monochromatic, too. This means that the global statistics vector also is contained in the convex hull that the balls contain. Consequently, both the global statistics vector and the estimate vector are on the same side of the threshold [5].

4 last statistics vector is updated pretty frequently, but still, it could be a bit outdated. Still it cannot be totally different than the local statistics vector.

5 This means the estimate vector is correct.

2.3.4 *Decentralized Protocol*

We elaborate the concept of the decentralized protocol in detail. The decentralized algorithm is oriented to fully distributed systems. Given the above, the concept to every protocol is that each node should be able to construct its own ball. In order for this to happen, since no coordinator or lead-node exists, every node should send its statistics (and ask for the others' statistics) to every other. Therefore, we do not really have a communication reduction here. Even naive, this protocol implements the fundamentals which is to not send any data if not necessary.

Specifically, a random node n_i obeys to the following two stages:

1. **initialization stage:**
   - broadcast a message containing the initial statistics vector,
   - update $\vec{v_i'}$ to hold the initial statistics vector,
   - after receiving every similar message from the other nodes, calculate the estimate vector $e_i(\vec{t})$.
2. **processing stage:**
   - Upon arrival of new data on the local stream:
     - recalculate $v_i(\vec{t})$
     - recalculate $u_i(\vec{t})$
     - check if the ball $(B(e(\vec{t}), u_i(\vec{t})))$, remains monochromatic. If not, go to initialization phase but broadcast $< i, v_i(\vec{t}) >$ instead of initial statistics vector.
   - Upon receipt of new message $< j, v_j(\vec{t}) >$:
     - update $\vec{v_j'}$ to hold $v_j(\vec{t})$
     - recalculate $e(\vec{t})$
     - check the ball for monochromicity. If ball is not monochromatic, then broadcast $< i, v_i(\vec{t}) >$ and update $\vec{v_i'}$ to hold $v_i(\vec{t})$.

The **pseudocode** that implements the protocol, is presented below:

The init phase is presented:

Algorithm 2.3.1: INITIALIZATION(n_i)

comment: Initialization phase of distributed protocol.

$msg \leftarrow$ BROADCAST($\vec{v_0}$)
UPDATE($\vec{v_i}, \vec{v_0}$)
$newMsg \leftarrow$ RECEIPT($-$)
while $newMsg \neq 0$
do { $list \leftarrow$ ADD($newMsg$); $newMsg \leftarrow$ RECEIPT($-$) }
$e_i(\vec{t}) \leftarrow$ CALC-ESTIMATE($list$)

Then, the processing stage at a random node p_i is presented:

Algorithm 2.3.2: LOCALARRIVAL($arrival$)

comment: Actions taken in a local arrival of new data.

$v_i(\vec{t}) \leftarrow$ RECALCLOCAL($arrival$)
$u_i(\vec{t}) \leftarrow$ RECALCDRIFT($-$)
if $ball$ is not monochromatic:
then { BROADCAST($(i, v_i(\vec{t}))$); UPDATE($\vec{v_i'}, v_i(\vec{t})$) }

Finally, the processing stage upon a new message receipt is presented:

Algorithm 2.3.3: MSGARRIVAL(msg)

comment: Actions taken in a new message arrival.

UPDATE($\vec{v_j'}, v_j(\vec{t})$)
RECALCESTIMATE($e(\vec{t})$)
if $ball$ is not monochromatic:
then { **comment:** i is the node's id.BROADCAST($i, v_i(\vec{t})$); UPDATE($\vec{v_i'}, v_i(\vec{t})$) }

2.3.5 *Coordinator-Based Protocol*

This protocol is more complex than the previous, naive one. It is necessary to dive into the details of this protocol, because it is effective in terms of communication. Another interesting aspect is that if something is not going as expected (i. e.a node's ball is not monochromatic), then it is able to gussy things up [6].

The main idea here is the existence of a special node, named coordinator which is responsible to compute the estimate vector and to send it to the other nodes.

coordinator

A new idea is also presented here, the idea of *balancing process*. According to the idea, if a node's ball is not monochromatic, then the coordinator is able to decide and choose a random node to balance things. The idea is that small "movings" of another node (or more), can restore the balance back to where it used to be. To implement this, we use a "helpful" vector, the slack vector. For the coordinator to ensure that the convexity property of the drift vectors is maintained, the sum of all slack vectors should bee zero.

When coordinator-based algorithms starts, first of all, all nodes send their initial statistics vector to the coordinator. If a local constraint is violated, then the coordinator sends a message containing its current drift vector and its current statistics vector. At first, the coordinator tries to resolve the constraint violation, executing a balancing process.

BALANCING PROCESS: During balancing process, the coordinator establishes a group of nodes chosen by coordinator, form a monochromatic ball with the estimate vector, such that the balancing vector creates a monochromatic ball with the estimate. Indeed, the balancing vector is $\vec{b} = \frac{\sum_{p_i \in P'} w_i u_i \vec{(t)}}{\sum_{p_i \in P'} w_i}$.

balancing vector definition

The details about balancing process is as follows: When a node p_i notifies the coordinator, that means that its local constraint has been violated. Therefore, its drift vector and its current statistics vector are appended to the message. If the ball $B(e(\vec{t}), \vec{b})$ is now monochromatic, then no more communication is needed. On the contrary, if there is some local violation, then more nodes are notified to participate in the balancing process.

The adjustment calculated to the slack vector is as follows:

6 This is not always possible, but small adjustments in a set of nodes can fix things up.

SIGNAL	FROM	TO	DESCRIPTION
$< \text{INIT}, \vec{v_i} >$	nodes	coord	Report initial statistics vector
REQ	coord	nodes	Request statistics and drift vector (balancing process)
$< \text{REP}, \vec{v_i}, \vec{u_i} >$	nodes	coord	Report local statistics vector when a local constraint has been violated.
$< \text{ADJ} - \text{SLK}, \Delta\vec{\delta}_i >$	coord	node	Report slack vector
$< \text{NEW} - \text{EST}, \vec{e} >$	coord	node	Report new estimate vector

Table 1: Necessary messages for coordinator-based protocol

$$\Delta\vec{\delta} = w_i\vec{b} - w_i u_i(t)$$

Now, every node adds the slack vector adjustment to its own slack vector: $\vec{\delta_i} = \vec{\delta_i} + \Delta\vec{\delta}$

As it is easily implied, after a successful balancing process procedure, we have $\sum_{i=1}^{n} \vec{\delta_i} = \vec{0}$. The bottom line here is that the atomic drift vectors have accepted small adjustments and due to the fact that the drift vector is computed with the help of the slack vector Section 2.3.1, the union of balls is still monochromatic.

balancing process failure

In the case that the balancing process has failed [7] , then it is unavoidable to recalculate everything from scratch. Thereafter, all nodes send their (most recent) statistics vectors and coordinator computes a new global estimate and broadcasts it to nodes. Moreover, all slack vectors are set to $\vec{0}$.

messages definition

A set of messages is defined for the protocol to run Table 1:

PROTOCOL'S INDIVIDUAL ALGORITHMS The coordinator-based protocol has four main points of view:

1. initialization,
2. processing stage at an ordinary node p_i,
3. processing stage at the coordinator and

7 A balancing process failure means that at last, every node has been asked for its statistics and drift vectors and it has sent them, but still no smoothing could be figured out.

4. balancing process at the coordinator.

INITIALIZATION Initialization is the first stage of the protocol. Both ordinary nodes and coordinator have it.

An "ordinary" node starts to review and collect data from its local stream [8]. At some point, a node decides that it has sufficient data and sends it to the coordinator. It sends a <INIT> signal with its initial statistics vector and sets its slacks vector to $\vec{0}$. Pseudo-code that describes the above, is presented:

Algorithm 2.3.4: NODEINIT(initStats)

---

comment: Node Initialization.

$msg \leftarrow$ CREATEMSG(initStats)
SEND(INIT, $\vec{v_0}$)
$\vec{v'} \leftarrow \vec{v_0}$
$\vec{\delta_i} \leftarrow \vec{0}$

Coordinator waits for messages to come. At the same time, the coordinator implements the "ordinary" node's protocol, as it collects data, too. At the time that every node has sent its initial vector, the coordinator calculates the estimate vector and broadcasts it to the other nodes.

Algorithm 2.3.5: COORDINIT(msg)

---

comment: Coordinator Initialization.

while msg is coming
  do $\{$ msgList $\leftarrow$ ADD(msg)
CALCESTIMATE(msgList)
BROADCAST(NEW − EST, $\vec{e}$)

---

8 or streams. A node may monitor more than one stream.

RANDOM NODE'S PROCESSING STAGE: Here the things start to become more interesting. We dive into an ordinary node's processing stage. The concept here is that the following algorithms are running in parallel. Each algorithm presents a functionality that should be implemented independently of the other algorithms. In this stage, events are happening simultaneously. The four different actions are:

1. new data arrival,
2. REQ signal received,
3. NEW-EST signal received and
4. ADJ-SLK signal received.

During arrival of new data from local stream, a node has to process the new data and finally to decide if sending data should take place. The decision is taken after ball checking, as described in Section 2.3.3. In other words, if a local constraint happens, then the node sends its statistics vector to the coordinator and waits for the appropriate answer. If coordinator manages to balance the situation, then the node receives and ADJ-SLK signal. Else, a NEW-EST signal is coming and node resends its statistics vector, etc.

Algorithm 2.3.6: NODELOCALARRIVAL(data)

comment: Actions taken in node, at a local arrival of new data.

$v_i(\vec{t}) \leftarrow$ RECALCLOCAL(data)
$u_i(\vec{t}) \leftarrow$ RECALCDRIFT(−)
if ball is not monochromatic:
then { SEND(REP, $(i, v_i(\vec{t}))$, coord)
WAITMSG(NEW − EST, ADJ − SLK)

During a receipt of a REQ message [9], the node sends a REP message to the coordinator and waits for a NEW-EST or an ADJ-SLK message. A useful observation here [10] is that the following

9 for messages meaning, check Table 1.
10 This observation is useful for implementing the algorithms. With extensive use of inheritance, the implementation becomes straightforward.

algorithm is exactly the same as the previous one, except for constraints checking and the drift vector sending.

Algorithm 2.3.7: REQ-RECEIPT(REQ)

---

comment: Actions taken in case of a REQ receipt.

$msg \leftarrow$ CREATEMSG(REP, $v_i(\vec{t}), u_i(\vec{t})$)
SEND($msg, coord$)
WAITMSG($NEW-EST, ADJ-SLK$)

During a NEW-EST receipt, the node updates its estimate vector and nullifies its slack vector. Everything is reseted and ready to start from the beginning.

Algorithm 2.3.8: NEW-EST-RECEIPT($NEW-EST$)

---

comment: Actions taken in case of a NEW-EST receipt.

UPDATE($e(\vec{t}), NEW-EST$)
$\vec{v'} \leftarrow \vec{v}$
$\vec{\delta} \leftarrow 0$

Finally, when an ADJ-SLK message is received, the node just adds the value that has come to its slack vector.

Algorithm 2.3.9: ADJ-SLK-RECEIPT($ADJ-SLK$)

---

comment: Actions taken in case of ADJ-SLK receipt.

$\vec{\delta} \leftarrow \vec{\delta} + (ADJ-SLK).\vec{\Delta\delta}$

COORDINATOR PROCESSING STAGE: Now let us examine the steps the coordinator follows. An important thing to remember is that coordinator actually **is** a node too. Hence, no special description about local arrival is in need, as the steps the coordinator follows are the same as the other nodes'. Here we will examine how the coordinator manipulates the messages arriving from nodes. Coordinator has two possible scenarios, which are described below:

1. new data arrival on local stream and
2. REP message receipt.

During the arrival of new data the coordinator follows the algorithm below:

Algorithm 2.3.10: CoordLocalArrival($data$)

---

comment: Actions taken in coordinator,

comment: at a local arrival of new data.

NodeLocalArrival($data$)
if $ball$ is not monochromatic:

then
- **comment:** P' is the balancing group.
- $P' \leftarrow \text{ADD}(1, v_l\vec{(t)}, u_l\vec{(t)})$
- BP-Init(P')

During receipt of a REP message from a node p_i, a balancing process is initiated. This node with its statistics vectors is added to the balancing group.

Algorithm 2.3.11: REP-Receipt(REP)

---

comment: Actions taken in coordinator upon a REP receipt.

$P' \leftarrow \text{ADD}((1, v_i\vec{(t)}, u_i\vec{(t)}), (\text{REP.i}, \text{REP.}v_i\vec{(t)}, \text{REP.}u_i\vec{(t)}))$
BP-Init(P')

BALANCING PROCESS: Finally, we consider the balancing process. Balancing process can be seen as an independent kind of protocol/algorithm. It is on the heart of the geometric approach.

Balancing Process also consists of three main steps. We appose the individual algorithms below:

Algorithm 2.3.12: BALANCINGPROCESS(−)

---

comment: Balancing Process.

CALCULATE($\vec{b}$)
if $ball$ is not monochromatic:
  then FAILEDBALANCE(−)

  else SUCCESSFULBALANCE(−)

Below, a balancing process in action is presented.

Algorithm 2.3.13: SUCCESSFULBALANCE(−)

---

comment: P′ is the balancing group. i is an item.

for each $i \in \mathcal{P}'$
  do $\begin{cases} \text{CALCSLACK}(\Delta\vec{\delta}_i) \\ \text{SEND}(p_i, ADJ - SLK) \end{cases}$
BP-EXIT(−)

Here, if the balancing process has failed, follow the relevant algorithm is presented:

Algorithm 2.3.14: FAILEDBALANCE(−)

comment: P is the set of nodes not contained into balancing group

$node \leftarrow$ SELECT(P)
SEND($node$, REQ)
while $node \in \mathcal{P}'$
do
- **if** $p_i.msg$ is REQ:
  - **then**
    - ADD(P', p_i)
    - GOTO(BalancingProcess)
  - **else**
    - CALCNEWEST(−)
    - BROADCAST($NEW-EST$)
    - BP-EXIT(−)
- $node \leftarrow$ SELECT(P)

2.4 PYTHON PROGRAMMING LANGUAGE

Python is a powerful programming language. Python was conceived in late 1980's by Guido van Rossum and its name is inspired by the famous british surreal comedy group Monty Python. Python is a script, object-oriented language. It is well-written, with a continuously growing supporting community. It is a general purpose language with many uses (web, simulations, applications are among them). Furthermore, python is a complete language, well documented, and tested in large project with great success. Another important fact about python is the high-level libraries that are being developed as well as the strict rules for developing them.

The main principles (and core philosophy) of the language are described perfectly below:

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Readability counts.

Comparing to other languages, python has the following advantages:

- *Shorter code:* Last years many comparisons [12] have been made between different programming languages. It turns out that the same piece of work needs six times less code to be implemented in python than in c. Also, we need to write as much as twice source code to implement the same functionality in java, than doing it in python.

- *Easy to read:* Python is a really elegant programming language. Writing code in python is a comprehensive, plain procedure. It is the exactly opposite of c++ and its messy templates, etc. It is an easy-to-read format. A novice programmer is usually able to read and understand python pretty easy, comparing to c/c++ code.

- *Easy to learn:* Python has an easy syntax. Everything is simple and plain. This is another reason why python is getting more and more popular amongst programmers. A pleasant consequence is that maintainabilty of source code is facilitated.

- *Extensive:* Python supports modules and packages. Therefore, program modularity and code reuse is encouraged.

- *Built-in functions:* Python implements a lot of useful functions and data types (such as dictionaries and lists) natively. As a result, there are many good theoritical (at least) reasons to trust these functions and data types and use it without doubts about their implementation.

To sum up, python is a well-written, clear, and powerful tool with a bright future.

2.5 SIMPY

Simulation in Python (SimPy) is an object-oriented, process-based Discrete Event Simulation (DES) language based on python and released under GNU GPL. It provides very solid and robust libraries for DES (discrete event simulation). It provides all elements of DES in a very elegant way. It includes components as processes, time, events, resources and it describes all the above in a complete way.Moreover, *SimPy* comes with data collection capabilities, real-time monitoring capabilities, GUI and plotting packages. Besides, it's easy to be interfaced to other packages as plotting, statistics, spreadsheets, data bases etc.

Simpy is being used for:

- modeling/simulation of epidemics
- traffic and network simulation
- industrial engineering
- computer hardware performance simulation
- workflow

It is a well-tested, well-documented and clean simulation implementation. Its robustness and clear components was a critical contribution to our selection for python and simpy.

Part II

PROBLEM AND APPROACH

Problem statement follows. After the problem, the main points in simulator's approach and implementation are exhibited as well.

3

PROBLEM STATEMENT

For every complex problem
there is an answer that is
clear, simple, and wrong.

— H. L. Mencken

3.1 GENERAL SIMULATION USES

Simulations appeared to a great extend last century. There are many different scientific sections where simulations took place and developed, mostly independently. At first simulations used to test physical concepts, but development of computing science revolutionized the concept of simulation. In our time, people are really familiar with simulations. They meet them from home games (such as flight simulator) to heavy industry applications and research.

simulation definition

Simulation is the imitation of the operation of real-world process or system, over time [**?**]. For a simulation to take place, first of all, a model is needed. A good model should represent physical characteristics to a detail, as well as behaviors that need to be tested. A model need not to describe every detail and parameter. On the contrary, it should be pretty abstract and describe *key* characteristics and concepts. The model represents the system (natural, human, abstract, imaginary) while the simulation represents the set of actions and operations that take place on system over time.

Simulation is heavy used. Some examples are simulations built for testing, training, education and video games. There are simulations that involve huge numbers of parameters and details. An example is a set of flight simulators, where potential pilots are trained to almost real-time conditions. Simulation concepts and applications are developing more and more. There are really detailed models developed for monitoring and testing durability and resistance of physical environments. An example is the simulations developed for volcanic activity and eruptions. Another one is the simulation of a natural, well-bounded physical space and the plant growing on it.

3.1.1 *Simulation purpose*

The main object of a simulation is to provide a tool for explaining and foreseeing systems' development over time. At many times it is not easy to test and install in a straightforward way a novel idea to a system that it is barely known. Let us think for example a totally new, novel way of airplane navigation. It is not wise to just implement and install the system to an airplane and let it fly the next day, since it is not heavily tested and tried to a big set of different, very demanding environments and conditions. If despite all of the above, someone decides to take the risk, then it is very possible for the plain to dramatically crash. In other words it is very important and useful to make an extensive use of simulation, because in a demanding simulation it is very easy to note omissions or oversights.

A batch of other, maybe less important problems may come up, such as big delays, undesired workload, sudden network overhead, communication problems, etc., depending on the problem and the model that is simulated. Another important fact of simulation importance, is the simulation of a time-consuming system, or more generally, a system that evolves really slow over time. In this case, when a novel approach is intended to be used, a simulation is necessary. A simulation can simulate even the time and hence there is the possibility to simulate months or years in a few minutes or hours.

Concluding, simulations are almost necessary to every big task, application, idea or algorithm and system that is going to be used in practice.

3.2 COMMUNICATION PROTOCOL SIMULATIONS

A lot of research has been done on communication protocols. Many algorithms have been invented and applied with different results each. The existence of protocol simulations is really important, because communication is expensive in terms of money and energy. Message exchange rate is usually frenetic and network and computer sources are crucial. Physical constraints are introduced in the particular scientific section and due to large scaling, there is a need for strict, detailed simulations.

Much research has been done recently about distributed data streams and data mining on them. Many algorithms and proto-

cols have been proposed. They usually manage to manipulate pretty efficiently specific challenges, but not many general purpose experiments have been done by now.

All algorithms present some basic efficiency and performance plots, but the experiments that have been done are simple (to naive). Parameters are very specific (and just a few). Datasets are of small variety and of specific format. Most important of all, to our knowledge, no real-time, real data experiments have been done.

More generally, we noted the absence of a good simulator that could handle and execute a variety of experiments with a variety to system parameters, variables and data sets. We would like to have a simulator, robust but also flexible. Such a simulator, it would also include a good, well documented Application Programming Interface (API)

problem definition

OUR PROBLEM: The overall problem we deal with, is as follows: Let a distributed system consisted of N nodes. Let a distributed stream $S = s_1, s_2, \ldots, s_m$ to be monitored by the nodes distributed system (see Section 2.1.2). In other words, each node n_i monitors one or more streams (i.e. $n_i, n_j, \ldots$). Some protocols have been proposed in order to deal with communication issues between the nodes. The problem is to evaluate these protocols in terms of effectiveness, quality and correctness. The main protocols we are interested in are described in Section 2.3. Moreover, we want at the same time to make experiments and check the output and effectiveness [1] of the protocols, if the nodes communicate with sketches Section 2.2.1 instead of raw data, extracted from local sub-streams.

Given the fact that installing the protocols to a real node-system as the one described above is risky and expensive, we need some detailed, good simulations of such a system. The system should be described with great detail and precise requirements. It also should implement many environment variables and parameters. It should also include a random factor in some of them (such as network delays, a missing message now and then). Finally, it should also deal with real time, real environments and use cases that a protocol in theory never deals with. An example is a case from coordinator based protocol Section 2.3.5 where the coordinator asks for a message from node

1 criteria for this approximation can be extracted by running the protocols for a long time and measure network metrics and statistics, such as total messages traveled through network, size of messages, etc.

n_i and at the same time the node n_i sends by itself this message to the coordinator.

To conclude, the problem is reduced to the following one: Build a simulator, focused on the implementation of the protocols that described above (Section 2.3), that implements DES and using sketches in messages. The simulator should be very flexible, in order to simulate (or add) any environment or technical variable the user may need. It has to consist of many independent parts, in order for the user to use what exactly he may need. It also should have have a well designed API and be well documented, so as to be absolutely portable and extensible.

4

RELATED WORK

There is a lot of independent, focused on particular protocols and parameters software for protocol simulation. Not much has been done for the implementation of a good, event-driven and general purpose, simulator. There is always need of libraries which help the experiments in a solid way. Variance is experimented as well.

Basic experiments and work have been done by Sharfman et al. where the efficiency of the algorithms is presented. Still, variety in nodes and network parameters is missing. For example, size of messages or network speed are absent. Special cases are not confronted, too. An example is the behavior of algorithm in a very easy-to-occur anomaly in real time exhibition.

A well-written set of libraries was developed in Technical University of Crete by Samoladas et al. This project has been written in C++ and is focused on the very same algorithms that this thesis does, too. The approach of simulation was different than the one of this thesis. The libraries have been written from time's perspective. In other words, a simulator using these libraries runs a "time-centric" simulation rather than discrete event simulations.

Another work worth mentioning, in terms of libraries is tortuga ([11]). Tortuga is a software framework for DES written in Java. A tortuga simulation can be written either as interacting processes or as scheduled events. Tortuga in theory is pretty similar to SimPy but to our knowledge, SimPy is more robust and extensible.

5

OUR APPROACH

The development of a DES tool focused on protocols [15], which meets the requirements stated on Chapter 3 was always high in our tasks list. To address the environment and monitoring needs, we have developed a simulator, named Elita [1] which is the subject of this thesis. Elita allows the user to inspect and use effectively each one of the basic modules of the existed code. This is done by having fully independent modules. System consists of several different modules that implement environmental and structural detail in several levels. User has just to use what he wants so as to run a simulation, according to her needs.

5.1 HIGH-LEVEL DESCRIPTION OF THE PROBLEM

First of all, we define the problem as follows, trying to re-state it and make it more clear. Let us define the problem as follows:

abstract problem definition

Given as input by user:

- **a monitoring query Q,**
- **a set of parameters P,**
- **an architecture and**
- **an algorithm among the given choices,**

run a simulation with the characteristics as described above.

5.1.1 *Analytical description of problem:*

We aim to develop a simulator that simulates the communication (of a peculiar type) of a distributed system consisting of nodes. The problem divides into many sub-problems.

1 The name is an inspiration given by Elita One character of Transformers (it is an old animation). Elita One is a devoted Autobot and powerful warrior, fearless in the face of the enemy, but compassionate to those who need her help. She is sometimes the wielder of a great special power: "the ability to stop time" (which is pretty much what a simulator is also capable of).

A sub-problem is the type of nodes. What are the characteristics of the nodes (CPU, RAM, network channels, etc.). Are all the nodes identical, or different? Do they have different values in some of their characteristics?

Another sub-problem is the type of system we want to simulate. How many nodes are there in the system? Which protocol do they use so as to communicate? P2p, client-server, a hybrid system between these two, something else? Also, how reliable is the network?

A third sub-problem is the query (which is a *monitoring query* - see Section 2.1.3) that the user wants to "install" to the system.

A fourth sub-problem is the kind of data that the nodes gather.

We try to answer to all problems above and as a consequence, to build the software that simulates all these parameters in a system, using the algorithms that are described in [15] in a satisfying way.

5.1.2 *Description of software system*

First of all, our simulator should include parameters that describe all the problems above. We define as components all these problems and using this as core, we build our system. As a result, we conclude to the following components: We have user-interface component in which the user pass all the parameters he is interested in. After that, these parameters form the nodes which number is set also by user. User also selects the architecture of the simulated system and its general characteristics.

All these parameters configure the simulation form, that also takes as input the nodes. Then, the simulation simulates the system and saves the results for further processing. A graphical representation is shown in Figure 3.

5.2 LEVELS OF DESIGN

Our simulator has been developed according to the strict programming principals that are described in [12]. Hence, many description levels of the system have been created. These levels present the software system from the most general, abstract one to the most detailed one. Here, three levels of design have been used. They all are high-levels. At first, we organize the

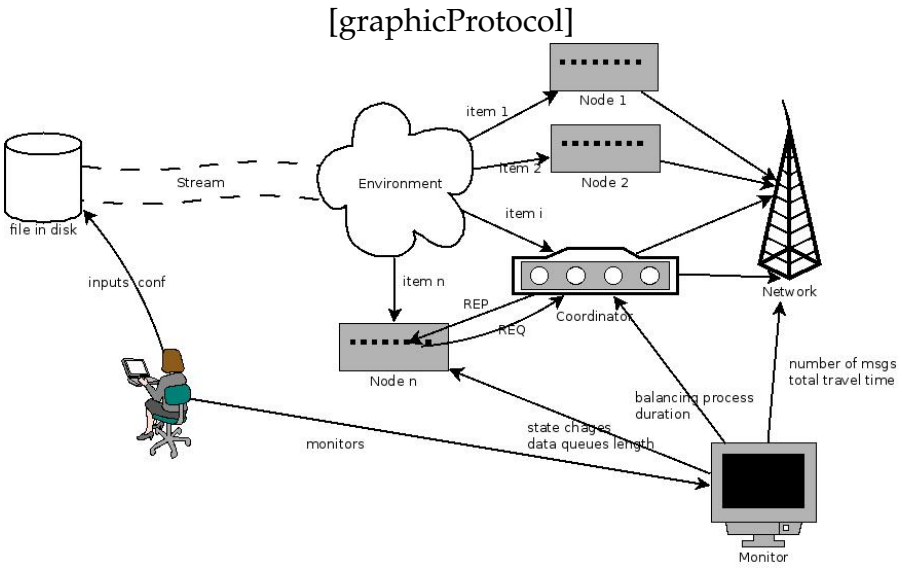


Figure 3: Visualization of protocol's operation

system into sub-systems. Secondly, sub-systems are further divided into classes, and finally the classes are divided into routines and data.

5.3 ELITA'S ARCHITECTURE

Here, we organize the system thinking through higher level combinations of classes, such as subsystems or packages.

We recognize five sub-systems/packages:

1. User interface sub-system
2. Node configuration sub-system
3. Simulation sub-system
4. Results sub-system
5. Data storage sub-system

The relationships between the sub-systems above is described in the Figure 4. We made a big effort in planning the communication rules between sub-systems, so as to simplify as much as possible the interactions between sub-systems. A good feature of this design is the directed graph representation. Representing each sub-system as a graph node and each communication arrow as a directed edge, we make the observation that our graph is a Directed Acyclic Graph. This is important because

when software components and their communication are represented in this way, and no cycles are observed, then the complexity is as low as possible. This happens because we avoid bad calls. A bad call example follows: a class B instance calls a class A's instance and this particular class A instance calls the previous class B's instance.

An important feature to observe here is that it is pretty straightforward for the user to run a simulation, without understand the whole simulator software. A user has only to give as input the environment's and system's parameters and wait for the simulation to finish. After that, the user just gathers the results. In this software description stage, an activity diagram is given for further comprehension of system's functionality in Figure 5

5.4 MAIN PACKAGES ANALYSIS

Our approach is directly related to the main packages. Hence, we bother to give a move detailed explanation of their utilities and uses. It is once more noted that they can be used independently.

To begin with, we have the userInterface package. The userInterface is responsible for reading configuration files (*.conf) files and prepare their data for further use. It also is its responsibility to handle streams. Due to the fact that our simulator should manipulate streams, this sub-system can handle both binary and text stream. Of course, if user wants to give a different stream than the default, then he should define stream's format. Other than that, user interface is ready to manipulate any stream and pass any parameters to the environment and the simulation.

NodeConfiguration package is responsible for creating a node and configure it. Here, a lot of individual jobs are made. The NodeConfiguration sub-system simulates nodes' hardware (such as CPU, RAM) and network's physical characteristics (such as overall bandwidth, channel's bandwidth etc.) and it is parameterized at will. All the configuration parameters and variables are already set from userInterface sub-system. Here the protocol is also set. Depending on configuration file, we can fully customize each node. Thus we can have a totally unique nodes, with unique characteristics each (RAM, CPU, even bandwidth). Of course, all these parameters are used in the simulation.

Simulation sub-system is responsible for everything related to simulation. It inputs from user (userInterface) as well as the nodes. It simulates the environment in big detail. It is responsible for the data incoming rating, the network, etc. This subsystem is the heart of simulation. Among others, here is the place where simulations run.

Results is the sub-system responsible for results manipulation and presentation This sub-system collects all variables responsible for the results. It then exports statistics, means, usage ratings, load ratings etc. When everything is collected and processed, this sub-system sends the results for storage.

Finally, DataStorage is responsible for storing the data (usually in files) and makes them available to the user. Note that the user interacts only with this part of software (except for the configuration files that gives as input to the program).

Before moving to implementation explanation and details we emphasize some key points on our approach. These are:

- Flexibility,
- Robustness,
- Well designed API and documentation.

5.4.1 *Flexibility*

By having independent sub-systems, we manage to maintain a fully flexible, fully-extensive piece of software. All sub-systems need minimal or no communication between them. Whereas minimal communication is required, this is well-defined by the API of the sub-system. Consequently, by using (or even implementing his own) API, the user can use the whole sub-system to his own simulation. Inspecting the same problem by a different perspective, if the user wants to use the whole simulator but for a sub-system, he can implement this particular subsystem's API and embody it to his own code. The rest of the packages are designed to collaborate with the new sub-system, given that the abstract API of the replaced one is implemented. In this way, we manage to maintain flexibility to the greatest extend.

5.4.2 *Robustness*

The simulator is pretty robust. We strengthen Elita's robustness in three ways:

1. technical inferiority and completeness
2. good API
3. many but solid packages

5.4.2.1 *Technical inferiority and completeness*

We use many corporation techniques in our software as described in [12] and [Pragmatic programmer]. We also implement heavy unit-testing on every package, module, class and function in our program. We include both simple and complex unit tests. We test each function and each module independently, but also in combination with others. We grant a sufficient error-message number and we follow a strict error policy [2]. As a result, a lot of assertions and use cases have been implemented. To further clarify this part, we present two examples:

PROTOCOL PROBLEM: Let us think of coordinator based protocol as described in Chapter 2. Let us think the special case of a lost message from coordinator to a node. That is a case that the protocol does not predict, but is is not a simulation error. The system will continue running in such a case.

CODE PROBLEM: Let us suppose the unlikely event of a null pointer, or of a data type that contains wrong data. This is an error that is not supposed to happen, and if the simulator meets such a bug, then it exits immediately.

Finally, in order to give more qualities to the source code implementing the simulator, we try to have good encapsulation, no global data, and very strict coupling criteria [12].

2 This means that if an error is encountered, then the program is terminated immediately. We prefer this strategy rather than "continue execution even with errors" strategy. Even if our strategy is more annoying to the user, we preferred it in our implementation because we need accurate measurements and precision, over a finished simulation. After all, we interested in precise results and performance.

5.4.2.2 *API Design*

A good API, minimal but also abstract in order for someone else than the usual user to implement it, helps the robustness. It clarifies the individual functions and sub-systems and helps the user to understand it. Moreover, by implementing the API correctly, the simulator's modules will work, no matter what the rest of third-party source code qualities are [3].

5.4.2.3 *Solid packages*

This part is strongly connected to Section 5.4.1. As mentioned above, simulator is designed to have several, discrete parts. Each one performs a well-defined, discrete from others, function. From this point, further split is not possible. This means that the modules that consist a subsystem can not be used separately For instance, the coordinator module which is a module that implements coordinator, can not be used outside its sub-system. There are several dependencies, inheritance issues, etc.

There are though a few modules, that can be used outside their packages but they are concerning to the simulation function exclusively. The network.py and environment.py modules are totally independent. Extra attention and effort have been given to these modules, mainly because they could be useful by their own in many other simulation implementations. Besides, network and environment are core elements in every simulation of this type. Same rules apply to some of the protocols too, but the external usage is not recommended.

Another characteristic that enforces robustness is that implementation details are encapsulated [12]. In addition to encapsulation, there is as much information hiding as the design allows. The concept of "black boxes" comes from information hiding. Python does not encourage such concepts (there is no formal private concept in python) but still, the idea is that the user who wants to heavily meddle with the simulation parts in order to create her own, fully-customized simulation, should not trouble herself with all the modules. Instead, all she has to do is to implement the API (and thus avoid the dragons ahead! [2])

hiding secrets

3 theoritically, at least.

5.4.3 *Documentation*

Last but not of least importance, is the documentation. Even if documentation should not be considered as a special feature, to our experience, even the most complete libraries and project lack documentation. The absence of good, full documentation leads to API misunderstandings as well as code misunderstandings. Our approximation to documentation is as follows:

1. API documentation
2. in-line documentation

API DOCUMENTATION API should be documented very well. No space for misunderstandings should exist. The aim as well as the parameters should be clearly discrete and described in detail. Returns should be described as well. If there are any constraints, they should be included, too. This approximation is specially designed for our simulator, because it is figured out that even a small misunderstanding can lead to wrong results.

INLINE DOCUMENTATION From our high-level schema, as described in Figure 4, we break it down to classes. We analyze the variables of classes as well as the API of each one. After the API documentation (see paragraph above) we further describe every function and variable. We build documentation for each function the same way we do it for API.

Finally, before ever writing a single line of code, we design an algorithm that implements the function's desired functionality in abstract, high-level, but detailed pseudocode. Then, after each pseudocode's line, we write the code which implements it. With this methodology, we clearly reveal the small but hidden details in source code and enhance readability.

DOCUMENTATION EXAMPLE An example of inline documentation is given below. A random function has been chosen for exhibition methods.

Listing 1: Documentation example

```
def findRecipientNode(self, message):
    """ finds the recipient node searching from message
        recipient
    and returns the recipient node.
```

```
    @param message (Message)is an incoming message, that
        contains
    information for the recipient.
    @return the recipient. If no recipient has been found,
       print an error
    message (recipient will contain a <None> value.
    """
    # init the recp to nothing
    self.recp = None
    # and search the node list for the recipient
    for node in self.NodeList:
        # if recipient exists and is found:
        if self.recp == node.name:
            # assign it to recp and return
            self.recp = node
            return self.recp
    # else, print an error message
    print 'No such recipient exists!'
```

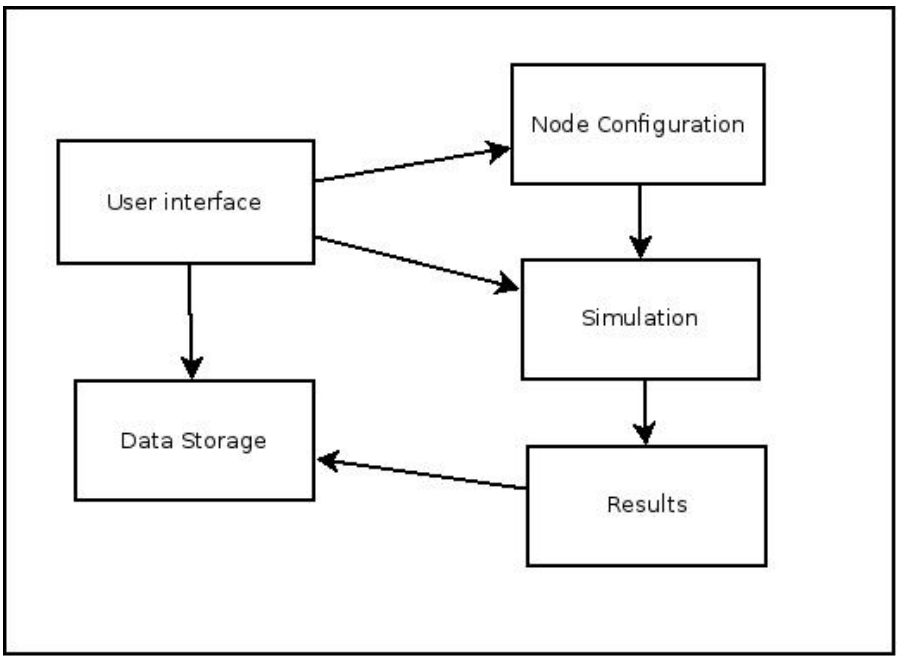


Figure 4: Top level simulator design. The main, (idependent) packages of simulator are distinguished. We observe no cyclic paths in packages' communication.

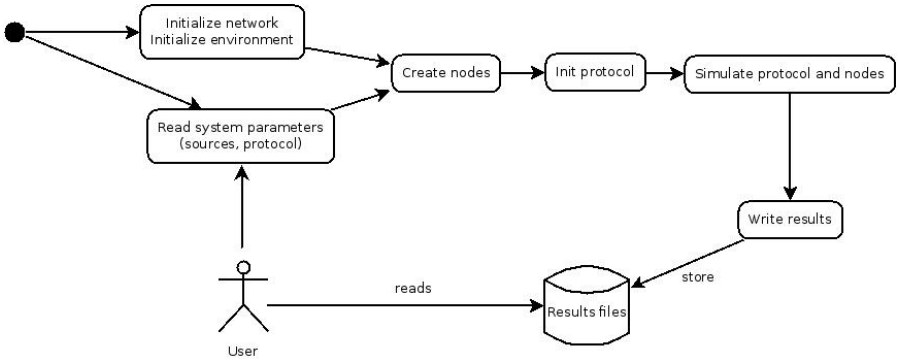


Figure 5: Activity diagram for the simulator.

6 IMPLEMENTATION

What I wanted to say, I finally get in, is that I've a set of instructions at home which open up great realms for the improvement of technical writing. They begin, "Assembly of Japanese bicycle require great peace of mind."

— Robert M. Pirsig [14]

6.1 LEVELS DESIGN

Continuing from Chapter 5, we further analyze the top level design as seen in Figure 4 to the sub-packages and modules that consist each system. This top-down approximation has the advantage of minimizing complexity when viewing the system. Each new level of design reveals more content than the previous one.

In Figure 6 more system pieces are revealed and many concepts described in [15] become visible in the design. Here, we observe that the userinterface package consists of NodeFeatures, where the hardware features of the node are placed, ParamsDefinition, where parameters relevant to simulation (such as nodes number, streams number, threshold, etc) are set, the QueryDefinition and ArchDefinition (arhcitecture definition, it concerns the algorithm/protocol used for communication between the nodes). Into the node the protocol and all the basic piece for a protocol to run are implemented and included.

6.1.1 *Classes and Modules*

The UML diagram of classes that consist API are presented in Figure 7. We insist on the analysis of the UML diagram as the heart of the system is revealed. The main module is the Simulation module. A simulation is set there. The user writes her own simulations calling all the necessary modules.

The most important pieces a simulation needs are

- the network,

- the environment,
- the node, and
- the simulation parameters.

The *Network* module implements the network. It is responsible for simulating the messages' traveling through it, the delays, the available bandwidth etc.

The *Environment* module generates data (in a rate) for the simulation. Our simulator is focused on stream, so the environment simulates a distributed stream.

The *Node* takes as input the hardware specification it supposed to have as well as the protocol. All this input comes from the user. It then is called by the simulation module.

Finally, other simulation parameters (such as network characteristics, protocols, etc.) are also being given as input by the user.

6.2 API ANALYSIS

In this section we attempt a more analytical description of API and its classes. We briefly describe each module and its functionality.

6.2.1 *User interface*

The userinterface package contains the readinput and streamhandler modules. The first is responsible for reading configuration files given by the user, while the second is responsible for handling and manipulating a stream. The only commitment the user has, is to give as input the path where the stream is. For more advance uses, (for example a different format in stream) the user gives as input the format of a stream's item (i.e. Integer-Integer- Integer- Character).

Let us analyze the simple case where the user wants to just change the data set. In this case, the user just gives to the simulator as input the path where the new data set is found. If the new data set has a different form of data from the default one, then the user just describes that form. For instance, if the user wants to override the current form (say it is I-I-I-I-C-C-C-C

[1]), she just types the new form (i.e. I-I-L-C-C). Note that there is no constraint both in type and length of the new form. The supported types are well documented.

Now, let us take into consideration the more complex scenario, where the user wants to deeply customize the system. Let us consider a user who wants to give unique names to the nodes and she also wants to connect the nodes in a particular way. The simulator supports configuration files as much customized as the user wants. Of course, the user needs to write some code so as to implement her special needs. It is not easy to predict and implement the user's wishes and needs, but an interface is offered to the user for doing this.

6.2.2 *Protocol*

The protocol package contains everything related to protocols. First of all, it contains some basic modules that implement math (vecOps.py module) as well as core modules (computationalmodel.py that implements the computational model, the monitoredfunction.py, an abstract protocol ready to implement, signals, system state variables, etc). Given these modules, a set of different protocol implementations have been made (decentralized protocol, ordinary node protocol, coordinator protocol). The most important (and independent) of them all is the protocol module. There, a basic API is required to be implemented by the advanced user, and then she is able to run any simulation she wants. This is a real simple API. It just requires the implementation of basic functionality: a sendMessage() function, a recvMessage() function and a broadcast one.

Of course, the user can use other modules as well. She can build on already given implementations of protocols, even integrate them to her code. For example, the user may want to use the decentralizedProtocol.py module, which implements the abstract protocol plus a few more functions in a complete way (unit testing included) and add her own code. This approximation and usage strategy though is risky, and it is not recommended, as it requires a complete understanding of the whole module's functionality. Moreover, it is possible for errors or bugs to exist (dragons ahead!). The *"clear"* way of implementing just the API is suggested, instead.

1 I corresponds to Integer while C: corresponds to character

Protocols are independent of the simulator. The only requirement of the simulator is a good implementation of the just three functions mentioned above. Other than that, no more dependencies exist. This is a useful feature because user can implement new protocols and run them to the simulator without any concern of external dependencies and increased debugging complexity. If her protocol operates in a wrong way, then the simulator guarantees that something goes wrong with her protocol and not into the simulator.

6.2.3 *Node*

The node package contains the basic elements a node consists of. It contains hardware specification in sources module (RAM, CPU, channels, etc), and a message data type, where all the information of the message (recipient(s), sender, content, size) is contained. The node module implements a node, but also a nodeProcess.py module, which is responsible for the simulation. NodeProcess.py module is a heavily dependent module and is not recommended to try to use it alone. Nevertheless, the node module is independent of the simulation and it can be used at will.

The node simulates a lot of things that are probably useless to some simulations (but at the same time, useful to some others). For instance, it implements CPU and RAM as resources that cost processing time if the user deems this feature necessary. In our exhibition simulations this feature is of no use, but another user would consider it essential to her simulations.

6.2.4 *Data*

Data package contains all files relative to data. It contains an AMS sketch implementation, probabilistic random number generators, operations to ams sketches. As mentioned above, the simulator is focused on simulating the protocol described in [**?** , lift:2006]ith heavy use of sketches. There is also a module named data.py. This last module exists in case the user wants a data type implementation of hers.

6.2.5 *Results*

Results package is responsible for the results collection and manipulation. Data that come of the program's execution are collected. When a simulation has finished, then this package is responsible for the elaboration of simulation's raw data and its presentation. A variety of functions implemented and are available to the user, so as she may select the most appropriate to her needs. The user can either have all the simulation's execution log file or just the processed, refined results of it. The user can also either save the results to one or more files, or just print them.

Monitoring data and results are deep intertwined. Monitoring elements are implemented in crucial points and monitor crucial variable and information of the simulation. After simulation's ending, they are summed up and combined so as to give the results. The only native, built-in monitoring element in our simulation concerns the network. Network data (such as number of messages and total traveling time of data into the network) are saved. In addition to the network monitoring, a lightweight monitoring is happening to the nodes, too. Data that accumulate in the nodes are also monitoring by our simulator.

Other kinds of monitoring should be implemented by the user in the protocols. Usually the user wants to measure her protocols' performance and overall attribution and hence, she is called to implement the monitoring elements needed. As it is expected, monitoring elements of the simulator are at her disposal as well.

6.2.6 *Simulation*

Finally, Simulation package is presented. This package is the heart of the simulator. It contains core modules such as network and environment, which implement what exactly their names describe, respectively. These modules are very important to any simulation and as a result they enjoy special treatment. They are two modules that can be used in every other kind of simulation and experiment.

NETWORK The network module is responsible for the network in the simulation. It has a really simple and straightforward API. Its primary job is to collect and redistribute mes-

sages. The network delays the incoming messages for a short period of time, in order to simulate the true delay which is observed during a message's sending.

Network has also physical restrictions and constraints, fully customizable. User can define the delay time as well as its capacity. She can also enable small crucial details such as message loss, if the protocols that she is called to implement, demand so.

ENVIRONMENT The environment module is perhaps the most important module of the simulator. It is responsible for coordinating all the modules. It reads the data set and produces streams, it finds the appropriate node for a stream item to be sent, etc. It includes the network and the nodes.

Simulation package also contains some simulations. In this package, simulations are created and run. This is where the user that runs conventional simulations, focus.

6.3 CODE DESCRIPTION

The UML class presented in Figure 7 does not present all pieces of software used for the implementation of Elita. There are lots of packages and sub-packages, as well as many more modules implemented which contribute to what Elita is.

It is not on the purposes of this chapter to show the analytical design tries, and all UML diagrams and classes that have been created and operate into Elita. More information can be found in Appendix. Thus, it deems appropriate that the user should have a complete picture of our work. Hence, the structure of the simulator is presented, with the help of the **tree** command [2] in a uniX-like system.

Listing 2: Project list of files

```
.
|-- code_samples
|   |-- events.py
|   `-- testEvents.py
|-- data
|   |-- ams_ops.py
|   |-- ams_ops.py~
|   |-- ams.py
|   |-- data.py
```

2 tree - list contents of directories in a tree-like format.

```
|   |-- __init__.py
|   `-- prng.py
|-- DEPENDENCIES
|-- __init__.py
|-- node
|   |-- __init__.py
|   |-- message.py
|   |-- node.py
|   |-- old-node.py
|   `-- sources.py
|-- protocol
|   |-- balancingprocess-old.py
|   |-- balancingprocess.py
|   |-- computationalmodel.py
|   |-- coordbasedProtocol
|   |   |-- coordProtocol.py
|   |   `-- nodeProtocol.py
|   |-- coordprotocol.py
|   |-- decentralizedProtocol.py
|   |-- __init__.py
|   |-- monitoredfunctionimpl.py
|   |-- monitoredfunction.py
|   |-- ordinarynodeprotocol.py
|   |-- protocol.py
|   |-- signals.py
|   |-- systemstate.py
|   `-- vec_ops.py
|-- README.txt
|-- results
|   |-- __init__.py
|   `-- results.py
|-- runit.py
|-- simulation
|   |-- coordbasedsim
|   |   |-- coordclass.py
|   |   |-- coord_sim.py
|   |   |-- final_sim.py
|   |   |-- __init__.py
|   |   `-- nodecoordclass.py
|   |-- environment.py
|   |-- __init__.py
|   |-- network.py
|   |-- nodeprocess.py
|   |-- sim_prototype.py
|   `-- simulation.py
|-- tests
```

```
|   |-- testAMS.py
|   |-- testCoordclass.py
|   |-- testCoordProtocol.py
|   |-- testDecentralizedProtocol.py
|   |-- testMessage.py
|   |-- testMonitorFuncImpl.py
|   |-- testNetwork.py
|   |-- testOrdinaryNodeProtocol.py
|   |-- testPrng.py
|   |-- testReadInput.py
|   |-- testStreamHandler.py
|   `-- testVectorOps.py
|-- test.txt
|-- TODO.txt
`-- userinterface
    |-- __init__.py
    |-- readinput.py
    `-- streamhandler.py

10 directories, 62 files
```

6.4 IMPLEMENTATION CHALLENGES

Here we describe some of the most interesting implementation challenges we met. Our target is to implement a good simulator that can really **simulate** real-life situations in high detail. Consequently, except to protocol special cases and problems, we deal with a range of problems about the network and the environment we simulate. Eventually the following questions occur:

- are the messages being sent in a non-deterministic way?
- more generally, is there any random factor in the system?
- is there enough randomness to ensure almost independent results and experiment executions?

Moreover, special irregularities arise. For the particular protocol we implement in order to exhibit our simulator, we have:

- What happens when two critical nodes send a message to each other?
- What happens when the system is to unsafe mode?

6.4.1 *Addressing Physical Challenges*

For the first set of challenges, we decided to attack the problem by importing to our system as much complexity as possible. Thereafter, we implement the environment and the network of our system with as much randomness imported as possible.

The environment reads a stream and sends its element to the nodes in a random way. It selects randomly a node and sends it an element. This procedure results to almost independent experiments and simulations, due to the fact that the nodes don't receive items from the stream in a determined way. Moreover, it is possible enough for a node to receive less elements than another node. Sometimes two, three, or more times less elements. This is also a real-life situation and results to more randomness in the system.

The network is programmed to add a random delay to each message. This simulates the delay of a message traveling through network. In addition, there is a possibility of having lost messages in the network. This feature is optional and it is not used in the simulations presented here. Though there is the probability of using this feature to other experiments.

6.4.2 *Addressing Protocol Challenges*

Protocols use to approach problems and describe algorithms and solutions from an abstract level. It is understood that they do not focus on practical problems or even theoretical extreme cases. The following problems represent a trivial, but still complex case that a simulator needs to solve, and a more difficult, almost to the limits of the protocol authorization.

SIMPLE SCENARIO: A node sends a REP message to the coordinator while at the same time, the coordinator sends to the node a REQ message. In this trivial, but yet special case a node observes a violation to its local constraint and sends its local statistics vector to the coordinator. At the same time, the coordinator is in the middle of a balancing process, which cannot yet balance, and hence it decides to request the statistics vector of the node. The messages happen to be sent and eventually arrive at the same time.

Node $\Longleftarrow$ receives REQ receives REP $\Longrightarrow$ Coord

The actions taken by each part are presented below:

Upon reception of the REQ message, the node ingores coordinator's request. It already has sent its last statistics. vector. Upon reception of the REP message, the coordinator just sends an ACK to the node. Even if a new REQ message arrives from the same node, this node will be in the balancing group and hence it is ignored.

COMPLEX SCENARIO: A node has stated that is in USAFE mode but items keep coming.

In this more complex case, a node observes a violation in its local constraint. It sends a REP message so as to notify the coordinator, but the data stream keeps flowing and new data arrive. Now the node has to deal with the situation. Should it keep collect the new data as they come or should ignore them, until the coordinator sends a notification?

The algorithm that is used is the following:

Algorithm 6.4.1: UNSAFE MODE(−)

---

comment: item is an item from stream.

$node \leftarrow$ SEND(coord, REP)

while REQ is not coming

do { **if** item has come:
  then { APPEND(list, item)

SENDTOSKETCH(list)

comment: Now, notify for new data arrival.

LOCALARRIVAL(arrival)

ALGORITHM EXPLANATION: Let us consider the scenario described above. Summing up, the node has to select between two choices: Either the node continues to update its local statistics vector, or stop updating it and wait for an answer. Still the node can not simply ignore the incoming data. The algorithm we propose, tries to combine both of the solutions above.

Therefore the node maintains another list where the stream is "pinning" the items. Then, it withdraws the items for this list. In other words this list of items is a broker between the environment and the protocol. If for some reason node's state is unsafe, then no items are promoted to the local statistics vector. On the contrary, the items wait there until the node is in safe mode. Then, they all are directed to the local statistics vector, as a batch of data. Since the list may be very big, we consider the local statistics vector as as sketch data structure Section 2.2.1, which have a great advantage over the conventional vectors, both in terms of space and time complexity.

EPILOGUE Concluding, we would like to point out that our simulator meets its purpose. It is a robust, flexible simulator, which simulates in great detail protocols and systems. Particular attention has been given to its portability. Moreover, our simulator comes ready to be used. A suite of tests supports and enhances it.

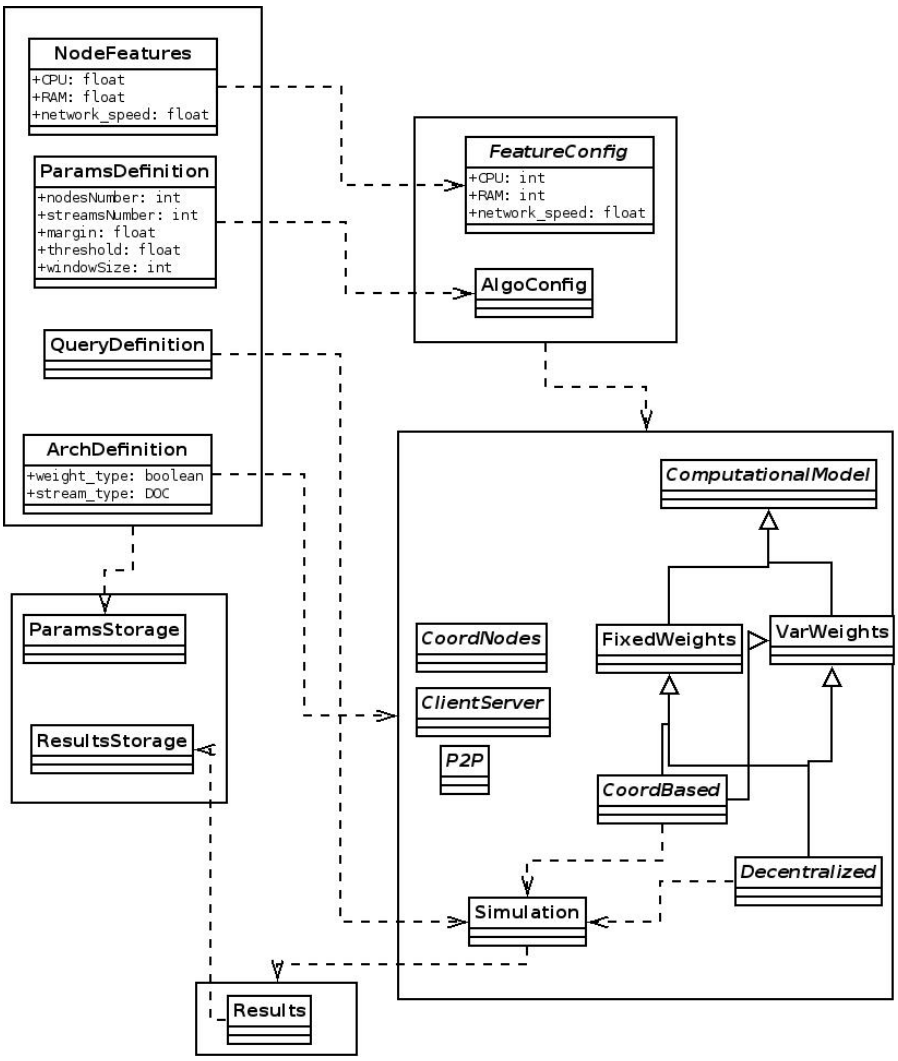


Figure 6: A more detailed sketch of **??**. Sub-systems that implement each system are presented.

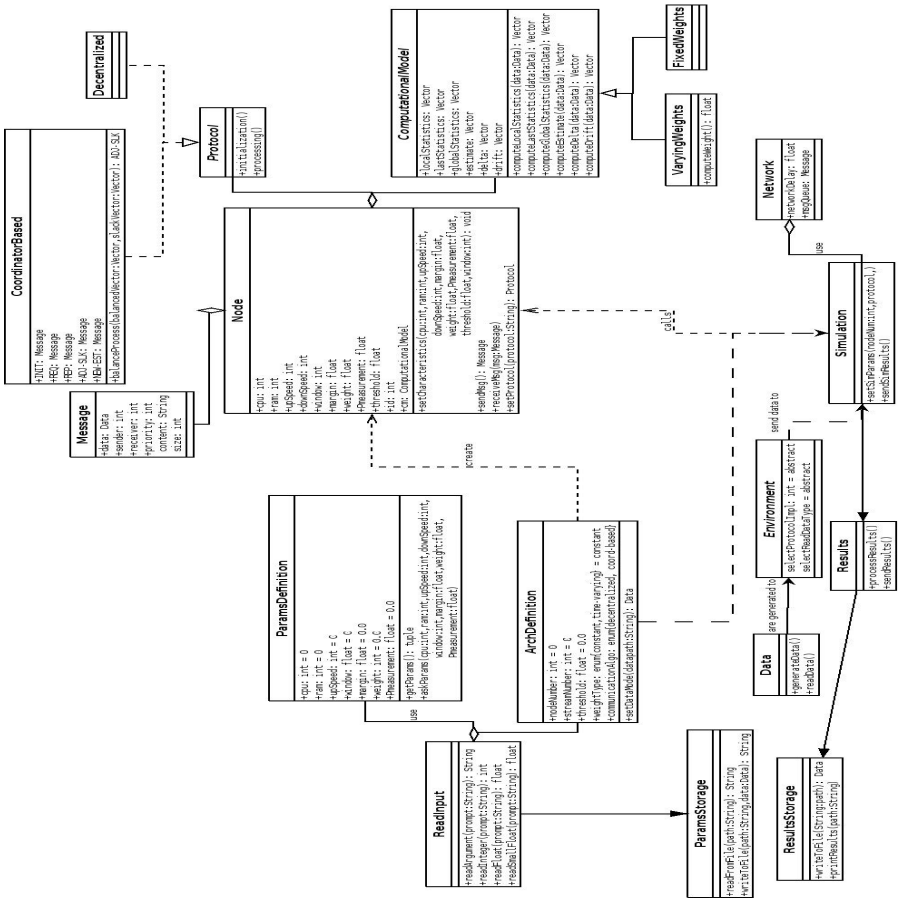


Figure 7: The uml diagram of main classes (that consist the API of our simulator) is presented. Diagram has been rotated 90 degrees.

Part III

THE SHOWCASE

Results are presented in this part. A brief set of conclusions is presented. Last chapter contains suggestions for future work on the simulator.

7

RESULTS

7.1 PUTTING IT ALL TOGETHER

Summing up, we created a simulation as described below: First of all, we built Elita. We also wrote for Elita the protocols described in [15]. Our main focus as well as the results we present are for the coordinator based protocol specifically. An additional problem with this protocol is the really big lists of waiting data that occur Section 6.4.2. Hence, taking advantage of Elita's flexibility features, we replaced the vector data structures in the computational model of the protocol with sketches. Now, Elita operates on sketches instead of heavy, large vectors. To sum up, until this point, we have a simulator written in Python and powered by SimPy, which is ready to run the protocols cited above but uses sketches instead of vectors.

Our only outstanding is the monitoring of a stream. The stream is produced by the environment. This is not happening in a random way, but instead, the WCup data set is used. Section 7.3.1 This data set is the data collected by many servers during a day of the World Cup '94 [4].

An exhibition of our simulator follows. The simulator runs the coordinator-based protocol and collects several different metrics. The results that are produced, they are compared to these in [15].

The protocol runs for several different parameters. The number of nodes is a variable. Another variable is the rate the items arrive to the nodes. The simulations run with full randomness enabled (both in network travel times and item arrivals in the nodes). No random loss is imported though. In other words, there is no message loss through the network.

THRESHOLD AND MONITORING FUNCTION Another important issue to the simulations is the monitoring function and the threshold. The problem becomes more interesting when every time there is a violation, a new threshold is computed. In this way the problem approaches even more a real situation. The way the new threshold is computed, is presented below.

Let a system be as described above. A variable with time threshold is presented.Let a function f be, where

$$\|f(sk)\| = median\{\|sk[j]\|^2, j = 1, \ldots, d\}$$

This function is the median of the norms of the sketches that a node maintains. The monitoring function checks if the new data affect to a significant extent the already existed threshold. To restate, the monitoring function checks if the new f, after the updating stage, is quite different from the original one. It checks the condition

monitoring function

$$f(sk) < (1 + \theta) f(sk)_{old}, \text{ where } \theta < \epsilon \quad [1]$$

If a violation is observed and after balancing process, a re-computation of new threshold is inevitable, the coordinator computes the new threshold.

7.2 DATA SUMMARIES

What we are interested in simulating the particular protocol, is some information about the waiting time of items before the node inserts them to its sketches and their size as well. Moreover, we are interested in state changes of the nodes. In particular, we are interested in the percentage of time during which a node is in safe mode or not. Furthermore, we are interested in balancing process duration and some protocol qualities. Finally, we are interested in network's metrics, both quantitative and qualitative. Thereafter we measure the total number of messages in the network and the total time of messages' traveling through network.

A feature of our simulator is that it gathers data and data summaries (count, total, mean, variance, time average, time variance - [6]) but at a performance cost. Due to the fact that we aim to a good simulator with detailed environment, we have to sacrifice performance for detailed and completed data summaries.

7.3 EXPERIMENTS

A set of simulations and experiments has been done using our simulator. During protocol testing, we focus on different environments and situations. We also pay special attention to the physical constraints that apply to the nodes and the stream.

1 θ is the margin of difference between the new and the old functions. ϵ is the margin error of sketches. Usually ϵ is much smaller than θ.

PARAMETERS The parameters we mostly are interested in are the number of nodes and the rate with which data arrive to each node. Passing different values to each parameter, we evaluate the performance, the possible bottlenecks, the efficiency, etc.

We run several simulations for different numbers of nodes (2, 4, 6, 8, 10, 15, 20, 30, 40, 50) and different values of rate 90.2, 0.4, 0.6, 0.8, 1.0, 2.0, 3.0, 6.0). Of course, results are heavily affected by stream's format and structure.

7.3.1 *Stream*

Some details about the stream should be presented here. Our stream comes from the wc day44 data set. This data set meets all the requirements for giving birth to a distributed data stream. It consists of the data collected by a distributed system of servers in a single day of the World Cup '94. In particular, it contains all the connections done that day.

Its structure is analyzed below. It is a binary file which contains millions of tuples (each one is a connection between the server and a random user's pc). A tuple's format is following:
(time stamp, node id, data, size of request, ...)
We obviously are interested in the fields that are described only. A characteristic of the stream is that it contains twenty to twenty-five same time stamps. This is a very convenient feature, because in this way, a concurrent item sending can be simulated. This very feature can also become a little tricky and potentially create a little anomaly to the results in few nodes setup. This happens because we send each tuple in the node with id that comes up from the following formula:

$$\boxed{\mathrm{id} = (\mathrm{timestamp} \mod \mathrm{nodesnumber})}$$

That means that in a setup with three only nodes, the nodes will probably receive several tuples each in a moment, while in a setup with let us say twenty nodes, the nodes will receive one or two tuples each, in a moment. This means that many more balancing processes will probably happen, as the sketches are updated much faster. In other words, a fair comparison between a twenty-node setup and a three-node setup would happen if the first setup has a five or maybe six times bigger rate than the other.

7.3.2 *Physical constraints*

In order to run simulations in a great detail and test the protocol in real-time conditions, we arbitrarily applied a physical constraint to our network. We decided to limit the number of messages that can concurrently travel through the network to ten. In this way, we could observe the network's performance in a heavy loaded environment.

Results are presented now:

EXPLANATION: We choose to present figures like Figure 11 because in our opinion this is the best way for a simulator to be exhibited. This plot's behavior is not foreseen by the protocol and it presents a new, yet inconvenient situation. Explanation for this is that that the setup with 2 nodes, updates its nodes six or more times more frequently than the setup with the 7 nodes does. As a result, a balancing process is happening more often (and each occurrence adds a little bit of time to the total waiting time).

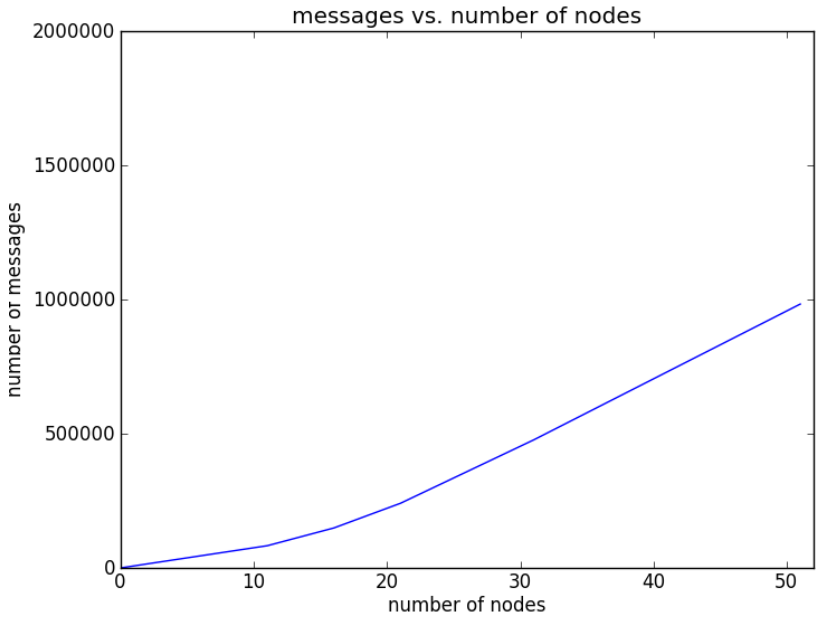


Figure 8: Messages number vs. nodes number. Rate here is 0.2.

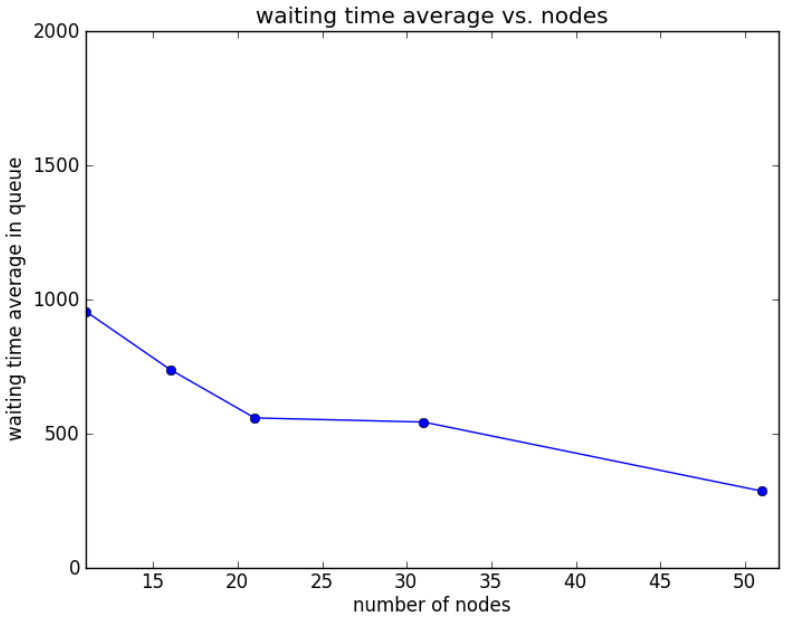


Figure 9: Messages are waiting before a node sketches them. Here the waiting time average to such a queue (but for all nodes) is presented. This figure needs a second read to be well understood. Waiting time average decreases over time. This makes sense, as even if a few nodes may accumulate many items during balancing process, the majority of them runs smoothly. Rate here is 0.2, time in miliseconds.

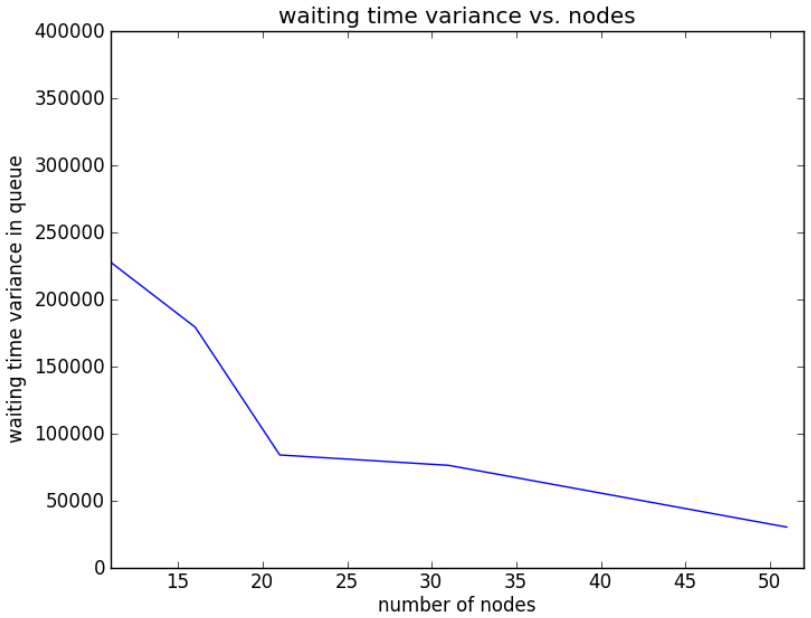


Figure 10: Messages are waiting before a node sketches them. Here the waiting time variance to such a queue (but for all nodes) is presented. Rate here is 0.2, time variance is squared.

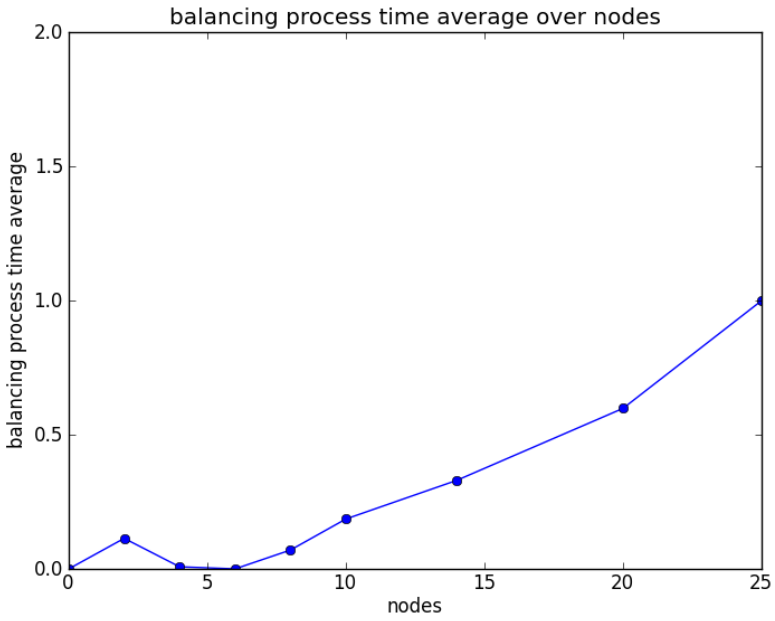


Figure 11: A balance process's duration is presented. Here an early peak is presented. This is because of the stream's structure Section 7.3.1. Rate here is 1, time is in seconds.

7.3.3 *Complex Figures and Results*

We present figures about three different monitoring stages. We monitor:

- the balancing process.
- the items being accumulated in the list of each node (before being placed to sketches) .
- the state of each node (safe-unsafe, depending on the balancing process).

FIGURES EXPLANATION Each figure shows one or more simulation results. The time average, time variance, mean an variance values are the mean values of all nodes for a simulation setup. For example, we compute the mean value of the waiting time average of all nodes and we present it.

7.3.3.1 *Data accumulation monitoring*

Here we present figures that are relative to Section 7.3.3. It refers to the accumulated items in a node, before the node sketches them. We present the system's behavior for a variety of different rates and number of nodes, as well.

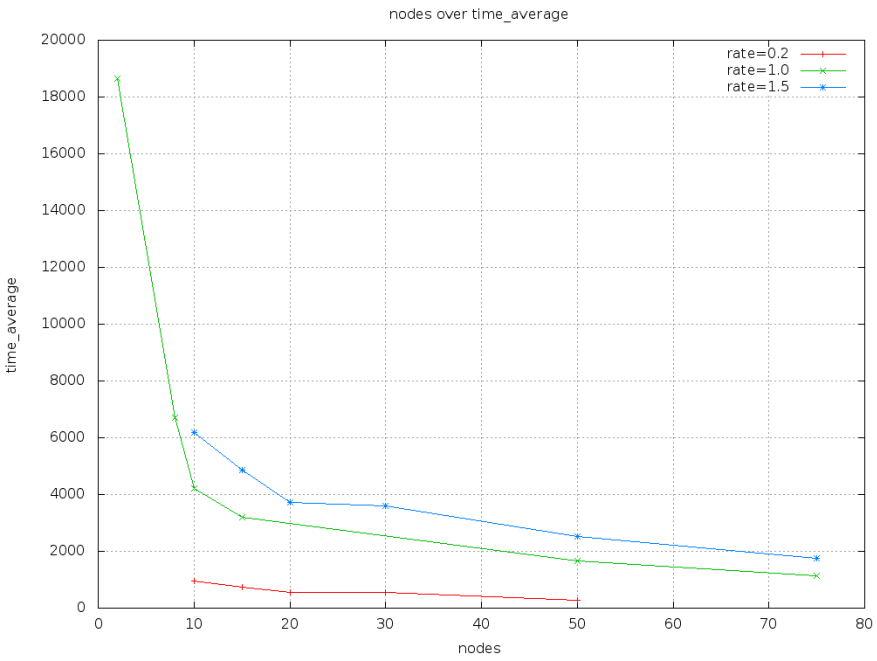


Figure 12: Mean waiting time average to the lists of the nodes. Green line (rate 1.0) presents the waiting time average for only few nodes. Explanation of the really high waiting time average is as follows: Nodes are too few and they all participate in the balancing process. Meanwhile, due to the nature of the stream, items tend to come more rapidly to the few nodes setup rather than a setup with more nodes. *Time in miliseconds.*

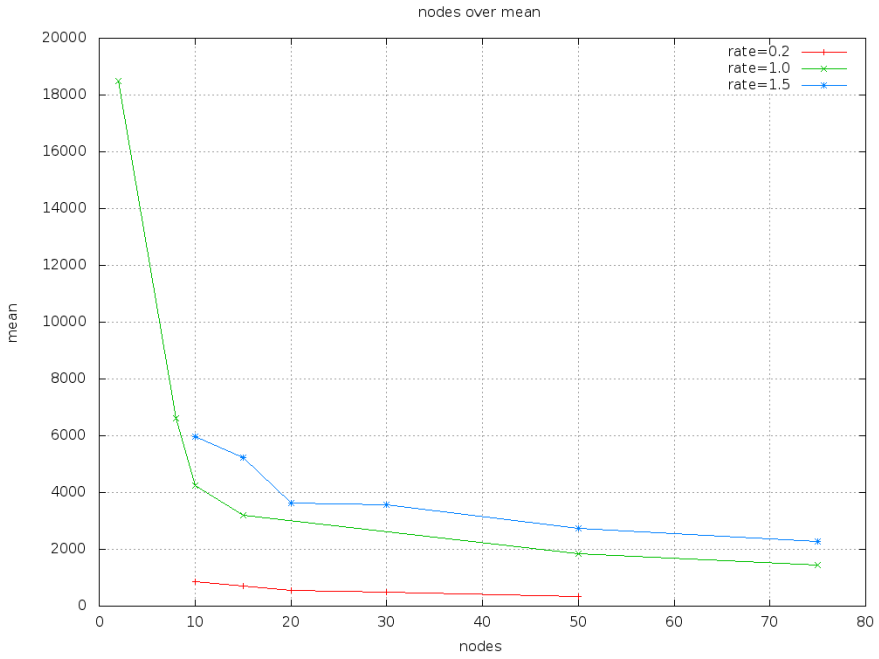


Figure 13: Mean value of waiting items is not much different than the waiting time average.

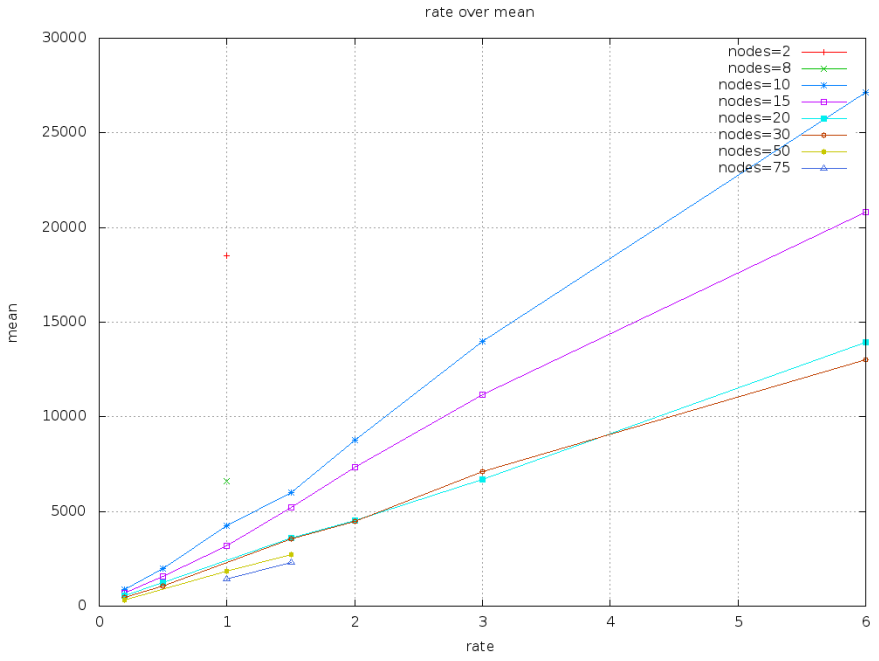


Figure 14: Total comparison figure. All nodes are represented here, for different rate values. Rate over mean is represented, for different number of nodes.

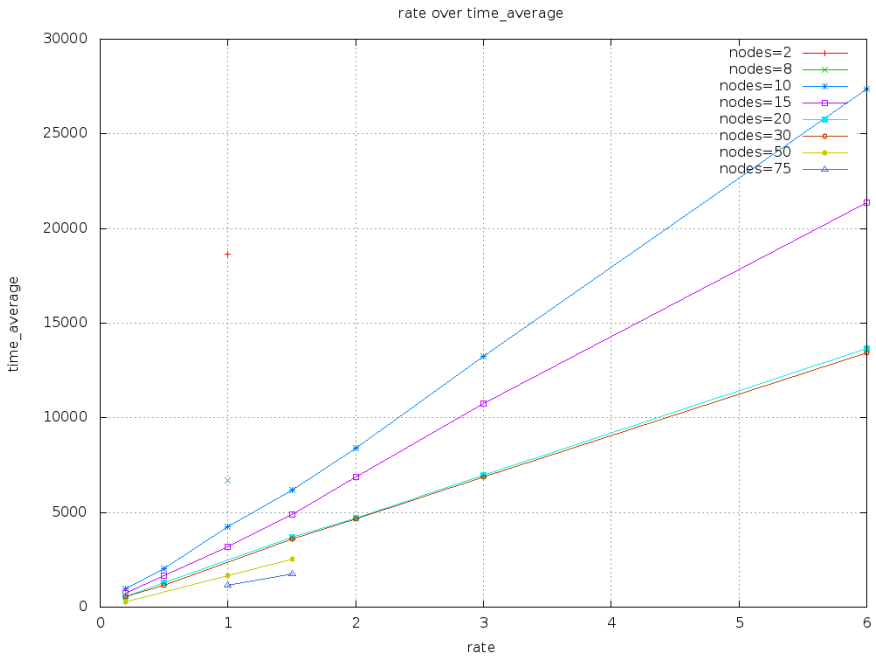


Figure 15: All nodes are represented for different values of rate. Time is in miliseconds.

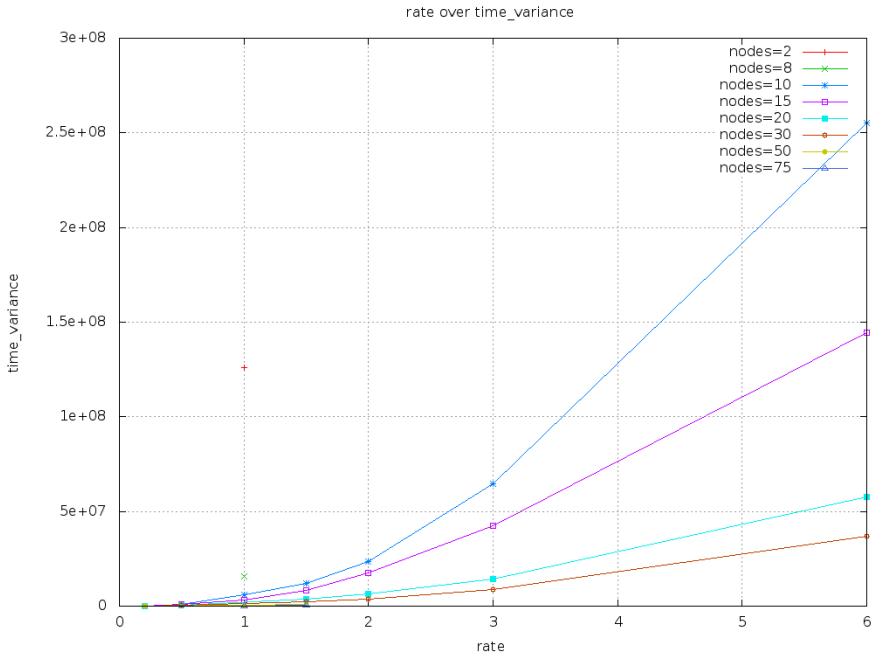


Figure 16: Wait time variance vs rate is presented here. Note that time variance has been squared.

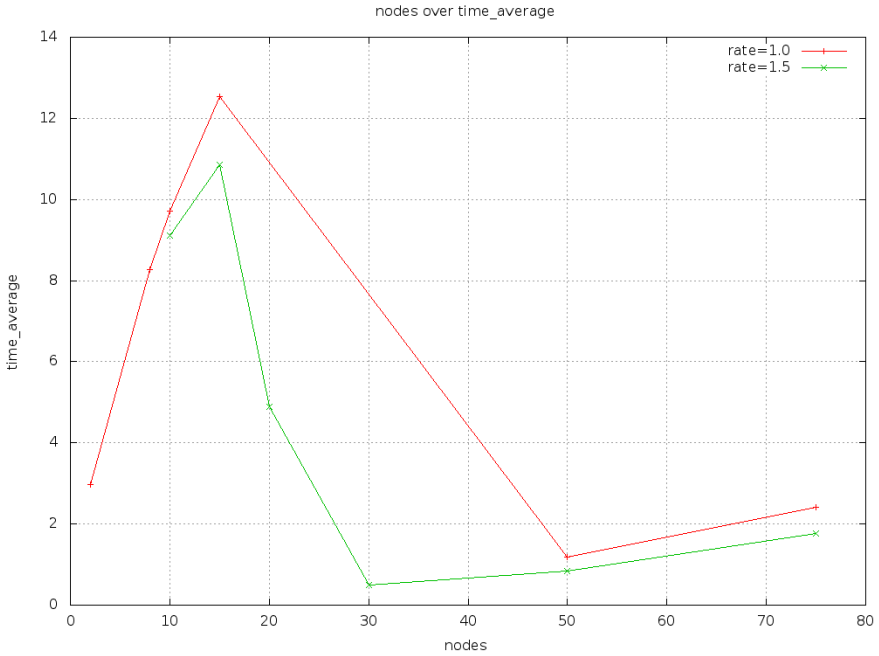


Figure 17: Let us have a function f where $f(x) = 1$, if state:safe and $f(x) = 0$, elsewhere. The more time $f(x) = 1$, the higher value will have in time. Hence, this figure makes sense, because for a bigger rate, more items are coming at the same time to the nodes and as a result, more balancing processes happen. *Time is in sec.*

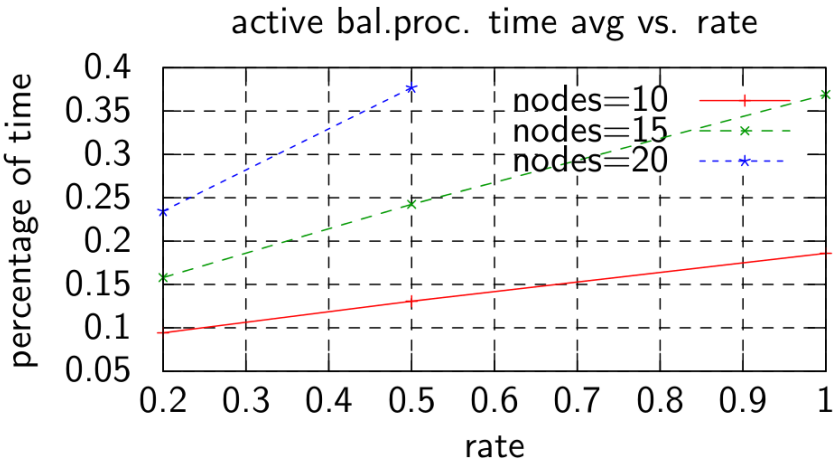


Figure 18: Active balancing process percentage of time vs rate

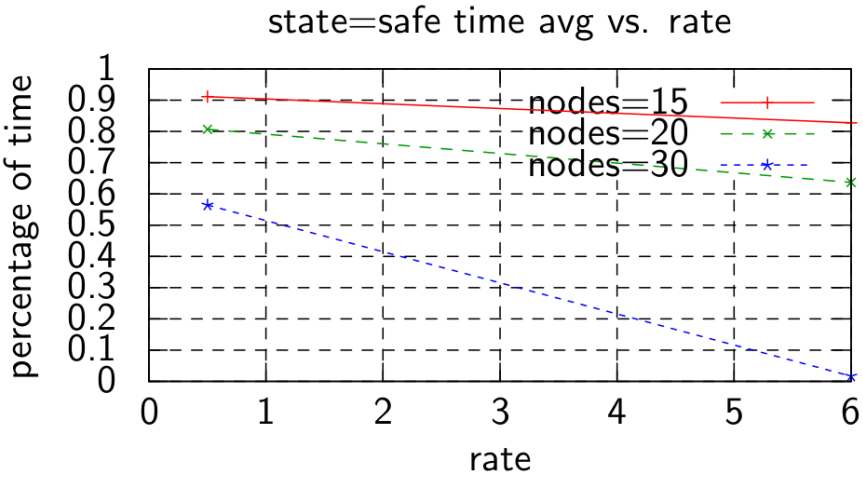


Figure 19: Active balancing process percentage of time vs nodes. While rate becomes bigger, the time a node is in safe state, becomes smaller.

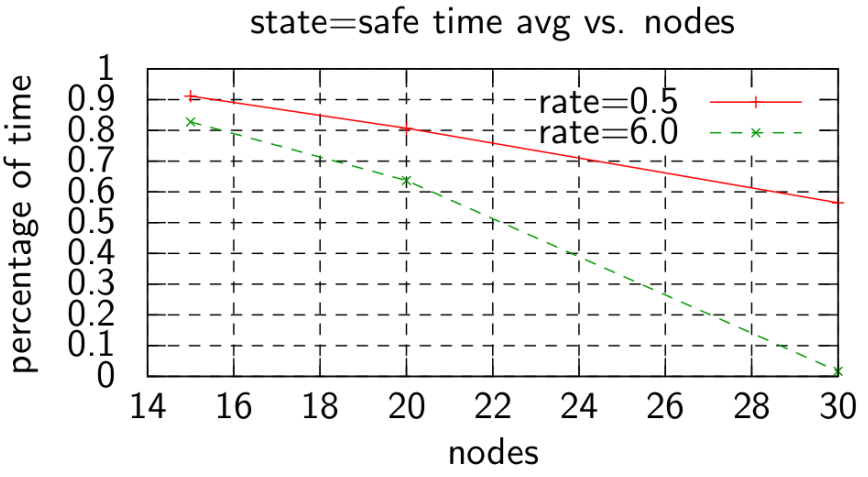


Figure 20: Active balancing process percentage of time vs nodes. While rate becomes bigger, the time a node is in safe state, becomes smaller.

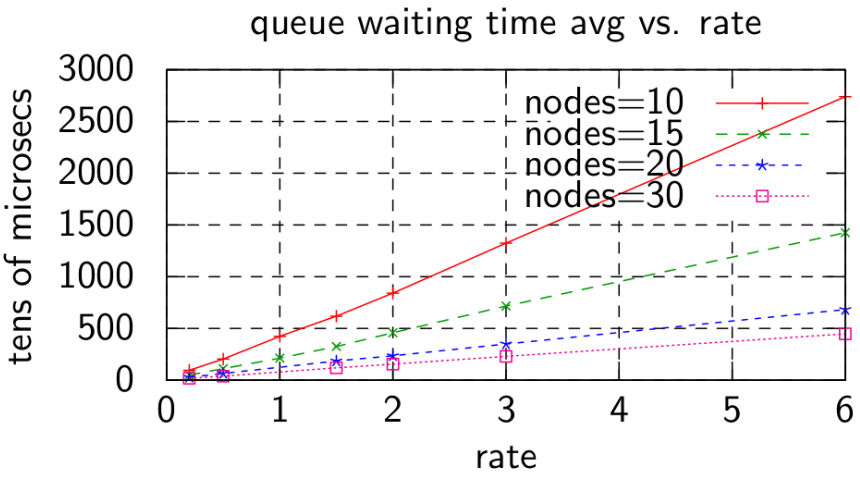


Figure 21: Average of waiting time of an item in a temporary queue vs. rate. The reason why the waiting time is bigger for more less nodes in our setup, is due to the specific form of our stream (see Section 7.3.1).

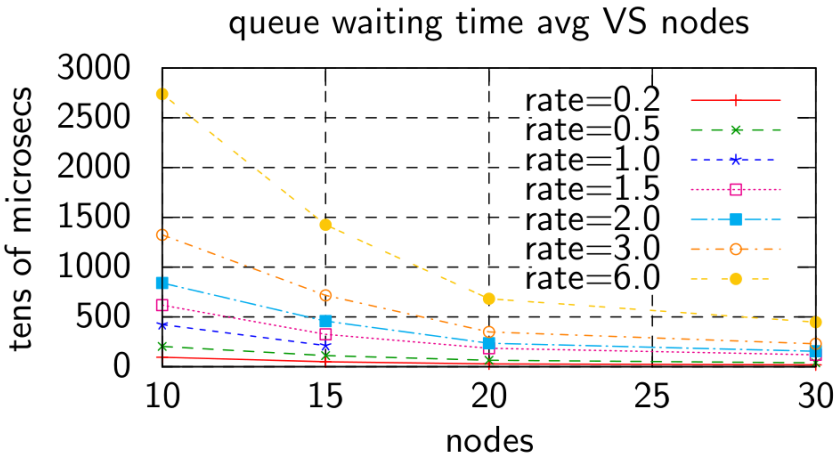


Figure 22: Same metrics as in figure Figure 21.

8

CONCLUSIONS / FUTURE WORK

8.1 CONCLUSIONS

After careful examination of the results, the user could note the usefulness of a good simulator. We have more than enough figures during which interesting conclusions and results come up. Through simulations we noticed the effectiveness of the protocols tested. We also took useful conclusions about protocol's performance. After extensive simulating of the protocol, we can predict its overall performance with accurate estimations.

Moreover, useful observations came up, like potential bottlenecks in the number of maximum nodes permitted to participate in the system or the maximum range of rate values.

INDIRECT RESULTS In addition to the above, some interesting stats came up. An example is that the waiting time average is smaller, if the number of nodes becomes bigger. That is happening due to the fact that not all the nodes participate to the balancing process. Hence, while some of them block the incoming items waiting for answer from the coordinator, the majority of system's nodes continues their operation, without intermission. Another reason for this is that items are coming to a lower rate (due to the form of the stream).

OVERALL CONCLUSIONS Finally, we can decide on the overall effectiveness and performance of the tested protocol. Combining all the individual sub-conclusions, we can evaluate the protocol as not easily-scalable over the number of nodes and over the rate, too. That means that by increasing the number of nodes or the rate, the protocol becomes especially "heavy", in terms of computational effort. Moreover, nodes that are found in unsafe mode, wait a long period of time being stand-by, and as a result, more and more items are accumulated to the nodes without being processed.

8.2 FUTURE WORK

8.2.1 *More protocols to test*

A significant amount of work can be done in testing a batch of different protocols. The protocols referred to [15] have been implemented, as well as the basic functionality of any protocol (sending messages, receiving messages and broadcasting messages is some of it). Still, many protocols should be further implemented. This will strengthen the simulator as a useful and productive tool, with reliable results. Moreover, implementing a lot of protocols could lead to many "hidden" situations to be revealed. In other words, situations and cases that are not covered by a protocol could come to the foreground and be revealed. This can lead to a more detailed and strict evaluation of the efficiency of a protocol. It can also lead to modifications or suggested improvements. An example of this has already been given in Section 6.4.2. Rare cases that a protocol only "confronts" in an abstract, high-leveled way or does not confront them at all, should be written down and be confronted.

hidden cases

8.2.2 *High-end user interface*

At this time, the simulator offers a wide range of ready-to-use interfaces. The user can just import her data to the system, if she wishes to execute a simulation exactly like the simulations that are presented here, or she can heavily modify the input making use of configuration files. The simulator implements an API which prompts the user to use configuration files (.conf) the way she likes it and totally modify the system (from the data, to the names of each node and their unique connections).

However, the simulator is addressed to the user who is familiar with Linux and Unix environments and has a little knowledge of shell commands. In other words, a graphical interface is missing. SimPy offers libraries which can help for building a graphical interface. By having a graphical interface, the system hides a big amount of its complexity while at the same time it becomes more attractive and eye-candy.

8.2.3 *Interconnection with other simulators*

Another future work for this simulator would be the interconnection of parts of the simulator (or even the whole simulator)

with other simulators. A very progressive step towards simulation would be the use of our simulator or parts of it to other simulators. For instance, the module that implements the network, it could be used unchanged by another simulator.

8.2.4 *Distributed System*

A high-quality feature, (even if its implementation would be of increased complexity) that could be added in the future is a distributed execution of the simulator. As of now, our simulator is not distributed. This means that it runs in a procedural way. Such a feature would be a great improvement over the simulator. Let us think the protocol we tested and examined in this thesis. Now, let us think of it operating in a distributed system. That means that every node (or small set of nodes) could run in one single node of a distributed computer system (i.e. grid) while the coordinator could acquire another node only for itself. Such a possibility could make the execution of simulations much faster, almost real-time. In this way, the only performance bottleneck in our system would be the network.

Of course, such a feature requires totally different algorithms, libraries and dependencies than the ones used in this thesis. Distributed algorithms are a great challenge but also add complexity to the project. Even more issues arise of such an approximation, including data safety algorithms and confidence guarantees about data receiving.

8.2.5 *Map Reduce Integration*

Last but not least, simulator's tasks parallelism could be another challenge. As long as we refer to streams monitoring and manipulation, and to data mining in general, we should always consider the size of our data. The latter, combined with the one-pass feature of the streams, demands extremely powerful machines to be effectively processed. To our disappointment, extremely powerful computers still have high costs and solutions like parallelism are preferred. Hence, a map-reduce implementation of demanding computational tasks that a detailed simulation could have, is crucial.

Of course, this challenge is advanced and even more complex than the above, as it usually demands a "parallel" way of thinking. Moreover the problems are way more complex, because not only should we confront the problems of the simulation effec-

tively, but also we have to think how the parallelization should work.

****

Part IV

APPENDIX

A

APPENDIX

A.1 TECHNICAL DETAILS

The concept of this thesis has been made exclusively with free and (only in a few points) open source software. We implemented our simulator in python (Section 2.4) and we also used some shell scripts (shell we used is the bash shell Bourne Again Shell (BaSh)). All the shapes and sketches have been designed in free software applications (dia and gimp in particular) as well. Finally, this thesis is written in LaTeX . We tested the simulator in a variety of different systems so as to be sure for its portability. Hence, the simulator has bee tested in both debian and fedora-based systems. In particular, we tested it in 64-bit Ubuntu, Debian and Fedora. Moreover, we ran the simulator in the Technical University of Crete's grid system (which runs a redhat operating system).We ran heavy simulations in grid in Torque (free software-[7]).

BIBLIOGRAPHY

[1] Charu C. Aggarwal and Philip S. Yu, editors. *Privacy-Preserving Data Mining - Models and Algorithms*, volume 34 of *Advances in Database Systems*. Springer, 2008. ISBN 978-0-387-70991-8.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

[3] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[4] Martin Arlitt. World cup 1994, 1994.

[5] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends databases*, 4 (1–3):1–294, January 2012. ISSN 1931-7883. doi: 10.1561/1900000004. URL http://dx.doi.org/10.1561/1900000004.

[6] SimPy developer team. Simpy data summaries, 2004-2012.

[7] Altair Engineering. Torque portable batch system, 1991.

[8] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, September 1985. ISSN 0022-0000. doi: 10.1016/0022-0000(85)90041-8. URL http://dx.doi.org/10.1016/0022-0000(85)90041-8.

[9] Minos Garofalakis. Distributed data streams.

[10] Donald E. Knuth. Computer Programming as an Art. *Communications of the ACM*, 17(12):667–673, December 1974.

[11] Fred Kuhl and Richard Weatherly. Tortuga, 2004-2006.

[12] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004. ISBN 0735619670, 9780735619678.

[13] S. Muthukrishnan. Data streams: algorithms and applications. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '03, pages 413–413, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics. ISBN 0-89871-538-5. URL http://dl.acm.org/citation.cfm?id=644108.644174.

[14] Robert M. Pirsig. *Zen and the Art of Motorcycle Maintenance: An Inquiry into Values*. William Morrow and Company, 25 edition, 1974.

[15] Izchak Sharfman, Assaf Schuster, and Daniel Keren. A geometric approach to monitoring threshold functions over distributed data streams. *ACM Trans. Database Syst.*, 32(4), 2007.

DECLARATION

Put your declaration here.

Technical University of Crete, February 2013

Babis Babalis